



A Siemens Business

Tessent® Scan and ATPG User's Manual

Software Version 2019.2

Document Revision 13

© 1999-2019 Mentor Graphics Corporation
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

Note - Viewing PDF files within a web browser causes some links not to function (see [MG595892](#)).
Use HTML for full navigation.

This document is for information and instruction purposes. Mentor Graphics 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 Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. 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 Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: mentor.com/trademarks.

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.

End-User License Agreement: You can print a copy of the End-User License Agreement from: mentor.com/eula.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: mentor.com
Support Center: support.mentor.com

Send Feedback on Documentation: support.mentor.com/doc_feedback_form

Revision History ISO-26262

Revision	Changes	Status/ Date
13	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 May 2019
12	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 2019
11	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 Dec 2018
10	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 Aug 2018

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 Mentor Graphics Technical Publication's source. For specific topic authors, contact Mentor Graphics Technical Publication 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		
Overview		27
What is Design-for-Test?.....		28
DFT Strategies		28
Top-Down Design Flow with DFT		29
Chapter 2		
Scan and ATPG Basics.....		31
Scan Design Overview.....		32
About Scan Design Methodology.....		34
About Wrapper Chains		35
Test Structure Insertion with Tesson Scan.....		37
ATPG Overview		39
The ATPG Process		40
Random Pattern Test Generation		40
Deterministic Pattern Test Generation		40
External Pattern Test Generation		41
Mentor Graphics ATPG Applications		41
Scan Sequential ATPG with the ATPG Tool		41
Overview of Test Types and Fault Models		43
Test Types.....		44
Functional Test.....		44
IDQ Test		44
At-Speed Test.....		46
Fault Modeling Overview.....		47
Test Types and Associated Fault Models.....		47
Fault Locations.....		47
Fault Collapsing		48
Supported Fault Model Types		49
Fault Manipulation		54
Overview of Fault Manipulation		54
Fault Manipulation Functionality.....		54
About User-Defined Fault Modeling		56
User-Defined Fault Modeling Usage Notes		56
UDFM File Format.....		58
Creating a User-Defined Fault Model		63
Generating UDFM Test Patterns		64
Multiple Detect		66
Bridge Coverage Estimate		66
Embedded Multiple Detect.....		68

Fault Detection	69
Fault Classes	72
Untestable (UT)	72
Testable (TE)	75
Fault Class Hierarchy	77
Fault Sub-classes	77
Fault Reporting	84
Testability Calculations	84
Chapter 3	
Common Tool Terminology and Concepts.	87
Scan Terminology	89
Scan Cells	89
Master Element	90
Slave Element	90
Shadow Element	91
Copy Element	92
Extra Element	93
Scan Chains	93
Scan Groups	94
Scan Clocks	95
Scan Architectures	96
Test Procedure Files	96
Model Flattening	98
Design Object Naming	98
The Flattening Process	99
Simulation Primitives of the Flattened Model	100
Learning Analysis	104
Equivalence Relationships	104
Logic Behavior	104
Implied Relationships	105
Forbidden Relationships	106
Dominance Relationships	106
ATPG Design Rules Checking	107
General Rules Checking	107
Procedure Rules Checking	107
Bus Mutual Exclusivity Analysis	108
Scan Chain Tracing	109
Shadow Latch Identification	109
Data Rules Checking	110
Transparent Latch Identification	110
Clock Rules Checking	111
RAM Rules Checking	111
Bus Keeper Analysis	111
Extra Rules Checking	112
Scannability Rules Checking	112
Constrained/Forbidden/Block Value Calculations	112
Clock Terminology	114

Table of Contents

Programmable Clock Chopper	114
Chapter 4	
Testability Issues.....	115
Synchronous Circuitry	115
Asynchronous Circuitry	116
Scannability Checking	116
Support for Special Testability Cases	118
Feedback Loops	118
Structural Combinational Loops and Loop-Cutting Methods	118
Structural Sequential Loops and Handling	124
Redundant Logic	125
Asynchronous Sets and Resets	125
Gated Clocks	126
Tri-State Devices	127
Non-Scan Cell Handling	127
Clock Dividers	131
Pulse Generators	132
JTAG-Based Circuits	133
RAM and ROM Test Overview	134
RAM/ROM Support	135
RAM/ROM Support Commands	138
Basic ROM/RAM Rules Checks	139
Incomplete Designs	140
Chapter 5	
Internal Scan and Test Circuitry Insertion	143
Introduction to Tessent Scan	145
Features of Tessent Scan	146
Tessent Scan Inputs and Outputs	146
Invoking Tessent Scan	148
Example Dofile	148
Test Structures Supported by Tessent Scan	149
Tool Terminology and Concepts	151
Scan Element	151
Scan Chain Family	153
Scan Mode	154
Unwrapped Cores Versus Wrapped Cores	155
Pre-Existing Scan Segments	156
Pre-Existing Scan Chains	158
Preparation for Test Structure Insertion	159
Scan Cell and Scan Output Mapping	159
Test Logic Insertion	160
User Clock Signals	163
How to Specify Existing Scan Information	164
How to Handle Existing Boundary Scan Circuitry	166
How to Run Rules Checking	166
Setup for a Basic Scan Insertion Run	167

Setup for Wrapper Chain Identification	168
Wrapper Chains Generation	168
Wrapper Cells Analysis	170
Clock Selection for Dedicated Wrapper Cells	174
How to Constrain Input Partition Pins	175
How to Mask Output Partition Pins	175
Child OCC Schematic	175
How to Manually Include and Exclude Cells for Scan	176
How to Report Scannability Information	178
Automatic Recognition of Existing Scan Chains In Cell Library Models	179
Automatic Recognition of Existing Shift Registers	182
Scan Cell Identification and Reporting	185
Multi-Mode Chains	186
Introduction to Multi-Mode Chains	186
Multi-Mode Gating Logic	187
Multi-Mode Scan Insertion	188
Chain Port Sharing	189
Population Control	190
Scan Insertion Flows	192
Scan Insertion Flow Steps	192
Controlling Scan Insertion	193
Scan Insertion for Unwrapped Core	193
Scan Insertion for Wrapped Core	197
Scan Insertion at the Parent Level	202
Dedicated Wrapper Cell Placement After Loading UPF/CPF File	204
Chapter 6	
Test Point Analysis and Insertion	207
What are Test Points?	208
Why You Use Test Points	208
Control Points and Observe Points	209
Control Points	209
Observe Points	210
Test Point Insertion Flows	212
Analyze and Insert Test Points in a Pre-Scan Design	213
Analyze and Insert Test Points in a Post-Scan Design	215
How to Insert Test Points and Perform Scan Stitching Using Third-Party Tools	220
How to Insert Test Points for Design Compiler or Genus™ Synthesis Solution	220
Test Point Usage Scenarios	224
Test Points for ATPG Pattern Count Reduction	225
Test Points for Reducing Pattern Count	226
Requirements for Test Point Analysis and Insertion	226
Specify Test Point Type	226
Inserting Test Points When Using the EDT Skeleton IP Flow	228
Test Points for LBIST Test Coverage Improvement	229
Test Point Analysis and Insertion Step Overview	229
Test Point Analysis With Small LBIST Pattern Counts	230

Table of Contents

Test Coverage Reporting During Test Point Analysis for LogicBIST Coverage Improvement	231
Test Points for Hybrid TK/LBIST Design	234
Improving Test Coverage and Pattern Count Reduction	234
Test Points for ATPG Test Coverage Improvement.	236
Test Points for Improving the Test Coverage of Deterministic Patterns	236
Test Points Special Topics	240
Control Point and Observe Point Enable Signal Handling	240
Test Point Analysis With Multiple Power Domains	241
Test Point Analysis Multicycle and False Path Handling.	242
Test Point Analysis Critical Path Handling	244
PrimeTime Script for Preventing Test Points on Critical Paths	245
User-Defined Test Points Handling	247
Test Point Deletion	249
delete_test_points Command	249
Modifying the Test Point Dofile	249
Back to Back Command Handling	250
Static Timing Analysis for Test Points	250
Test Points and At-Speed Testing.	251
Clock Selection for Test Point Flops	252
Example of Test Point Insertion With Custom Prefix for Inserted Logic	252
Chapter 7	
Running ATPG Patterns	255
Running ATPG Patterns after Tessent Scan	255
Running ATPG Patterns without Tessent Scan	257
Core Mapping for ATPG Process Overview	257
Core Mapping Examples	265
Limitations	271
Chapter 8	
Test Pattern Generation.	273
ATPG Basic Tool Flow	275
ATPG Tool Inputs and Outputs	277
ATPG Process Overview	279
Basic ATPG Process.	279
ATPG Tool Timing Model	280
ATPG Tool Pattern Types	280
ATPG Procedures	285
Tool Invocation	285
Circuit Behavior Setup	286
Equivalent or Inverted Primary Input Definition	286
Primary Inputs and Outputs Addition.	286
Bidirectional Pins as Primary Inputs or Outputs	287
How to Tie Undriven Signals.	290
Primary Input Constraints.	290
How to Mask Primary Outputs.	290
Slow Pad Addition	291

Tool Behavior Setup	292
Bus Contention Checks	292
Multi-Driven Net Behavior Setup	293
Z-State Handling Setup	293
The ATPG Learning Process	294
Capture Handling Setup	294
Transient Detection Setup	295
Scan Data Definition.	296
Scan Clocks Definition.	296
Scan Groups Definition	296
Scan Chains Definition.	297
Clock Restriction Setup	297
How to Add Constraint to Scan Cells.	297
Nofault Settings	298
Rules Checks and Violations	298
Good/Fault Simulation with Existing Patterns	299
Fault Simulation	299
Good-Machine Simulation	301
Random Pattern Simulation	302
Tool Setup for Random Pattern Simulation	302
Faults List Addition	302
Running Random Pattern Simulation	302
Fault Information Setup for ATPG	303
Tool Setup for ATPG	303
Faults List Creation.	304
Faults Addition to an Existing List	304
How to Load Faults from an External List	304
How to Write Faults to an External File	305
Fault Sampling Percentage Setup	305
Fault Mode Setup	305
Possible-Detect Credit Setup	305
ATPG Operations	306
ATPG Setup	307
ATPG Constraints Definition.	307
Power and Ground Ports Exclusion from the Pattern Set.	309
ATPG Limits Setup	310
Event Simulation for DFFs and Latches	310
Flattened Model Saves Time and Memory	312
Pattern Buffer Area Creation	312
Fault Sampling to Save Processing Time	313
Checkpointing Setup	313
Pattern Creation with Default Settings	314
Approaches for Improving ATPG Efficiency	316
Reasons for Low Test Coverage.	316
Analysis of a Specific Fault	317
Aborted Faults Report	317
Abort Limit Setup.	317
Random Pattern Usage Setup	318
Decision Reordering for Pattern Creation	318

Table of Contents

How to Save the Test Patterns	318
Low-Power ATPG	319
Low-Power Capture	319
Low-Power Shift	319
Setting up Low-Power ATPG	320
IDDQ Test Set Creation	322
IDDQ Test Set Generation	323
When to Perform the Measures	323
Pattern Generation	323
Running ATPG for IDDQ	323
Leakage Current Checks	324
Delay Test Set Creation	325
Transition Delay Test Set Creation	326
Transition Fault Detection	327
Generating a Transition Test Set	330
Timing for Transition Delay Tests	331
Transition Fault Detection and Multiple Clocks	334
Pattern Failures Due to Timing Exception Paths	337
Types of Timing Exception Paths	338
Timing Exception Paths from an SDC File	340
Does the SDC File Contain Valid SDC Information?	341
Manual Definition of False Paths	342
SDC Timing Exception Effects	343
Debugging Test Coverage with Timing Exceptions	347
Path Delay Test Set Creation	349
Path Delay Fault Detection	349
The Path Definition File	352
Path Definition Checks	355
Generating a Path Delay Test Set	357
Path Delay Testing Limitations	358
At-Speed Test With Named Capture Procedures	359
Support for On-Chip Clocks (PLLs)	360
Internal and External Modes Definition	360
Named Capture Procedures Display	363
Achieving the Test Coverage Goal Using Named Capture Procedures	364
Debugging Low Test Coverage Using Pre-Defined Named Capture Procedures	365
At-speed Fault Simulation Clocking Considerations	365
Internal Signals and Clocks	365
How to Save Internal and External Patterns	366
Mux-DFF Example	367
Support for Internal Clock Control	373
Per-Cycle Clock Control	373
Sequence Clock Control	376
Capture Cycle Determination	377
Applying Internal Clock Control	378
Generating Test Patterns for Different Fault Models and Fault Grading	380
Timing-Aware ATPG	383
Slack Calculation	383
Synopsys Design Constraint Derived Clock Timing	384

Delay Test Coverage Calculation	384
Timing-Aware ATPG vs. Transition ATPG	385
Timing-Aware ATPG Limitations	385
Inaccuracies in Timing Calculations	386
Running Timing-Aware ATPG	387
Troubleshooting Topics	390
Bridge and Open, and Cell Neighborhood Defects UDFM Creation	396
Interconnect Bridge and Open Extraction and UDFM Creation	397
Critical Area Calculation	398
Viewing Extracted Bridge and Open Defects	405
Bridge and Open Extraction Output Files	409
Critical Area Based Bridge Pattern Generation	413
Critical Area Based Open Pattern Generation	417
Cell Neighborhood Defects UDFM Creation	419
Inter-Cell Data File	422
The Marker File	424
Pattern Generation for a Boundary Scan Circuit	426
About Dofile	426
TAP Controller State Machine	427
About the Test Procedure File	428
MacroTest Overview	434
The MacroTest Process Flow	435
Macro Qualification for MacroTest	437
When to Use MacroTest	438
Macro Boundary Definition	441
Macro Boundary Definition by Instance Name	441
Macro Boundary Definition Without Using an Instance Name	442
Observation Site Specification and Reporting	443
Macro Boundary Definition With Trailing Edge Inputs	443
Test Values Definition	445
Recommendations for MacroTest Use	447
Macro Test Examples	449
Verifying Test Patterns	456
Design Simulation with Timing	457
The Verilog Test Bench	458
Parallel Versus Serial Patterns	464
Potential Causes of Simulation Mismatches	465
Simulation Mismatch Investigation	466
DRC Issues	467
Shadow Cells	467
Library Problems	467
Timing Violations	468
Simulation Data Analysis	468
Simulation Mismatch Analysis	471
Chapter 9 Multiprocessing for ATPG and Simulation	481
Definition of Multiprocessing Terms	481

Table of Contents

Multiprocessing to Reduce Runtime	483
Multiprocessing Requirements	483
Procedures for Multiprocessing	486
Using Multiprocessing for ATPG and Simulation	486
SSH Environment and Passphrase Errors	487
How to Disable Multithreading Functionality	489
Threads Addition to the Master	489
Processor Addition in Manual Mode	490
Processor Addition in Grid Mode	492
Processor Addition to the LSF Grid	493
Manual Mode Processor Deletion	494
Grid Mode Processor Deletion	494
 Chapter 10	
Scan Pattern Retargeting	497
Tools and Licensing Required for Scan Retargeting	498
Scan Pattern Retargeting Process Overview	498
Core-Level Pattern Generation	501
Scan Pattern Retargeting of Test Patterns	504
Chip-Level Test Procedures	504
External_capture Procedure	505
Test Procedure Retargeting For Scan Pattern Retargeting	505
Scan Pattern Retargeting Without a Netlist	508
Retargeting Support For Patterns Generated at the Top Level	509
Generating Patterns at the Core Level	512
Retargeting Patterns in Internal Mode	513
Extracting Connectivity Between the Top Level and the Core Boundary	515
Retargeting Patterns Without a Top-Level Netlist	516
Generating Patterns in External Mode	516
Retargeting Example	518
Core-level Pattern Generation for CPU Cores	519
Core-level Pattern Generation for NB Core	520
Retargeting of Patterns at the Chip Level	520
Retargeting of Patterns at the Chip Level Without a Netlist	521
Core-in-Core Pattern Retargeting	523
Core-in-Core Pattern Retargeting Flow	524
Generation of Child Core Internal Mode Patterns	526
Retargeting of Child Core Internal Mode Patterns	526
Generation of Parent Internal Mode Patterns	527
Retargeting of Parent Internal Mode Patterns	528
Scan Pattern Retargeting Limitations	528
 Chapter 11	
Test Pattern Formatting and Timing	531
Test Pattern Timing Overview	532
Timing Terminology	532
General Timing Issues	533
Generating a Procedure File	534

Defining and Modifying Timeplates	534
Delaying Clock Pulses in Shift and Capture to Handle Slow Scan Enable Transitions	536
Delaying Clocks After All Scan Enable Transitions (Recommended)	536
Transitions From Shift to Capture	538
Transition from Capture to Shift	541
Saving Timing Patterns	544
Features of the Formatter	544
Pattern Formatting Issues	546
Serial Versus Parallel Scan Chain Loading	546
Parallel Scan Chain Loading	546
Reduce Serial Loading Simulation Time with Sampling	548
Test Pattern Data Support for IDDQ	548
Basic Test Data Formats for Patterns	549
Text Format	549
Comparing the Text Format with Other Test Data Formats	549
Binary	552
Verilog	552
Wave Generation Language (ASCII)	552
Standard Test Interface Language (STIL)	553
ASIC Vendor Data Formats	554
TI TDL 91	554
Fujitsu FTDL-E	554
Mitsubishi TDL	555
Toshiba TSTL2	555
Vector Creation and Modification	556
Port List Modification	556
Vector Data Callbacks	557
Used Ports and Vector Callback Examples	562
Chapter 12	
Test Pattern File Formats	567
ASCII File Format	568
Header_Data	568
ASCII_Setup_Data	568
Functional_Chain_Test	571
Scan_Test	572
Scan_Cell	575
BIST Pattern File Format	577
BIST_Pattern_Setup_Data	577
Scan_Test_for_BIST	578
Chapter 13	
Power-Aware DRC and ATPG	581
Power-Aware Overview	582
Assumptions and Limitations	582
Multiple Power Mode Test Flow	583
Power-Aware ATPG for Traditional Fault Models	583
CPF and UPF Parser	584

Table of Contents

Power-Aware ATPG Procedure	586
Power-Aware Flow Examples	588
Chapter 14	
Low-Power Design Test	593
Low-Power Testing Overview	594
Low-Power Assumptions and Limitations	594
Low-Power CPF/UPF Parameters	594
Test Insertion	596
Low-Power Test Flow	596
Scan Insertion with Tessonnt Scan	598
Managing Scan Enable Signals for Multiple Power Domains	598
Managing Wrapper Cells with Power Domains	599
Managing Test Point Insertion with Power Domains	601
Power-Aware Design Rule Checks	603
Low-Power DRCs	603
Low-Power DRC Troubleshooting	603
Power State-Aware ATPG	606
Power Domain Testing	606
Low-Power Cell Testing	608
Level Shifters	608
Isolation Cells	608
Chapter 15	
Using MTFI Files	609
MTFI File Format	610
MTFI Features	613
Support of Fault Classes and Sub-Classes	613
Support of Stuck and Transition Fault Information	615
Support of N-Detect Values	615
Support of Different Fault Types in the Same File	617
Support for Hierarchical Fault Accounting	617
Commands that Support MTFI	619
Chapter 16	
Graybox Overview	621
What Is a Graybox?	621
Graybox Process Overview	624
Example dofile for Creating a Graybox Netlist	625
Graybox Netlist Generation for EDT Logic Inserted Blocks	626
Chapter 17	
Tessonnt On-Chip Clock Controller	629
Tessonnt OCC Overview	629
Primary OCC Functions	630
On-Chip Clock Controller Design Description	632
The Standard OCC	633
Design Placement	633

Standard OCC Schematic	634
OCC With Capture Enable	644
Clock Control Operation Modes	644
Timing Diagrams	647
Standard OCC With IJTAG Interface	648
The Parent OCC	650
The Child OCC	652
Tessent OCC Insertion	654
Inserting the OCC	654
occ.dft_spec	655
post_dft_insertion_procedure.tcl	656
OCC Insertion When Using an Existing Clock as the test_clk DFT Signal	656
Core OCC Recommendation	658
Appendix A	
Clock Gaters	663
Basic Clock Gater Cell	663
Two Types of Embedding	665
Non-DRC Violation Clock Gater Embedding (Type-A)	666
Potential DRC Violation Clock Gater Embedding (Type-B)	667
Type-B Scan Chain Tracing Failures (T3 Violations)	667
Scan Cell Data Captures When Clocks are Off	668
Driving LE Flops With Type-B Clock Gaters	668
Cascaded Clock Gaters	670
Level-2 Clock Gater	670
Example Combinations of Cascaded Clock Gaters	670
Clock Gater Configuration Support Summary	671
Appendix B	
State Stability Issues	673
Display of State Stability Data	673
State Stability Data Format	674
State Stability Examples	677
Example 1 — Basic Example	680
Example 2 — Multiple Cycles in Load_unload Prior to Shift	683
Example 3 — Drc_pattern Reporting for Pulse Generators	685
Example 4 — Single Post Shift	686
Example 5 — Single Post Shift with Cycles Between Main and Post Shift	687
Example 6 — Cycles After Apply Shift Statement in Load_unload	689
Example 7 — No Statements in Load_unload Prior to Apply Shift	691
Example 8 — Basic with Specified Capture Clock	691
Example 9 — Setting Stability Check to Off and All_shift	692
Example 10 — Pin Constraints, Test_setup, and State Stability	693
Example 11 — Single Pre Shift	694
Example 12 — Basic with Enhanced Stability Check for NCPs	696

Table of Contents

Appendix C	
Tessent Shell Batch Jobs	697
Commands and Variables for the dofile	697
Command Line Options	700
Scheduling a Batch Job for Execution Later	700
Appendix D	
Net Pair Identification with Calibre for Bridge Fault Test Patterns.....	701
The Static Bridge Fault Model	701
Four-Way Dominant Fault Model	702
Top-level Bridging ATPG	702
Incremental Multi-Fault ATPG	703
The Bridge Parameters File	705
Appendix E	
Getting Help	709
The Tessent Documentation System	709
Mentor Support Services	710
Index	
Third-Party Information	
End-User License Agreement	

List of Figures

Figure 1-1. Top-Down Design Flow Tasks and Products	30
Figure 2-1. Design Before and After Adding Scan	33
Figure 2-2. Scan Representation	34
Figure 2-3. Example of Partitioned Design	36
Figure 2-4. Wrapper Chains Added to Partition A	37
Figure 2-5. Manufacturing Defect Space for a Design	43
Figure 2-6. Internal Faulting Example	48
Figure 2-7. Single Stuck-At Faults for AND Gate	49
Figure 2-8. IDDQ Fault Testing	51
Figure 2-9. Transition Fault Detection Process	52
Figure 2-10. Fault Selection Cone	55
Figure 2-11. Fault Detection Process	70
Figure 2-12. Path Sensitization Example	70
Figure 2-13. Example of Unused Fault in Circuitry	72
Figure 2-14. Example of Tied Fault in Circuitry	73
Figure 2-15. Example of Blocked Fault in Circuitry	74
Figure 2-16. Example of Redundant Fault in Circuitry	74
Figure 3-1. Generic Scan Cell	89
Figure 3-2. Generic Mux-DFF Scan Cell Implementation	90
Figure 3-3. LSSD Master/Slave Element Example	91
Figure 3-4. Dependently-clocked Mux-DFF/Shadow Element Example	91
Figure 3-5. Independently-clocked Mux-DFF/Shadow Element Example	92
Figure 3-6. Mux-DFF/Copy Element Example	92
Figure 3-7. Generic Scan Chain	93
Figure 3-8. Scan Group - Two Scan Inputs	94
Figure 3-9. Scan Group - One Scan Input	95
Figure 3-10. Scan Clocks Example	95
Figure 3-11. Mux-DFF Replacement	96
Figure 3-12. Design Before Flattening	99
Figure 3-13. Design After Flattening	99
Figure 3-14. 2x1 MUX Example	101
Figure 3-15. LA, DFF Example	101
Figure 3-16. TSD, TSH Example	102
Figure 3-17. PBUS, SWBUS Example	102
Figure 3-18. Equivalence Relationship Example	104
Figure 3-19. Example of Learned Logic Behavior	105
Figure 3-20. Example of Implied Relationship Learning	105
Figure 3-21. Forbidden Relationship Example	106
Figure 3-22. Dominance Relationship Example	106
Figure 3-23. Bus Contention Example	108

Figure 3-24. Bus Contention Analysis	108
Figure 3-25. Simulation Model with Bus Keeper	111
Figure 3-26. Constrained Values in Circuitry	113
Figure 3-27. Forbidden Values in Circuitry	113
Figure 3-28. Blocked Values in Circuitry	113
Figure 3-29. Programmable Clock Chopper	114
Figure 4-1. Structural Combinational Loop Example	119
Figure 4-2. Loop Naturally-Blocked by Constant Value	119
Figure 4-3. Cutting Constant Value Loops	120
Figure 4-4. Cutting Single Multiple-Fanout Loops	120
Figure 4-5. Loop Candidate for Duplication	121
Figure 4-6. TIE-X Insertion Simulation Pessimism	121
Figure 4-7. Cutting Loops by Gate Duplication	122
Figure 4-8. Cutting Coupling Loops	123
Figure 4-9. Sequential Feedback Loop	124
Figure 4-10. Test Logic Added to Control Asynchronous Reset	125
Figure 4-11. Test Logic Added to Control Gated Clock	126
Figure 4-12. Tri-state Bus Contention	127
Figure 4-13. Requirement for Combinationally Transparent Latches	128
Figure 4-14. Example of Sequential Transparency	129
Figure 4-15. Clocked Sequential Scan Pattern Events	130
Figure 4-16. Clock Divider	132
Figure 4-17. Example Pulse Generator Circuitry	132
Figure 4-18. Long Path Input Gate Must Go to Gates of the Same Type	133
Figure 4-19. Design with Embedded RAM	134
Figure 4-20. RAM Sequential Example	136
Figure 5-1. Generic Usage of Tessent Scan on a Synthesized Netlist	145
Figure 5-2. The Inputs and Outputs of Tessent Scan	147
Figure 5-3. A Generic Representation of a Scan Element Object	151
Figure 5-4. Single Bit Scan Element	152
Figure 5-5. Multi-Bit Scan Element	152
Figure 5-6. Scan Element Memory with Sub-Chains / Segments	153
Figure 5-7. Unwrapped Core Versus Wrapped Core	156
Figure 5-8. Test Logic Insertion	161
Figure 5-9. Child On-Chip Controller Logic Schematic	176
Figure 5-10. Child On-Chip Controller Logic Schematic, no Clock-Gater	176
Figure 5-11. Simple Example of Two Modes	186
Figure 5-12. Scan Insertion Flow for Unwrapped Core	194
Figure 5-13. Scan Insertion Flow for Wrapped Core	198
Figure 6-1. Uncontrollable and Unobservable Circuitry	208
Figure 6-2. AND Control Point	209
Figure 6-3. OR Control Point	209
Figure 6-4. Observe Point Sharing	211
Figure 6-5. Test Point Analysis & Insertion Starting With a Gate-Level Netlist	212
Figure 6-6. Test Point Sharing Across Power Domains	242

List of Figures

Figure 7-1. Generation of Core Description Files	261
Figure 7-2. Core Mapping to Top Level	262
Figure 7-3. Generation of Top-Level TCD File for Top-Level Logic	263
Figure 7-4. Alternate Process for Core Mapping to Top Level	264
Figure 7-5. Mapping of Cores to Chip Level.....	266
Figure 8-1. Overview of ATPG Tool Usage	275
Figure 8-2. ATPG Tool Inputs and Outputs	277
Figure 8-3. Clock-PO Circuitry	281
Figure 8-4. Data Capture Handling Example	295
Figure 8-5. Efficient ATPG Flow	306
Figure 8-6. Circuitry with Natural “Select” Functionality	308
Figure 8-7. Simulation Frames	311
Figure 8-8. Waveform Modeling for DFFs and Latches	311
Figure 8-9. Flow for Creating a Delay Test Set.....	325
Figure 8-10. Transition Delay	326
Figure 8-11. Transition Launch and Capture Events.....	327
Figure 8-12. Basic Broadside Timing	328
Figure 8-13. Pseudo Launch Off Shift Timing	328
Figure 8-14. Native Launch-Off-Shift Timing	329
Figure 8-15. Broadside Timing Example	331
Figure 8-16. Launch Off Shift (Skewed) Timing Example.....	333
Figure 8-17. Same Clocks Between Loads On	334
Figure 8-18. Compatible Clocks Between Loads On	335
Figure 8-19. domain_clock	336
Figure 8-20. Multicycle Path Example	337
Figure 8-21. Setup Time and Hold Time Exceptions	338
Figure 8-22. Across Clock Domain Hold Time Exception	339
Figure 8-23. Effect Cone of a Non-specific False Path Definition	343
Figure 8-24. SDC No Masking	344
Figure 8-25. SDC False Path Hold Time Effect	345
Figure 8-26. SDC False Path Setup Time Effect.....	346
Figure 8-27. SDC Multicycle Path Effect	347
Figure 8-28. Path Delay Launch and Capture Events	349
Figure 8-29. Hazard-free Detection Example	350
Figure 8-30. Robust Detection Example	351
Figure 8-31. Non-robust Detection Example.....	351
Figure 8-32. Functional Detection Example	352
Figure 8-33. Example Use of Transition_condition Statement	353
Figure 8-34. Example of Ambiguous Path Definition.....	356
Figure 8-35. Example of Ambiguous Path Edges	356
Figure 8-36. On-chip Clock Generation	360
Figure 8-37. PLL-Generated Clock and Control Signals	361
Figure 8-38. Cycles Merged for ATPG	364
Figure 8-39. Cycles Expanded for ATPG	364
Figure 8-40. Mux-DFF Example Design.....	368

Figure 8-41. Mux-DFF Broadside Timing, Cell to Cell	368
Figure 8-42. Broadside Timing, Clock Pulses in Non-adjacent Cycles	370
Figure 8-43. Mux-DFF Cell to PO Timing	371
Figure 8-44. Mux-DFF PI to Cell Timing	372
Figure 8-45. Simplified Per-Cycle Clock Control Model	374
Figure 8-46. Per-Cycle Clock Control Waveform	375
Figure 8-47. Simplified Counter-based Clock Control Example	376
Figure 8-48. Counter-based Clock Control Capture Waveform	377
Figure 8-49. Clock Control Capture Cycles	378
Figure 8-50. Timing Slack	383
Figure 8-51. Testcase 1 Logic	388
Figure 8-52. Dofile	389
Figure 8-53. Reports	389
Figure 8-54. Glitch Detection Case	394
Figure 8-55. Tessent Bridge and Open Extraction Flow	397
Figure 8-56. Probability and Critical Area	398
Figure 8-57. Bridge Area Calculation	398
Figure 8-58. Critical Area Calculation	399
Figure 8-59. High Bridge Probability	399
Figure 8-60. Low Bridge Probability	399
Figure 8-61. High Open Probability	400
Figure 8-62. Low Open Probability	400
Figure 8-63. Critical Area Calculation Formula for Bridges	400
Figure 8-64. Critical Area Example - Side-to-Side Bridges	401
Figure 8-65. Critical Area Example - Corner-to-Corner Bridge	402
Figure 8-66. Critical Area Calculation Formula for Opens and Vias	402
Figure 8-67. Open Example - Net with Seven Layout Segments	403
Figure 8-68. One Via	403
Figure 8-69. Four Vias	404
Figure 8-70. Alternative Critical Area Calculation Formula for Bridges	404
Figure 8-71. GDS View with Calibre RVE	405
Figure 8-72. Calibre RVE View for Bridges	406
Figure 8-73. Calibre RVE View for Opens	407
Figure 8-74. Highlighted Bridge Fault B1	408
Figure 8-75. Four-Way Bridge Static Model	414
Figure 8-76. Two-Way Bridge Delay Model	415
Figure 8-77. Open Fault Model - Static and Delay	417
Figure 8-78. Neighborhood Cell Layout	419
Figure 8-79. Cell Neighborhood Defects	420
Figure 8-80. Neighborhood UDFM Generation Flow	420
Figure 8-81. Complete Neighborhood Flow	421
Figure 8-82. Marker File	425
Figure 8-83. State Diagram of TAP Controller Circuitry	427
Figure 8-84. Conceptual View of MacroTest	434
Figure 8-85. Basic Scan Pattern Creation Flow with MacroTest	436

List of Figures

Figure 8-86. Mismatch Diagnosis Guidelines	465
Figure 8-87. Simulation Transcript	469
Figure 8-88. Automatic Simulation Mismatch Analysis Flow	473
Figure 8-89. Manual Simulation Mismatch Analysis Flow	476
Figure 8-90. Clock-Skew Example	480
Figure 10-1. Core-level Pattern Generation Process	499
Figure 10-2. Scan Pattern Retargeting - Internal Mode	500
Figure 10-3. Pattern Generation - External Mode	501
Figure 10-4. Merging Patterns for Multiple Views of a Core at the Top Level	511
Figure 10-5. Merging Patterns for Top Level and Core at the Top Level	512
Figure 10-6. Retargeting of Core-level Patterns	518
Figure 10-7. Three Levels of Wrapped Cores	524
Figure 10-8. Core-in-Core Retargeting Process Overview	525
Figure 11-1. Defining Basic Timing Process Flow	531
Figure 11-2. Clock Pulses Requiring Additional Delay	536
Figure 11-3. Stretched Waveform	537
Figure 11-4. Stretched Timeplate in Capture	538
Figure 11-5. Extended Capture Procedure	539
Figure 11-6. Extended Capture Procedure (Clock Sequential)	539
Figure 11-7. Extended Capture Procedure (Dead Cycle Removed)	540
Figure 11-8. External Capture Procedure (Dead Cycles Added)	541
Figure 11-9. Stretched Timeplate (load_unload)	542
Figure 11-10. Capture to Shift	543
Figure 11-11. Example Vector Callback	558
Figure 12-1. Example Scan Circuit	576
Figure 13-1. PON Reporting in DFTVisualizer	587
Figure 14-1. Scan Chain Insertion by Power Domain	596
Figure 14-2. Input Wrapper Cells and Power Domains	600
Figure 14-3. Output Wrapper Cells and Power Domains	601
Figure 15-1. Using MTFI with Hierarchical Designs	619
Figure 16-1. Full Hierarchical Block Netlist	622
Figure 16-2. Graybox Version of Block Netlist	623
Figure 17-1. OCC Clock Control Components	631
Figure 17-2. Standard OCC Example	633
Figure 17-3. Clock Control Logic Design Placement	634
Figure 17-4. On-Chip Clock Controller Logic Schematic	635
Figure 17-5. Sample Dofile Variables	639
Figure 17-6. Fast Capture, Serial Load	643
Figure 17-7. Slow Capture, Serial Load	643
Figure 17-8. OCC with capture_en	644
Figure 17-9. Functional Mode Operation	645
Figure 17-10. Shift Mode Operation	645
Figure 17-11. OCC Shift Clocking	646
Figure 17-12. Slow Capture Mode	646
Figure 17-13. Fast Capture Mode Operation	647

Figure 17-14. Slow Speed Capture Timing Diagram	647
Figure 17-15. Fast Capture Timing Diagram.....	648
Figure 17-16. Standard OCC with IJTAG Interface	649
Figure 17-17. OCC With IJTAG Interface: <code>ijtag_tck</code> injected for slow clock.....	649
Figure 17-18. Parent OCC in Standard Mode (Extest Mode)	650
Figure 17-19. Parent OCC in Parent Mode (Intest Mode).....	650
Figure 17-20. Parent On-Chip Controller Logic Schematic	651
Figure 17-21. Child OCC Gates and Creates Clocks (with Parent OCC).....	652
Figure 17-22. Child On-Chip Controller Logic Schematic	652
Figure 17-23. Child OCC Creates Enable for Clock Gaters (with Parent OCC).	653
Figure 17-24. Child On-Chip Controller Logic Schematic, no Clock-Gater.	653
Figure 17-25. Original Clocking	657
Figure 17-26. Incorrect OCC Placement	657
Figure 17-27. Correct OCC Placement	658
Figure 17-28. Child OCC in Core	659
Figure 17-29. Standard OCC in Core	660
Figure A-1. Basic Clock Gater Cell.....	663
Figure A-2. Two Types of Embedding for the Basic Clock Gater	665
Figure A-3. Type-B Clock Gater Causes Tracing Failure.....	667
Figure A-4. Sample EDT Test Procedure Waveforms	668
Figure A-5. Two-level Clock Gating.....	670
Figure B-1. Design Used in State Stability Examples	678
Figure B-2. Typical Initialization Problem	681
Figure B-3. Three-bit Shift Register (Excerpted from Figure D-1).....	682
Figure B-4. Initialization with a Non-Shift Clock.....	683
Figure B-5. Clocking <code>ff20</code> with a Pulse Generator	685

List of Tables

Table 2-1. Test Type/Fault Model Relationship	47
Table 2-2. UDFM Keywords	58
Table 2-3. Fault Class Hierarchy	77
Table 2-4. Fault Sub-classes	78
Table 4-1. RAM/ROM Commands	138
Table 5-1. Scan Direction and Active Values	162
Table 8-1. ATPG Inputs	277
Table 8-2. ATPG Outputs	278
Table 8-3. ATPG Constraint Conditions	309
Table 8-4. Typical Events in a Launch-Off-Shift Pattern	329
Table 8-5. Testcase 2 Data	385
Table 13-1. Power Data Commands Directly Related to ATPG and DRC	584
Table 13-2. Example Design With Four Power domains and Power Modes	588
Table 14-1. Power Data Commands Directly Related to ATPG	595
Table 15-1. MTFI Command Summary	619
Table 16-1. Graybox Command Summary	621
Table 17-1. Clock Controller I/O Signals	635
Table 17-2. Parent OCC I/O Signals	651
Table A-1. Clock Gater Configuration Support Summary	671
Table D-1. Bridge Definition File Keywords	705

Chapter 1 Overview

The following is an overview of ASIC/IC Design-for-Test (DFT) strategies and shows how to use Mentor Graphics ASIC/IC DFT products as part of typical DFT design processes. This overview discusses the Tessent products that use Scan and ATPG technology.

- Tessent Scan, which is Tessent Shell operating in “dft -scan” context
- Tessent FastScan, which is Tessent Shell operating in “patterns -scan” context
- Tessent TestKompress (with EDT off), which is Tessent Shell operating in “patterns -scan” context

This manual uses the term “ATPG tool” to refer to any of the following products: Tessent FastScan, Tessent TestKompress (with EDT off), or Tessent Shell operating in the “patterns -scan” context.

For information about contexts in Tessent Shell, refer to “[Contexts and System Modes](#)” in the *Tessent Shell User’s Manual*. For information about any of the commands mentioned in this manual, refer to the [Tessent Shell Reference Manual](#).

What is Design-for-Test?.....	28
DFT Strategies	28
Top-Down Design Flow with DFT	29

What is Design-for-Test?

Testability is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design's quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a design-for-test (DFT) process flow.

Testable circuitry is both *controllable* and *observable*. In a testable design, setting specific values on the primary inputs results in values on the primary outputs that indicate whether or not the internal circuitry works properly. To ensure maximum design testability, designers must employ special DFT techniques at specific stages in the development process.

DFT Strategies	28
----------------------	----

DFT Strategies

At the highest level, there are two main approaches to DFT: ad hoc and structured. The following subsections discuss these DFT strategies.

Ad Hoc DFT

Ad hoc DFT implies using good design practices to enhance the testability of a design without making major changes to the design style.

Some ad hoc techniques include:

- Minimizing redundant logic
- Minimizing asynchronous logic
- Isolating clocks from the logic
- Adding internal control and observation points

Using these practices throughout the design process improves the overall testability of your design. However, using structured DFT techniques with the Mentor Graphics DFT tools yields far greater improvement. Thus, the remainder of this document concentrates on structured DFT techniques.

Structured DFT

Structured DFT provides a more systematic and automatic approach to enhancing design testability.

Structured DFT's goal is to increase the controllability and observability of a circuit. Various methods exist for accomplishing this. The most common is the *scan design* technique, which modifies the internal sequential circuitry of the design. You can also use the Built-in Self-Test

(BIST) method, which inserts a device's testing function within the device itself. Another method is *boundary scan*, which increases board testability by adding circuitry to a chip. “[Scan and ATPG Basics](#)” describes these methods in detail.

Top-Down Design Flow with DFT

The following text and figure shows the basic steps and the Mentor Graphics tools you would use during a typical ASIC top-down design flow.

This document discusses those steps shown in gray; it also mentions certain aspects of other design steps, where applicable. This flow is just a general description of a top-down design process flow using a structured DFT strategy.

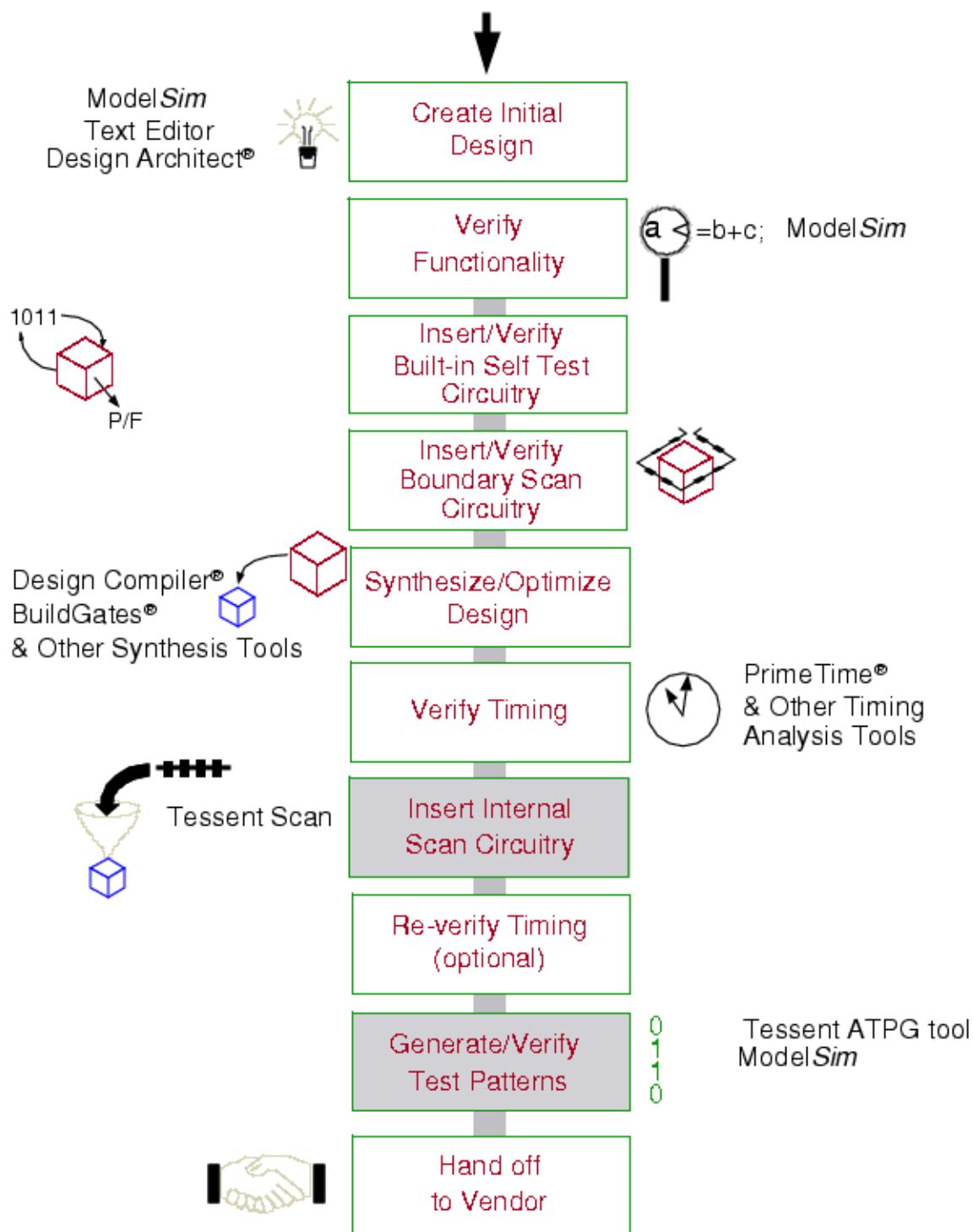
As [Figure 1-1](#) shows, the first task in any design flow is creating the initial RTL-level design, through whatever means you choose. In the Mentor Graphics environment, you may choose to create a high-level Verilog description using ModelSim® or a schematic using Design Architect®. You then verify the design's functionality by performing a functional simulation, using ModelSim or another vendor's Verilog simulator.

At this point in the flow you are ready to insert internal scan circuitry into your design using Tesson Scan®. You may then want to re-verify the timing because you added scan circuitry. Once you are sure the design is functioning as desired, you can generate test patterns. You can use the ATPG tool to generate a test pattern set in the appropriate format.

Now you should verify that the design and patterns still function correctly with the proper timing information applied. You can use ModelSim or some other simulator to achieve this goal. You may then have to perform a few additional steps required by your ASIC vendor before handing the design off for manufacture and testing.

Note

 It is important for you to check with your vendor early on in your design process for specific requirements and restrictions that may affect your DFT strategies. For example, the vendor's test equipment may only be able to handle single scan chains (see “[Scan Design Overview](#)” on page 32), have memory limitations, or have special timing requirements that affect the way you generate scan circuitry and test patterns.

Figure 1-1. Top-Down Design Flow Tasks and Products

Chapter 2

Scan and ATPG Basics

Before you begin the testing process, you must first have an understanding of certain testing concepts. Once you understand these concepts, you can determine the best test strategy for your particular design.

Scan circuitry facilitates test generation and can reduce external tester usage. There are two main types of scan circuitry: internal scan and boundary scan. *Internal scan* (also referred to as *scan design*) is the internal modification of your design's circuitry to increase its testability.

While scan design modifies circuitry within the original design, *boundary scan* adds scan circuitry around the periphery of the design to make internal circuitry on a chip accessible via a standard board interface. The added circuitry enhances board testability of the chip, the chip I/O pads, and the interconnections of the chip to other board circuitry.

Scan Design Overview	32
About Scan Design Methodology	34
About Wrapper Chains	35
Test Structure Insertion with Tesson Scan	37
ATPG Overview	39
The ATPG Process	40
Mentor Graphics ATPG Applications	41
Scan Sequential ATPG with the ATPG Tool	41
Overview of Test Types and Fault Models	43
Test Types	44
Fault Modeling Overview	47
Fault Manipulation	54
About User-Defined Fault Modeling	56
Multiple Detect	66
Embedded Multiple Detect	68
Fault Detection	69
Fault Classes	72
Fault Reporting	84
Testability Calculations	84

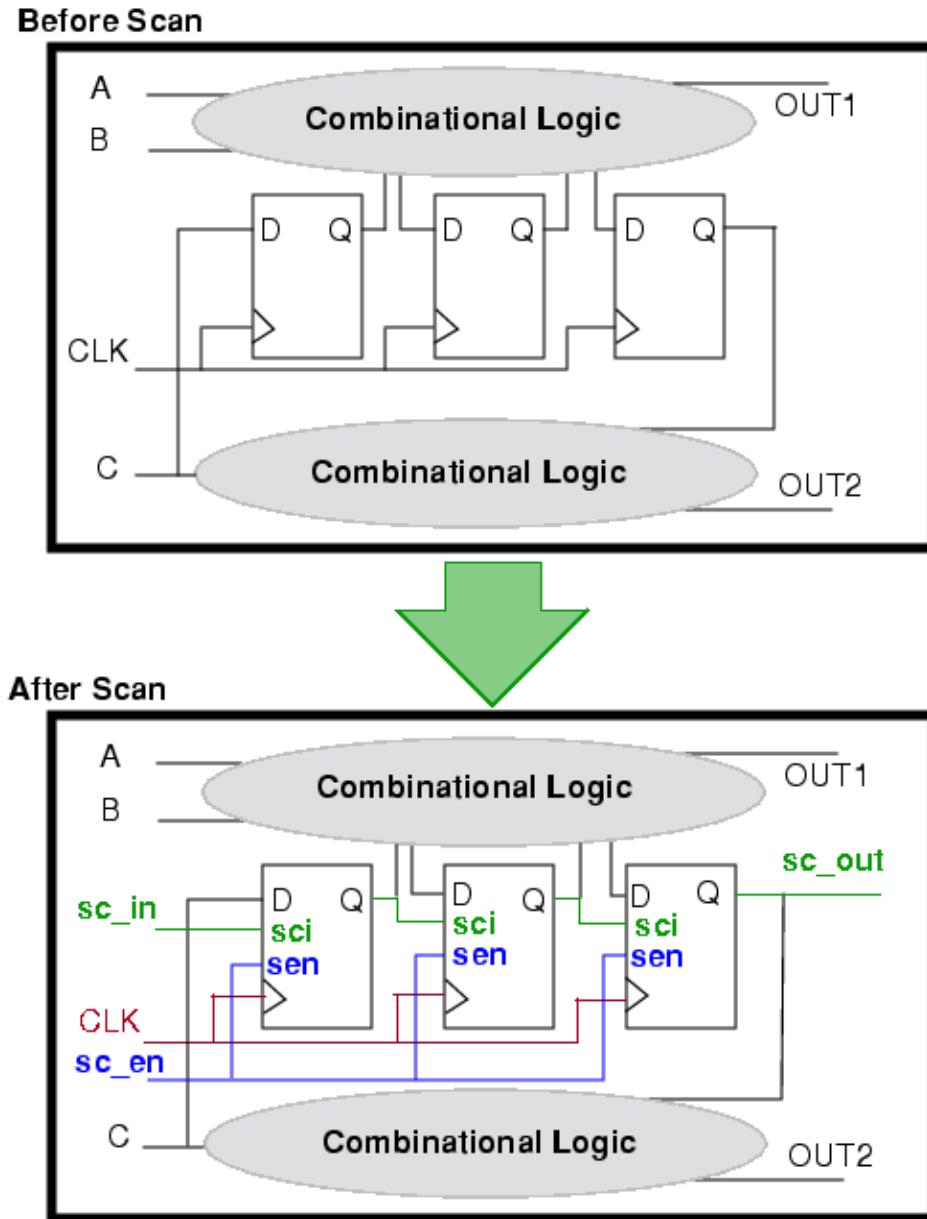
Scan Design Overview

Internal scan (or scan design) is the internal modification of your design's circuitry to increase its testability.

The goal of scan design is to make a difficult-to-test sequential circuit behave (during the testing process) like an easier-to-test combinational circuit. Achieving this goal involves replacing sequential elements with scannable sequential elements (scan cells) and then stitching the scan cells together into scan registers, or scan chains. You can then use these serially-connected scan cells to shift data in and out when the design is in scan mode.

The design shown in [Figure 2-1](#) contains both combinational and sequential portions. Before adding scan, the design had three inputs, A, B, and C, and two outputs, OUT1 and OUT2. This “Before Scan” version is difficult to initialize to a known state, making it difficult to both control the internal circuitry and observe its behavior using the primary inputs and outputs of the design.

Figure 2-1. Design Before and After Adding Scan



After adding scan circuitry, the design has two additional inputs, sc_in and sc_en, and one additional output, sc_out. Scan memory elements replace the original memory elements so that when shifting is enabled (the sc_en line is active), scan data is read in from the sc_in line.

The operating procedure of the scan circuitry is as follows:

1. Enable the scan operation to permit shifting (to initialize scan cells).
2. After loading the scan cells, hold the scan clocks off and then apply stimulus to the primary inputs.

3. Measure the outputs.
4. Pulse the clock to capture new values into scan cells.
5. Enable the scan operation to unload and measure the captured values while simultaneously loading in new values via the shifting procedure (as in step 1).

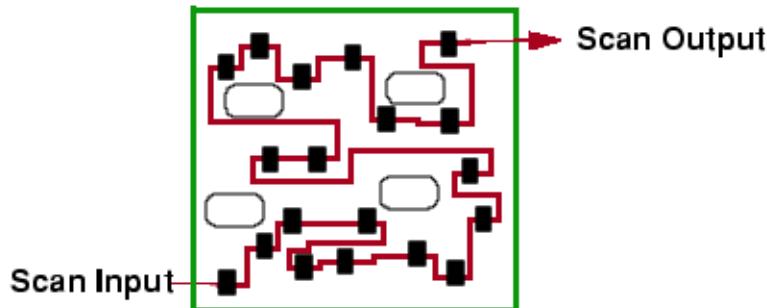
About Scan Design Methodology	34
About Wrapper Chains	35
Test Structure Insertion with Tesson Scan	37

About Scan Design Methodology

Scan is a scan design methodology that replaces all memory elements in the design with their scannable equivalents and then stitches (connects) them into scan chains. The idea is to control and observe the values in all the design's storage elements so you can make the sequential circuit's test generation and fault simulation tasks as simple as those of a combinational circuit.

[Figure 2-2](#) gives a symbolic representation of a scan design.

Figure 2-2. Scan Representation



The black rectangles in [Figure 2-2](#) represent scan elements. The line connecting them is the scan path. Because this is a scan design, all storage elements were converted and connected in the scan path. The rounded boxes represent combinational portions of the circuit.

For information on implementing a scan strategy for your design, refer to "[Test Structures Supported by Tesson Scan](#)" on page 149.

Scan Benefits

The following are benefits of employing a scan strategy:

- **Highly automated process**

Using scan insertion tools, the process for inserting scan circuitry into a design is highly-automated, thus requiring very little manual effort.

- **Highly-effective, predictable method**

Scan design is a highly-effective, well-understood, and well-accepted method for generating high test coverage for your design.

- **Ease of use**

Using scan methodology, you can insert both scan circuitry and run ATPG without the aid of a test engineer.

- **Assured quality**

Scan assures quality because parts containing such circuitry can be tested thoroughly during chip manufacture. If your end products are going to be used in market segments that demand high quality, such as aircraft or medical electronics—and you can afford the added circuitry—then you should take advantage of the scan methodology.

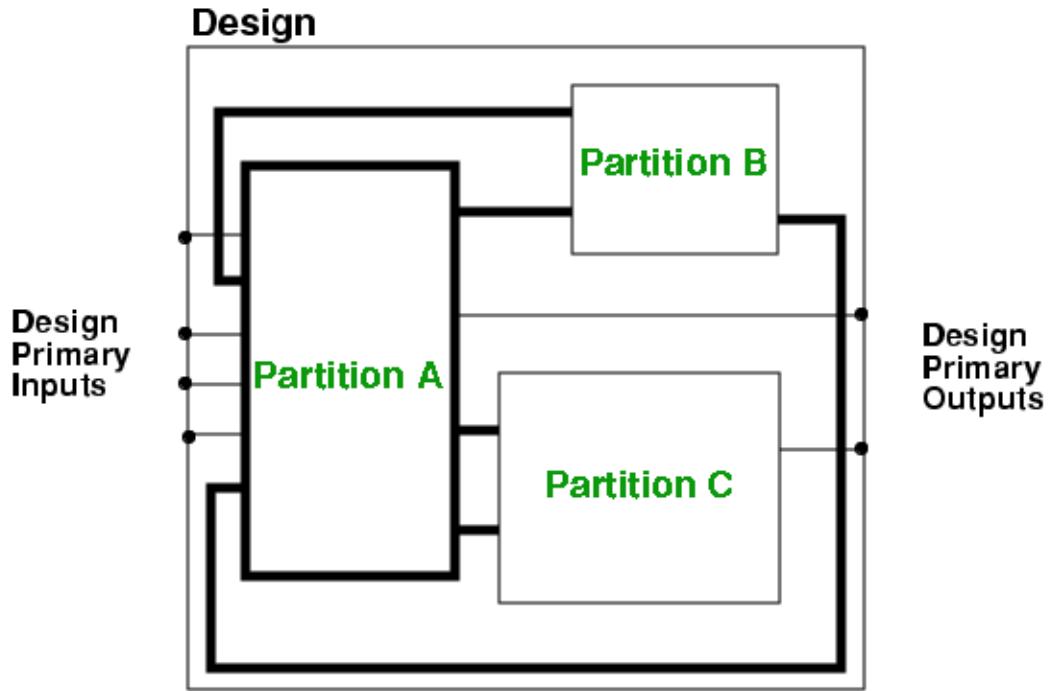
About Wrapper Chains

The ATPG process on very large, complex designs can often be unpredictable. This problem is especially true for large sequential or partial scan designs. To reduce this unpredictability, a number of hierarchical techniques for test structure insertion and test generation are beginning to emerge. Creating wrapper chains is one of these techniques. Large designs that are split into a number of design blocks benefit most from wrapper chains.

Wrapper chains add controllability and observability to the design via a hierarchical wrapper scan chain. A wrapper chain is a series of scan cells connected around the boundary of a design partition that is accessible at the design level. The wrapper chain improves both test coverage and run time by converting sequential elements to scan cells at inputs (outputs) that have low controllability (observability) from outside the block.

The architecture of wrapper chains is illustrated in the following two figures. [Figure 2-3](#) shows a design with three partitions, A, B, and C.

Figure 2-3. Example of Partitioned Design



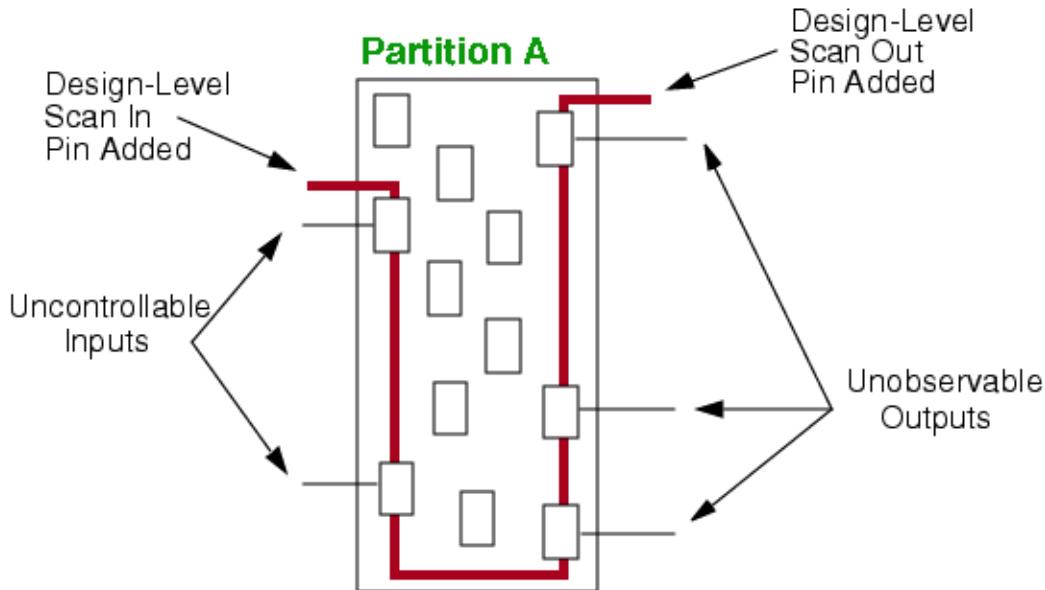
The bold lines in [Figure 2-3](#) indicate inputs and outputs of partition A that are not directly controllable or observable from the design level. Because these lines are not directly accessible at the design level, the circuitry controlled by these pins can cause testability problems for the design.

[Figure 2-4](#) shows how adding wrapper chain structures to partition A increases the controllability and observability (testability) of partition A from the design level.

Note

 Only the first elements that are directly connected to the uncontrollable (unobservable) primary inputs (primary outputs) become part of the wrapper chain.

Figure 2-4. Wrapper Chains Added to Partition A



The wrapper chain consists of two types of elements: sequential elements connected directly to uncontrolled primary inputs of the partition, and sequential elements connected directly to unobservable (or masked) outputs of the partition. The partition also acquires two design-level pins, scan in and scan out, to give direct access to the previously uncontrollable or unobservable circuitry.

You can also use wrapper chains in conjunction with scan structures. Sequential elements not eligible for wrapper chains become candidates for internal scan.

For information on implementing a scan strategy for your design, refer to “[Setup for Wrapper Chain Identification](#)”.

Test Structure Insertion with Tessent Scan

Tessent Scan, the Mentor Graphics internal scan synthesis tool, can identify sequential elements for conversion to scan cells and then stitch those scan cells into scan chains.

Tessent Scan contains the following features:

- **Verilog format**
Reads and writes a Verilog gate-level netlist.
- **Multiple scan types**
Supports insertion of three different scan types, or methodologies: mux-DFF, clocked-scan, and LSSD.
- **Multiple test structures**

Supports identification and insertion of scan (both sequential ATPG-based and scan sequential procedure-based), wrapper chains, and test points.

- **Scannability checking**

Provides powerful scannability checking/reporting capabilities for sequential elements in the design.

- **Design rules checking**

Performs design rules checking to ensure scan setup and operation are correct—before scan is actually inserted. This rules checking also guarantees that the scan insertion done by Tessent Scan produces results that function properly in an ATPG tool.

- **Interface to ATPG tools**

Automatically generates information for the ATPG tools on how to operate the scan circuitry Tessent Scan creates.

- **Optimal partial scan selection**

Provides optimal partial scan analysis and insertion capabilities.

- **Flexible scan configurations**

Enables flexibility in the scan stitching process, such as stitching scan cells in fixed or random order, creating either single- or multiple-scan chains, and using multiple clocks on a single-scan chain.

- **Test logic**

Provides capabilities for inserting test logic circuitry on uncontrollable set, reset, clock, tri-state enable, and RAM read/write control lines.

- **User specified pins**

Enables user-specified pin names for test and other I/O pins.

- **Multiple model levels**

Handles gate-level, as well as gate/transistor-level models.

- **Online help**

Provides online help for every command along with online manuals.

For information about using Tessent Scan to insert scan circuitry into your design, refer to “[Internal Scan and Test Circuitry Insertion](#)” on page 143.

ATPG Overview

ATPG stands for Automatic Test Pattern Generation. Test patterns, sometimes called test vectors, are sets of 1s and 0s placed on primary input pins during the manufacturing test process to determine if the chip is functioning properly. When the test pattern is applied, the Automatic Test Equipment (ATE) determines if the circuit is free from manufacturing defects by comparing the fault-free output—which is also contained in the test pattern—with the actual output measured by the ATE.

The ATPG Process.....	40
Mentor Graphics ATPG Applications.....	41
Scan Sequential ATPG with the ATPG Tool	41

The ATPG Process

The goal of ATPG is to create a set of patterns that achieves a given test coverage, where test coverage is the total percentage of testable faults the pattern set actually detects.

For a more precise definition of test coverage, see “[Testability Calculations](#)” on page 84.

ATPG consists of two main steps: 1) generating patterns and, 2) performing fault simulation to determine which faults the patterns detect. Mentor Graphics ATPG tools automate these two steps into a single operation or ATPG process. This ATPG process results in patterns you can then save with added tester-specific formatting that enables a tester to load the pattern data into a chip’s scan cells and otherwise apply the patterns correctly.

The two most typical methods for pattern generation are random and deterministic. Additionally, the ATPG tools can fault simulate patterns from an external set and place those patterns detecting faults in a test set.

For information on the fault simulation process see “[Fault Classes](#)” on page 72. For details on each type of test pattern generation, see the following:

Random Pattern Test Generation	40
Deterministic Pattern Test Generation	40
External Pattern Test Generation	41

Random Pattern Test Generation

An ATPG tool uses *random pattern test generation* when it produces a number of random patterns and identifies only those patterns that detect faults. It then stores only those patterns in the test pattern set.

The type of fault simulation used in random pattern test generation cannot replace deterministic test generation because it can never identify redundant faults. Nor can it create test patterns for faults that have a very low probability of detection. However, it can be useful on testable faults terminated by deterministic test generation. As an initial step, using a small number of random patterns can improve ATPG performance.

Deterministic Pattern Test Generation

An ATPG tool uses *deterministic test pattern generation* when it creates a test pattern intended to detect a given fault. The procedure is to pick a fault from the fault list, create a pattern to detect the fault, fault simulate the pattern, and check to make sure the pattern detects the fault.

More specifically, the tool assigns a set of values to control points that force the fault site to the state opposite the fault-free state, so there is a detectable difference between the fault value and the fault-free value. The tool must then find a way to propagate this difference to a point where it can observe the fault effect. To satisfy the conditions necessary to create a test pattern, the test

generation process makes intelligent decisions on how best to place a desired value on a gate. If a conflict prevents the placing of those values on the gate, the tool refines those decisions as it attempts to find a successful test pattern.

If the tool exhausts all possible choices without finding a successful test pattern, it must perform further analysis before classifying the fault. Faults requiring this analysis include redundant, ATPG-untestable, and possible-detected-untestable categories (see “[Fault Classes](#)” on page 72 for more information on fault classes). Identifying these fault types is an important by-product of deterministic test generation and is critical to achieving high test coverage. For example, if a fault is proven redundant, the tool may safely mark it as untestable. Otherwise, it is classified as a potentially detectable fault and counts as an untested fault when calculating test coverage.

External Pattern Test Generation

An ATPG tool uses *external pattern test generation* when the preliminary source of ATPG is a pre-existing set of external patterns.

The tool analyzes this external pattern set to determine which patterns detect faults from the active fault list. It then places these effective patterns into an internal test pattern set. The “generated patterns,” in this case, include the patterns (selected from the external set) that can efficiently obtain the highest test coverage for the design.

Mentor Graphics ATPG Applications

Tessent FastScan and Tessent TestKompress (EDT off) are the Mentor Graphics scan sequential ATPG products, and are the same thing as Tessent Shell operating in “patterns -scan” context. This manual refers to all three of these products as the “ATPG tool.”

The following subsections introduce the features of Tessent Shell operating in “patterns -scan” context. “[Test Pattern Generation](#)” on page 273 discusses the ATPG products in greater detail.

Scan Sequential ATPG with the ATPG Tool

Mentor Graphics ATPG products include many features that support scan sequential ATPG.

Scan sequential ATPG features included in the tool:

- Deliver high performance ATPG for designs with structured scan
- Reduce run time with no effect on coverage or pattern count using distributed ATPG
- Maximize test coverage by minimizing the impact of X’s caused by false and multicycle paths
- Identify testability problems early using comprehensive design rule checking
- Reduce test validation time with automatic simulation mismatch debugging

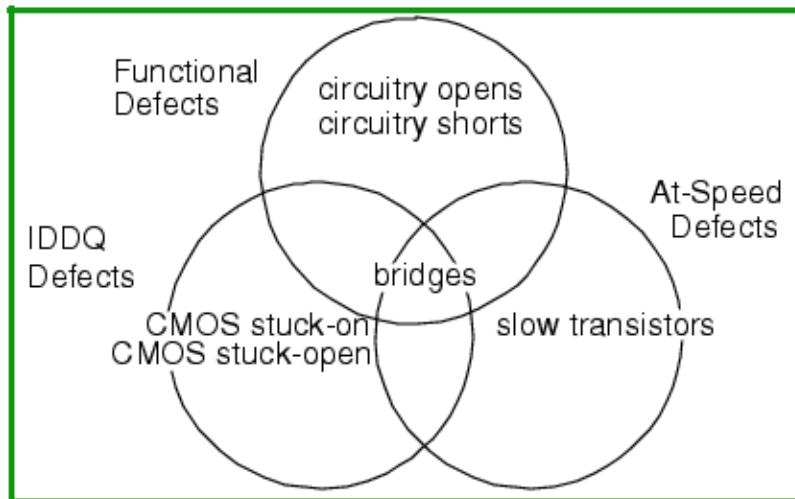
- Ensure shorter time to market with integration into all design flows and foundry support
- Have extensive fault model support, including stuck-at, IDDQ, transition, path delay and bridge
- Have on-chip PLL support for accurate at-speed test
- Automate testing small embedded memories and cores with scan
- Supported in the Tessent SoCScan hierarchical silicon test environment

Overview of Test Types and Fault Models

A manufacturing defect is a physical problem that occurs during the manufacturing process, causing device malfunctions of some kind. The purpose of test generation is to create a set of test patterns that detect as many manufacturing defects as possible.

Figure 2-5 gives an example of possible device defect types.

Figure 2-5. Manufacturing Defect Space for a Design



Each of these defects has an associated detection strategy. The following subsection discusses the three main types of test strategies.

Test Types.....	44
Fault Modeling Overview	47
Fault Manipulation	54
About User-Defined Fault Modeling	56
Multiple Detect.....	66
Embedded Multiple Detect	68
Fault Detection.....	69
Fault Classes.....	72
Fault Reporting	84
Testability Calculations.....	84

Test Types

The previous figure shows three main categories of defects and their associated test types: functional, IDDQ, and at-speed. Functional testing checks the logic levels of output pins for a “0” and “1” response. IDDQ testing measures the current going through the circuit devices. At-speed testing checks the amount of time it takes for a device to change logic states. The following subsections discuss each of these test types in more detail.

Functional Test	44
IDDQ Test	44
At-Speed Test	46

Functional Test

Functional test continues to be the most widely-accepted test type. Functional test typically consists of user-generated test patterns, simulation patterns, and ATPG patterns.

Functional testing uses logic levels at the device input pins to detect the most common manufacturing process-caused problem, static defects (for example, open, short, stuck-on, and stuck-open conditions). Functional testing applies a pattern of 1s and 0s to the input pins of a circuit and then measures the logical results at the output pins. In general, a defect produces a logical value at the outputs different from the expected output value.

IDDQ Test

IDDQ testing measures quiescent power supply current rather than pin voltage, detecting device failures not easily detected by functional testing—such as CMOS transistor stuck-on faults or adjacent bridging faults. IDDQ testing equipment applies a set of patterns to the design, lets the current settle, then measures for excessive current draw. Devices that draw excessive current may have internal manufacturing defects.

Because IDDQ tests do not have to propagate values to output pins, the set of test vectors for detecting and measuring a high percentage of faults may be very compact. The ATPG tool efficiently creates this compact test vector set.

In addition, IDDQ testing detects some static faults, tests reliability, and reduces the number of required burn-in tests. You can increase your overall test coverage by augmenting functional testing with IDDQ testing.

IDDQ test generation methodologies break down into these categories:

- **Every-vector**

This methodology monitors the power-supply current for every vector in a functional or stuck-at fault test set. Unfortunately, this method is relatively slow—on the order of 10-

100 milliseconds per measurement—making it impractical in a manufacturing environment.

- **Supplemental**

This methodology bypasses the timing limitation by using a smaller set of IDDQ measurement test vectors (typically generated automatically) to augment the existing test set.

Three test vector types serve to further classify IDDQ test methodologies:

- **Ideal**

Ideal IDDQ test vectors produce a nearly zero quiescent power supply current during testing of a good device. Most methodologies expect such a result.

- **Non-ideal**

Non-ideal IDDQ test vectors produce a small, deterministic quiescent power supply current in a good circuit.

- **Illegal**

If the test vector cannot produce an accurate current component estimate for a good device, it is an illegal IDDQ test vector. You should never perform IDDQ testing with illegal IDDQ test vectors.

IDDQ testing classifies CMOS circuits based on the quiescent-current-producing circuitry contained inside as follows:

- **Fully static**

Fully static CMOS circuits consume close to zero IDDQ current for all circuit states. Such circuits do not have pull-up or pull-down resistors, and there can be one and only one active driver at a time in tri-state buses. For such circuits, you can use any vector for ideal IDDQ current measurement.

- **Resistive**

Resistive CMOS circuits can have pull-up or pull-down resistors and tristate buses that generate high IDDQ current in a good circuit.

- **Dynamic**

Dynamic CMOS circuits have macros (library cells or library primitives) that generate high IDDQ current in some states. Diffused RAM macros belong to this category.

Some designs have a low current mode, which makes the circuit behave like a fully static circuit. This behavior makes it easier to generate ideal IDDQ tests for these circuits.

The ATPG tool currently supports only the ideal IDDQ test methodology for fully static, resistive, and some dynamic CMOS circuits. The tools can also perform IDDQ checks during

ATPG to ensure the vectors they produce meet the ideal requirements. For information on creating IDDQ test sets, refer to “[IDDQ Test Set Creation](#)”.

At-Speed Test

Timing failures can occur when a circuit operates correctly at a slow clock rate, and then fails when run at the normal system speed. Delay variations exist in the chip due to statistical variations in the manufacturing process, resulting in defects such as partially conducting transistors and resistive bridges.

The purpose of at-speed testing is to detect these types of problems. At-speed testing runs the test patterns through the circuit at the normal system clock speed.

Fault Modeling Overview

Fault models are a means of abstractly representing manufacturing defects in the logical model of your design. Each type of testing—functional, IDDQ, and at-speed—targets a different set of defects.

Test Types and Associated Fault Models	47
Fault Locations	47
Fault Collapsing	48
Supported Fault Model Types	49

Test Types and Associated Fault Models

There is a relationship between test types, fault models, and the types of manufacturing defects targeted for detection.

Table 2-1 lists those relationships.

Table 2-1. Test Type/Fault Model Relationship

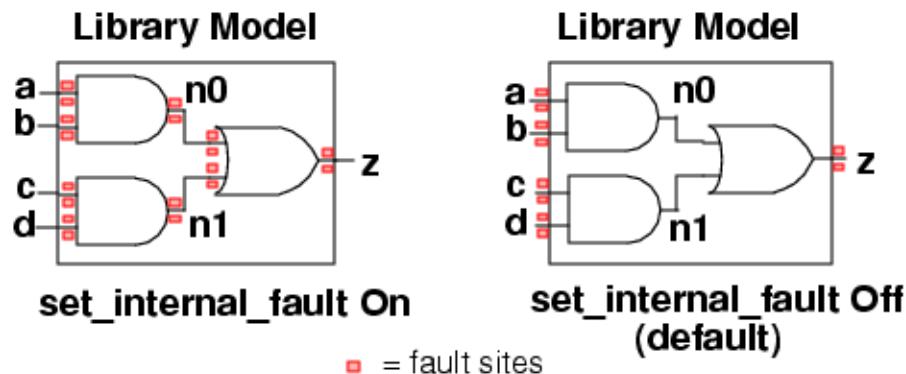
Test Type	Fault Model	Examples of Mfg. Defects Detected
Functional	Stuck-at, toggle	Some opens/shorts in circuit interconnections
IDDQ	Pseudo stuck-at	CMOS transistor stuck-on/some stuck-open conditions, resistive bridging faults, partially conducting transistors
At-speed	Transition, path delay	Partially conducting transistors, resistive bridges

Fault Locations

By default, faults reside at the inputs and outputs of library models. However, faults can instead reside at the inputs and outputs of gates within library models if you turn internal faulting on.

Figure 2-6 shows the fault sites for both cases.

Figure 2-6. Internal Faulting Example



To locate a fault site, you need a unique, hierarchical instance pathname plus the pin name.

You can also use Verilog `celldesign statements to extend cell boundaries beyond library models. Using this technique has several implications:

- The default fault population changes. By default, all fault locations are at library boundary pins. However, when the library boundary moves from the ATPG library level up to the `celldesign level, the fault locations and fault population change as a result.
- The flattened model can be different because the logic inside `celldesign module might be optimized to reduce the flattened model size.
- Hierarchical instance/pin names inside `celldesign module are not treated as legal instance/pin names.

Fault Collapsing

A circuit can contain a significant number of faults that behave identically to other faults. That is, the test may identify a fault, but may not be able to distinguish it from another fault. In this case, the faults are said to be equivalent, and the fault identification process reduces the faults to one equivalent fault in a process known as fault collapsing.

For performance reasons, early in the fault identification process the ATPG tool singles out a member of the set of equivalent faults and use this “representative” fault in subsequent algorithms. Also for performance reasons, these applications only evaluate the one equivalent fault, or collapsed fault, during fault simulation and test pattern generation. The tools retain information on both collapsed and uncollapsed faults, however, so they can still make fault reports and test coverage calculations.

Supported Fault Model Types

The ATPG tool supports stuck-at, pseudo stuck-at, toggle, and transition fault models. In addition to these, the tool supports static bridge and path delay fault models. The following subsections discuss these supported fault models, along with their fault collapsing rules.

In addition to the above standard fault models, the ATPG tool supports User-Defined Fault Models (UDFM). For more information, refer to [About User-Defined Fault Modeling](#).

Functional Testing and the Stuck-At Fault Model	49
Functional Testing and the Toggle Fault Model.....	50
IDDQ Testing and the Pseudo Stuck-At Fault Model	50
At-Speed Testing and the Transition Fault Model.....	52
At-Speed Testing and the Path Delay Fault Model	53

Functional Testing and the Stuck-At Fault Model

Functional testing uses the *single stuck-at model*, the most common fault model used in fault simulation, because of its effectiveness in finding many common defect types. The stuck-at fault models the behavior that occurs if the terminals of a gate are stuck at either a high (stuck-at-1) or low (stuck-at-0) voltage. The fault sites for this fault model include the pins of primitive instances.

[Figure 2-7](#) shows the possible stuck-at faults that could occur on a single AND gate.

Figure 2-7. Single Stuck-At Faults for AND Gate



Possible Errors: 6

- "a" s-a-1, "a" s-a-0
- "b" s-a-1, "b" s-a-0
- "c" s-a-1, "c" s-a-0

For a single-output, n-input gate, there are $2(n+1)$ possible stuck-at errors. In this case, with $n=2$, six stuck-at errors are possible.

The ATPG tool uses the following fault collapsing rules for the single stuck-at model:

- **Buffer** — Input stuck-at-0 is equivalent to output stuck-at-0. Input stuck-at-1 is equivalent to output stuck-at-1.

- **Inverter** — Input stuck-at-0 is equivalent to output stuck-at-1. Input stuck-at-1 is equivalent to output stuck-at-0.
- **AND** — Output stuck-at-0 is equivalent to any input stuck-at-0.
- **NAND** — Output stuck-at-1 is equivalent to any input stuck-at-0.
- **OR** — Output stuck-at-1 is equivalent to any input stuck-at-1.
- **NOR** — Output stuck-at-0 is equivalent to any input stuck-at-1.
- **Net between single output pin and single input pin** — Output pin stuck-at-0 is equivalent to input pin stuck-at-0. Output pin stuck-at-1 is equivalent to input pin stuck-at-1.

Functional Testing and the Toggle Fault Model

Toggle fault testing ensures that a node can be driven to both a logical 0 and a logical 1 voltage. This type of test indicates the extent of your control over circuit nodes. Because the toggle fault model is faster and requires less overhead to run than stuck-at fault testing, you can experiment with different circuit configurations and get a quick indication of how much control you have over your circuit nodes.

The ATPG tool uses the following fault collapsing rules for the toggle fault model:

- **Buffer** — A fault on the input is equivalent to the same fault value at the output.
- **Inverter** — A fault on the input is equivalent to the opposite fault value at the output.
- **Net between single output pin and multiple input pin** — All faults of the same value are equivalent.

IDDQ Testing and the Pseudo Stuck-At Fault Model

IDDQ testing, in general, can use several different types of fault models, including node toggle, pseudo stuck-at, transistor leakage, transistor stuck, and general node shorts.

The ATPG tool supports the *pseudo stuck-at* fault model for IDDQ testing. Testing detects a pseudo stuck-at model at a node if the fault is excited and propagated to the output of a cell (library model instance or primitive). Because library models can be hierarchical, fault modeling occurs at different levels of detail.

The pseudo stuck-at fault model detects all defects found by transistor-based fault models—if used at a sufficiently low level. The pseudo stuck-at fault model also detects several other types of defects that the traditional stuck-at fault model cannot detect, such as some adjacent bridging defects and CMOS transistor stuck-on conditions.

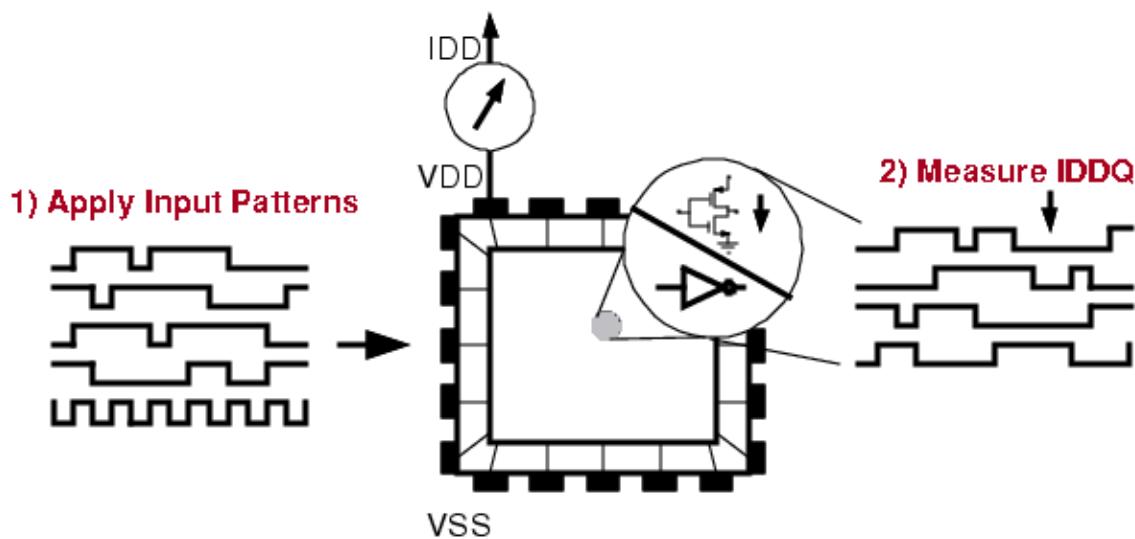
The benefit of using the pseudo stuck-at fault model is that it lets you obtain high defect coverage using IDDQ testing, without having to generate accurate transistor-level models for all library components.

The transistor leakage fault model is another fault model commonly used for IDDQ testing. This fault model models each transistor as a four terminal device, with six associated faults. The six faults for an NMOS transistor include G-S, G-D, D-S, G-SS, D-SS, and S-SS (where G, D, S, and SS are the gate, drain, source, and substrate, respectively).

You can only use the transistor level fault model on gate-level designs if each of the library models contains detailed transistor level information. Pseudo stuck-at faults on gate-level models equate to the corresponding transistor leakage faults for all primitive gates and fanout-free combinational primitives. Thus, without the detailed transistor-level information, you should use the pseudo stuck-at fault model as a convenient and accurate way to model faults in a gate-level design for IDDQ testing.

[Figure 2-8](#) shows the IDDQ testing process using the pseudo stuck-at fault model.

Figure 2-8. IDDQ Fault Testing



The pseudo stuck-at model detects internal transistor shorts, as well as “hard” stuck-ats (a node actually shorted to VDD or GND), using the principle that current flows when you try to drive two connected nodes to different values. While stuck-at fault models require propagation of the fault effects to a primary output, pseudo stuck-at fault models enable fault detection at the output of primitive gates or library cells.

IDDQ testing detects output pseudo stuck-at faults if the primitive or library cell output pin goes to the opposite value. Likewise, IDDQ testing detects input pseudo stuck-at faults when the input pin has the opposite value of the fault and the fault effect propagates to the output of the primitive or library cell.

By combining IDDQ testing with traditional stuck-at fault testing, you can greatly improve the overall test coverage of your design. However, because it is costly and impractical to monitor current for every vector in the test set, you can supplement an existing stuck-at test set with a compact set of test vectors for measuring IDDQ. This set of IDDQ vectors can be generated automatically. Refer to section “[IDDQ Test Set Creation](#)” on page 322 for information.

The fault collapsing rule for the pseudo stuck-at fault model is as follows: for faults associated with a single cell, pseudo stuck-at faults are considered equivalent if the corresponding stuck-at faults are equivalent.

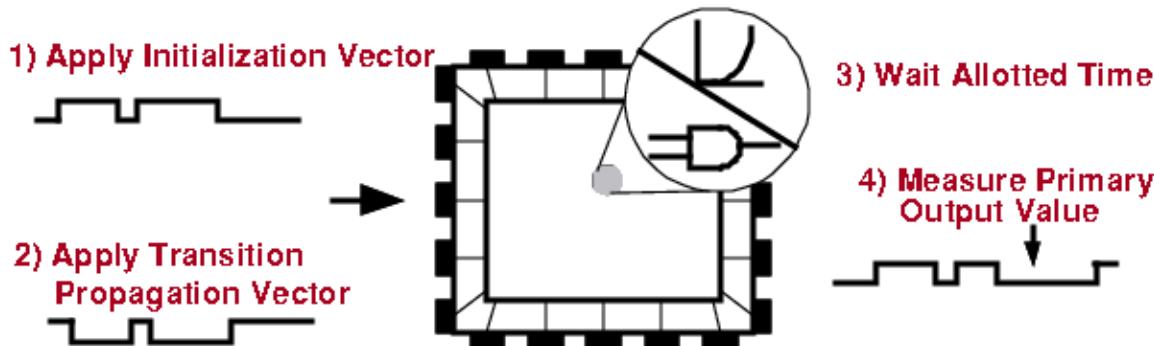
set_transition_holdpi — Freezes all primary inputs values other than clocks and RAM controls during multiple cycles of pattern generation.

At-Speed Testing and the Transition Fault Model

Transition faults model large delay defects at gate terminals in the circuit under test. The transition fault model behaves as a stuck-at fault for a temporary period of time. The slow-to-rise transition fault models a device pin that is defective because its value is slow to change from a 0 to a 1. The slow-to-fall transition fault models a device pin that is defective because its value is slow to change from a 1 to a 0.

Figure 2-9 demonstrates the at-speed testing process using the transition fault model. In this example, the process could be testing for a slow-to-rise or slow-to-fall fault on any of the pins of the AND gate.

Figure 2-9. Transition Fault Detection Process



A transition fault requires two test vectors for detection: an *initialization vector* and a *transition propagation vector*. The initialization vector propagates the initial transition value to the fault site. The transition vector, which is identical to the stuck-at fault pattern, propagates the final transition value to the fault site. To detect the fault, the tool applies proper at-speed timing relative to the second vector, and measures the propagated effect at an external observation point.

The tool uses the following fault collapsing rules for the transition fault model:

- **Buffer** — a fault on the input is equivalent to the same fault value at the output.
- **Inverter** — a fault on the input is equivalent to the opposite fault value at the output.
- **Net between single output pin and single input pin** — all faults of the same value are equivalent.

`set_fault_type` — Specifies the fault model for which the tool develops or selects ATPG patterns. The transition option for this command specifies the tool to develop or select ATPG patterns for the transition fault model.

For more information on generating transition test sets, refer to “[Transition Delay Test Set Creation](#)” on page 326.

At-Speed Testing and the Path Delay Fault Model

Path delay faults model defects in circuit paths. Unlike the other fault types, path delay faults do not have localized fault sites. Rather, they are associated with testing the combined delay through all gates of specific paths (typically critical paths).

Path topology and edge type identify path delay faults. The path topology describes a user-specified path from beginning, or *launch point*, through a combinational path to the end, or *capture point*. The launch point is either a primary input or a state element. The capture point is either a primary output or a state element. State elements used for launch or capture points are either scan elements or non-scan elements that qualify for clock-sequential handling. A path definition file defines the paths for which you want patterns generated. The `add_faults` command specifies the edge type for path delay faults.

The edge type defines the type of transition placed on the launch point that you want to detect at the capture point. A “0” indicates a rising edge type, which is consistent with the slow-to-rise transition fault and is similar to a temporary stuck-at-0 fault. A “1” indicates a falling edge type, which is consistent with the slow-to-fall transition fault and is similar to a temporary stuck-at-1 fault.

The ATPG tool targets multiple path delay faults for each pattern it generates. Within the (ASCII) test pattern set, patterns that detect path delay faults include comments after the pattern statement identifying the path fault, type of detection, time and point of launch event, time and point of capture event, and the observation point. Information about which paths were detected by each pattern is also included.

For more information on generating path delay test sets, refer to “[Path Delay Test Set Creation](#)” on page 349.

Fault Manipulation

Fault manipulation is also known as fault bucketing. You manipulate faults when you group faults of different regions into different classes and then remove them from the list or avoid targeting them during ATPG.

Overview of Fault Manipulation	54
Fault Manipulation Functionality	54

Overview of Fault Manipulation

Fault manipulation (report/write/delete faults) is based upon a user-specified region so certain faults can be excluded from ATPG or be re-classified into a user-defined AU or DI fault subclass.

The user-specified region is defined by a logic cone using at least one of these options:

- **start pins** — A list of pins that are the start points of the region
- **through pins** — A list of pins that are the through pins of the region
- **end pins** — A list of pins that are the end points of the region.

When you specify more than one of the above options, the region is the intersection cone of all the options specified. The tool enables you to trace the region either only within the combinational cone, or trace through the non-scan cells.

Note

 The options for fault manipulation support all pin-based fault models, such as stuck-at and transition fault models, except user-defined fault models (UDFM). They also do not support path-delay and bridging fault models.

Fault Manipulation Functionality

You access fault manipulation functionality with the report_faults, write_faults, and delete_faults commands with additional options to identify the start, through, and stop pins of the fault selection cone. You can also specify an option to indicated whether the cone trace should stop at any sequential or scan cells.

Examples of Fault Manipulation

Example 1

This example writes the faults in the user-specified region as a new user defined AU subclass, AU.USR1. It then reads the faults back so the selected faults are treated as AU.USR1 and excluded from the create_patterns target.

```
// the first command changes default for the fanin/fanout switches
add_faults -all
write_faults faults.usr1 -from inst1/* -to flop*/D \
    -change_classification_to AU.USR1
read_faults faults.usr1 -retain // these faults are changed into AU.USR1
set_capture_procedure on cap1
create_patterns // ATPG will not target the AU.USR1 faults
reset_au_faults // AU.USR1 faults will remain as AU.USR1
set_capture_procedure off -all
set_capture_procedure on cap2
create_patterns // ATPG will not target the AU.USR1 faults
```

Example 2

This example demonstrates the “report_faults -through pins...” command. The -through switch enables the tool to trace forwards and backwards through the specified pin or pins. The tool traces the fan-in and fanout until it reaches an instance driving or driven by an instance outside of the fault selection cone.

Figure 2-10. Fault Selection Cone

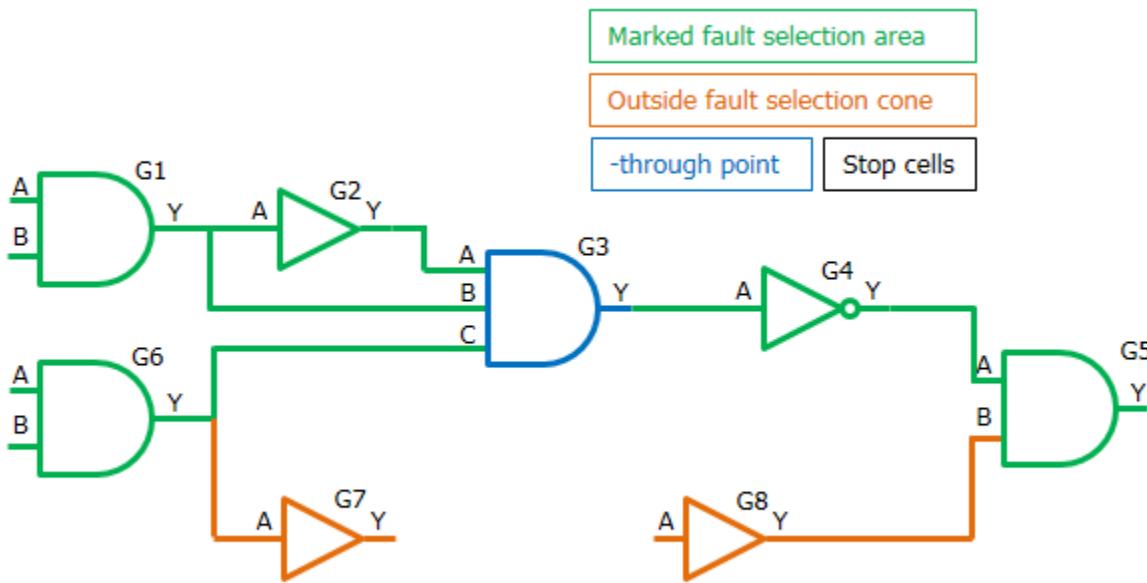


Figure 2-10 shows the fault selection cone in green, with the through point, G3/Y, in blue.

report_faults -through G3/Y

The forward trace stops at G5/A, because G5/B is driven by pin Y on gate G8, shown in orange, which is outside the cone. The backward trace of the fault selection cone includes pins on G3, G2, and G1. While it includes the pin G6/Y, the fault selection cone stop point is the gate G6, because it fans out to G7, which is outside of the fault selection cone.

About User-Defined Fault Modeling

The standard fault models supported by the tool are typically sufficient to create a highly efficient pattern set that detects the majority of potential defects. However, a number of defects are not covered by these models and are detected only by accident; these defects require specific conditions that cannot be defined for the existing fault models.

User-Defined Fault Modeling Usage Notes	56
UDFM File Format	58
Creating a User-Defined Fault Model	63
Generating UDFM Test Patterns	64

User-Defined Fault Modeling Usage Notes

You can use user-defined fault models (UDFMs) to define custom fault models. UDFMs extend the natively-supported fault models (primarily stuck-at, transition, and IDDQ) by adding combinational or sequential constraints on other pins/nets. These custom models enable you to generate specific test patterns for process-related defects like intra-cell bridges or any other kind of defect that requires additional constraints. You can also use UDFMs to define required conditions in an additional fault model to reduce the need for functional test patterns.

Specifically, UDFMs provide the following capabilities:

- Support for generating high compact pattern sets.
- Close integration with the existing ATPG flow.
- Support of static (stuck-at), delay (transition), and IDDQ fault models.
- Definition of test alternatives.
- Definition of additional single or multiple-cycle conditions.
- Definition on library or hierarchy levels.
- Definition on specific instances.
- Write and load of fault status information using the MTFI format. For more information, refer to “[Using MTFI Files](#)” on page 609.
- Generate gate-exhaustive UDFM files without executing cell-aware characterization. For more information, refer to the `-gate_exhaustive` switch for [cellmodelgen](#) in the *Tessent CellModelGen Tool Reference* manual.

The following restrictions apply to UDFMs:

- No support for multiple detection during pattern generation (`set_multiple_detection` command).

- No support for the analyze fault functionality (analyze_fault command).
- Separation of static and delay fault models. Both fault types cannot be handled together in one pattern generation step. The definitions must be handled in different pattern generation runs; therefore, this leads to multiple pattern sets.
- IDDQ pattern generation is supported for static UDFM fault model only.

Note

 UDFM requires additional memory. The amount of additional memory needed depends on the number of UDFM definitions and their complexity.

UDFM File Format

Input for: read_fault_sites

The User-Defined Fault Model (UDFM) file is an ASCII file that models custom fault sites. The UDFM file is created manually or is created using Tessent CellModelGen. UDFM provides a method for expanding native ATPG faults such as stuck-at to exhaustively test library cell models.

Format

A UDFM file must conform to the following syntax rules:

- Precede each line of comment text with a pair of slashes (//).
- Keywords are not case-sensitive.
- Use a colon sign to define a value for a keyword.
- Enclose all string values in double quotation marks (" ").
- Enclose all UDFM declarations in braces ({}).
- Separate each entry within the UDFM declaration with a semicolon (;).

Parameters

In a UDFM file, you use keywords to define the fault models you are creating and their behavior.

Table 2-2. UDFM Keywords

Keyword	Description
UDFM	Required. Specifies the version of the UDFM syntax. The syntax version number must precede the UDFM declaration.
UdfmType	Required. Specifies a user-defined identifier for all faults declared within the UDFM file. This identifier enables you to target the group of faults declared in a UDFM file for an ATPG run or other test pattern manipulations. The following commands enable you to use the UdfmType identifier as an argument: add_faults, delete_faults, report_fault_sites, report_faults, and write_faults.
Cell	Optional. Maps a set of defined faults to a specified library cell. When the UDFM is loaded, the defined faults are applied to all instances of the specified library cell. The cell definition supports only input and output ports.
Module	Optional. Maps a set of defined faults to a specified module. When the UDFM is loaded, the defined faults are applied to all instances of the specified module. The specified value can be a library cell or a module at any hierarchy in the design.

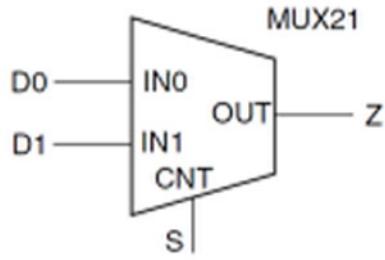
Table 2-2. UDFM Keywords (cont.)

Keyword	Description
Instance	Optional. Maps a set of defined faults to a specified instance. When the UDFM is loaded, the defined faults are applied to the specified instance. Complete pathname must be specified either the full hierarchical path (starting with "/") or relative to the instance path.
Fault	Required. Defines one or more faults. The definition contains two parts: <ul style="list-style-type: none"> • Fault identification string — Name used to reference the fault. • List of test alternatives — When more than one test is listed, the fault is tested if any test alternatives are satisfied.
Test	Required. Specifies the fault conditions to test for. You must minimally specify the defect type with the faulty value and, optionally, a list of conditions. For more information, see the StaticFault or DelayFault keyword.
Conditions	Optional. Defines the nodes and values that represent all fault conditions. To fulfill a <i>condition</i> , the specified node must be set to the specified value: <ul style="list-style-type: none"> • 0: state low • 1: state high • —: no condition
StaticFault or DelayFault	One of these keywords is required, except when you also use the “Observation” keyword set to “false.” Specifies the faulty value at the specified location. If this statement applies to a cell, you can specify only cell ports; if it applies to module definitions, you can specify only ports and module internal elements.
Observation	Optional. Enables you to disable fault propagation as required by faults models like toggle bridge. By default, fault propagation is enabled. Valid arguments are “true” or “false.” Note that when you disable observation (observation:false), the definition of the UDFM faulty value (StaticFault or DelayFault) becomes optional. This attribute is supported only on the “UDFMTyp” level and disables the fault propagation for all fault models that follow.
Properties	Optional. Stores information for future uses or by external user tools.
EncryptedTest	Optional. Automatically used by Tesson CellModelGen to protect the IP. For more information, refer to the Tesson CellModelGen Tool Reference .

Examples

Intra-cell Bridge Fault Example

The following is an example that tests for a cell internal bridge fault of a 2-to-1 multiplexer (MUX21) from the CMOS123_std library. This example specifies two tests with the required activation conditions on the input pins for observing on pin Z the effects of the bridge fault.



```

UDFM {
    version : 1;
    Properties {
        "library-name" : "CMOS123_std";
        "create-by" : "my tool 1.0";
    }
    UdfmType ("intra-cell-bridges") {
        Cell ("MUX21") {
            Fault ("myFlt-N1-Z") {
                Test {
                    StaticFault {"Z" : 1;}
                    Conditions {"D0" : 0; "D1" : 0; "S" : 0;}
                }
                Test {
                    StaticFault {"Z" : 0;}
                    Conditions {"D0" : 0; "D1" : 1; "S" : 0;}
                }
            }
        }
    }
}
    
```

Delay Fault Example

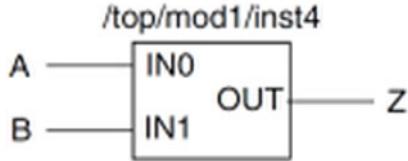
The following is an example that tests for delay faults on the input pins of a 2-to-1 multiplexer (MUX21, shown in the previous example). This example specifies a test where the D0 input transitions from 1 to 0 while observing that the Z output is faulty if it remains at 1.

```

UDFM {
    version : 1;
    UdfmType ("Special-delay") {
        Cell ("MUX21") {
            Fault ("myFlt-D0") {
                Test {
                    DelayFault {"Z" : 1;}
                    Conditions {"D0" : 10; "D1" : 00; "S" : 00;}
                }
            }
        }
    }
}
    
```

Instance-based Fault Example

The following is an example that tests for a bridge fault on the input pins of a single generic 2-input instance. This example specifies two tests for observing on pin Z the effects of a bridge



fault on the input pins.

```

UDFM {
    version : 1;
    UdfmType ("intra-cell-bridges") {
        Instance ("/top/mod1/inst4") {
            Fault ("f001") {
                Test{ StaticFault{"Z":0;} Conditions{"A":0;"B":1;} }
            }
            Fault ("f002") {
                Test{ StaticFault{"Z":1;} Conditions{"A":1;"B":1;} }
            }
        }
    }
}
    
```

Toggle Bridges Example

The following is an example that tests for two specific toggle bridge faults within an entire design. This example tests for bridge faults between the following two sets of nets:

Bridge1: /top/mod2/i3/y /top/mod2/i5/a

Bridge2: /top/mod1/i47/z /top/mod1/iop/c

This example specifies two tests to detect each bridge fault by ensuring that each net pair can be set to different values simultaneously.

```

UDFM {
    Version:2;
    UdfmType("toggle-bridges") {
        Observation:false;
        Instance("/") {
            Fault("Bridge1") {
                Test{Conditions{"top/mod2/i3/y":1;"top/mod2/i5/a":0;}}
                Test{Conditions{"top/mod2/i3/y":0;"top/mod2/i5/a":1;}}
            }
            Fault("Bridge2") {
                Test{Conditions{"top/mod1/i47/z":1;"top/mod1/iop/c":0;}}
                Test{Conditions{"top/mod1/i47/z":0;"top/mod1/iop/c":1;}}
            }
        }
    }
}
    
```

IDDQ Example

The UDFM IDDQ model is restricted to static fault sites only. In general, existing UDFM files with static fault site definitions used for static patterns generation can be used for IDDQ in the same way. The following is an example that tests for one specific bridge fault within an entire design, by applying the four way model. This example tests for the bridge fault between the following two nets:

Bridge1: /top/mod2/i3/y /top/mod2/i5/a

This example specifies four fault sites to detect a bridge between the two nets by ensuring that the net pair will be set to different values simultaneously.

Fault sites:

- Bridge1_A_B_dom0 — Net A is set to 1 and net B to 0. The fault effect is observed at net A.
- Bridge1_A_B_dom1 — Net A is set to 0 and net B to 1. The fault effect is observed at net A
- Bridge1_B_A_dom0 —Net B is set to 1 and net A to 0. The fault effect is observed at net B
- Bridge1_B_A_dom1 — Net B is set to 0 and net A to 1. The fault effect is observed at net B

```
UDFM {
    Version:2;
    UdfmType("bridges") {
        Instance("/")
            Fault("Bridge1_A_B_dom0") {Test{StaticFault{"top/mod2/i3/y":0;}}
                Conditions{"top/mod2/i5/a":0;}}}
            Fault("Bridge1_A_B_dom1") {Test{StaticFault{"top/mod2/i3/y":1;}}
                Conditions{"top/mod2/i5/a":1;}}}
            Fault("Bridge1_B_A_dom0") {Test{StaticFault{"top/mod2/i5/a":0;}}
                Conditions{"top/mod2/i3/y":0;}}}
            Fault("Bridge1_B_A_dom1") {Test{StaticFault{"top/mod2/i5/a":1;}}
                Conditions{"top/mod2/i3/y":1;}}}
    }
}
```

This UDFM definition would be in a file read by the `read_fault_sites` command. These commands are an example of how you use the UDFM definition for IDDQ pattern generation:

```
set_fault_type UDFM -iddq_faults
read_fault_sites <UDFM file name>
add_faults -all
create_patterns
```

Creating a User-Defined Fault Model

You can manually define your own fault models to detect faults that are not included in standard fault models supported by the tool.

Prerequisites

- Text editor

Procedure

Note



In this procedure, entries added in the current step are shown in **bold**.

1. In a text editor, enter the UDFM statement as shown here:

```
UDFM { }
```

All of the additional statements you use to define all fault models will be contained within the curly brackets of this statement.

2. Add the UDFM version number statement next.

```
UDFM {
    version : 1;
}
```

3. Add the UdfmType keyword to the ASCII file under the version number statement to create a fault model name.

```
UDFM {
    version : 1; // Syntax version ensures future compatibility
    UdfmType ("udfm_fault_model")
{
}
}
```

4. Specify the type of object you want to attach the fault to using the Cell, Module, and Instance keywords. For more information on these keywords, see [UDFM Keywords](#).

```
UDFM {
    version : 1;
    UdfmType ("udfm_fault_model") {
        Cell ("MUX21") {
        }
    }
}
```

5. Define a unique fault model name using the Fault keyword. You can define multiple faults.

```
UDFM {
    version : 1;
    UdfmType ("udfm_fault_model") {
        Cell ("MUX21") {
            Fault ("myFlt-D0") {
            }
        }
    }
}
```

6. Define how the defect can be tested using the Test, StaticFault or DelayFault, and Conditions keywords. Notice that the following example also shows how to use the conditions statement for a single assignment or a list of assignments. See “[UDFM Keywords](#)” for the complete list of keywords.

```
UDFM {
    version : 1;
    UdfmType ("udfm_fault_model") {
        Cell ("MUX21") {
            Fault ("myFlt-N1-Z") {
                Test {
                    StaticFault {"Z": 1;}
                    Conditions {"D0": 0; "D1" : 0; "S" : 0;}
                }
                Test {
                    StaticFault {"Z" : 0;}
                    Conditions {"D0" : 0; "D1": 1; "S" : 0;}
                }
            }
        }
    }
}
```

7. Save the file using a *.udfm* extension.

Generating UDFM Test Patterns

UDFM test pattern generation is similar to pattern generation for bridge logic faults. In both cases, fault definitions are accessed from an external file and the internal fault list is built using that data. It is important to understand what is needed for UDFM pattern generation.

UDFM test pattern generation for static fault models is similar to test pattern generation for stuck-at faults. As a result, most ATPG settings for stuck-at faults can be used for UDFM faults. However, because UDFM fault definitions must be imported, the following commands will work slightly differently than they do for stuck-at faults:

add_fault	delete_fault_site	report_faults	set_fault_type
delete_faults	read_fault_sites	report_udfm_statistic	write_faults

Prerequisites

- UDFM file that contains fault definitions.

Procedure

1. Set the fault type to UDFM using the [set_fault_type](#) command with the UDFM option:

set_fault_type udfm

Note

 When you change the fault type, the current fault list and internal test pattern set are deleted.

2. Load the fault definitions from a specified UDFM file into your current tool session using the [read_fault_sites](#) command:

read_fault_sites <filename>.udfm

3. Create the internal fault list using all of the fault definitions from the UDFM file with the [add_faults](#) command:

add_faults -All

4. Generate test patterns using the [create_patterns](#) command:

create_patterns

For more information, refer to the [Tessent Shell Reference Manual](#).

5. Print out the resulting statistics using the [report_statistics](#) command.

report_statistics

For more information, refer to the [Tessent Shell Reference Manual](#).

Related Topics

[create_patterns](#) [Tessent Shell Reference Manual]

[report_statistics](#) [Tessent Shell Reference Manual]

Multiple Detect

The basic idea of multiple detect (n-detect) is to randomly target each fault multiple times. By changing the way the fault is targeted and the other values in the pattern set, the potential to detect a bridge increases.

This approach starts with a standard stuck-at or transition pattern set. Each fault is graded for multiple detect. Additional ATPG is then performed and patterns created targeting the faults that have lower than the multiple detect target threshold.

Bridge Coverage Estimate..... 66

Bridge Coverage Estimate

Bridge coverage estimate (BCE) is a metric for reporting the ability of multiple detect to statistically detect a bridge defect.

If a bridge fault exists between the target fault site and another net then there is a 50 percent chance of detecting the fault with one pattern. If a second randomly different pattern targets the same fault then the probability of detecting the bridge is $1 - 0.5^2$. BCE performs this type of calculation for all faults in the target list and will always be lower than the test coverage value.

The following shows the statistics report that is automatically produced when multiple detection or embedded multiple detect is enabled:

Statistics Report		
Transition Faults		
Fault Classes	#faults	(total)
FU (full)	1114	
DS (det_simulation)	1039	(93.27%)
UU (unused)	30	(2.69%)
TI (tied)	4	(0.36%)
RE (redundant)	3	(0.27%)
AU (atpg_untestable)	38	(3.41%)
<hr/>		
Coverage		
test_coverage	96.47%	
fault_coverage	93.27%	
atpg_effectiveness	100.00%	
<hr/>		
#test_patterns	130	
#clock_sequential_patterns	130	
#simulated_patterns	256	
CPU_time (secs)	0.5	
<hr/>		
Multiple Detection Statistics		
<hr/>		
Detections	DS Faults	Test Coverage
(N)	(Detection == N)	(Detection >= N)
<hr/>		
1	0 (0.00%)	1039 (96.47%)
2	0 (0.00%)	1039 (96.47%)
3	93 (8.35%)	1039 (96.47%)
4	56 (5.03%)	946 (87.84%)
5	54 (4.85%)	890 (82.64%)
6	56 (5.03%)	836 (77.62%)
7	50 (4.49%)	780 (72.42%)
8	46 (4.13%)	730 (67.78%)
9	54 (4.85%)	684 (63.51%)
10+	630 (56.55%)	630 (58.50%)
<hr/>		
bridge_coverage_estimate		94.71%
<hr/>		

The report includes the BCE value and the number of faults with various detects less than the target detection value.

Related Topics

[The Bridge Parameters File](#)

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

[set_diagnosis_options \[Tessent Shell Reference Manual\]](#)
[report_fault_sites \[Tessent Shell Reference Manual\]](#)

Embedded Multiple Detect

Embedded multiple detect (EMD) is a method of improving the multiple detect of a pattern set without increasing the number of patterns within that pattern set. Essentially, EMD produces the same quality patterns as a standard pattern set but in addition adds improved multiple detection for free. The only cost for the extra multiple detection is a longer run time during pattern creation of about 30 to 50 percent. As a result, EMD is often considered a no-cost additional value and is used for ATPG.

When performing ATPG, the tool tries to detect as many previously undetected faults in parallel within the same pattern as possible. However, even though ATPG maximizes the number of previously undetected faults detected per pattern, only a small percentage of scan cells will have specific values necessary for the detection. These *specified bits* that need to be loaded in scan cells for that pattern are referred to as the test cube. The remaining scan cells that are not filled with test cube values are randomly filled for fortuitous detection of untargeted faults. EMD uses the same ATPG starting point to produce a test cube but then determines if there are some faults that previously had a low number of detections. For these faults, EMD will put additional scan cell values added to the test cube to improve multiple detection on top of the new detection pattern.

EMD will have a multiple detection that is better than normal ATPG but might not be as high a BCE as n-detect with additional patterns could produce. In a design containing EDT circuitry, the amount of detection will be dependent on the how aggressive the compression is. The more aggressive (higher) compression, the lower the encoding capacity and the fewer test cube bits can be specified per pattern. If a design is targeting 200x compression then the available test cube bits might be mostly filled up for many of the patterns with values for the undetected fault detection. As a result, the additional EMD multiple detection might not be significantly higher than BCE for the standard pattern set.

Standard multiple detect will have a cost of additional patterns but will also have a higher multiple detection than EMD. How much difference between EMD and multiple detect will be dependent on the particular design's pattern set and the level of compression used¹.

1. J. Geuzebroek, et al., “Embedded Multi-Detect ATPG and Its Effect on the Detection of Unmodeled Defects”, Proceedings IEEE Int. Test Conference, 2007

Multiple Detect for EMD

You can enable either EMD or multiple detect using the `set_multiple_detection` command. Either approach is supported with stuck-at and transition patterns. You can use the following arguments with the `set_multiple_detection` command:

- **-Guaranteed_atpg_detection** — Sets the multiple detect target for each fault. ATPG will try to target each fault the specified number of times. ATPG does not guarantee that the fault will be detected in a completely different path but randomly changes the way it excites and propagates the fault. In addition, the random fill is different so values around the target fault are likely to be randomly different than previous detections.
- **-Desired_atpg_detections** — Sets the EMD target. Users often set this target to 5 or a value in that range.
- **-Simulation_drop_limit** — This is the accuracy of the BCE calculation. In general, there is no reason to change this value from the default of 10. This means that the BCE simulations stop once a fault is learned to be detected 10 times. A fault multiple detected 10 times will have a BCE and statistical chance of detecting a defect of $1 - 1/2e10$ or 0.99902. This is only a 0.0009 percent inaccuracy which is slightly conservative but insignificant.

Logic BIST

Logic BIST has a natural very high multiple detection. The faults that are detected with logic BIST would often have multiple detection well above 10. This is in part due to the very large number of patterns typically used for logic BIST. In addition, many of the hard to detect areas of a circuit are made randomly testable and easier to produce high multiple detect coverage with test logic inserted during logic BIST.

Multiple Detect and AU Faults

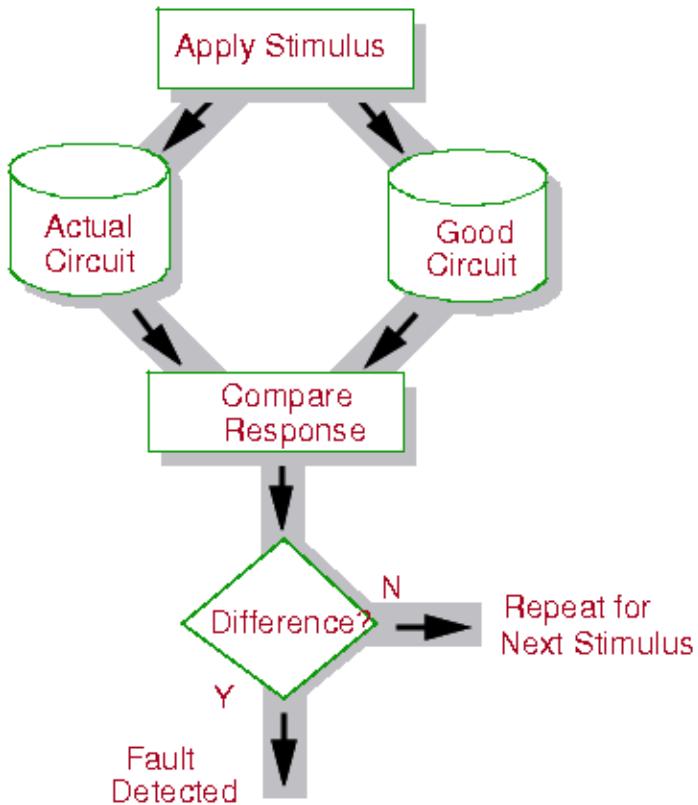
During multiple detect ATPG, the AU (ATPG_untestable) fault count changes only at the beginning of each ATPG loop rather than during the loop. This is normal behavior when using NCPs (named capture procedures) for ATPG. The tool updates AU faults after going through all the NCPs at the end of the loop.

Fault Detection

Faults detection works by comparing the response of a known-good version of the circuit to that of the actual circuit, for a given stimulus set. A fault exists if there is any difference in the responses. You then repeat the process for each stimulus set.

Figure 2-11 shows the basic fault detection process.

Figure 2-11. Fault Detection Process



Path Sensitization and Fault Detection

One common fault detection approach is *path sensitization*. The path sensitization method, which is used by the tool to detect stuck-at faults, starts at the fault site and tries to construct a vector to propagate the fault effect to a primary output. When successful, the tools create a stimulus set (a test pattern) to detect the fault. They attempt to do this for each fault in the circuit's fault universe.

Figure 2-12 shows an example circuit for which path sensitization is appropriate.

Figure 2-12. Path Sensitization Example

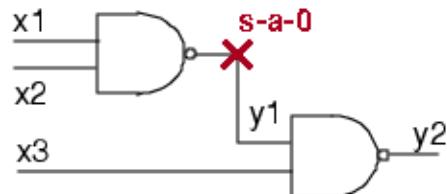


Figure 2-12 has a stuck-at-0 on line y1 as the target fault. The x1, x2, and x3 signals are the primary inputs, and y2 is the primary output. The path sensitization procedure for this example follows:

1. Find an input value that sets the fault site to the opposite of the desired value. In this case, the process needs to determine the input values necessary at x1 and/or x2 that set y1 to a 1, since the target fault is s-a-0. Setting x1 (or x2) to a 0 properly sets y1 to a 1.
2. Select a path to propagate the response of the fault site to a primary output. In this case, the fault response propagates to primary output y2.
3. Specify the input values (in addition to those specified in step 1) to enable detection at the primary output. In this case, in order to detect the fault at y1, the x3 input must be set to a 1.

Fault Classes

The tool categorizes faults into fault classes, based on how the faults were detected or why they could not be detected. Each fault class has a unique name and two character class code. When reporting faults, the tool uses either the class name or the class code to identify the fault class to which the fault belongs.

Note

 The tools may classify a fault in different categories, depending on the selected fault type.

Untestable (UT)	72
Testable (TE)	75
Fault Class Hierarchy	77
Fault Sub-classes	77

Untestable (UT)

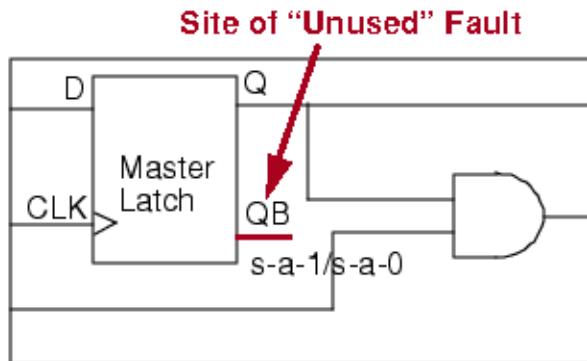
Untestable (UT) faults are faults for which no pattern can exist to either detect or possible-detect them. Untestable faults cannot cause functional failures, so the tools exclude them when calculating test coverage. Because the tools acquire some knowledge of faults prior to ATPG, they classify certain unused, tied, or blocked faults before ATPG runs. When ATPG runs, it immediately places these faults in the appropriate categories. However, redundant fault detection requires further analysis.

The following list discusses each of the untestable fault classes.

1. Unused (UU)

The unused fault class includes all faults on circuitry unconnected to any circuit observation point and faults on floating primary outputs. For information about UU fault sub-classes, refer to [Table 2-4](#) on page 78. [Figure 2-13](#) shows the site of an unused fault.

Figure 2-13. Example of Unused Fault in Circuitry



2. Tied (TI)

The tied fault class includes faults on gates where the point of the fault is tied to a value identical to the fault stuck value. These are possible causes of tied circuitry:

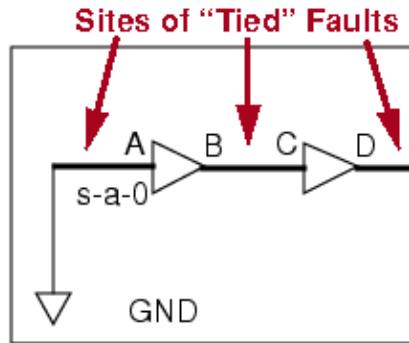
- Tied signals
- AND and OR gates with complementary inputs
- Exclusive-OR gates with common inputs
- Line holds due to primary input pins held at a constant logic value during test by CT0 or CT1 pin constraints you applied with the [add_input_constraints](#) command

Note

 The tools do not use line holds set by the “add_input_constraints -C0” (or C1) command to determine tied circuitry. C0 and C1 pin constraints (as distinct from CT0 and CT1 constraints) result in [ATPG_untestable \(AU\)](#) faults, not tied faults. For more information, refer to the [add_input_constraints](#) command.

Figure 2-14 shows the site of a tied fault.

Figure 2-14. Example of Tied Fault in Circuitry



Because tied values propagate, the tied circuitry at A causes tied faults at A, B, C, and D.

3. Blocked (BL)

The blocked fault class includes faults on circuitry for which tied logic blocks all paths to an observable point. These are possible causes of tied circuitry:

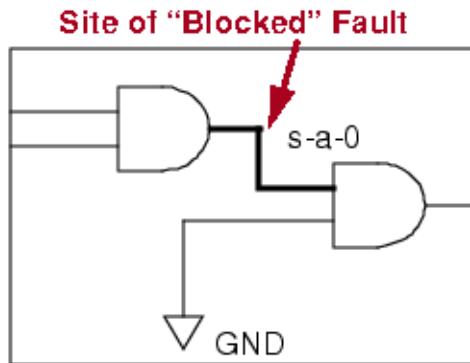
- Tied signals
- AND and OR gates with complementary inputs.
- Exclusive-OR gates with common inputs.
- Line holds due to primary input pins held at a constant logic value during test by CT0 or CT1 pin constraints you applied with the [add_input_constraints](#) command.

Note

The tools do not use line holds set by the “add_input_constraints -C0” (or C1) command to determine tied circuitry. C0 and C1 pin constraints (as distinct from CT0 and CT1 constraints) result in **ATPG_untestable (AU)** faults, not blocked faults. For more information, refer to the [add_input_constraints](#) command.

This class also includes faults on selector lines of multiplexers that have identical data lines. Figure 2-15 shows the site of a blocked fault.

Figure 2-15. Example of Blocked Fault in Circuitry



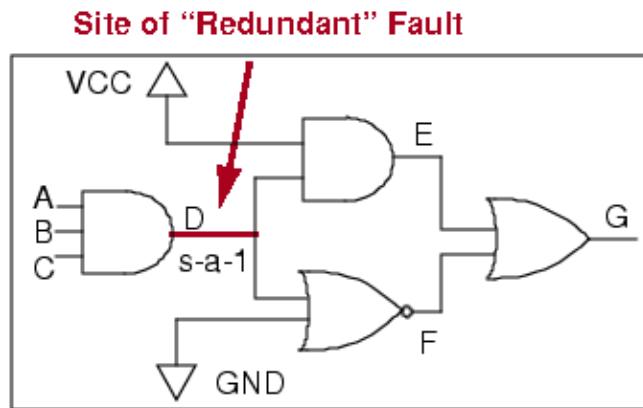
Note

Tied faults and blocked faults can be equivalent faults.

4. Redundant (RE)

The redundant fault class includes faults the test generator considers undetectable. After the test pattern generator exhausts all patterns, it performs a special analysis to verify that the fault is undetectable under any conditions. Figure 2-16 shows the site of a redundant fault.

Figure 2-16. Example of Redundant Fault in Circuitry



In this circuit, signal G always has the value of 1, no matter what the values of A, B, and C. If D is stuck at 1, this fault is undetectable because the value of G can never change, regardless of the value at D.

Testable (TE)

Testable (TE) faults are all those faults that cannot be proven untestable.

These are the testable fault classes:

1. Detected (DT)

The detected fault class includes all faults that the ATPG process identifies as detected. The detected fault class contains two groups:

- det_simulation (DS) - faults detected when the tool performs fault simulation.
- det_implication (DI) - faults detected when the tool performs learning analysis.

The det_implication group normally includes faults in the scan path circuitry, as well as faults that propagate ungated to the shift clock input of scan cells. The scan chain functional test, which detects a binary difference at an observation point, guarantees detection of these faults. The tool also classifies scan enable stuck-in-system-mode faults on the multiplexer select line of mux-DFFs as DI.

The tool provides the update_implication_detections command, which lets you specify additional types of faults for this category. Refer to the [update_implication_detections](#) command description in the *Tessent Shell Reference Manual*.

For path delay testing, the detected fault class includes two other groups:

- det_robust (DR) - robust detected faults.
- det_functional (DF) - functionally detected faults.

For detailed information on the path delay groups, refer to “[Path Delay Fault Detection](#)” on page 349.

2. Posdet (PD)

The posdet, or possible-detected fault class includes all faults that fault simulation identifies as possible-detected but not hard detected. A possible-detected fault results from a good-machine simulation observing 0 or 1 and the faulty machine observing X. A hard-detected fault results from binary (not X) differences between the good and faulty machine simulations. The posdet class contains two groups:

- posdet_testable (PT) - potentially detectable posdet faults. PT faults result when the tool cannot prove the 0/X or 1/X difference is the only possible outcome. A higher abort limit may reduce the number of these faults.

- posdet_untestable (PU) - proven ATPG_untestable during pattern generation and hard undetectable posdet faults. Typically, faults may be classified as PU during ATPG or when the “compress_patterns -reset_au” command is used.

By default, the calculations give 50 percent credit for posdet faults. You can adjust the credit percentage with the [set_possible_credit](#) command.

3. **ATPG_untestable (AU)**

The ATPG_untestable fault class includes all faults for which the test generator is unable to find a pattern to create a test, and yet cannot prove the fault redundant. Testable faults become ATPG_untestable faults because of constraints, or limitations, placed on the ATPG tool (such as a pin constraint or an insufficient sequential depth). These faults may be possible-detectable, or detectable, if you remove some constraint, or change some limitation, on the test generator (such as removing a pin constraint or changing the sequential depth). You cannot detect them by increasing the test generator abort limit.

The tools place faults in the AU category based on the type of deterministic test generation method used. That is, different test methods create different AU fault sets. Likewise, the tool can create different AU fault sets even using the same test method. Thus, if you switch test methods (that is, change the fault type) or tools, you should reset the AU fault list using the [reset_au_faults](#) command.

Note

 During multiple detect ATPG, the AU fault count changes only at the beginning of each ATPG loop rather than during the loop. This is normal behavior when using NCPs (named capture procedures) for ATPG. The tool updates AU faults after going through all the NCPs at the end of the loop.

AU faults are categorized into several predefined fault sub-classes, as listed in [Table 2-4](#) on page 78.

4. **Undetected (UD)**

The undetected fault class includes undetected faults that cannot be proven untestable or ATPG_untestable. The undetected class contains groups:

- **uncontrolled (UC)** - undetected faults, which during pattern simulation, never achieve the value at the point of the fault required for fault detection—that is, they are uncontrollable.
- **unobserved (UO)** - faults whose effects do not propagate to an observable point.

All testable faults prior to ATPG are put in the UC category. Faults that remain UC or UO after ATPG are aborted, which means that a higher abort limit may reduce the number of UC or UO faults.

Note

 Uncontrolled and unobserved faults can be equivalent faults. If a fault is both uncontrolled and unobserved, it is categorized as UC.

Fault Class Hierarchy

Fault classes are hierarchical. The highest level, Full, includes all faults in the fault list.

Within Full, faults are classified into untestable and testable fault classes, and so on, as shown in Table 2-3.

Table 2-3. Fault Class Hierarchy

1. Full (FU)
1.1 TESTable (TE)
a. DETECTed (DT) <ul style="list-style-type: none"> i. DET_Simulation (DS) ii. DET_Implication (DI) iii. DET_Robust (DR)—Path Delay Testing Only iv. DET_Functional (DF)—Path Delay Testing Only
b. POSDET (PD) <ul style="list-style-type: none"> i. POSDET_Untestable (PU) ii. POSDET_Testable (PT)
c. AtPg_untestable (AU)
d. UNDETECTED (UD) <ul style="list-style-type: none"> i. UNControlled (UC) ii. UNObserved (UO)
1.2 UNTESTable (UT)
a. UNUSED (UU)
b. TIed (TI)
c. Blocked (BL)
d. Redundant (RE)

For any given level of the hierarchy, the tool assigns a fault to one—and only one—class. If the tool can place a fault in more than one class of the same level, the tool places the fault in the class that occurs first in the list of fault classes.

Fault Sub-classes

The DI, AU, UD, and UU fault classes are further categorized into fault sub-classes.

Table 2-4 lists the fault sub-classes. For more information about each AU and UD fault sub-classes, click the hyperlinks in the table.

The DI and AU fault classes can also contain user-defined sub-classes, which you create by grouping a set of faults into one sub-class and assigning a name to the group.

You can use the `reset_di_faults` command to reclassify the DI faults to the uncontrolled (UC) category. Note that the tool does not reclassify DI faults defined by other Tesson plugins, for example DI.MBIST and DI.MBISR, which are defined by Tesson MemoryBIST.

Table 2-4. Fault Sub-classes

Fault Class	Fault Sub-class	Code	Description
DI	EDT_LOGIC	EDT	<p>Faults in EDT logic detected by implication when EDT Finder is On (EDT Finder is on by default). Supported for stuck and transition fault types</p> <ul style="list-style-type: none"> • Includes faults implicitly tested when applying the EDT patterns • Excludes redundant faults in EDT logic, faults in bypass logic or other dual configurations • Excludes faults already identified as implicitly tested
	SCAN_PATH	SCAN	DI faults that are directly in the scan path.
	SCAN_ENABLE	SEN	DI faults that can propagate to and disrupt scan shifting.
	CLOCK	CLK	DI faults that are in either the scan or functional clock cone of state elements.
	SET_RESET	SR	DI faults that are in the SET/RESET cone of state elements.
	DATA_IN	DIN	DI faults that are in the data inputs of nonscan state elements.
	MEMORY	MEM	DI faults that are in read_enable, read_clock, write_enable, write_clock, or set/reset ports of RAM/ROMs.
	MEMORY BIST	MBIST	Faults in the fan-in and fan-out of the memory detected during Tessent MemoryBIST test. These faults are not detectable by ATPG when the memory is bypassed, but might be detectable using multi-load or RAM sequential patterns.
	MEMORY BISR	MBISR	Faults in the fan-in and fan-out of the Tessent MemoryBIST BISR registers. These faults are detected during BISR chain operations. Faults are not detectable by ATPG because BISR registers are not part of the ATPG scan chains.
	User-defined	none	User-defined DI fault subclass.

Table 2-4. Fault Sub-classes (cont.)

Fault Class	Fault Sub-class	Code	Description
AU	AU.BB — BLACK_BOXES	BB	Fault untestable due to black box
	AU.EDT — EDT_BLOCKS	EDT	AU faults in EDT block
	AU.LBIST — HYBRID_LBIST	LBIST	AU faults in the hybrid IP controller
	AU.PC — PIN_CONSTRAINTS	PC	Tied or blocked by input constraint
	AU.TC — TIED_CELLS	TC	Tied or blocked by tied non-scan cell
	AU.OCC — ON_CHIP_CLOCK_CONTROL	OCC	AU faults in the OCC.
	AU.IJTAG — IJTAG	IJTAG	AU faults in the IJTAG instrument.
	AU.LPCT — LOW_PIN_COUNT_TEST	LPCT	AU faults in the low pin count test controller.
	AU.CC — CELL_CONSTRAINTS	CC	Tied or blocked by cell constraint
	AU.FP — FALSE_PATHS	FP	Fault untestable due to false path
	AU.HPI — HOLD_PI	HPI	Faults in the fanout cone of the hold PIs
	AU.MCP — MULTICYCLE_PATHS	MCP	Fault untestable due to multicycle path
	AU.MPO — MASK_PO	MPO	Faults in the fanin cone of the masked POs
	AU.SEQ — SEQUENTIAL_DEPTH	SEQ	Untestable due to insufficient sequential depth
	AU.UDN — UNDRIVEN	UDN	Undetectable faults caused by undriven input pins.
	AU.WIRE — WIRE	WIRE	Faults that drive a WIRE gate with more than one input
	User-defined	none	User-defined AU fault subclass
UD	UD.AAB — ATPG_ABORT	AAB	Fault aborted by ATPG tool
	UD.UNS — UNSUCCESS	UNS	Fault undetected due to unsuccessful test
	UD.EAB — EDT_ABORT	EAB	Fault aborted due to insufficient EDT encoding capacity
UU	CONNECTED	CON	Fault in the logic cone but with no observation point
	FLOATING	FL	Fault on the unconnected output of a cell

AU.BB — BLACK_BOXES

These are faults that are untestable due to a black box, which includes faults that need to be propagated through a black box to reach an observation point, as well as faults whose control or observation requires values from the output(s) of a black box. You can use the [report_black_boxes](#) command to identify the black boxes in your design. Your only potential option for resolving AU.BB faults is to create a model for each black box, although this may not fix the entire problem.

AU.EDT — EDT_BLOCKS

These are faults inside an instance identified as an EDT instance using the [set_edt_instances](#) command. If EDT Finder is turned on (EDT Finder is on by default), most of those faults (regardless of the use of [set_edt_instances](#)) are classified as DI.EDT faults.

To further reduce the number of AU.EDT faults, you must either use default naming for EDT blocks, or you must use the [add_edt_blocks](#) command to explicitly name EDT blocks. Note that the tool excludes AU.EDT faults from the relevant fault coverage by default because the chain and IP test actually checks most of the EDT logic in addition to the scan chains, so other than properly naming EDT blocks, you need not do anything more. For more information about relevant coverage, refer to the description of the [set_relevant_coverage](#) command in the *Tessent Shell Reference Manual*.

AU.LBIST — HYBRID_LBIST

These are faults located within the hybrid IP controller when you are using the hybrid TK/LBIST flow. This is the default fault classification for faults found within the hybrid IP controller if they have not already been identified with another designator, such as unused (UU), tied (TI), or blocked (BL). These faults cannot be detected or improved upon. Refer to “[Fault Coverage Report for the Hybrid IP](#)” in the *Hybrid TK/LBIST Flow User’s Manual* for more information.

AU.PC — PIN_CONSTRAINTS

These are faults that are uncontrollable or that cannot be propagated to an observation point, in the presence of a constraint value. That is, because the tool cannot toggle the pin, the tool cannot test the fanout. The only possible solution is to evaluate whether you really need the input constraint, and if not, to remove the constraint. To help troubleshoot, you can use the “[report_statistics -detailed_analysis](#)” command to report specific pins.

AU.TC — TIED_CELLS

These are faults associated with cells that are always 0 (TIE0) or 1 (TIE1) or X (TIEX) during capture. One example is test data registers that are loaded during [test_setup](#) and then constrained so as to preserve that value during the rest of scan test. The only possible solution is to verify that the tied state is really required, and if not, modify the [test_setup](#) procedure. To

To help troubleshoot, you can use the “[report_statistics -detailed_analysis](#)” command to report specific cells.

AU.OCC — ON_CHIP_CLOCK_CONTROL

These are faults located within the On-Chip Clock Controller (OCC). This is the default classification for the faults found within the OCC if they have not already been identified with another designator.

For more information, refer to “[Tessent On-Chip Clock Controller](#)” on page 629.

AU.IJTAG — IJTAG

These are faults located within the IJTAG instruments. This is the default classification for the faults found within the IJTAG instrument if they have not already been identified with another designator.

Specifically for AU.IJTAG, the tool only identifies faults in ICL instances that have either of the following attributes:

- The [keep_active_during_scan_test](#) ICL module attribute explicitly set to “true”.
- The [keep_active_during_scan_test](#) ICL module attribute is not set, and the [tessent_instrument_type](#) ICL module attribute equals: “mentor::ijtag_node”.

For more information, refer to “[A Detailed IJTAG ATPG Flow](#)” in the *Tessent IJTAG User's Manual*.

AU.LPCT — LOW_PIN_COUNT_TEST

These are faults located within the Low Pin Count Test (LPCT) controller. This is the default classification for the faults found within the LPCT if they have not already been identified with another designator.

See “[Low Pin Count Test Controller](#)” in the *Tessent TestKompress User's Manual* for complete information.

AU.CC — CELL_CONSTRAINTS

These are faults that are uncontrollable or that cannot be propagated to an observation point, in the presence of a constraint value. That is, because the tool cannot toggle a state within the cell, the tool cannot test the fanout or the faults along the input path that need to propagate through the cell. The only possible solution is to evaluate whether you really need the cell constraint, and if not, to remove the constraint. To help troubleshoot, you can use the “[report_statistics -detailed_analysis](#)” command to report specific cells.

For more information about cell constraints and their affect on ATPG, refer to the [add_cell_constraints](#) command description in the *Tessent Shell Reference Manual*.

AU.FP — FALSE_PATHS

These are faults that can be tested only through a false path. However, if there is any other path possible that is not false in which the fault exists, then the fault remains in the fault list.

Therefore, many faults along false paths might not be classified as AU.FP. For example, assume that the end of a long false path is connected to both a flop A and a flop B. If flop B is not in a false path, then the tool can test all the setup faults using flop B and can therefore classify none as AU.FP.

In the case of hold-time false paths, the tool cannot classify them as AU.FP because a test is almost always possible that does not cause a hold-time violation. For example, consider a circuit with a hold-time false path that contains a flop A that feeds a buffer connected to flop B. If a pattern places a 1 on flop A's D input and a 0 on Q, then ATPG simulation would capture X in flop B because the tool cannot be certain the capture cycle would appropriately capture 0 in flop B. That is, the hold-time violation could cause flop A to update and then propagate to flop B before the clock captures at flop B. However, if flop A's D input is 0 and Q is 0, then the 0 would be properly captured regardless of the hold-time false path. This is why the tool cannot remove the faults in this circuit from the fault list and define them as AU.FP.

Note that the tool excludes AU.FP faults from the relevant fault coverage by default. For more information about relevant coverage, refer to the description of the [set_relevant_coverage](#) command in the *Tessent Shell Reference Manual*.

AU.HPI — HOLD_PI

These are faults for transition fault models with launch off-shift disabled that can only be controlled by hold PIs. They are marked AU.HPI because there is no way to launch such a transition from the fault site.

AU.HPI analysis is performed at any of these events:

- At the end of system mode transition, when the tool performs static AU analysis of other subclasses.
- When you change holdpi settings in non-setup mode.
- When you issue the “set_capture_procedure” command in non-setup mode, and the named capture procedure holdpi is different. Analysis is performed because the named capture procedure might override hold PI values.
- When you issue the “read_procedure” command in non-setup mode, and the named capture procedure holdpi is different. Analysis is performed because the named capture procedure might override hold PI values.

AU.MCP — MULTICYCLE_PATHS

These are faults that can be tested only through a multicycle path, which means that the fault cannot propagate between the launch and capture point within a single clock cycle. For

information about resolving multicycle path issues, refer to “[Pattern Failures Due to Timing Exception Paths](#)” on page 337.

AU.MPO — MASK_PO

These are faults that exist in the fanin cone of the masked POs and have no path to other observation points. This category of faults is updated when you make changes to the PO masks by using commands such as `add_output_masks`, `delete_output_masks`, and `set_output_masks`.

AU.SEQ — SEQUENTIAL_DEPTH

These are faults associated with non-scan cells that require multiple clock cycles to propagate to an observe point. One possible solution is to increase the sequential depth using the “`set_pattern_type -sequential`” command. Another solution is to ensure that you have defined all clocks in the circuit using the `add_clocks` command.

AU.UDN — UNDRIVEN

These are faults that cannot be tested due to undriven input pins. For the purpose of this analysis, undriven input pins include inputs driven by X values. Faults on undriven pins and on pins that are untestable due to other pins being undriven are classified as AU.UDN. This analysis is performed by default.

AU.WIRE — WIRE

WIRE faults are identified during AU analysis if they drive WIRE gates with more than one input and it is in a fanout free region dominated by any WIRE gate input.

UD.AAB — ATPG_ABORT

These are faults that are undetected because the tool reached its abort limit. You can gain additional information about UD.AAB faults using the `report_aborted_faults` command. You can raise the abort limit to increase coverage using the `set_abort_limit` command.

UD.UNS — UNSUCCESS

These are faults that are undetected for unknown reasons. There is nothing you can do about faults in this sub-class.

UD.EAB — EDT_ABORT

These are faults that are undetected for one of these reasons:

- Your design’s chain:channel ratio is insufficient to detect the fault. One solution is to run compression analysis on your design to determine the most aggressive chain:channel ratio using the `analyze_compression` command, and then re-create the EDT logic in your design. Another solution is to insert test points to improve the compressibility of faults.

- The tool cannot compress the test due to the scan cells being clustered (which is reported at the end of ATPG). The solution is to insert test points to improve the compressibility of faults.

Fault Reporting

When reporting faults, the tool identifies each fault by three ordered fields.

- Fault value (0 for stuck-at-0 or “slow-to-rise” transition faults; 1 for stuck-at-1 or “slow-to-fall” transition faults)
- Two-character fault class code
- Pin pathname of the fault site

If the tools report uncollapsed faults, they display faults of a collapsed fault group together, with the representative fault first followed by the other members (with EQ fault codes). Use the [report_faults](#) command to report faults.

Testability Calculations

The tool reports several measures of testability. Three important measures are test coverage, fault coverage, and ATPG effectiveness.

- **Test Coverage**

Test coverage, which is a measure of test quality, is the percentage of faults detected from among all testable faults. Typically, this is the number of most concern when you consider the testability of your design.

The tool calculates test coverage using the formula:

$$\frac{\#DT + (\#PD * \text{posdet_credit})}{\#testable} \times 100$$

In this formula, posdet_credit is the user-selectable detection credit (the default is 50 percent) given to possible detected faults with the [set_possible_credit](#) command.

- **Fault Coverage**

Fault coverage consists of the percentage of faults detected from among all faults that the test pattern set tests—treating untestable faults the same as undetected faults.

The tool calculates fault coverage using the formula:

$$\frac{\#DT + (\#PD * \text{posdet_credit})}{\#full} \times 100$$

- **ATPG Effectiveness**

ATPG effectiveness measures the ATPG tool's ability to either create a test for a fault, or prove that a test cannot be created for the fault under the restrictions placed on the tool.

The tool calculates ATPG effectiveness using the formula:

$$\frac{\#DT + \#UT + \#AU + \#PU + (\#PT * posdet_credit)}{\#full} \times 100$$

Chapter 3

Common Tool Terminology and Concepts

Once you understand the basic ideas behind DFT, scan design, and ATPG, you can concentrate on the Mentor Graphics DFT tools and how they operate. Tesson Scan and the ATPG tools not only work toward a common goal (to improve test coverage), they also share common terminology, internal processes, and other tool concepts, such as how to view the design and the scan circuitry.

These are subjects common to the tools:

Scan Terminology	89
Scan Cells	89
Master Element	90
Slave Element	90
Shadow Element	91
Copy Element	92
Extra Element	93
Scan Chains	93
Scan Groups	94
Scan Clocks	95
Scan Architectures	96
Test Procedure Files	96
Model Flattening	98
Design Object Naming	98
The Flattening Process	99
Simulation Primitives of the Flattened Model	100
Learning Analysis	104
Equivalence Relationships	104
Logic Behavior	104
Implied Relationships	105
Forbidden Relationships	106
Dominance Relationships	106
ATPG Design Rules Checking	107
General Rules Checking	107
Procedure Rules Checking	107
Bus Mutual Exclusivity Analysis	108
Scan Chain Tracing	109
Shadow Latch Identification	109
Data Rules Checking	110
Transparent Latch Identification	110

Clock Rules Checking	111
RAM Rules Checking	111
Bus Keeper Analysis	111
Extra Rules Checking	112
Scannability Rules Checking	112
Constrained/Forbidden/Block Value Calculations	112
Clock Terminology	114
Programmable Clock Chopper	114

Scan Terminology

You need to become familiar with the scan terminology common to Tesson Scan and the ATPG tools.

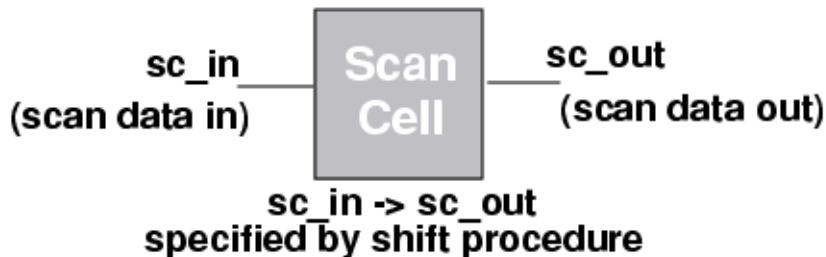
Scan Cells	89
Master Element	90
Slave Element	90
Shadow Element	91
Copy Element	92
Extra Element	93
Scan Chains	93
Scan Groups	94
Scan Clocks	95

Scan Cells

A scan cell is the fundamental, independently-accessible unit of scan circuitry, serving both as a control and observation point for ATPG and fault simulation. You can think of a scan cell as a black box composed of an input, an output, and a procedure specifying how data gets from the input to the output. The circuitry inside the black box is not important as long as the specified procedure shifts data from input to output properly.

Because scan cell operation depends on an external procedure, scan cells are tightly linked to the notion of test procedure files. “[Test Procedure Files](#)” on page 96 discusses test procedure files in detail. [Figure 3-1](#) illustrates the black box concept of a scan cell and its reliance on a test procedure.

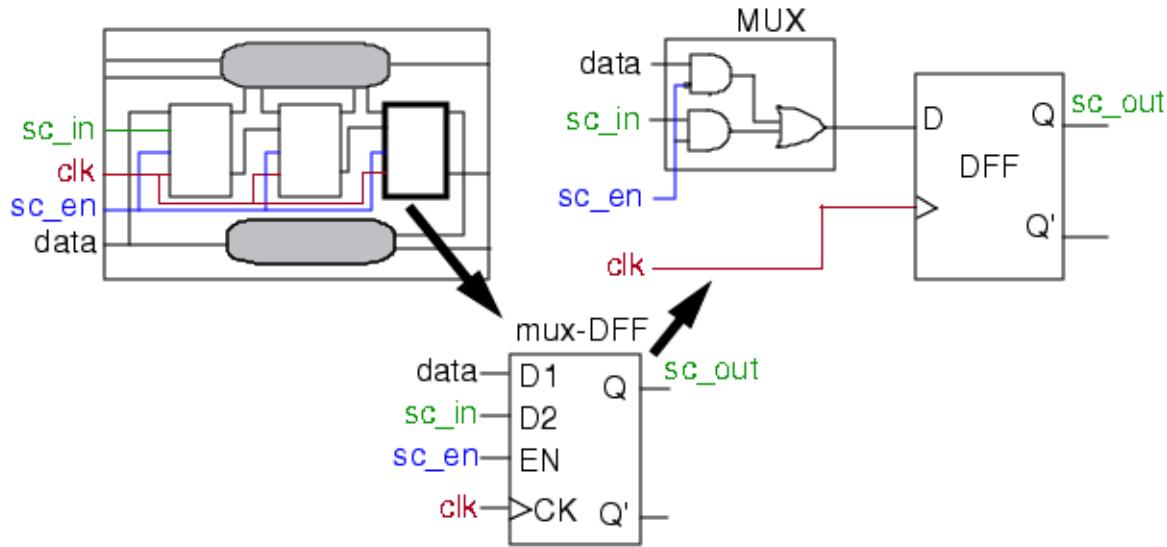
Figure 3-1. Generic Scan Cell



A scan cell contains at least one memory element (flip-flop or latch) that lies in the scan chain path. The cell can also contain additional memory elements that may or may not be in the scan chain path, as well as data inversion and gated logic between the memory elements.

[Figure 3-2](#) gives one example of a scan cell implementation (for the mux-DFF scan type).

Figure 3-2. Generic Mux-DFF Scan Cell Implementation



Each memory element may have a set and/or reset line in addition to clock-data ports. The ATPG process controls the scan cell by placing either normal or inverted data into its memory elements. The scan cell observation point is the memory element at the output of the scan cell. Other memory elements can also be observable, but may require a procedure for propagating their values to the scan cell's output. The following subsections describe the different memory elements a scan cell may contain.

Master Element

The master element (master cell), the primary memory element of a scan cell, captures data directly from the output of the previous scan cell. Each scan cell must contain one and only one master element.

For example, [Figure 3-2](#) shows a mux-DFF scan cell, which contains only a master element. However, scan cells can contain memory elements in addition to the master. Figures [3-3](#) through [3-6](#) illustrate examples of master elements in a variety of other scan cells.

The **shift** procedure in the test procedure file controls the master element. If the scan cell contains no additional independently-clocked memory elements in the scan path, this procedure also observes the master. If the scan cell contains additional memory elements, you may need to define a separate observation procedure (called **master_observe**) for propagating the master element's value to the output of the scan cell.

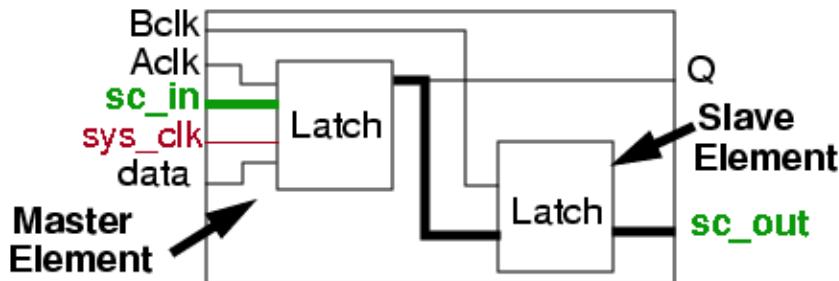
Slave Element

The slave element (slave cell), an independently-clocked scan cell memory element, resides in the scan chain path. It cannot capture data directly from the previous scan cell. When used, it

stores the output of the scan cell. The shift procedure both controls and observes the slave element. The value of the slave may be inverted relative to the master element.

Figure 3-3 shows a slave element within a scan cell.

Figure 3-3. LSSD Master/Slave Element Example



In the example of Figure 3-3, Aclk controls scan data input. Activating Aclk, with sys_clk (which controls system data) held off, shifts scan data into the scan cell. Activating Bclk propagates scan data to the output.

Shadow Element

The shadow element (shadow cell), either dependently- or independently-clocked, resides outside the scan chain path. It can be inside or outside of a scan cell.

Figure 3-4 gives an example of a scan cell with a dependently-clocked, non-observable shadow element with a non-inverted value.

Figure 3-4. Dependently-clocked Mux-DFF/Shadow Element Example

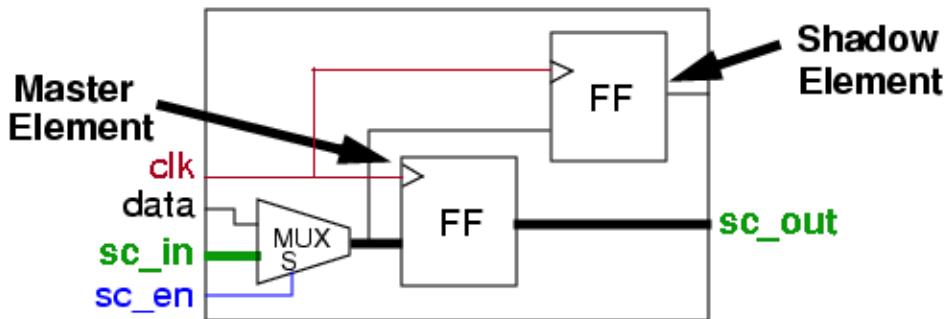
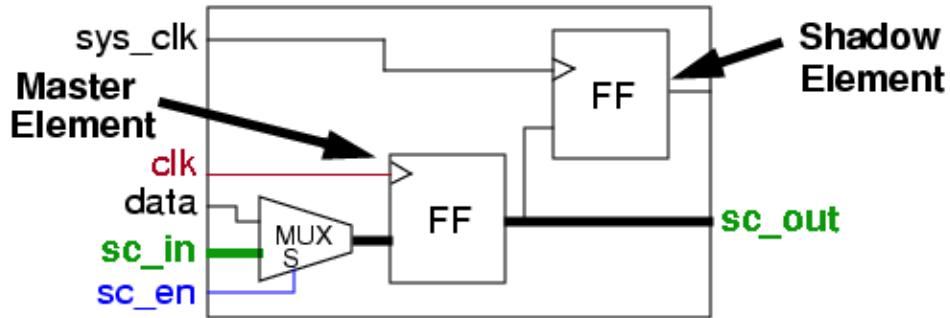


Figure 3-5 shows a similar example where the shadow element is independently-clocked.

Figure 3-5. Independently-clocked Mux-DFF/Shadow Element Example



You load a data value into the dependently-clocked shadow element with the **shift** procedure. If the shadow element is independently clocked, you use a separate procedure called **shadow_control** to load it. You can optionally make a shadow observable using the **shadow_observe** procedure. A scan cell may contain multiple shadows but only one may be observable, because the tools allow only one **shadow_observe** procedure. A shadow element's value may be the inverse of the master's value.

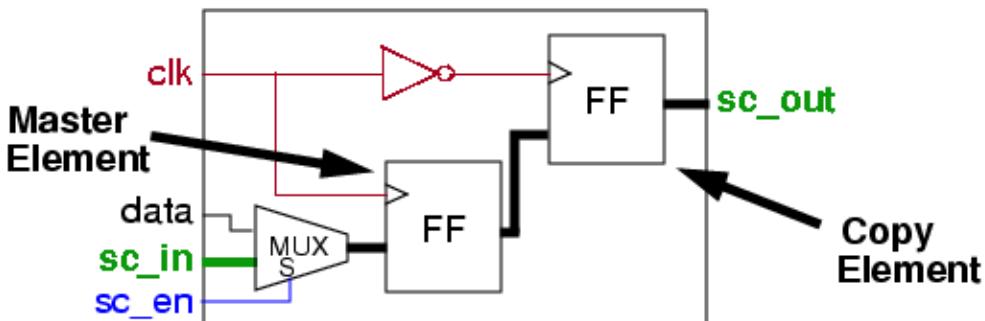
The definition of a shadow element is based on the shadow having the same (or inverse) value as the master element it shadows. A variety of interconnections of the master and shadow will accomplish this. In Figure 3-4, the shadow's data input is connected to the master's data input, and both FFs are triggered by the same clock edge. The definition would also be met if the shadow's data input was connected to the master's output and the shadow was triggered on the trailing edge, the master on the leading edge, of the same clock.

Copy Element

The copy element (copy cell) is a memory element that lies in the scan chain path and can contain the same (or inverted) data as the associated master or slave element in the scan cell.

Figure 3-6 gives an example of a copy element within a scan cell in which a master element provides data to the copy.

Figure 3-6. Mux-DFF/Copy Element Example



The clock pulse that captures data into the copy's associated scan cell element also captures data into the copy. Data transfers from the associated scan cell element to the copy element in the second half of the same clock cycle.

During the **shift** procedure, a copy contains the same data as that in its associated memory element. However, during system data capture, some types of scan cells allow copy elements to capture different data. When the copy's value differs from its associated element, the copy becomes the observation point of the scan cell. When the copy holds the same data as its associated element, the associated element becomes the observation point.

Extra Element

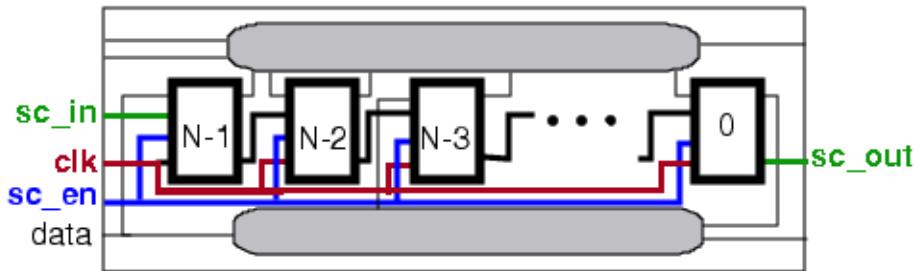
The extra element is an additional, independently-clocked memory element of a scan cell. An extra element is any element that lies in the scan chain path between the master and slave elements. The shift procedure controls data capture into the extra elements. These elements are not observable. Scan cells can contain multiple extras. Extras can contain inverted data with respect to the master element.

Scan Chains

A scan chain is a set of serially linked scan cells. Each scan chain contains an external input pin and an external output pin that provide access to the scan cells.

Figure 3-7 shows a scan chain, with scan input “sc_in” and scan output “sc_out”.

Figure 3-7. Generic Scan Chain



The scan chain length (N) is the number of scan cells within the scan chain. By convention, the scan cell closest to the external output pin is number 0, its predecessor is number 1, and so on. Because the numbering starts at 0, the number for the scan cell connected to the external input pin is equal to the scan chain length minus one ($N-1$). The minimum number of cells in a scan chain is one.

Scan Groups

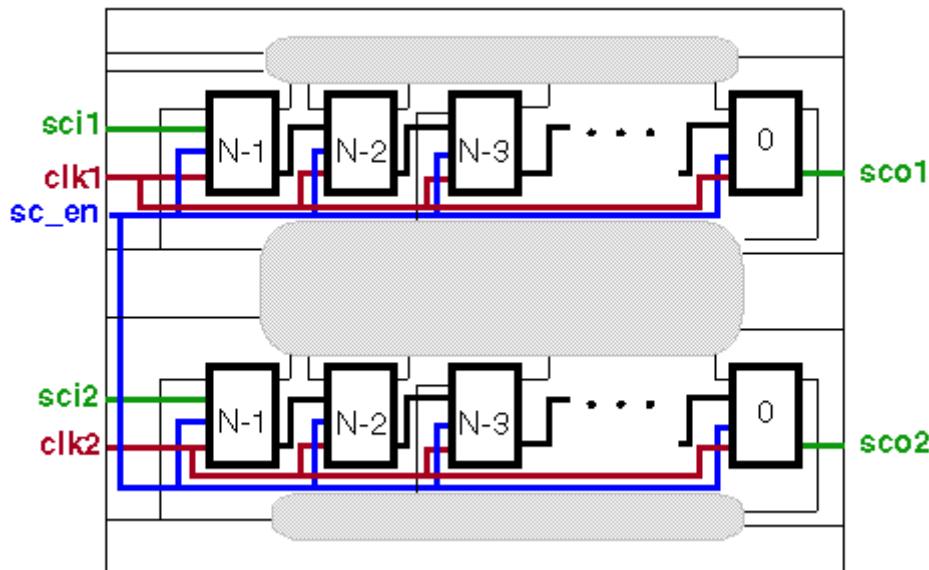
A scan chain group is a set of scan chains that operate in parallel and share a common test procedure file.

The test procedure file defines how to access the scan cells in all of the scan chains of the group. For more information on the test procedure file see the chapter “[Test Procedure File](#)” in the *Tessent Shell User’s Manual*.

Normally, all of a circuit’s scan chains operate in parallel and are thus in a single scan chain group.

You can have two clocks, clk1 and clk2, as shown in [Figure 3-8](#), each of which clocks different scan chains. You may also have a single clock to operate the scan chains. Regardless of operation, all defined scan chains in a circuit must be associated with a scan group. A scan group is a concept used by Tessent tools.

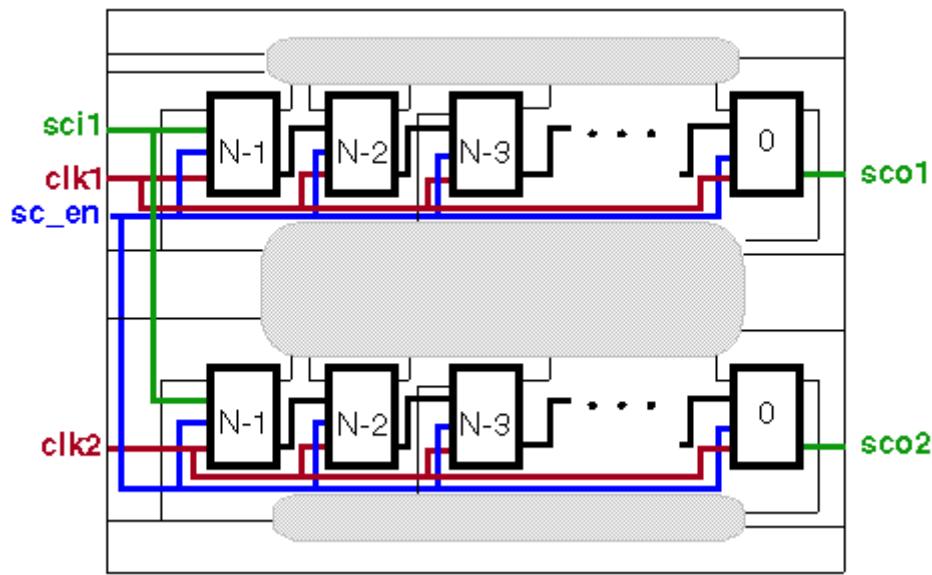
Figure 3-8. Scan Group - Two Scan Inputs



Scan groups are a way to group scan chains based on operation. All scan chains in a group must be able to operate in parallel, which is normal for scan chains in a circuit. However when scan chains cannot operate in parallel, the operation of each must be specified separately. This means the scan chains belong to different scan groups.

An example is shown in [Figure 3-9](#). In this case, the two scan chains share a single scan input, **sci1**.

Figure 3-9. Scan Group - One Scan Input



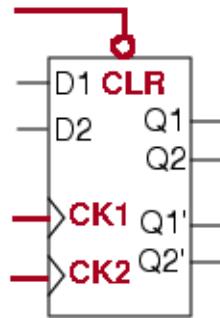
When these scan chains operate in parallel, their dependency on the same scan input will have an impact on test coverage. You can remedy this situation by defining another scan group and assigning one of the scan chains to it. Another reason to define multiple scan groups is when you want to perform sequential loads for low-power situations. Be aware that multiple scan groups are not allowed in EDT.

Scan Clocks

Scan clocks are external pins capable of capturing values into scan cell elements. Scan clocks include set and reset lines, as well as traditional clocks. Any pin defined as a clock can act as a capture clock during ATPG.

Figure 3-10 shows a scan cell whose scan clock signals are shown in bold.

Figure 3-10. Scan Clocks Example



In addition to capturing data into scan cells, scan clocks, in their off state, ensure that the cells hold their data. Design rule checks ensure that clocks perform both functions. A clock's *off-state* is the primary input value that results in a scan element's clock input being at its inactive

state (for latches) or state prior to a capturing transition (for edge-triggered devices). In the case of [Figure 3-10](#), the off-state for the CLR signal is 1, and the off-states for CK1 and CK2 are both 0.

Scan Architectures

Tessent Scan supports the insertion of mux-DFF (mux-scan) architecture. Additionally, Tessent Scan supports all standard scan types, or combinations thereof, in designs containing pre-existing scan circuitry.

Each scan style provides different benefits. Mux-DFF or clocked-scan are generally the best choice for designs with edge-triggered flip-flops.

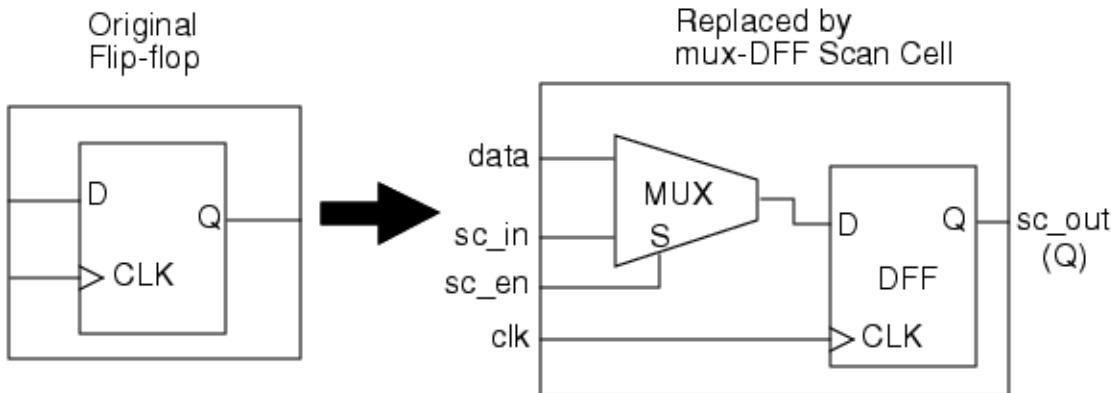
The following subsection details the mux-DFF architecture.

Mux-DFF

A mux-DFF cell contains a single D flip-flop with a multiplexed input line that allows selection of either normal system data or scan data.

[Figure 3-11](#) shows the replacement of an original design flip-flop with mux-DFF circuitry.

Figure 3-11. Mux-DFF Replacement



In normal operation ($sc_en = 0$), system data passes through the multiplexer to the D input of the flip-flop, and then to the output Q. In scan mode ($sc_en = 1$), scan input data (sc_in) passes to the flip-flop, and then to the scan output (sc_out).

Test Procedure Files

Test procedure files describe, for the ATPG tool, the scan circuitry operation within a design. Test procedure files contain cycle-based procedures and timing definitions that tell the ATPG tool how to operate the scan structures within a design.

In order to use the scan circuitry in your design, you must do the following:

- Define the scan circuitry for the tool.
- Create a test procedure file to describe the scan circuitry operation. Tessent Scan can create test procedure files for you.
- Perform DRC process. This occurs when you exit from setup mode.

Once the scan circuitry operation passes DRC, the ATPG tool processes assume the scan circuitry works properly.

If your design contains scan circuitry, the ATPG tool requires a test procedure file. You must create one before running ATPG.

For more information about the test procedure file format, see “[Test Procedure File](#)” in the *Tessent Shell User’s Manual*, which describes 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.

Model Flattening

To work properly, the ATPG tool and Tesson Scan must use their own internal representations of the design. The tools create these internal design models by flattening the model and replacing the design cells in the netlist (described in the library) with their own primitives. The tools flatten the model when you initially attempt to exit setup mode, just prior to design rules checking. The ATPG tool also provides the `create_flat_model` command, which allows flattening of the design model while still in setup mode.

If a flattened model already exists when you exit setup mode, the tools will only reflatten the model if you have since issued commands that would affect the internal representation of the design. For example, adding or deleting primary inputs, tying signals, and changing the internal faulting strategy are changes that affect the design model. With these types of changes, the tool must re-create or re-flatten the design model. If the model is undisturbed, the tool keeps the original flattened model and does not attempt to reflatten.

For a list of the specific Tesson Scan commands that cause flattening, refer to the [set_system_mode](#) description in the *Tesson Shell Reference Manual*.

Other commands that are useful for this task:

- [create_flat_model](#) — Creates a primitive gate simulation representation of the design.
- [report_flattener_rules](#) — Displays either a summary of all the flattening rule violations or the data for a specific violation.
- [set_flattener_rule_handling](#) — Specifies how the tool handles flattening violations.

Design Object Naming	98
The Flattening Process	99
Simulation Primitives of the Flattened Model	100

Design Object Naming

Tesson Scan and the ATPG tool use special terminology to describe different objects in the design hierarchy.

The following list describes the most common:

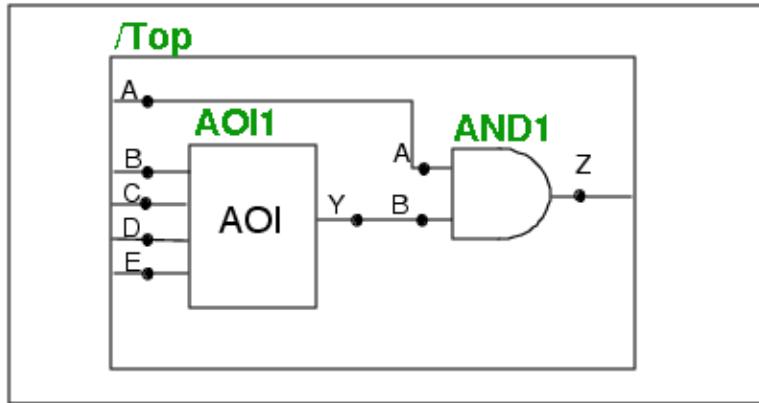
- Instance — A specific occurrence of a library model or functional block in the design.
- Hierarchical instance — An instance that contains additional instances and/or gates underneath it.
- *Module* — A Verilog functional block (module) that can be repeated multiple times. Each occurrence of the module is a hierarchical instance.

The Flattening Process

The flattened model contains only simulation primitives and connectivity, which makes it an optimal representation for the processes of fault simulation and ATPG.

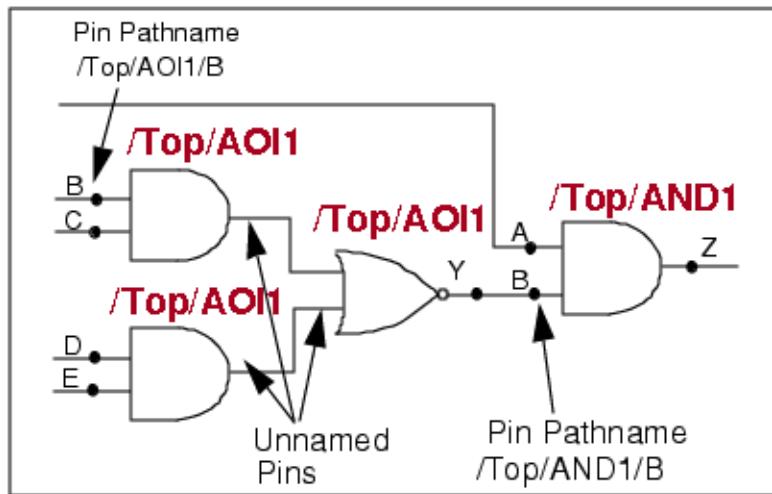
[Figure 3-12](#) shows an example of circuitry containing an AND-OR-Invert cell and an AND gate, before flattening.

Figure 3-12. Design Before Flattening



[Figure 3-13](#) shows this same design once it has been flattened.

Figure 3-13. Design After Flattening



After flattening, only naming preserves the design hierarchy; that is, the flattened netlist maintains the hierarchy through instance naming. Figures [3-12](#) and [3-13](#) show this hierarchy preservation. */Top* is the name of the hierarchy's top level. The simulation primitives (two AND gates and a NOR gate) represent the flattened instance *AOII* within */Top*. Each of these flattened gates retains the original design hierarchy in its naming—in this case, */Top/AOII*.

The tools identify pins from the original instances by hierarchical pathnames as well. For example, */Top/AOI1/B* in the flattened design specifies input pin B of instance AOI1. This naming distinguishes it from input pin B of instance AND1, which has the pathname */Top/AND1/B*. By default, pins introduced by the flattening process remain unnamed and are not valid fault sites. If you request gate reporting on one of the flattened gates, the NOR gate for example, you will see a system-defined pin name shown in quotes. If you want internal faulting in your library cells, you must specify internal pin names within the library model. The flattening process then retains these pin names.

You should be aware that in some cases, the design flattening process can appear to introduce new gates into the design. For example, flattening decomposes a DFF gate into a DFF simulation primitive, the Q and Q' outputs require buffer and inverter gates, respectively. If your design wires together multiple drivers, flattening would add wire gates or bus gates. Bidirectional pins are another special case that requires additional gates in the flattened representation.

Simulation Primitives of the Flattened Model

Tessent Scan and the ATPG tool select from a number of simulation primitives when they create the flattened circuitry. The simulation primitives are multiple-input (zero to four), single-output gates, except for the RAM, ROM, LA, and DFF primitives.

The following list describes these simulation primitives:

- **PI, PO** — Primary inputs are gates with no inputs and a single output, while primary outputs are gates with a single input and no fanout.
- **BUF** — A single-input gate that passes the values 0, 1, or X through to the output.
- **FB_BUF** — A single-input gate, similar to the BUF gate, that provides a one iteration delay in the data evaluation phase of a simulation. The tools use the FB_BUF gate to break some combinational loops and provide more optimistic behavior than when TIEX gates are used.

Note

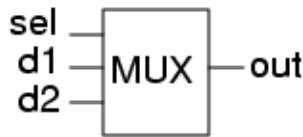
 There can be one or more loops in a feedback path. In analysis mode, you can display the loops with the [report_loops](#) command. In setup mode, use [report_feedback_paths](#).

The default loop handling is simulation-based, with the tools using the FB_BUF to break the combinational loops. In setup mode, you can change the default with the [set_loop_handling](#) command. Be aware that changes to loop handling will have an impact during the flattening process.

- **ZVAL** — A single-input gate that acts as a buffer unless Z is the input value. When a Z is the input value, the output is an X. You can modify this behavior with the [set_z_handling](#) command.

- **INV**— A single-input gate whose output value is the opposite of the input value. The INV gate cannot accept a Z input value.
- **AND, NAND**— Multiple-input gates (two to four) that act as standard AND and NAND gates.
- **OR, NOR**— Multiple-input (two to four) gates that act as standard OR and NOR gates.
- **XOR, XNOR**— 2-input gates that act as XOR and XNOR gates, except that when either input is an X, the output is an X.
- **MUX** — A 2x1 mux gate whose pins are order dependent, as shown in [Figure 3-14](#).

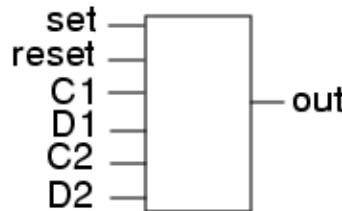
Figure 3-14. 2x1 MUX Example



The sel input is the first defined pin, followed by the first data input and then the second data input. When sel=0, the output is d1. When sel=1, the output is d2.

- **LA, DFF** — State elements, whose order dependent inputs include set, reset, and clock/data pairs, as shown in [Figure 3-15](#).

Figure 3-15. LA, DFF Example

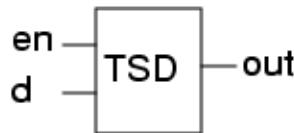


Set and reset lines are always level sensitive, active high signals. DFF clock ports are edge-triggered while LA clock ports are level sensitive. When set=1, out=1. When reset=1, out=0. When a clock is active (for example C1=1), the output reflects its associated data line value (D1). If multiple clocks are active and the data they are trying to place on the output differs, the output becomes an X.

- **TLA, STLA, STFF**— Special types of learned gates that act as, and pass the design rule checks for, transparent latch, sequential transparent latch, or sequential transparent flip-flop. These gates propagate values without holding state.
- **TIE0, TIE1, TIEX, TIEZ**— Zero-input, single-output gates that represent the effect of a signal tied to ground or power, or a pin or state element constrained to a specific value (0,1,X, or Z). The rules checker may also determine that state elements exhibit tied behavior and replace them with the appropriate tie gates.

- **TSD, TSH** — A 2-input gate that acts as a tri-state™ driver, as shown in [Figure 3-16](#).

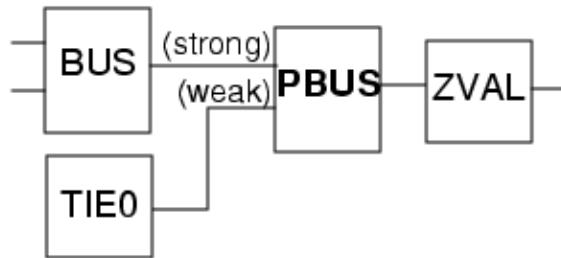
Figure 3-16. TSD, TSH Example



When $\text{en}=1$, $\text{out}=\text{d}$. When $\text{en}=0$, $\text{out}=Z$. The data line, d , cannot be a Z . The ATPG tool uses the TSD gate for the same purpose.

- **SW, NMOS** — A 2-input gate that acts like a tri-state driver but can also propagate a Z from input to output. The ATPG tool uses the SW gate uses the NMOS gate for the same purpose.
- **BUS** — A multiple-input (up to four) gate whose drivers must include at least one TSD or SW gate. If you bus more than four tri-state drivers together, the tool creates cascaded BUS gates. The last bus gate in the cascade is considered the dominant bus gate.
- **WIRE** — a multiple-input gate that differs from a bus in that none of its drivers are tri-statable.
- **PBUS, SWBUS** — A 2-input pull bus gate, for use when you combine strong bus and weak bus signals together, as shown in [Figure 3-17](#).

Figure 3-17. PBUS, SWBUS Example



The strong value always goes to the output, unless the value is a Z , in which case the weak value propagates to the output. These gates model pull-up and pull-down resistors. The ATPG tool uses the PBUS gate.

- **ZHOLD** — A single-input buskeeper gate (see [page 111](#) for more information on buskeepers) associated with a tri-state network that exhibits sequential behavior. If the input is a binary value, the gate acts as a buffer. If the input value is a Z , the output depends on the gate's hold capability. There are three ZHOLD gate types, each with a different hold capability:
 - ZHOLD0 - When the input is a Z , the output is a 0 if its previous state was 0. If its previous state was a 1, the output is a Z .

- ZHOLD1 - When the input is a Z, the output is a 1 if its previous state was a 1. If its previous state was a 0, the output is a Z.
- ZHOLD0,1 - When the input is a Z, the output is a 0 if its previous state was a 0, or the output is a 1 if its previous state was a 1.

In all three cases, if the previous value is unknown, the output is X.

- **RAM, ROM**— Multiple-input gates that model the effects of RAM and ROM in the circuit. RAM and ROM differ from other gates in that they have multiple outputs.
- **OUT**— Gates that convert the outputs of multiple output gates (such as RAM and ROM simulation gates) to a single output.

Learning Analysis

After design flattening, the ATPG tool performs extensive analysis on the design to learn behavior that may be useful for intelligent decision making in later processes, such as fault simulation and ATPG. You have the ability to turn learning analysis off, which may be desirable if you do not want to perform ATPG during the session.

For more information on turning learning analysis off, refer to the [set_static_learning](#) description in the *Tessent Shell Reference Manual*.

The ATPG tools perform static learning only once—after flattening. Because pin and ATPG constraints can change the behavior of the design, static learning does not consider these constraints. Static learning involves gate-by-gate local simulation to determine information about the design. The following subsections describe the types of analysis performed during static learning.

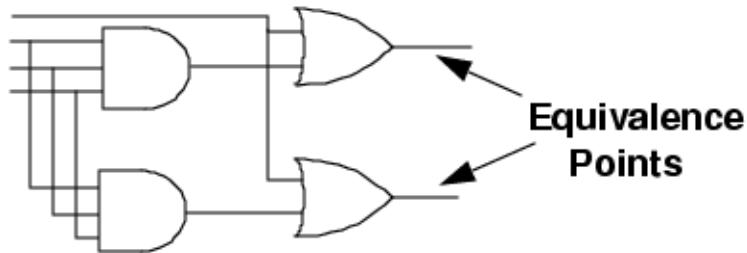
Equivalence Relationships	104
Logic Behavior	104
Implied Relationships	105
Forbidden Relationships	106
Dominance Relationships	106

Equivalence Relationships

During this analysis, simulation traces back from the inputs of a multiple-input gate through a limited number of gates to identify points in the circuit that always have the same values in the good machine.

[Figure 3-18](#) shows an example of two of these equivalence points within some circuitry.

Figure 3-18. Equivalence Relationship Example

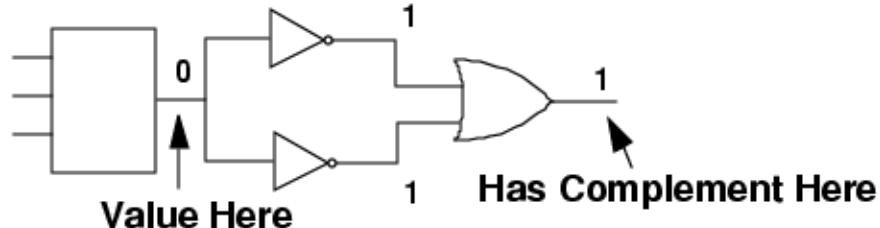


Logic Behavior

During logic behavior analysis, simulation determines a circuit's functional behavior.

For example, [Figure 3-19](#) shows some circuitry that, according to the analysis, acts as an inverter.

Figure 3-19. Example of Learned Logic Behavior



During gate function learning, the tool identifies the circuitry that acts as gate types TIE (tied 0, 1, or X values), BUF (buffer), INV (inverter), XOR (2-input exclusive OR), MUX (single select line, 2-data-line MUX gate), AND (2-input AND), and OR (2-input OR). For AND and OR function checking, the tool checks for busses acting as 2-input AND or OR gates. The tool then reports the learned logic gate function information with the messages:

```
Learned gate functions: #<gatetype>=<number> ...
Learned tied gates:      #<gatetype>=<number> ...
```

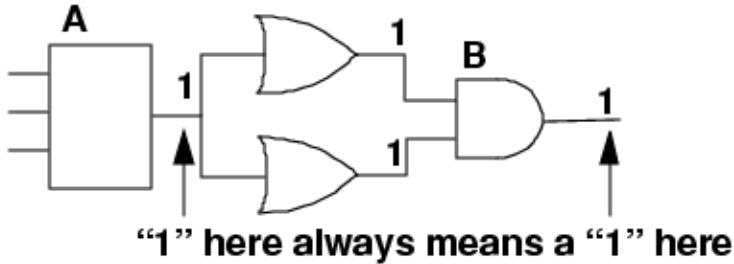
If the analysis process yields no information for a particular category, it does not issue the corresponding message.

Implied Relationships

This type of analysis consists of contrapositive relation learning, or learning implications, to determine that one value implies another. This learning analysis simulates nearly every gate in the design, attempting to learn every relationship possible.

[Figure 3-20](#) shows the implied learning the analysis derives from a piece of circuitry.

Figure 3-20. Example of Implied Relationship Learning



The analysis process can derive a very powerful relationship from this circuitry. If the value of gate A=1 implies that the value of gate B=1, then B=0 implies A=0. This type of learning establishes circuit dependencies due to reconvergent fanout and buses, which are the main

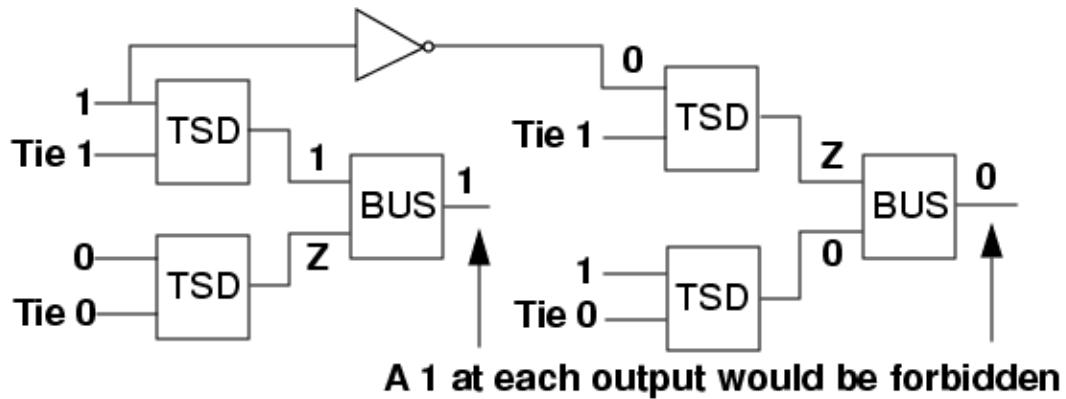
obstacles for ATPG. Thus, implied relationship learning significantly reduces the number of bad ATPG decisions.

Forbidden Relationships

During forbidden relationship analysis, which is restricted to bus gates, simulation determines that one gate cannot be at a certain value if another gate is at a certain value.

Figure 3-21 shows an example of such behavior.

Figure 3-21. Forbidden Relationship Example

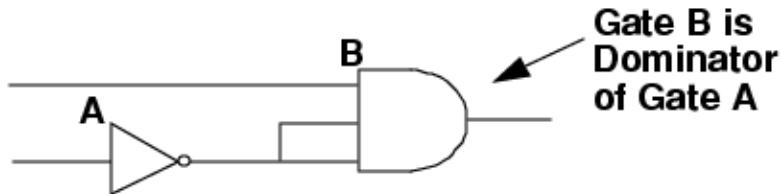


Dominance Relationships

During dominance relationship analysis, simulation determines which gates are dominators. If all the fanouts of a gate go to a second gate, the second gate is the dominator of the first.

Figure 3-22 shows an example of this relationship.

Figure 3-22. Dominance Relationship Example



ATPG Design Rules Checking

Tessent Scan and the ATPG tool perform design rules checking (DRC) after design flattening.

While not all of the tools perform the exact same checks, design rules checking generally consists of the following processes, done in the order shown:

General Rules Checking	107
Procedure Rules Checking	107
Bus Mutual Exclusivity Analysis	108
Scan Chain Tracing.....	109
Shadow Latch Identification.....	109
Data Rules Checking	110
Transparent Latch Identification.....	110
Clock Rules Checking	111
RAM Rules Checking	111
Bus Keeper Analysis	111
Extra Rules Checking	112
Scannability Rules Checking	112
Constrained/Forbidden/Block Value Calculations.....	112

General Rules Checking

General rules checking searches for very-high-level problems in the information defined for the design. For example, it checks to ensure the scan circuitry, clock, and RAM definitions all make sense. General rules violations are errors and you cannot change their handling.

The “[General Rules](#)” section in the *Tessent Shell Reference Manual* describes the general rules in detail.

Procedure Rules Checking

Procedure rules checking examines the test procedure file. These checks look for parsing or syntax errors and ensure adherence to each procedure’s rules. Procedure rules violations are errors and you cannot change their handling.

The “[Procedure Rules](#)” section in the *Tessent Shell Reference Manual* describes the procedure rules in detail.

Bus Mutual Exclusivity Analysis

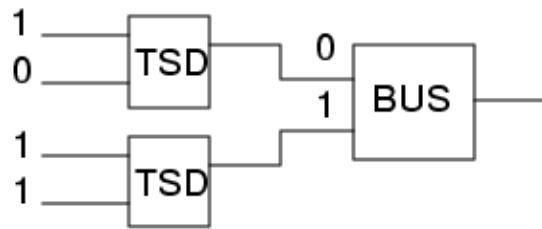
Buses in circuitry can cause problems for ATPG. For example, bus contention during ATPG and stuck-at faults on tri-state bus drivers.

- Bus contention during ATPG
- Testing stuck-at faults on tri-state drivers of buses.

This section addresses the first concern, that ATPG must place buses in a non-contending state. For information on how to handle testing of tri-state devices, see “[Tri-State Devices](#)” on page 127.

[Figure 3-23](#) shows a bus system that can have contention.

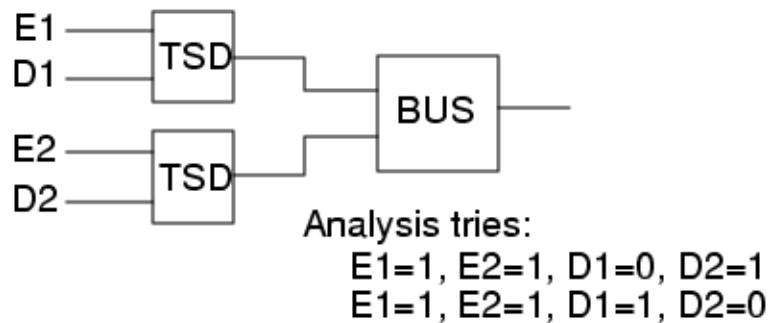
Figure 3-23. Bus Contention Example



Many designs contain buses, but good design practices usually prevent bus contention. As a check, the learning analysis for buses determines if a contention condition can occur within the given circuitry. Once learning determines that contention cannot occur, none of the later processes, such as ATPG, ever check for the condition.

Buses in a Z-state network can be classified as dominant or non-dominant and strong or weak. Weak buses and pull buses are allowed to have contention. Thus the process only analyzes strong, dominant buses, examining all drivers of these gates and performing full ATPG analysis of all combinations of two drivers being forced to opposite values. [Figure 3-24](#) demonstrates this process on a simple bus system.

Figure 3-24. Bus Contention Analysis



If ATPG analysis determines that either of the two conditions shown can be met, the bus fails bus mutual-exclusivity checking. Likewise, if the analysis proves the condition is never possible, the bus passes these checks. A third possibility is that the analysis aborts before it completes trying all of the possibilities. In this circuit, there are only two drivers, so ATPG analysis need try only two combinations. However, as the number of drivers increases, the ATPG analysis effort grows significantly.

You should resolve bus mutual-exclusivity before ATPG. Extra rules E4, E7, E9, E10, E11, E12, and E13 perform bus analysis and contention checking. Refer to “[Extra Rules](#)” in the *Tessent Shell Reference Manual* for more information on these bus checking rules.

Scan Chain Tracing

The purpose of scan chain tracing is for the tool to identify the scan cells in the chain and determine how to use them for control and observe points.

Using the information from the test procedure file (which has already been checked for general errors during the procedure rules checks) and the defined scan data, the tool identifies the scan cells in each defined chain and simulates the operation specified by the **load_unload** procedure to ensure proper operation. Scan chain tracing takes place during the trace rules checks, which trace back through the sensitized path from output to input. Successful scan chain tracing ensures that the tools can use the cells in the chain as control and observe points during ATPG.

Trace rules violations are either errors or warnings, and for most rules you cannot change the handling. The “[Scan Chain Trace Rules](#)” section in the *Tessent Shell Reference Manual* describes the trace rules in detail.

Shadow Latch Identification

Shadows are state elements that contain the same data as an associated scan cell element, but do not lie in the scan chain path. So while these elements are technically non-scan elements, their identification facilitates the ATPG process. This is because if a shadow element’s content is the same as the associated element’s content, you always know the shadow’s state at that point. Thus, a shadow can be used as a control point in the circuit.

If the circuitry allows, you can also make a shadow an observation point by writing a **shadow_observe** test procedure. The section entitled “[Shadow Element](#)” on page 91 discusses shadows in more detail.

The DRC process identifies shadow latches under the following conditions:

1. The element must not be part of an already identified scan cell.

2. Plus any one of the following:
 - At the time the clock to the shadow latch is active, there must be a single sensitized path from the data input of the shadow latch up to the output of a scan latch. Additionally the final shift pulse must occur at the scan latch no later than the clock pulse to the shadow latch (strictly before, if the shadow is edge triggered).
 - The shadow latch is loaded before the final shift pulse to the scan latch is identified by tracing back the data input of the shadow latch. In this case, the shadow will be a shadow of the next scan cell closer to scan out than the scan cell identified by tracing. If there is no scan cell close to scan out, then the sequential element is not a valid shadow.
 - The shadow latch is sensitized to a scan chain input pin during the last shift cycle. In this case, the shadow latch will be a shadow of the scan cell closest to scan in.

Data Rules Checking

Data rules checking ensures the proper transfer of data within the scan chain. Data rules violations are either errors or warnings, however, you can change the handling.

The “[Scan Cell Data Rules](#)” section in the *Tessent Shell Reference Manual* describes the data rules in detail.

Transparent Latch Identification

Transparent latches need to be identified because they can propagate values but do not hold state.

A basic scan pattern contains the following events.

1. Load scan chain
2. Force values on primary inputs

 **Note**

Latch must behave as transparent between events 2 and 3.

3. Measure values on primary outputs
4. Pulse the capture clock
5. Unload the scan chain

Between the PI force and PO measure, the tool constrains all pins and sets all clocks off. Thus, for a latch to qualify as transparent, the analysis must determine that it can be turned on when clocks are off and pins are constrained. TLA simulation gates, which rank as combinational, represent transparent latches.

Clock Rules Checking

After the scan chain trace, clock rules checking is the next most important analysis. Clock rules checks ensure data stability and capturability in the chain. Clock rules violations are either errors or warnings, however, you can change the handling.

The “[Clock Rules](#)” section in the *Tessent Shell Reference Manual* describes the clock rules in detail.

RAM Rules Checking

RAM rules checking ensures consistency with the defined RAM information and the chosen testing mode. RAM rules violations are all warnings, however, you can change their handling.

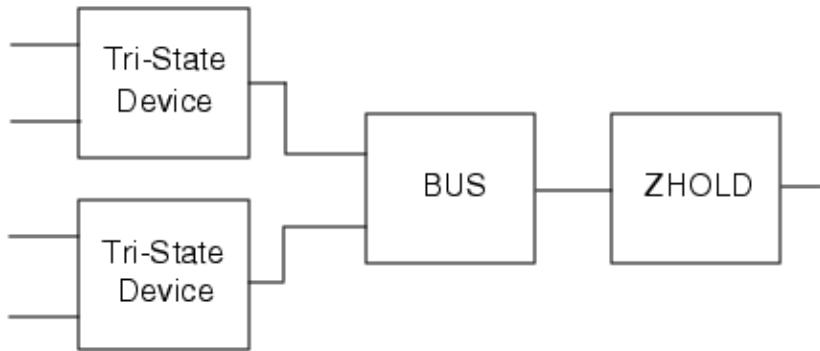
The “[RAM Rules](#)” section in the *Tessent Shell Reference Manual* describes the RAM rules in detail.

Bus Keeper Analysis

Bus keepers model the ability of an undriven bus to retain its previous binary state. You specify bus keeper modeling with a **bus_keeper** attribute in the model definition. When you use the **bus_keeper** attribute, the tool uses a ZHOLD gate to model the bus keeper behavior during design flattening.

In this situation, the design’s simulation model becomes that shown in [Figure 3-25](#).

Figure 3-25. Simulation Model with Bus Keeper



Rules checking determines the values of ZHOLD gates when clocks are off, pin constraints are set, and the gates are connected to clock, write, and read lines. ZHOLD gates connected to clock, write, and read lines do not retain values unless the clock off-states and constrained pins result in binary values.

During rules checking, if a design contains ZHOLD gates, messages indicate when ZHOLD checking begins, the number and type of ZHOLD gates, the number of ZHOLD gates connected

to clock, write, and read lines, and the number of ZHOLD gates set to a binary value during the clock off-state condition.

Note

 Only the ATPG tool requires this type of analysis, because of the way it “flattens” or simulates a number of events in a single operation.

For information on the bus_keeper model attribute, refer to “[Attributes](#)” in the *Tessent Cell Library Manual*.

Extra Rules Checking

Excluding rule E10, which performs bus mutual-exclusivity checking, most extra rules checks do not have an impact on Tessent Scan and the ATPG tool processes. However, they may be useful for enforcing certain design rules. By default, most extra rules violations are set to ignore, which means they are not even checked during DRC. However, you may change the handling.

For more information, refer to “[Extra Rules](#)” in the *Tessent Shell Reference Manual* for more information.

Scannability Rules Checking

Each design contains a certain number of memory elements. Tessent Scan examines all these elements and performs scannability checking on them, which consists mainly of the audits performed by rules S1, S2, S3, and S4. Scannability rules are all warnings, and you cannot change their handling, except for the S3 rule.

For more information, refer to “[Scannability Rules \(S Rules\)](#)” in the *Tessent Shell Reference Manual*.

Constrained/Forbidden/Block Value Calculations

This analysis determines constrained, forbidden, and blocked circuitry. The checking process simulates forward from the point of the constrained, forbidden, or blocked circuitry to determine its effects on other circuitry. This information facilitates downstream processes, such as ATPG.

[Figure 3-26](#) gives an example of a tie value gate that constrains some surrounding circuitry.

Figure 3-26. Constrained Values in Circuitry

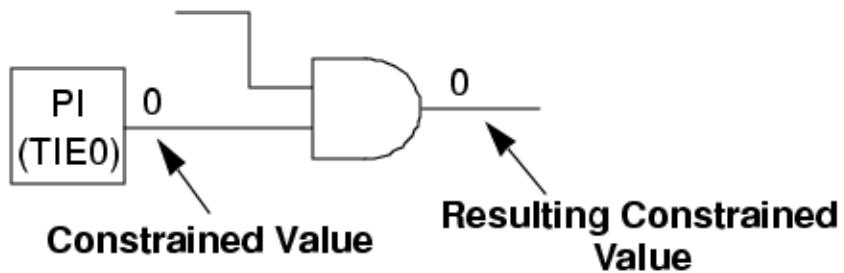


Figure 3-27 gives an example of a tied gate, and the resulting forbidden values of the surrounding circuitry.

Figure 3-27. Forbidden Values in Circuitry

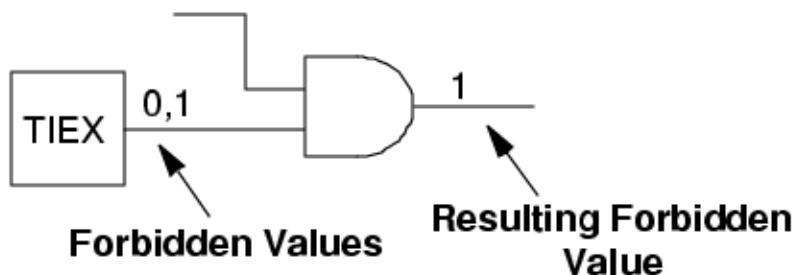
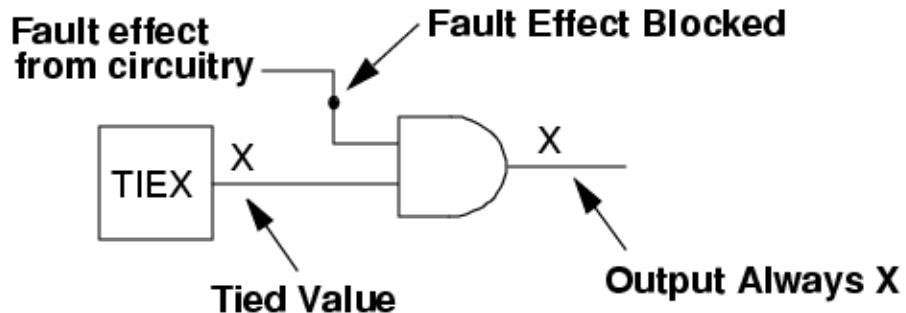


Figure 3-28 gives an example of a tied gate that blocks fault effects in the surrounding circuitry.

Figure 3-28. Blocked Values in Circuitry



Clock Terminology

The following text discusses clock-related concepts.

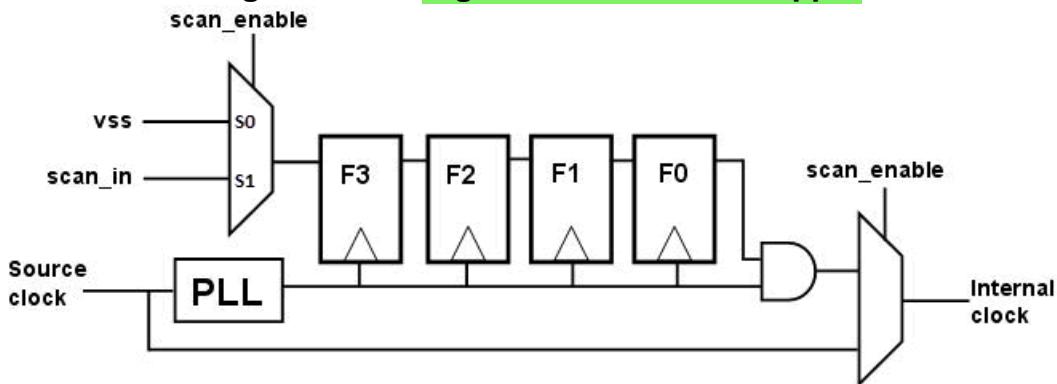
Programmable Clock Chopper 114

Programmable Clock Chopper

A programmable clock chopper is a type of clock controller that shapes the outgoing clock signal by allowing or suppressing (chopping) specific clock pulses.

The programmable OCC (On-Chip Controller) in [Figure 3-29](#) is a simple example of a clock chopper. You can program the OCC to generate up to four clock pulses based on an input clock, and you can also chop out up to four pulses from the output. You would typically program a clock controller during scan chain loading by loading specific control data into a shift register that controls a clock gating cell on the output of the OCC. [Figure 3-29](#) shows an example of this, where each bit in the shift register controls the clock in each of the four cycles.

Figure 3-29. Programmable Clock Chopper



Chapter 4

Testability Issues

Testability naturally varies from design to design. Some features and design styles make a design difficult, if not impossible, to test, while others enhance the testability of a design.

The following subsections discuss these design features and describe their effect on the testability of a design:

Synchronous Circuitry	115
Asynchronous Circuitry	116
Scannability Checking.....	116
Support for Special Testability Cases	118
Feedback Loops	118
Structural Combinational Loops and Loop-Cutting Methods	118
Structural Sequential Loops and Handling	124
Redundant Logic	125
Asynchronous Sets and Resets	125
Gated Clocks	126
Tri-State Devices	127
Non-Scan Cell Handling	127
Clock Dividers	131
Pulse Generators	132
JTAG-Based Circuits	133
RAM and ROM Test Overview	134
Incomplete Designs.....	140

Synchronous Circuitry

Using synchronous design practices, you can help ensure that your design will be both testable and manufacturable. In the past, designers used asynchronous design techniques with TTL and small PAL-based circuits. Today, however, designers can no longer use those techniques because the organization of most gate arrays and FPGAs necessitates the use of synchronous logic in their design.

A synchronous circuit operates properly and predictably in all modes of operation, from static DC up to the maximum clock rate. Inputs to the circuit do not cause the circuit to assume unknown states. And regardless of the relationship between the clock and input signals, the circuit avoids improper operation.

Truly synchronous designs are inherently testable designs. You can implement many scan strategies, and run the ATPG process with greater success, if you use synchronous design techniques. Moreover, you can create most designs following these practices with no loss of speed or functionality.

Synchronous Design Techniques

Your design's level of synchronicity depends on how closely you observe the following techniques:

- The system has a minimum number of clocks—optimally only one.
- You register all design inputs and account for metastability. That is, you should treat the metastability time as another delay in the path. If the propagation delay plus the metastability time is less than the clock period, the system is synchronous. If it is greater than or equal to the clock period, you need to add an extra flip-flop to ensure the proper data enters the circuit.
- No combinational logic drives the set, reset, or clock inputs of the flip-flops.
- No asynchronous signals set or reset the flip-flops.
- Buffers or other delay elements do not delay clock signals.
- Do not use logic to delay signals.
- Do not assume logic delays are longer than routing delays.

If you adhere to these design rules, you are much more likely to produce a design that is manufacturable, testable, and operates properly over a wide range of temperature, voltage, and other circuit parameters.

Asynchronous Circuitry

A small percentage of designs need some asynchronous circuitry due to the nature of the system. Because asynchronous circuitry is often very difficult to test, you should place the asynchronous portions of your design in one block and isolate it from the rest of the circuitry. In this way, you can still utilize DFT techniques on the synchronous portions of your design.

Scannability Checking

Tessent Scan performs the scannability checking process on a design's sequential elements. For the tool to insert scan circuitry into a design, it must replace existing sequential elements with their scannable equivalents. Before beginning substitution, the original sequential elements in the design must pass *scannability checks*; that is, the tool determines if it can convert sequential elements to scan elements without additional circuit modifications.

Scannable sequential elements pass the following checks:

1. When all clocks are off, all clock inputs (including set and reset inputs) of the sequential element must be in their inactive state (initial state of a capturing transition). This prevents disturbance of the scan chain data before application of the test pattern at the primary input. If the sequential element does not pass this check, its scan values could become unstable when the test tool applies primary input values. This checking is a modification of rule C1. For more information on this rule, refer to “[C1](#)” in the *Tessent Shell Reference Manual*.
2. Each clock input (not including set and reset inputs) of the sequential element must be capable of capturing data when a single clock primary input goes active while all other clocks are inactive. This rule ensures that this particular storage element can capture system data. If the sequential element does not meet this rule, some loss of test coverage could result. This checking is a modification of rule C7. For more information on this rule, refer to “[C7](#)” in the *Tessent Shell Reference Manual*.

When a sequential element passes these checks, it becomes a *scan candidate*, meaning that Tessent Scan can insert its scan equivalent into the scan chain. However, even if the element fails to pass one of these checks, it may still be possible to convert the element to scan. In many cases, you can add additional logic, called *test logic*, to the design to remedy the situation. For more information on test logic, refer to “[Test Logic Insertion](#)” on page 160.

Note

 If TIE0 and TIE1 nonscan cells are scannable, they are considered for scan. However, if these cells are used to hold off sets and resets of other cells so that another cell can be scannable, you must use the [add_nonscan_instances](#) command to make them nonscan.

Scannability Checking of Latches

By default, Tessent Scan performs scannability checking on all flip-flops and latches.

When latches do not pass scannability checks, Tessent Scan considers them non-scan elements and then classifies them into one of the categories explained in “[Non-Scan Cell Handling](#)” on page 127. However, if you want Tessent Scan to perform transparency checking on the non-scan latches, you must turn off checking of rule D6 prior to scannability checking. For more information on this rule, refer to “[D6](#)” in the *Tessent Shell Reference Manual*.

Support for Special Testability Cases

Certain design features can pose design testability problems. Mentor Graphics DFT tools can handle special testability cases.

Each topic in this section describes how the Mentor Graphics DFT tools supports design testability cases:

Feedback Loops	118
Structural Combinational Loops and Loop-Cutting Methods	118
Structural Sequential Loops and Handling.....	124
Redundant Logic	125
Asynchronous Sets and Resets	125
Gated Clocks	126
Tri-State Devices	127
Non-Scan Cell Handling	127
Clock Dividers	131
Pulse Generators	132
JTAG-Based Circuits	133
RAM and ROM Test Overview	134
Incomplete Designs	140

Feedback Loops

Designs containing loop circuitry have inherent testability problems. A *structural loop* exists when a design contains a portion of circuitry whose output, in some manner, feeds back to one of its inputs. A *structural combinational loop* occurs when the feedback loop, the path from the output back to the input, passes through only combinational logic. A *structural sequential loop* occurs when the feedback path passes through one or more sequential elements.

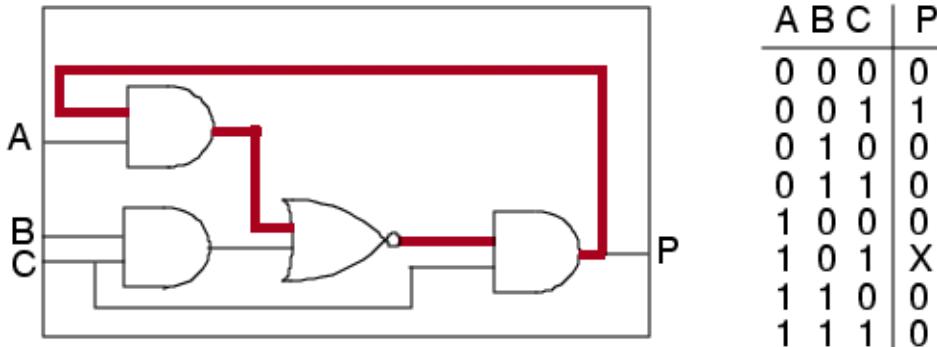
The ATPG tool and Tesson Scan all provide some common loop analysis and handling. However, loop treatment can vary depending on the tool. The following subsections discuss the treatment of structural combinational and structural sequential loops.

Structural Combinational Loops and Loop-Cutting Methods

Structural combinational loops are hardwired feedback paths in combinational circuits that make that circuit difficult to test.

[Figure 4-1](#) shows an example of a structural combinational loop. Notice that the A=1, B=0, C=1 state causes unknown (oscillatory) behavior, which poses a testability problem.

Figure 4-1. Structural Combinational Loop Example



The flattening process, which each tool runs as it attempts to exit setup mode, identifies and cuts, or breaks, all structural combinational loops. The tools classify and cut each loop using the appropriate methods for each category.

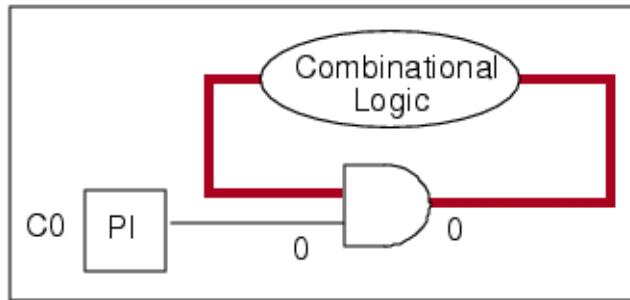
The following list presents the loop classifications, as well as the loop-cutting methods established for each. The order of the categories presented indicates the least to most pessimistic loop cutting solutions.

1. Constant value

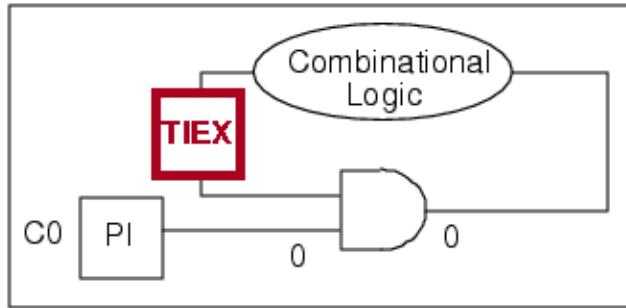
This loop cutting method involves those loops blocked by tied logic or pin constraints. After the initial loop identification, the tools simulate TIE0/TIE1 gates and constrained inputs. Loops containing constant value gates as a result of this simulation, fall into this category.

[Figure 4-2](#) shows a loop with a constrained primary input value that blocks the loop's feedback effects.

Figure 4-2. Loop Naturally-Blocked by Constant Value



These types of loops lend themselves to the simplest and least pessimistic breaking procedures. For this class of loops, the tool inserts a TIE-X gate at a non-constrained input (which lies in the feedback path) of the constant value gate, as [Figure 4-3](#) shows.

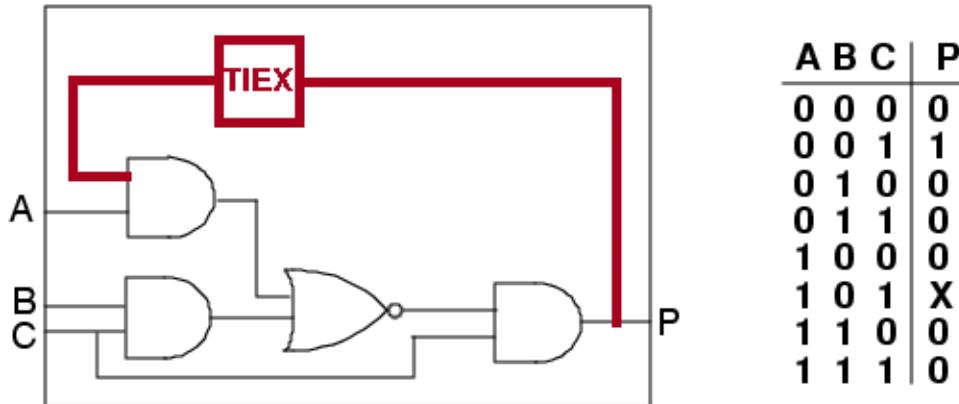
Figure 4-3. Cutting Constant Value Loops

This loop cutting technique yields good circuit simulation that always matches the actual circuit behavior, and thus, the tools employ this technique whenever possible. The tools can use this loop cutting method for blocked loops containing AND, OR, NAND, and NOR gates, as well as MUX gates with constrained select lines and tri-state drivers with constrained enable lines.

2. Single gate with “multiple fanout”

This loop cutting method involves loops containing only a single gate with multiple fanout.

[Figure 4-1](#) on page 119 shows the circuitry and truth table for a single multiple-fanout loop. For this class of loops, the tool cuts the loop by inserting a TIE-X gate at one of the fanouts of this “multiple fanout gate” that lie in the loop path, as [Figure 4-4](#) shows.

Figure 4-4. Cutting Single Multiple-Fanout Loops

3. Gate duplication for multiple gate with multiple fanout

This method involves duplicating some of the loop logic—when it proves practical to do so. The tools use this method when it can reduce the simulation pessimism caused by breaking combinational loops with TIE-X gates. The process analyzes a loop, picks a connection point, duplicates the logic (inserting a TIE-X gate into the copy), and connects the original circuitry to the copy at the connection point.

Figure 4-5 shows a simple loop that the tools would target for gate duplication.

Figure 4-5. Loop Candidate for Duplication

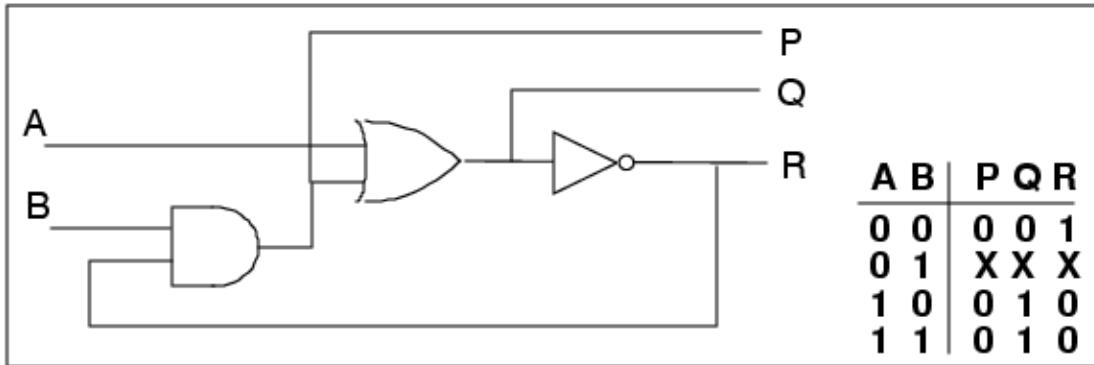
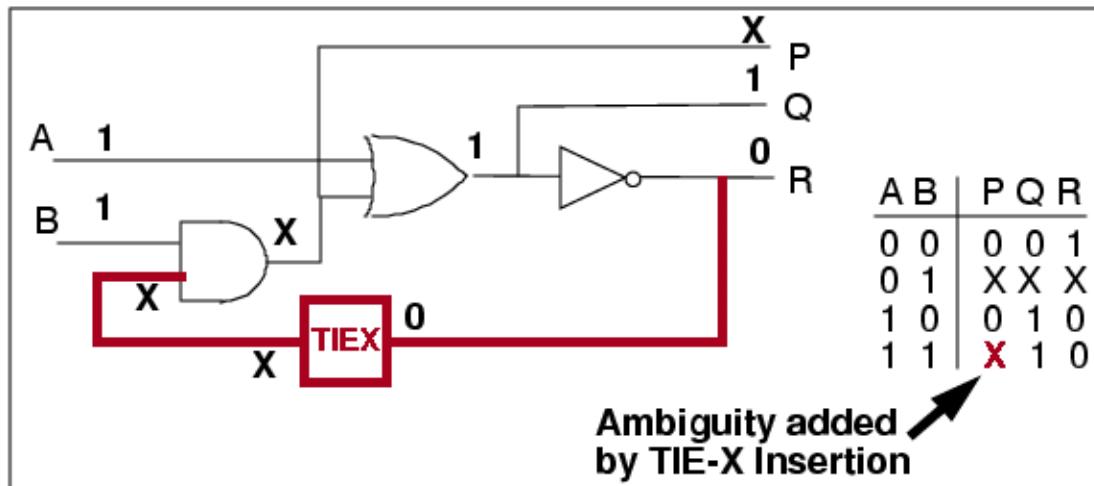


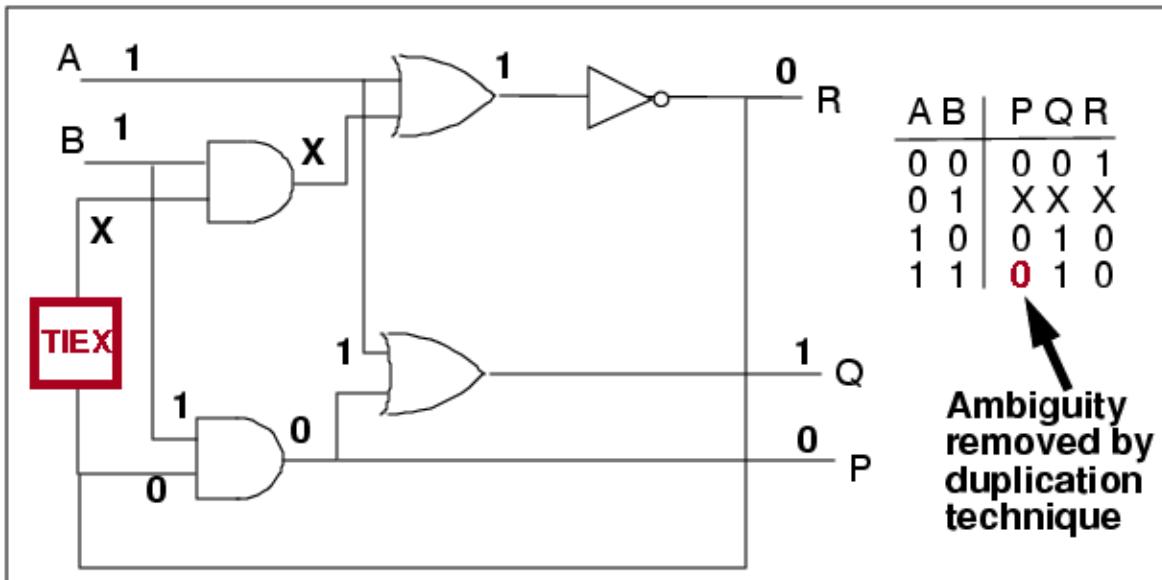
Figure 4-6 shows how TIE-X insertion would add some pessimism to the simulation at output P.

Figure 4-6. TIE-X Insertion Simulation Pessimism



The loop breaking technique proves beneficial in many cases. Figure 4-7 provides a more accurate simulation model than the direct TIE-X insertion approach.

Figure 4-7. Cutting Loops by Gate Duplication



However, it also has some drawbacks. While less pessimistic than the other approaches (except breaking constant value loops), the gate duplication process can still introduce some pessimism into the simulation model.

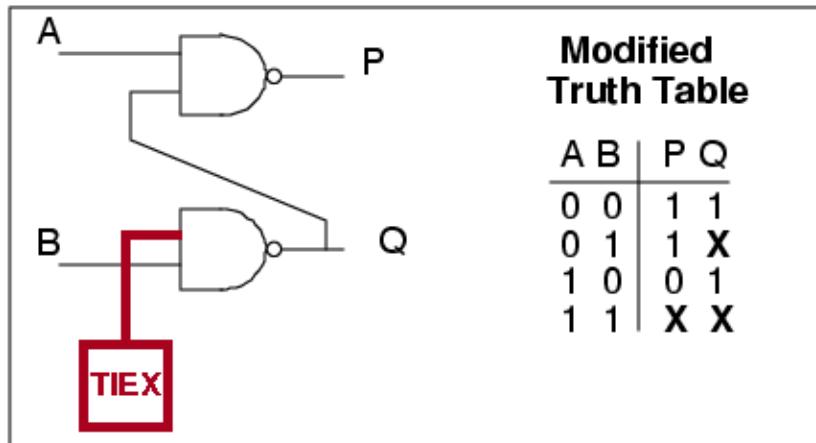
Additionally, this technique can prove costly in terms of gate count as the loop size increases. Also, the tools cannot use this method on complex or *coupled loops* — those loops that connect with other loops (because gate duplication may create loops as well).

4. Coupling loops

The tools use this technique to break loops when two or more loops share a common gate. This method involves inserting a TIE-X gate at the input of one of the components within a loop. The process selects the cut point carefully to ensure the TIE-X gate cuts as many of the coupled loops as possible.

For example, assume the SR latch shown in Figure 4-5 was part of a larger, more complex, loop coupling network. In this case, loop circuitry duplication would turn into an iterative process that would never converge. So, the tools would have to cut the loop as shown in Figure 4-8.

Figure 4-8. Cutting Coupling Loops



The modified truth table shown in Figure 4-8 demonstrates that this method yields the most pessimistic simulation results of all the loop-cutting methods. Because this is the most pessimistic solution to the loop cutting problem, the tools only use this technique when they cannot use any of the previous methods.

ATPG-Specific Combinational Loop Handling Issues

By default, the ATPG tool performs parallel pattern simulation of circuits containing combinational feedback networks. This is controlled by using the `set_loop_handling` command.

A learning process identifies feedback networks after flattening, and an iterative simulation is used in the feedback network. For an iterative simulation, the ATPG tool inserts FB_BUF gates to break the combinational loops.

The ATPG tool also has the ability to insert TIE-X gates to break the combinational loops. The gate duplication option reduces the impact that a TIE-X gate places on the circuit to break combinational loops. By default, this duplication switch is off.

Tessent Scan-Specific Combinational Loop Handling Issues

Tessent Scan identifies combinational loops during flattening. By default, it performs TIE-X insertion using the methods specified in “[Structural Combinational Loops and Loop-Cutting Methods](#)” on page 118 to break all loops detected by the initial loop analysis. You can turn loop duplication off using the “`set_loop_handling -duplication off`” command.

You can report on loops using the `report_loops` or the `report_feedback_paths` commands. While both involved with loop reporting, these commands behave somewhat differently. You can write all identified structural combinational loops to a file using the `write_loops` command.

You can use the loop information Tessent Scan provides to handle each loop in the most desirable way. For example, assuming you wanted to improve the test coverage for a coupling

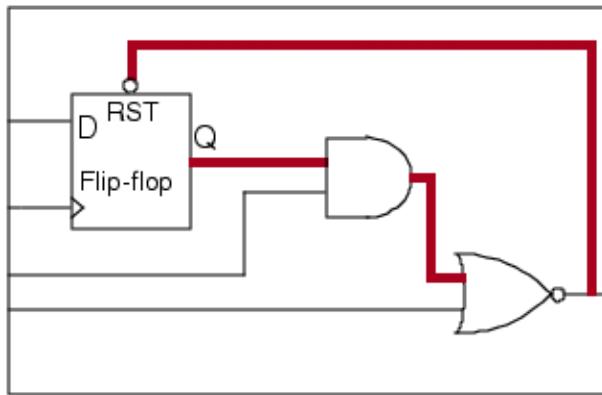
loop, you could use the [add_control_points](#)/[add_observe_points](#) commands within Tesson Scan to insert a test point to control or observe values at a certain location within the loop.

Structural Sequential Loops and Handling

Sequential feedback loops occur when the output of a latch or flip-flop feeds back to one of its inputs, either directly or through some other logic.

[Figure 4-9](#) shows an example of a structural sequential feedback loop.

Figure 4-9. Sequential Feedback Loop



Note

- ❑ The tools model RAM and ROM gates as combinational gates, and thus, they consider loops involving only combinational gates and RAMs (or ROMs) as combinational loops—not sequential loops.
-

The following sections provide tool-specific issues regarding sequential loop handling.

ATPG-Specific Sequential Loop Handling

While the ATPG tool can suffer some loss of test coverage due to sequential loops, these loops do not cause the tool the extensive problems that combinational loops do. By its very nature, the ATPG tool re-models the non-scan sequential elements in the design using the simulation primitives described in “[ATPG Handling of Non-Scan Cells](#)” on page 127. Each of these primitives, when inserted, automatically breaks the loops in some manner.

Within the ATPG tool, sequential loops typically trigger C3 and C4 design rules violations. When one sequential element (a source gate) feeds a value to another sequential element (a sink gate), the tool simulates old data at the sink. For more information on the C3 and C4 rules, refer to “[Clock Rules](#)” in the *Tesson Shell Reference Manual*.

Redundant Logic

In most cases, you should avoid using redundant logic because a circuit with redundant logic poses testability problems. First, classifying redundant faults takes a great deal of analysis effort.

Additionally, redundant faults, by their nature, are untestable and therefore lower your fault coverage. [Figure 2-16](#) on page 74 gives an example of redundant circuitry.

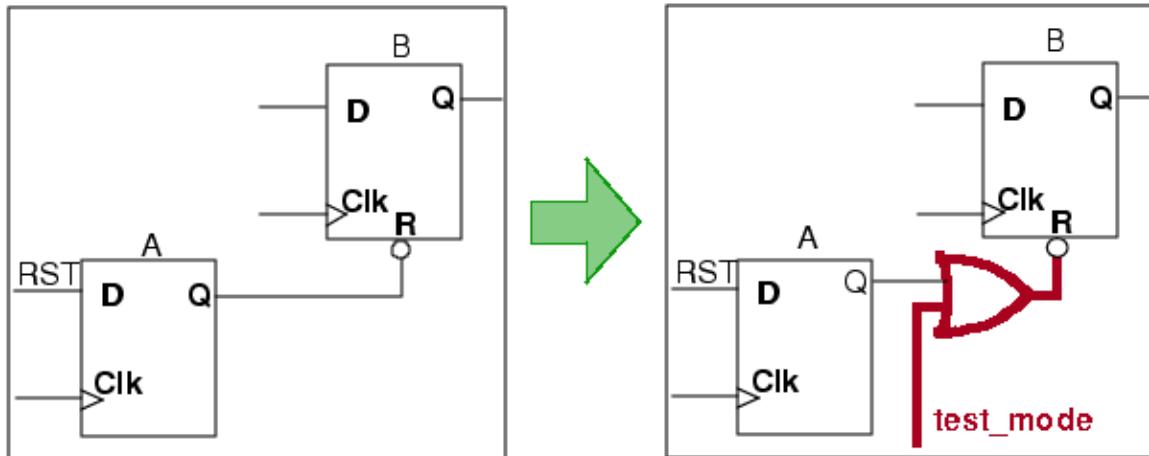
Some circuitry requires redundant logic; for example, circuitry to eliminate race conditions or circuitry which builds high reliability into the design. In these cases, you should add test points to remove redundancy during the testing process.

Asynchronous Sets and Resets

Scannability checking treats sequential elements driven by uncontrollable set and reset lines as unscannable. You can remedy this situation in one of two ways: you can add test logic to make the signals controllable, or you can use initialization patterns during test to control these internally-generated signals. Tessent Scan provides capabilities to aid you in both solutions.

[Figure 4-10](#) shows a situation with an asynchronous reset line and the test logic added to control the asynchronous reset line.

Figure 4-10. Test Logic Added to Control Asynchronous Reset



In this example, Tessent Scan adds an OR gate that uses the test_mode (not scan_enable) signal to keep the reset of flip-flop B inactive during the testing process. You would then constrain the test_mode signal to be a 1, so flip-flop B could never be reset during testing. To insert this type of test logic, you can use the Tessent Scan command set_test_logic (see “[Test Logic Insertion](#)” on page 160 for more information).

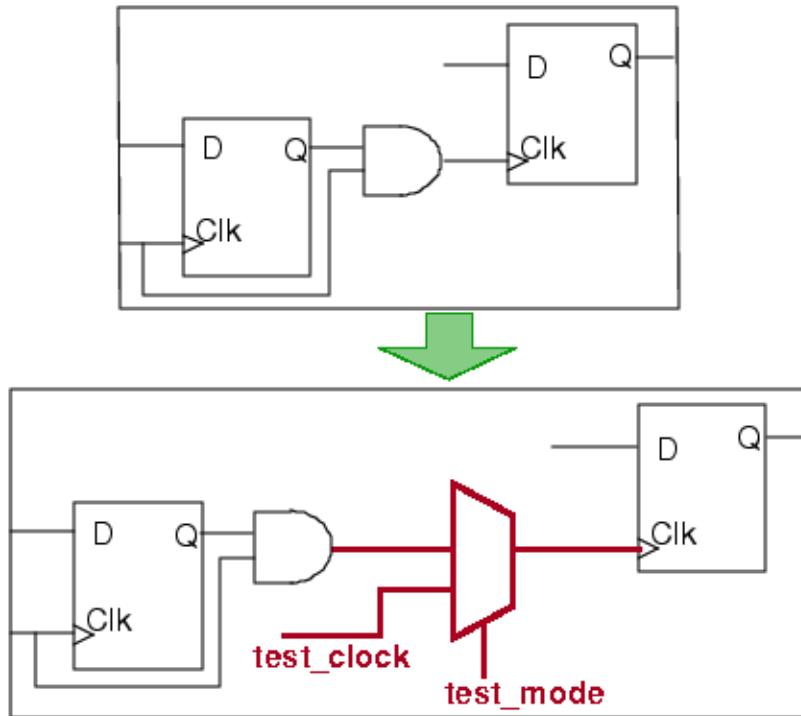
Tessent Scan also allows you to specify an initialization sequence in the test procedure file to avoid the use of this additional test logic. For additional information, refer to the [add_scan_groups](#) description in the *Tessent Shell Reference Manual*.

Gated Clocks

Primary inputs typically cannot control the gated clock signals of sequential devices. In order to make some of these sequential elements scannable, you may need to add test logic to modify their clock circuitry.

For example, [Figure 4-11](#) shows an example of a clock that requires some test logic to control it during test mode.

Figure 4-11. Test Logic Added to Control Gated Clock



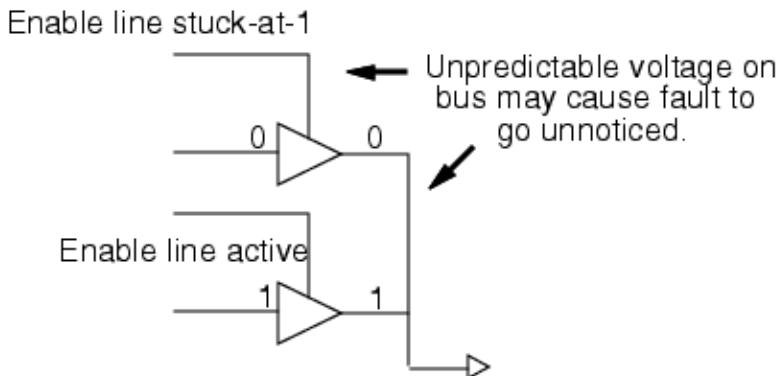
In this example, Tessent Scan makes the element scannable by adding a test clock, for both scan loading/unloading and data capture, and multiplexing it with the original clock signal. It also adds a signal called `test_mode` to control the added multiplexer. The `test_mode` signal differs from the `scan_mode` or `scan_enable` signals in that it is active during the entire duration of the test—not just during scan chain loading/unloading. To add this type of test logic into your design, you can use the [set_test_logic](#) and [set_scan_signals](#) commands

Tri-State Devices

Tri-state™ buses are another testability challenge. Faults on tri-state bus enables can cause one of two problems: bus contention, which means there is more than one active driver, or bus float, which means there is no active driver. Either of these conditions can cause unpredictable logic values on the bus, which allows the enable line fault to go undetected.

Figure 4-12 shows a tri-state bus with bus contention caused by a stuck-at-1 fault.

Figure 4-12. Tri-state Bus Contention



Tessent Scan can add gating logic that turns off the tri-state devices during scan chain shifting. The tool gates the tri-state device enable lines with the `scan_enable` signal so they are inactive and thus prevent bus contention during scan data shifting. To insert this type of gating logic, you can use the Tessent Scan command `set_tristate_gating` (see “[Test Logic Insertion](#)” on page 160 for more information).

In addition, the ATPG tool lets you specify the fault effect of bus contention on tri-state nets. This capability increases the testability of the enable line of the tri-state drivers. Refer to the [`set_net_dominance`](#) description in the *Tessent Shell Reference Manual* for details.

Non-Scan Cell Handling

During rules checking and learning analysis, the ATPG tool learns the behavior of all state elements that are not part of the scan circuitry. This learning involves how the non-scan element behaves after the scan loading operation. As a result of the learning analysis, the ATPG tool categorizes each of the non-scan cells.

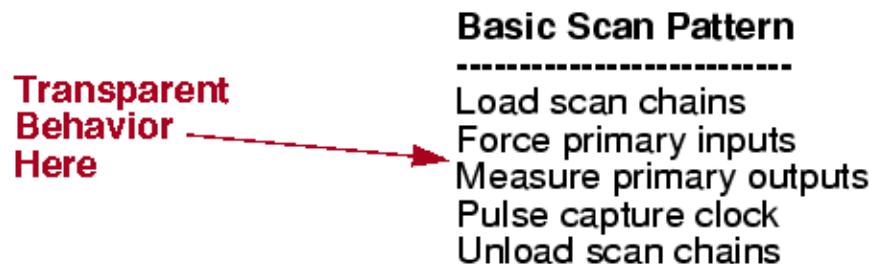
ATPG Handling of Non-Scan Cells

The ATPG tool places non-scan cells in one of the following categories:

- **TIEX** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be an X value during test. This condition may prevent the detection of a number of faults.

- **TIE0** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be a 0 value during test. This condition may prevent the detection of a number of faults.
- **TIE1** — In this category, the ATPG tool considers the output of a flip-flop or latch to always be a 1 value during test. This condition may prevent the detection of a number of faults.
- **Transparent (combinational)** — In this category, the non-scan cell is a latch, and the latch behaves transparently. When a latch behaves transparently, it acts, in effect, as a buffer—passing the data input value to the data output. The TLA simulation gate models this behavior. [Figure 4-13](#) shows the point at which the latch must exhibit transparent behavior.

Figure 4-13. Requirement for Combinatorially Transparent Latches



Transparency occurs if the clock input of the latch is inactive during the time between the force of the primary inputs and the measure of the primary outputs. If your latch is set up to behave transparently, you should not experience any significant fault detection problems (except for faults on the clock, set, and reset lines). However, only in limited cases do non-scan cells truly behave transparently. For the tool to consider the latch transparent, it must meet the following conditions:

- The latch must not create a potential feedback path, unless the path is broken by scan cells or non-scan cells (other than transparent latches).
- The latch must have a path that propagates to an observable point.
- The latch must be able to pass a data value to the output when all clocks are off.
- The latch must have clock, set, and reset signals that can be set to a determined value.

For more information on the transparent latch checking procedure, refer to "[D6](#)" in the *Tessent Shell Reference Manual*.

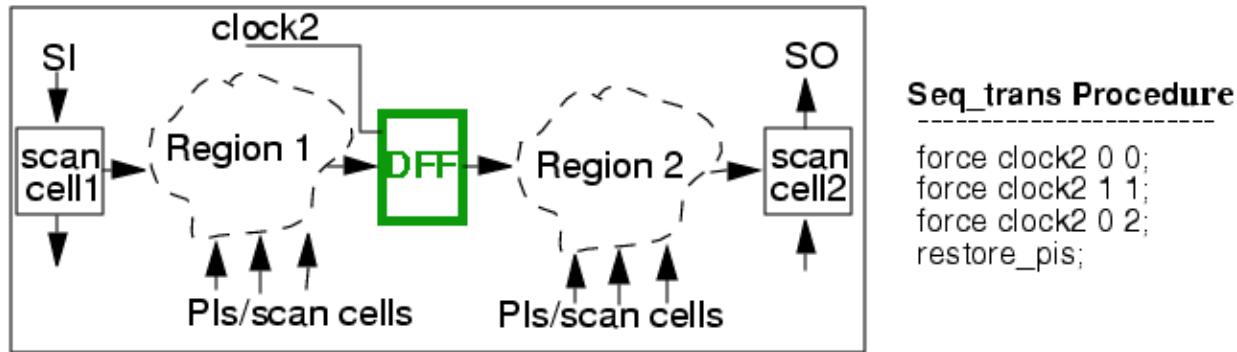
- **Sequential transparent** — Sequential transparency extends the notion of transparency to include non-scan elements that can be forced to behave transparently at the same point in which natural transparency occurs. In this case, the non-scan element can be either a flip-flop, a latch, or a RAM read port. A non-scan cell behaves as sequentially

transparent if, given a sequence of events, it can capture a value and pass this value to its output, without disturbing critical scan cells.

Sequential transparent handling of non-scan cells lets you describe the events that place the non-scan cell in transparent mode. You do this by specifying a procedure, called **seq_transparent**, in your test procedure file. This procedure contains the events necessary to create transparent behavior of the non-scan cell(s). After the tool loads the scan chain, forces the primary inputs, and forces all clocks off, the **seq_transparent** procedure pulses the clocks of all the non-scan cells or performs other specified events to pass data through the cell “transparently”.

[Figure 4-14](#) shows an example of a scan design with a non-scan element that is a candidate for sequential transparency.

Figure 4-14. Example of Sequential Transparency



The DFF shown in [Figure 4-14](#) behaves sequentially transparent when the tool pulses its clock input, clock2. The sequential transparent procedure shows the events that enable transparent behavior.

Note

To be compatible with combinational ATPG, the value on the data input line of the non-scan cell must have combinational behavior, as depicted by the combinational Region 1. Also, the output of the state element, in order to be useful for ATPG, must propagate to an observable point.

Benefits of sequential transparent handling include more flexibility of use compared to transparent handling, and the ability to use this technique for creating “structured partial scan” (to minimize area overhead while still obtaining predictable high test coverage). Also, the notion of sequential transparency supports the design practice of using a cell called a *transparent slave*. A transparent slave is a non-scan latch that uses the slave clock to capture its data. Additionally, you can define and use up to 32 different, uniquely-named **seq_transparent** procedures in your test procedure file to handle the various types of non-scan cell circuitry in your design.

Rules checking determines if non-scan cells qualify for sequential transparency via these procedures. Specifically, the cells must satisfy rules P5, P6, P41, P44, P45, P46, D3, and

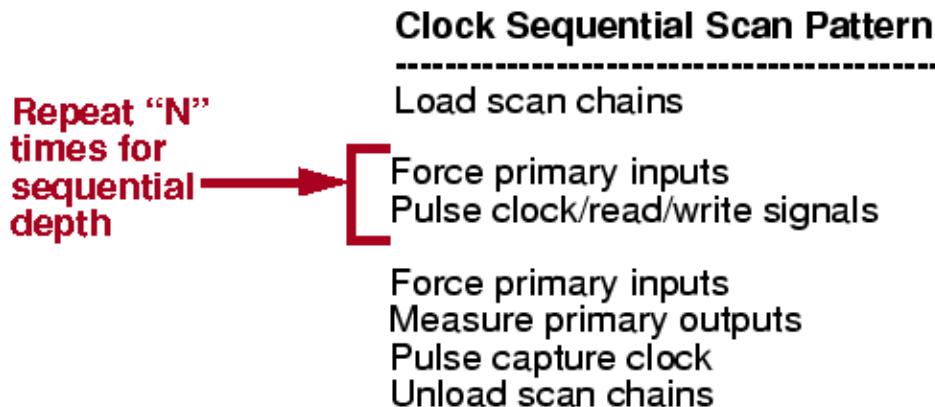
D9. For more information on these rules, refer to “[Design Rule Checking](#)” in the *Tessent Shell Reference Manual*. Clock rules checking treats sequential transparent elements the same as scan cells.

Limitations of sequential transparent cell handling include the following:

- Impaired ability to detect AC defects (transition fault type causes sequential transparent elements to appear as tie-X gates).
- Cannot make non-scan cells clocked by scan cells sequentially transparent without **condition** statements.
- Limited usability of the sequential transparent procedure if applying it disturbs the scan cells (contents of scan cells change during the **seq_transparent** procedure).
- Feedback paths to non-scan cells, unless broken by scan cells, prevent treating the non-scan cells as sequentially transparent.
- **Clock sequential** — If a non-scan cell obeys the standard scan clock rules—that is, if the cell holds its value with all clocks off—the tool treats it as a clock sequential cell. In this case, after the tool loads the scan chains, it forces the primary inputs and pulses the clock/write/read lines multiple times (based on the sequential depth of the non-scan cells) to set up the conditions for a test. A normal observe cycle then follows.

[Figure 4-15](#) shows a clock sequential scan pattern.

Figure 4-15. Clocked Sequential Scan Pattern Events



This technique of repeating the primary input force and clock pulse allows the tool to keep track of new values on scan cells and within feedback paths.

When DRC performs scan cell checking, it also checks non-scan cells. When the checking process completes, the rules checker issues a message indicating the number of non-scan cells that qualify for clock sequential handling.

You instruct the tool to use clock sequential handling by selecting the -Sequential option to the [set_pattern_type](#) command. During test generation, the tool generates test patterns for target faults by first attempting combinational, and then RAM sequential techniques.

If unsuccessful with these techniques, the tool performs clock sequential test generation if you specify a non-zero sequential depth.

Note

 Setting the -Sequential switch to either 0 (the default) or 1 results in patterns with a maximum sequential depth of one, but the tool creates clock sequential patterns only if the setting is 1 or higher.

To report on clock sequential cells, you use the `report_nonscan_cells` command. For more information on setting up and reporting on clock sequential test generation, refer to the `set_pattern_type` and `report_nonscan_cells` descriptions in the *Tessent Shell Reference Manual*.

Limitations of clock sequential non-scan cell handling include:

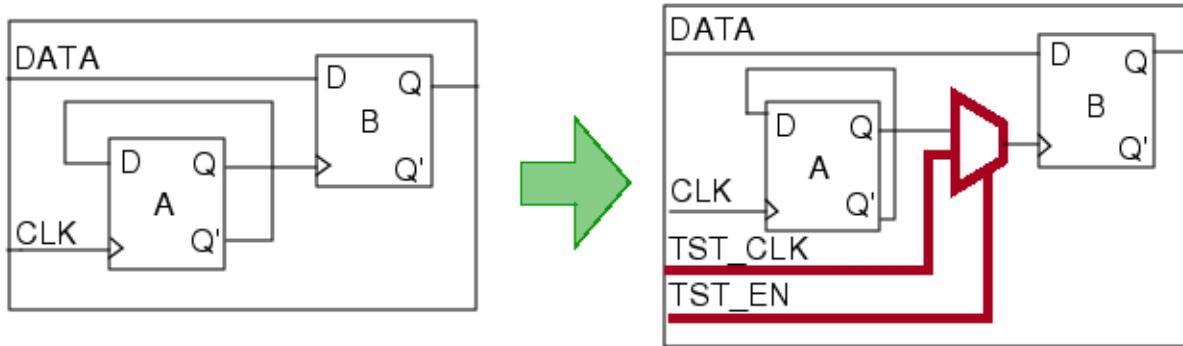
- The maximum allowable sequential depth is 255 (a typical depth would range from 2 to 5).
- Copy and shadow cells cannot behave sequentially.
- The tool cannot detect faults on clock/set/reset lines.
- You cannot use the read-only mode of RAM testing with clock sequential pattern generation.
- The tool simulates cells that capture data on a trailing clock edge (when data changes on the leading edge) using the original values on the data inputs.
- Non-scan cells that maintain a constant value after load_unload simulation are treated as tied latches.
- This type of testing has high memory and performance costs.

Clock Dividers

Some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, Tessent Scan will not consider the sequential elements controlled by these signals “scannable”. And consequently, they could disturb sequential elements during scan shifting. Thus, the system cannot convert these elements to scan.

Figure 4-16 shows an example of a sequential element (B) driven by a clock divider signal and with the appropriate circuitry added to control the divided clock signal.

Figure 4-16. Clock Divider



Tesson Scan can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) by inserting special circuitry, called *test logic*, at these nodes when necessary. Tesson Scan typically gates the uncontrollable circuitry with chip-level test pins. In the case of uncontrollable clocks, Tesson Scan adds a MUX controlled by the test_clk and test_en signals.

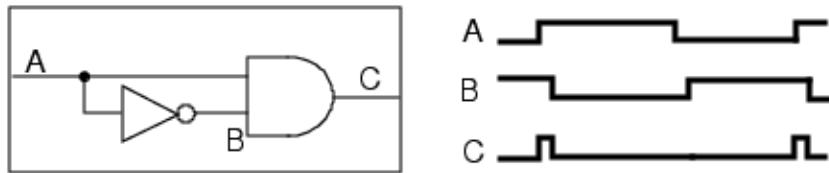
For more information on test logic, refer to “[Test Logic Insertion](#)” on page 160.

Pulse Generators

A pulse generator is circuitry that creates a pulse at its output when active.

[Figure 4-17](#) gives an example of pulse generator circuitry.

Figure 4-17. Example Pulse Generator Circuitry



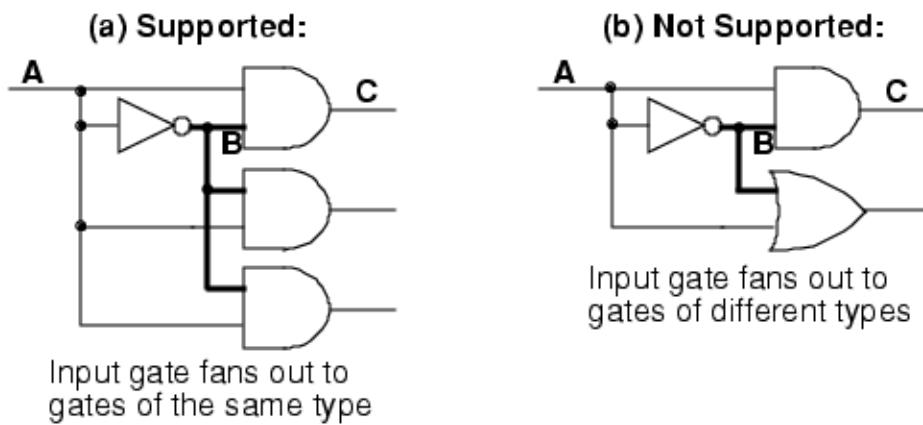
When designers use this circuitry in clock paths, there is no way to create a stable on state. Without a stable on state, the fault simulator and test generator have no way to capture data into the scan cells. Pulse generators also find use in write control circuitry, a use that impedes RAM testing.

By default, the ATPG tool identifies the reconvergent pulse generator sink (PGS) gates, or simply “pulse generators”, during the learning process. For the tools to provide support, a “pulse generator” must satisfy the following requirements:

- The “pulse generator” gate must have a connection (at C in [Figure 4-17](#)) to a clock input of a memory element or a write line of a RAM.

- The “pulse generator” gate must be an AND, NAND, OR, or NOR gate.
- Two inputs of the “pulse generator” gate must come from one reconvergent source gate.
- The two reconvergent paths may only contain inverters and buffers.
- There must be an inversion difference in the two reconvergent paths.
- The two paths must have different lengths (propagation times).
- In the long path, the inverter or buffer that connects to the “pulse generator” input must only go to gates of the same gate type as shown in (a) in [Figure 4-18](#). A fanout to gates of different types as in (b) in the figure is not supported. The tools model this input gate as tied to the non-controlling value of the “pulse generator” gate (TIE1 for AND and NAND gates, TIE0 for OR and NOR gates).

Figure 4-18. Long Path Input Gate Must Go to Gates of the Same Type



Rules checking includes some checking for “pulse generator” gates. Specifically, Trace rules #16 and #17 check to ensure proper usage of “pulse generator” gates. Refer to “[T16](#)” and “[T17](#)” in the *Tessent Shell Reference Manual* for more details about these rules.

The ATPG tool supports pulse generators with multiple timed outputs. For detailed information about this support, refer to “[Pulse Generators with User Defined Timing](#)” in the *Tessent Cell Library User’s Manual*.

JTAG-Based Circuits

Boundary scan circuitry, as defined by IEEE standard 1149.1, can result in a complex environment for the internal scan structure and the ATPG process. The two main issues with boundary scan circuitry are 1) connecting the boundary scan circuitry with the internal scan circuitry, and 2) ensuring that the boundary scan circuitry is set up properly during ATPG.

RAM and ROM Test Overview

The three basic problems of testing designs that contain RAM and ROM are 1) modeling the behavior, 2) passing rules checking to allow testing, and 3) detecting faults during ATPG.

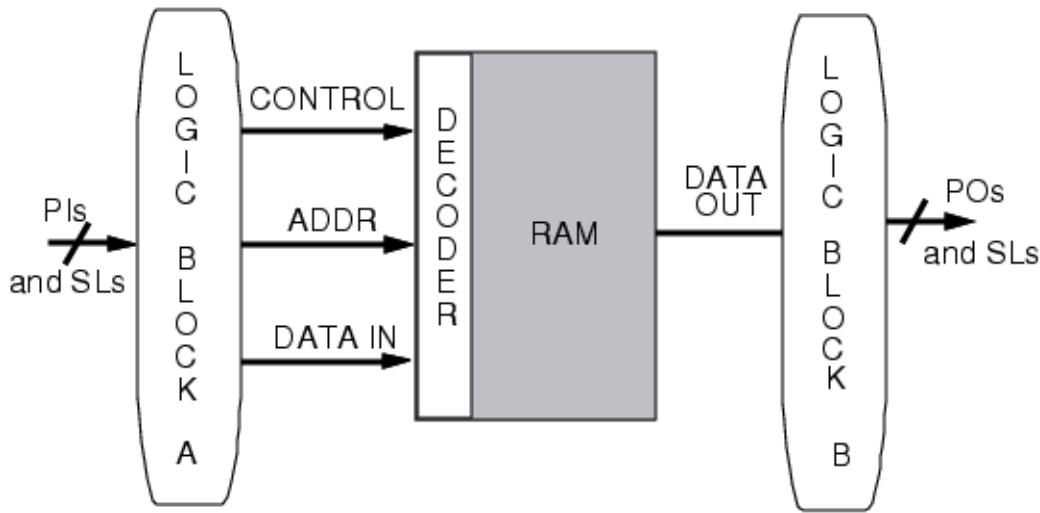
The “[RAM and ROM](#)” section in the *Tessent Cell Library User’s Manual* discusses modeling RAM and ROM behavior. The “[RAM Rules](#)” section in the *Tessent Shell Reference Manual* discusses RAM rules checking. This section primarily discusses the techniques for detecting faults in circuits with RAM and ROM during ATPG. The “[RAM Summary Results and Test Capability](#)” section of the *Tessent Shell Reference Manual* discusses displayed DRC summary results upon completion of RAM rules checking.

The ATPG tool does not test the internals of the RAM/ROM, although MacroTest (separately licensed but available in the ATPG tool) lets you create tests for small memories such as register files by converting a functional test sequence or algorithm into a sequence of scan tests. For large memories, built-in test structures within the chip itself are the best methods of testing the internal RAM or ROM.

However, the ATPG tool needs to model the behavior of the RAM/ROM so that tests can be generated for the logic on either side of the embedded memory. This allows the tool to generate tests for the circuitry around the RAM/ROM, as well as the read and write controls, data lines, and address lines of the RAM/ROM unit itself.

[Figure 4-19](#) shows a typical configuration for a circuit containing embedded RAM.

Figure 4-19. Design with Embedded RAM



ATPG must be able to operate the illustrated RAM to observe faults in logic block A, as well as to control the values in logic block B to test faults located there. The ATPG tool has unique strategies for operating the RAMs.

RAM/ROM Support [135](#)

RAM/ROM Support Commands	138
Basic ROM/RAM Rules Checks.....	139

RAM/ROM Support

The tool treats a ROM as a strictly combinational gate. Once a ROM is initialized, it is a simple task to generate tests because the contents of the ROM do not change. Testing RAM however, is more of a challenge, because of the sequential behavior of writing data to and reading data from the RAM.

The tool supports the following strategies for propagating fault effects through the RAM:

- **Read-only mode** — The tool assumes the RAM is initialized prior to scan test and this initialization must not change during scan. This assumption allows the tool to treat a RAM as a ROM. As such, there is no requirement to write to the RAM prior to reading, so the test pattern only performs a read operation. Important considerations for read-only mode test patterns are as follows:
 - The read-only testing mode of RAM only tests for faults on data out and read address lines, just as it would for a ROM. The tool does not test the write port I/O.
 - To use read-only mode, the circuit must pass rules A1 and A6.
 - Values placed on the RAM are limited to initialized values.
 - Random patterns can be useful for all RAM configurations.
 - You must define initial values and assume responsibility that those values are successfully placed on the correct RAM memory cells. The tool does not perform any audit to verify this is correct, nor will the patterns reflect what needs to be done for this to occur.
 - Because the tester may require excessive time to fully initialize the RAM, it is allowed to do a partial initialization.
- **Pass-through mode** — The tool has two separate pass-through testing modes:
 - **Static pass-through** — To detect faults on data input lines, you must write a known value into some address, read that value from the address, and propagate the effect to an observation point. In this situation, the tool handles RAM transparently, similar to the handling of a transparent latch. This requires several simultaneous operations. The write and read operations are both active and thus writing to and reading from the same address. While this is a typical RAM operation, it allows testing faults on the data input and data output lines. It is not adequate for testing faults on read and write address lines.
 - **Dynamic pass-through** — This testing technique is similar to static pass-through testing except one pulse of the write clock performs both the write and read operation (if the write and read control lines are complementary). While static pass-

through testing is comparable to transparent latch handling, dynamic pass-through testing compares to sequential transparent testing.

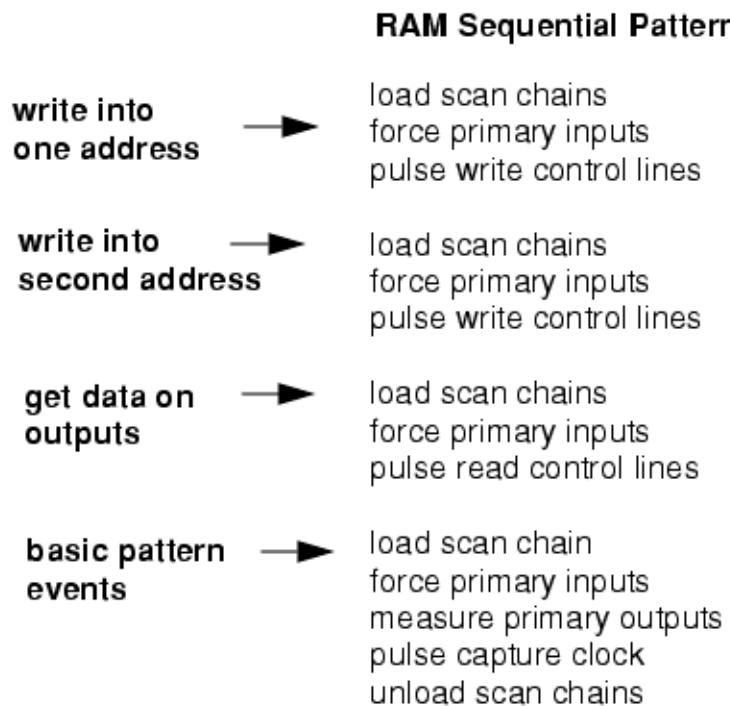
- **Sequential RAM test mode** — This is the recommended approach to RAM testing. While the previous testing modes provide techniques for detecting some faults, they treat the RAM operations as combinational. Thus, they are generally inadequate for generating tests for circuits with embedded RAM. In contrast, this testing mode tries to separately model all events necessary to test a RAM, which requires modeling sequential behavior. This enables testing of faults that require detection of multiple pulses of the write control lines. These faults include RAM address and write control lines.

RAM sequential testing requires its own specialized pattern type. RAM sequential patterns consist of one scan pattern with multiple scan chain loads. A typical RAM sequential pattern contains the events shown in [Figure 4-20](#).

Note

For RAM sequential testing, the RAM's read_enable/write_enable control(s) can be generated internally. However, the RAM's read/write clock should be generated from a PI. This ensures RAM sequencing is synchronized with the RAM sequential patterns.

Figure 4-20. RAM Sequential Example



In this example of an address line test, assume that the MSB address line is stuck at 0. The first write would write data into an address whose MSB is 0 to match the faulty

value, such as 0000. The second write operation would write different data into a different address (the one obtained by complementing the faulty bit). For this example, it would write into 1000. The read operation then reads from the first address, 0000. If the highest order address bit is stuck-at-0, the 2nd write would have overwritten the original data at address 0, and faulty circuitry data would be read from that address in the 3rd step.

Another technique that may be useful for detecting faults in circuits with embedded RAM is clock sequential test generation. It is a more flexible technique, which effectively detects faults associated with RAM. “[Clock Sequential Patterns](#)” on page 282 discusses clock sequential test generation in more detail.

Common Read and Clock Lines

Ram_sequential simulation supports RAMs whose read line is common with a scan clock. The tool assumes that the read and capture operation can occur at the same time and that the value captured into the scan cell is a function of the value read out from the RAM.

If the clock that captures the data from the RAM is the same clock which is used for reading, the tool issues a C6 clock rules violation. This indicates that you must set the clock timing so that the scan cell can successfully capture the newly read data.

If the clock that captures the data from the RAM is not the same clock that is used for reading, you will likely need to turn on multiple clocks to detect faults. The default “set_clock_restriction On” command is conservative, so the tool will not allow these patterns, resulting in a loss in test coverage. If you issue the “set_clock_restriction Off” command, the tool allows these patterns, but there is a risk of inaccurate simulation results because the simulator does not propagate captured data effects.

Common Write and Clock Lines

The tool supports common write and clock lines. The following shows the support for common write and clock lines:

- You can define a pin as both a write control line and a clock if the off-states are the same value. the tool then displays a warning message indicating that a common write control and clock has been defined.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a write line of a RAM, and the corresponding address or data-in lines are connected to scan latches which has a connection to the same clock.
- The rules checker issues a C3 clock rule violation if a clock can propagate to a read line of a RAM, and the corresponding address lines are connected to scan latches which has a connection to the same clock.

- The rules checker issues a C3 clock rule violation if a clock can capture data into a scan latch that comes from a RAM read port that has input connectivity to latches which has a connection to the same clock.
- If you set the simulation mode to Ram_sequential, the rules checker will not issue an A2 RAM rule violation if a clock is connected to a write input of a RAM. Any clock connection to any other input (including the read lines) will continue to be a violation.
- If a RAM write line is connected to a clock, you cannot use the dynamic pass through test mode.
- Patterns which use a common clock and write control for writing into a RAM will be in the form of ram_sequential patterns. This requires you to set the simulation mode to Ram_sequential.
- If you change the value of a common write control and clock line during a test procedure, you must hold all write, set, and reset inputs of a RAM off. The tool considers failure to satisfy this condition as an A6 RAM rule violation and disqualifies the RAM from being tested using read_only and ram_sequential patterns.

RAM/ROM Support Commands

The tool requires certain knowledge about the design prior to test generation. For circuits with RAM, you must define write controls, and if the RAM has data hold capabilities, you must also define read controls. Just as you must define clocks so the tool can effectively write scan patterns, you must also define these control lines so it can effectively write patterns for testing RAM. And similar to clocks, you must define these signals in setup mode, prior to rules checking.

The commands in [Table 4-1](#) support the testing of designs with RAM and/or ROM.

Table 4-1. RAM/ROM Commands

Command Name	Description
add_read_controls	Defines a PI as a read control and specifies its off value.
add_write_controls	Defines a PI as a write control and specifies its off value.
create_initialization_patterns	Creates RAM initialization patterns and places them in the internal pattern set.
delete_read_controls	Removes the read control line definitions from the specified primary input pins.
delete_write_controls	Removes the write control line definitions from the specified primary input pins.
read_modelfile	Initializes the specified RAM or ROM gate using the memory states contained in the specified modelfile.
report_read_controls	Displays all of the currently defined read control lines.

Table 4-1. RAM/ROM Commands (cont.)

Command Name	Description
report_write_controls	Displays all of the currently defined write control lines.
set_pattern_type	Specifies whether the ATPG simulation run uses combinational or sequential RAM test patterns.
set_ram_initialization	Specifies whether to initialize RAM and ROM gates that do not have initialization files.
write_modelfile	Writes all internal states for a RAM or ROM gate into the file that you specify.

Basic ROM/RAM Rules Checks

It is important the tool provides basic ROM/RAM rules checks.

The rules checker performs the following audits for RAMs and ROMs:

- The checker reads the RAM/ROM initialization files and checks them for errors. If you selected random value initialization, the tool gives random values to all RAM and ROM gates without an initialized file. If there are no initialized RAMs, you cannot use the read-only test mode. If any ROM is not initialized, an error condition occurs. A ROM must have an initialization file but it may contain all Xs. Refer to the [read_modelfile](#) description in the *Tessent Shell Reference Manual* for details on initialization of RAM/ROM.
- The RAM/ROM instance name given must contain a single RAM or ROM gate. If no RAM or ROM gate exists in the specified instance, an error condition occurs.
- If you define write control lines and there are no RAM gates in the circuit, an error condition occurs. To correct this error, delete the write control lines.
- When the write control lines are off, the RAM set and reset inputs must be off and the write enable inputs of all write ports must be off. You cannot use RAMs that fail this rule in read-only test mode. If any RAM fails this check, you cannot use dynamic pass-through. If you defined an initialization file for a RAM that failed this check, an error condition occurs. To correct this error, properly define all write control lines or use lineholds (pin constraints).
- A RAM gate must not propagate to another RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- A defined scan clock must not propagate directly (unbroken by scan or non-scan cells) to a RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- The tool checks the write and read control lines for connectivity to the address and data inputs of all RAM gates. It gives a warning message for all occurrences and if connectivity fails, there is a risk of race conditions for all pass-through patterns.

- A RAM that uses the edge-triggered attribute must also have the **read_off** attribute set to hold. Failure to satisfy this condition results in an error condition when the design flattening process is complete.
- If the RAM rules checking identifies at least one RAM that the tool can test in read-only mode, it sets the RAM test mode to read-only. Otherwise, if the RAM rules checking passes all checks, it sets the RAM test mode to dynamic pass-through. If it cannot set the RAM test mode to read-only or dynamic pass-through, it sets the test mode to static pass-through.
- A RAM with the **read_off** attribute set to hold must pass Design Rule A7 (when read control lines are off, place read inputs at 0). The tool treats RAMs that fail this rule as:
 - A TIE-X gate, if the read lines are edge-triggered.
 - A **read_off** value of X, if the read lines are not edge-triggered.
- The read inputs of RAMs that have the **read_off** attribute set to hold must be at 0 during all times of all test procedures, except the **test_setup** procedure.
- The read control lines must be off at time 0 of the **load_unload** procedure.
- A clock cone stops at read ports of RAMs that have the **read_off** attribute set to hold, and the effect cone propagates from its outputs.

For more information on the RAM rules checking process, refer to “[RAM Rules](#)” in the *Tessent Shell Reference Manual*.

Incomplete Designs

The ATPG tool and Tessent Scan can read incomplete Verilog designs due to their ability to generate black boxes. The Verilog parser can blackbox any instantiated module or instance that is not defined in either the ATPG library or the design netlist.

The tool issues a warning message for each blackboxed module similar to the following:

```
// WARNING: Following modules are undefined:  
//          ao21  
//          and02  
// Use "add_black_box -auto" to treat these as black boxes.
```

If the tool instantiates an undefined module, it generates a module declaration based on the instantiation. If ports are connected by name, the tool uses those port names in the generated module. If ports are connected by position, the parser generates the port names. Calculating port directions is problematic and must be done by looking at the other pins on the net connected to the given instance pin. For each instance pin, if the connected net has a non-Z-producing driver, the tool considers the generated module port an input, otherwise the port is an output. The tool never generates inout ports since they cannot be inferred from the other pins on the net.

Modules that are automatically blackboxed default to driving X on their outputs. Faults that propagate to the black box inputs are classified as ATPG_untestable (AU). To change the output values driven, refer to the [add_black_box](#) description in the *Tessent Shell Reference Manual*.

Chapter 5

Internal Scan and Test Circuitry Insertion

Tessent Scan is used to insert scan circuitry in your design.

The process of inserting scan and other test circuitry in your design using Tessent Scan or Tessent Shell operating in “dft -scan” context is described in these sections.

Note

 When you invoke Tessent Scan in the “dft -scan” context, it is invoked using the hierarchical scan capability (HSI) by default. If you want to use Tessent Scan in the legacy mode (without hierarchical scan insertion), you can do so by setting context to “dft -scan -legacy.”

Introduction to Tessent Scan	145
Features of Tessent Scan.....	146
Tessent Scan Inputs and Outputs	146
Invoking Tessent Scan	148
Example Dofile	148
Test Structures Supported by Tessent Scan	149
Tool Terminology and Concepts	151
Scan Element	151
Scan Chain Family	153
Scan Mode	154
Unwrapped Cores Versus Wrapped Cores	155
Pre-Existing Scan Segments	156
Pre-Existing Scan Chains	158
Preparation for Test Structure Insertion	159
Scan Cell and Scan Output Mapping	159
Test Logic Insertion	160
User Clock Signals	163
How to Specify Existing Scan Information	164
How to Handle Existing Boundary Scan Circuitry	166
How to Run Rules Checking	166
Setup for a Basic Scan Insertion Run	167
Setup for Wrapper Chain Identification	168
Child OCC Schematic	175
How to Manually Include and Exclude Cells for Scan	176
How to Report Scannability Information	178
Automatic Recognition of Existing Scan Chains In Cell Library Models	179
Automatic Recognition of Existing Shift Registers	182

Scan Cell Identification and Reporting	185
Multi-Mode Chains	186
Introduction to Multi-Mode Chains	186
Multi-Mode Gating Logic	187
Multi-Mode Scan Insertion	188
Scan Insertion Flows	192
Scan Insertion Flow Steps	192
Controlling Scan Insertion	193

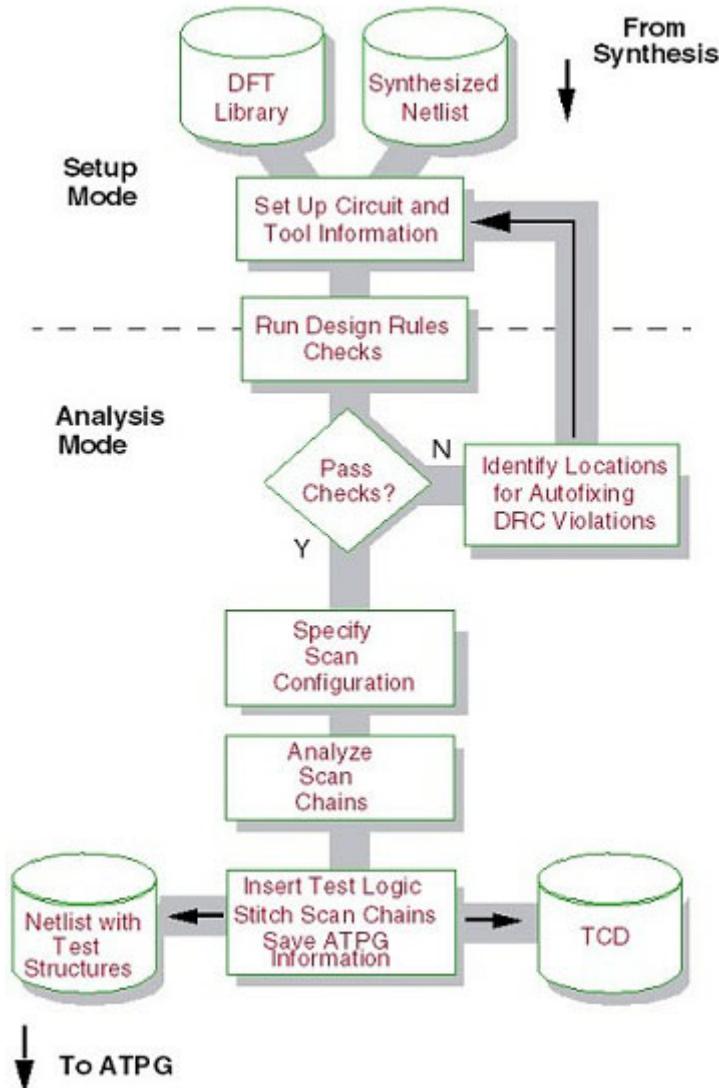
Introduction to Tessent Scan

Tessent Scan is extremely flexible to use and has various options and features that will be described in detail in this manual.

With Tessent Scan, you can analyze the design, allocate new scan chains based on design and user constraints, and introspect the scan chain distribution and balancing before performing the actual scan insertion on the design.

Another valuable feature of Tessent Scan is multi-mode scan insertion capabilities. With multi-mode, any number of scan modes or scan configurations can be specified.

Figure 5-1. Generic Usage of Tessent Scan on a Synthesized Netlist



[Figure 5-1](#) shows how a netlist out of synthesis tools is used with Tesson Scan. The netlist can have scan cells replaced and not stitched or just non-scan cells which can be replaced with scan-cells while performing scan stitching. The Design Rule Checks are the S-Rules that are run when transitioning from system mode Setup to Analysis.

Identification of wrapper cells is required and used only when performing scan insertion on a hierarchical region where hierarchical DFT is to be implemented, where the test patterns to test the logic for this block/module can be run stand-alone and/or planned to be retargeted from the next level. If you are using hierarchical test application, then you need to identify wrapper cells before specifying Scan Configuration. This is described in the [Scan Insertion for Wrapped Core](#) section.

You can provide input to specify the desired Scan Configuration to analyze the scan chains before it gets inserted and stitched up. This is very powerful, as you change the scan configuration specification to see how the scan chains will end up after it gets stitched by just performing analysis. After the scan chains have been inserted, the netlist and TCD (Tesson Core Description) that has details of how the scan changes have been stitched are written out.

Features of Tesson Scan	146
Tesson Scan Inputs and Outputs.....	146
Invoking Tesson Scan	148
Example Dofile	148
Test Structures Supported by Tesson Scan	149

Features of Tesson Scan

These are some of the features of Tesson Scan.

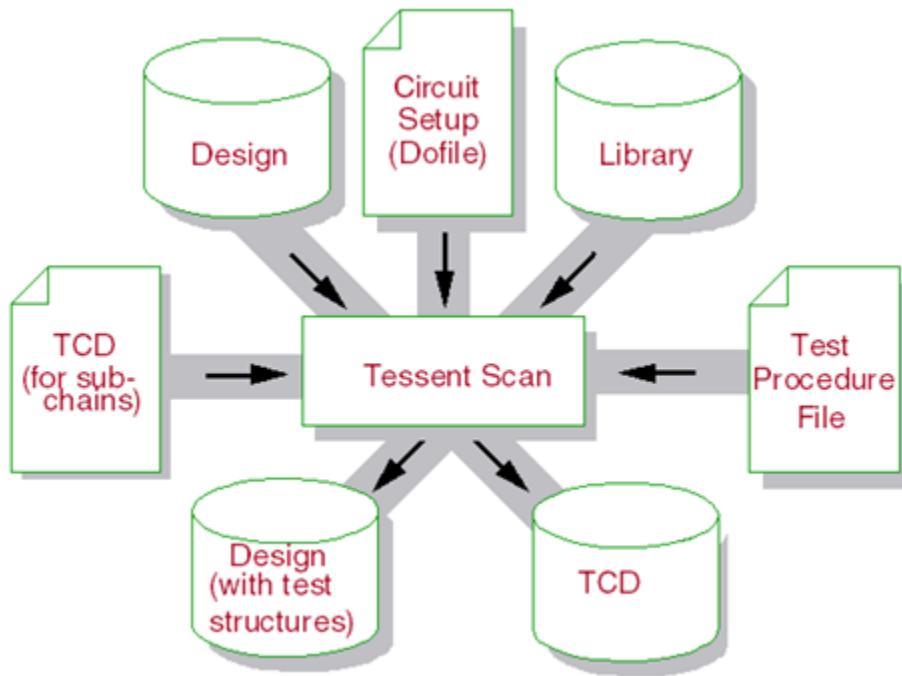
1. It supports multi-mode scan insertion with optimal scan chain lengths across all modes.
2. It allows you to separate the scan analysis phase from the actual stitching of scan chains and provides the scan chain map prior to stitching.
3. It enables introspection of the scan data model and allows you to specify how scan cells should be grouped during scan stitching.

Tesson Scan Inputs and Outputs

Tesson Scan uses multiple inputs and produces several outputs during the scan insertion process.

[Figure 5-2](#) shows the inputs used and the outputs produced by Tesson Scan.

Figure 5-2. The Inputs and Outputs of Tessent Scan



Tessent Scan uses the following inputs:

- **Design (netlist)** — A Verilog gate-level netlist needs to be provided as input.
- **Circuit Setup (or Dofile or Tcl file)** — This is the set of commands that gives the tool information about the circuit and how to insert test structures. You can issue these commands interactively in the tool session or place them in a Tcl file.
- **Library** — The Tessent Cell library contains descriptions of all the cells the design uses. The library also includes information that the tool uses to map non-scan cells to scan cells and to select components for added test logic circuitry.
- **Input TCD File** — If there are pre-existing scan segments that are described in a *.tcd_scan file, then you need to provide them as input to Scan Insertion. You can read them in using the set_design_sources command. The full syntax of tcd_scan file is found in [Scan](#) section of the *Tessent Shell Reference Manual*. Also, if there are CTL models for pre-existing scan segments, these can be converted using stil2mgc and then read in.
- **Test Procedure File** — This file defines the stimulus for shifting scan data through the defined scan chains. This input is only necessary on designs containing pre-existing scan circuitry or requiring initialization for test mode (test setup).

Tessent Scan produces the following outputs:

- **Design (Netlist)** — This netlist contains the original design modified with the inserted test structures. The output netlist format is gate-level Verilog.

- **TCD (Tessent Core Description)** — This file contains all the Scan modes that were specified during scan insertion. This file is used by ATPG tool to generate patterns. If any *.tcd* files from a previous insertion pass or from a lower level core are read in, then the scan modes are appended to the input **.tcd* file and written out into the *tsdb_outdir* directory.

Invoking Tessent Scan

You access Tessent Scan functionality by invoking Tessent Shell and then setting the context to “dft -scan.”

Procedure

Enter the following:

```
% tessent -shell  
SETUP> set_context dft -scan
```

The tool invokes in setup mode, ready for you to begin loading or working on your design. You use this setup mode prepare the tool to define the circuit and scan data which is the next step in the process.

Example Dofile

This is an example of a dofile using Tessent Scan.

```
# SETUP
# Set the context
>set_context dft -scan

# Read the verilog
>read_verilog ./design/cpu.v

# Read the library
>read_cell_library ./library/adk.tcelllib
>read_cell_library ./library/ram.lib
>set_current_design cpu

# Add clocks in the design
>add_clock 0 clk1
>add_clock 0 clk2
>add_clock 0 clk3
>add_clock 0 clk4
>add_clock 0 ramclk

//RUN DRC
>set_system_mode analysis

# Specify constraints to stitch up scan chains
>set_scan_insertion_options -chain_count 100
>analyze_scan_chains
>report_scan_chains
>insert_test_logic -write_in_tsdb On
>report_scan_chains
```

The insert_test_logic -write_in_tsdb on command creates the following:

1. The scan stitched and inserted design.
2. TCD file containing the Scan Configurations to be used during ATPG.

Test Structures Supported by Tessent Scan

Tessent Scan can identify and insert a variety of test structures, including several different scan architectures and test points.

The tool supports the following test structures:

- **Scan**— A flow where the tool converts all sequential elements that pass scannability checking into scan cells. [About Scan Design Methodology](#) discusses the full scan style.
- **Wrapper chains**— A flow where the tool identifies sequential elements that interact with input and output pins. These memory elements are converted into scan chains, and the remaining sequential elements are not affected. For more information, see [About Wrapper Chains](#).
- **Scan and/or Wrapper**— A flow where the tool converts into scan cells those sequential elements that interact with primary input and output pins, and then stitches the scan cells

into dedicated wrapper chains. The tool converts the remaining sequential elements into scan cells and stitches them into separate chains, which are called core chains.

- **Test points** — A flow where the tool inserts control and observe points at user specified locations. “[What are Test Points?](#)” on page 208 discusses the test points method.

Tessent Scan provides the ability to insert test points at user specified locations. If both scan and test points are enabled during an identification run, the tool performs scan identification followed by test point identification.

Tool Terminology and Concepts

This section introduces and describes some terminology that will help you understand how to view the design and how the scan chains are to be stitched.

Scan Element	151
Scan Chain Family.....	153
Scan Mode	154
Unwrapped Cores Versus Wrapped Cores.....	155
Pre-Existing Scan Segments	156
Pre-Existing Scan Chains	158

Scan Element

A scan element, in simple terms, can be either a library cell (lowest level model) called leaf cell or a sub-chain / segments. A library cell/leaf cell can either be single-bit (for example, a flip-flop) or a multi-bit scan element.

Figure 5-3. A Generic Representation of a Scan Element Object

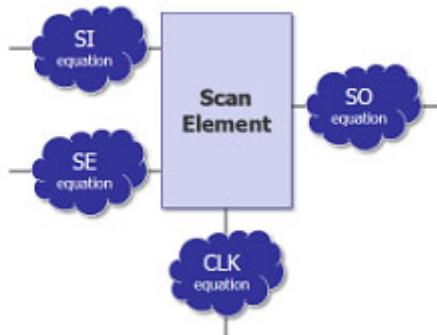


Figure 5-4 is an example of a scan element with a single bit library / leaf cell.

Figure 5-4. Single Bit Scan Element

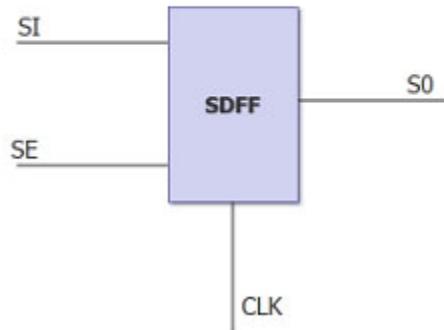


Figure 5-5 is an example of a scan element with multi-bit library / leaf cell.

Figure 5-5. Multi-Bit Scan Element

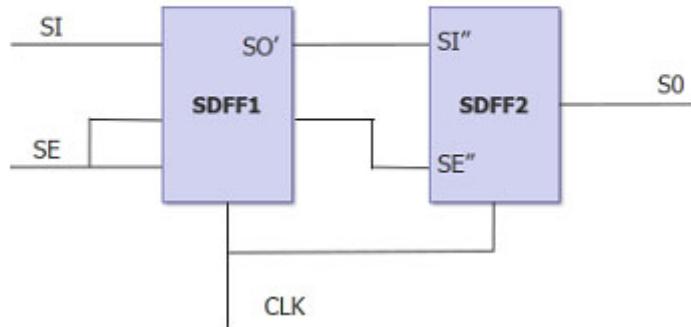
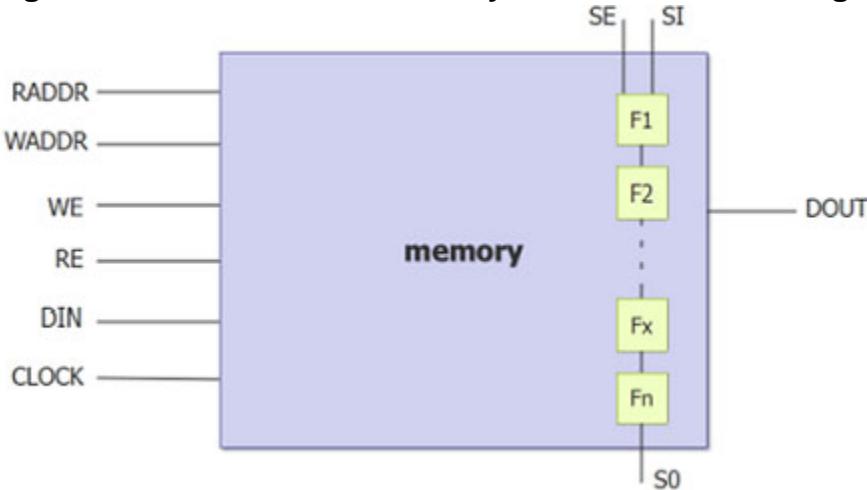


Figure 5-6 is an example of a scan element with sub-chain/segments that are described inside a memory module. This scan element can be described using the input tcd_scan file or using the add_scan_segments command.

Figure 5-6. Scan Element Memory with Sub-Chains / Segments



The Scan Element Object Type is described in the [Scan Data Model](#) section of the Tessent Shell Reference Manual. It can be introspected in the design and is attribute friendly. This aids in describing the desired scan configuration.

Most scan element leaves are populated from DRC results, but some get inferred by the planned insertion of certain types of modules like dedicated wrapper cells; these scan elements are deemed "virtual" as they do not exist yet but are assumed to exist for analysis and planning purposes.

Scan Chain Family

A `scan_chain_family` is an intelligent container that controls the allocation of new scan chains from a specific population of scan elements.

The command `create_scan_chain_family` is used to generate new chain families. The related commands `delete_scan_chain_families` and `get_scan_chain_families` are also available.

Examples

Example 1

The following example shows how a new scan chain family can be created using the `create_scan_chain_family` command. The `scan_chain_family` called `family_clk1` is created for all the `scan_elements` in clock domain `clk1`. The `scan_chain_family` named `family_clk23` is created for clock domains `clk2` and `clk3` below. The example also shows to use `get_scan_chain_families` to report the two new `scan_chain_families` that were created.

```
>create_scan_chain_family family_clk1 -include_elements \
[get_scan_element -filter "clock_domain == clk1"]

>create_scan_chain_family family_clk23 -include_elements \
[get_scan_element -filter "clock_domain == clk2 || clock_domain == clk3"]
```

>get_scan_chain_families

Example 2

The command `create_scan_chain_family` enables more advanced users to control how a specific scan element sub-population gets allocated into chains. For instance, if you want to use special naming only for the external chain ports, this could be achieved by creating a scan chain family for the sub-population of external scan elements as follows:

```
> analyze_wrapper_cells
> create_scan_chain_family ext_chains \
    -si_port_format { ext_si[%d] } \
    -so_port_format { ext_so[%d] } \
    -include_elements [get_scan_elements -class wrapper]
```

These `scan_chain_family` objects constitute larger and more complex building blocks than `scan_element` objects when allocating scan chains.

Scan Mode

Hierarchical Scan Insertion allows you to specify any number of scan modes.

A mode typically applies to an entire scan element population or a large subset (for example, wrapper elements), and is populated by including `scan_element` objects directly and/or `scan_chain_family` objects. The Scan Mode built-in attributes are described in the [Scan Data Model](#) section of the Tesson Shell Reference Manual.

The population of each scan mode is defined by including `scan_element` and `scan_chain_family` objects. If you do not explicitly add at least one scan mode, the tool will infer a default mode which includes every `scan_element` and `scan_chain_family` object (if present).

Implicit to the definition of a scan mode is the inclusion of a scan element population particular for that mode. The mode definition also lets you override general scan insertion options with some specific to the mode's requirement. `Scan_mode` objects selectively include `scan_element` and `scan_chain_family` objects to create the best chain allocation building block.

Examples

Example 1

Here is an example that specifies to create a scan mode `mode1` with `chain_count` of up to 100 chains.

```
> add_scan_mode mode1 -chain_count 100
```

Example 2

This example shows how to create a scan mode `mode2` with a maximum chain length of 300 scan elements per chain.

```
> add_scan_mode mode2 -chain_length 300
```

Example 3

Example 3 shows how to find sub_chains inside OCCs and to keep in separate scan chain than the remaining scan elements:

1. Using the –single_cluster_chains option:

```
>set_attribute_value [get_scan_elements -below_instances *_occ_* ] \  
    -name cluster_name -value is_occ  
>set edt_instance [get_instances -of_icl_instances \  
    [get_icl_instances -filter tessent_instrument_type==mentor::edt]]  
>add_scan_mode edt -single_cluster_chains on -edt_instance $edt_instance
```

2. Using chain families:

```
>register_attribute -name is_occ -obj_type scan_element -value_type boolean  
>set_attribute_value [get_scan_elements -below_instances *_occ_* ] -name is_occ  
>create_scan_chain_family occ_chain -include_elements \  
    [get_scan_elements -filter is_occ ]  
>create_scan_chain_family not_occ -include_elements \  
    [get_scan_elements -filter !is_occ ]  
>set edt_instance [get_instances -of_icl_instances \  
    [get_icl_instances -filter tessent_instrument_type==mentor::edt]]  
>add_scan_mode edt -include_chain_families {occ_chain not_occ} \  
    -edt_instance $edt_instance
```

Unwrapped Cores Versus Wrapped Cores

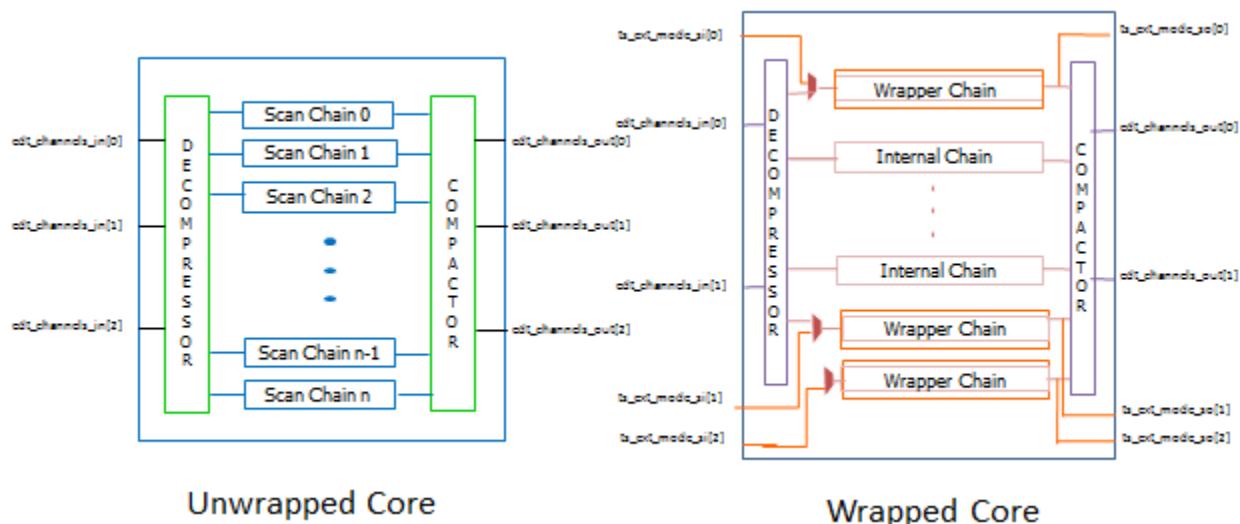
Unwrapped cores are physical blocks that do not contain wrapper cells. Wrapped cores are physical blocks that contain wrapper cells. Wrapped cores are used for the RTL and scan DFT insertion flow for hierarchical designs.

For more information, refer to “[Tessent Shell Flow for Hierarchical Designs](#)” in the *Tessent Shell User’s Manual*.

For unwrapped cores, scan insertion is performed as a single view of the core. For wrapped cores, scan insertion is performed for two views of the core: internal mode and external mode. During scan insertion, the wrapper cells get stitched into separate scan chains called wrapper chains. The logic inside the wrapper chains is called the internal mode of the core, and the logic outside the wrapper chains is called external mode of the core.

The following figure shows an unwrapped core and a wrapped core.

Figure 5-7. Unwrapped Core Versus Wrapped Core



Wrapper cells may be of type dedicated cell or shared wrapper cell with functional flop.

Use a wrapped core if the ATPG patterns that are generated are to be retargeted at the next parent level where the core is instantiated. Wrapped cores are also necessary when a number of wrapped cores are combined or grouped at the next parent level to generate ATPG patterns.

If you are not going to retarget the ATPG patterns, then an unwrapped core may be sufficient.

For details about pattern retargeting, refer to “[Scan Pattern Retargeting](#).”

Pre-Existing Scan Segments

A scan segment is a scan chain that exists inside a lower level/sub-module, and the unconnected scan input and scan output of the scan segment are at the boundary of the lower level/sub-module. Pre-existing scan segments get concatenated with other scan elements during `analyze_scan_chains` to form new scan chains.

You must specify the scan input, scan_output and scan_enable pins for each pre-existing scan segment. In addition to stitching the segment into a chain by connecting the scan_in and scan_out, the tool will also connect the scan_enable port to the appropriate signal depending on the type of scan chain that it is stitched into. In the event of an existing connection to the port, that connection will be removed and replaced as described above.

Note

- ❑ There is one exception to scan_enable processing. If the scan_segment is part of a Mentor Graphics OCC (as defined by the ICL), then the existing connection is assumed to be correct and will not be modified.

There are two methods by which these scan segment can be described.

1. Using tcd_scan:

If there are scan segment in the design, then describe these scan segment using the input tcd_scan file of the sub-module. The complete syntax of the input tcd_scan file is described in the [Scan](#) section of the *Tessent Shell Reference Manual*. This method is independent of the dofile, and just requires the tcd_scan file to be read in along with the design. This works well for IPs that come with pre-existing scan segments or for memories that have built-in bypass scan chains.

Here is an example of how to read in a TCD file:

```
>set_context dft -scan -hierarchical_scan_insertion
>read_cell_library ..library/adk.tcell_library
>read_verilog ..from_synthesis/piccpu.v
>set_design_sources -format tcd_scan -Y ..for_sub_modules -extensions tcd_scan
>set_current_design piccpu
```

Here is an example of a tcd_scan file:

```
Core(display_header_sync_10) {
    Scan {
        allow_internal_pins : 1;
        is_hard_module : 1;
        internal_scan_only : 0;
        Mode(mode1) {
            type : unwrapped;
            traceable : 1;
            ScanChain {
                length : 8;
                scan_in_clock : ~cmp_gclk;
                scan_out_clock : cmp_rclk;
                scan_in_port : ts_si[0];
                scan_out_port : ts_so[0];
            }
            ScanEn(scan_en) {
                active_polarity : all_ones;
            }
            Clock(cmp_gclk) {
                off_state : 1'b0;
            }
            Clock(cmp_rclk) {
                off_state : 1'b0;
            }
        }
    }
}
```

2. Using add_scan_segments:

The scan segment that are present in the design can be described using the [add_scan_segments](#) command.

In every design where the sub-modules with pre-existing scan segment are present, the [add_scan_segments](#) command needs to be used. A load/unload testproc file needs to be provided that aids the tool in tracing these sub chains.

Example:

```
>dofile trace.testproc
>add_scan_segments mychain1 -length 8 -on_module display_header_sync_10 \
    -si_connections ts_si[0] -so_connections ts_so[0] \
    -clock_pins {cmp_gclk cmp_rclk} -clock_update_edges {trailing leading} \
    -scan_enable_pins scan_en -scan_enable_inversion false
```

Pre-Existing Scan Chains

If there are pre-existing scan chains that are already connected to the module boundaries where scan insertion needs to be performed, then you need to connect them to the EDT IP if the EDT IP already exists in the pre-scan netlist.

If the pre-existing scan chains are not connected to the EDT IP, then the pre-existing scan chains will exist in all the scan modes specified using the `add_scan_mode` command and will also remain as uncompressed scan chains even in EDT mode. The pre-existing scan chains do not automatically get connected to the EDT IP.

If pre-existing scan chains are present in the design and you decide to connect them to the EDT IP, make sure to build the bypass inside the EDT IP for these pre-existing scan chains to be included along with the bypass (either multi chain bypass and/or single chain bypass).

Once the pre-existing scan chains are connected to the EDT IP, these chains need to be declared using `add_scan_chains` command and a test procedure file with load/unload and shift procedures is required. If the design needs to be setup in a specific state for the scan chains to be traced, then a `test_setup` procedure is required as well.

Here is an example of how to specify the pre-existing scan chains after connecting to EDT IP:

```
>add_scan_group grp1 chain_trace.testproc
>add_scan_chains -internal my_chain0 grp1 \
    corea_gate1_tessent_edt_c1_inst/edt_scan_in[0] \
    corea_gate1_tessent_edt_c1_inst/edt_scan_out[0]
>add_scan_chains -internal my_chain1 grp1 \
    corea_gate1_tessent_edt_c1_inst/edt_scan_in[1] \
    corea_gate1_tessent_edt_c1_inst/edt_scan_out[1]
```

Preparation for Test Structure Insertion

You must complete a series of steps to prepare for the insertion of test structures into your design. When the tool invokes, you are in setup mode by default. All of the setup steps shown in the following subsections occur in setup mode.

Scan Cell and Scan Output Mapping.....	159
Test Logic Insertion.....	160
User Clock Signals.....	163
How to Specify Existing Scan Information	164
How to Handle Existing Boundary Scan Circuitry	166
How to Run Rules Checking.....	166
Setup for a Basic Scan Insertion Run	167
Setup for Wrapper Chain Identification.....	168
Wrapper Chains Generation	168
Wrapper Cells Analysis	170
Clock Selection for Dedicated Wrapper Cells	174
How to Constrain Input Partition Pins	175
How to Mask Output Partition Pins	175
Child OCC Schematic	175
How to Manually Include and Exclude Cells for Scan.....	176
How to Report Scannability Information	178
Automatic Recognition of Existing Scan Chains In Cell Library Models	179
Automatic Recognition of Existing Shift Registers.....	182
Scan Cell Identification and Reporting	185

Scan Cell and Scan Output Mapping

The tool uses the default mapping defined within the ATPG library. Each scan model in the library describes how the non-scan models map to scan model in the `scan_definition` section of the model.

For more information on the default mapping of the library model, refer to “[Defining Cell Information](#)” in the *Tessent Cell Library Manual*.

You have the option to customize the scan cell and the cell’s scan output mapping behavior. You can change the mapping for an individual instance, all instances under a hierarchical instance, all instances in all occurrences of a module in the design, or all occurrences of the model in the entire design, using the [set_cell_model_mapping](#) command.

For example, you can map the fd1 nonscan model to the fd1s scan model for all occurrences of the model in the design by entering:

```
set_cell_model_mapping -new_model fd1s -model fd1
```

The following example maps the fd1 nonscan model to the fd1s scan model for all matching instances in the “counter” module and for all occurrences of that module in the design:

```
set_cell_model_mapping -new_model fd1s -model fd1 -module counter
```

Additionally, you can change the scan output pin of the scan model in the same manner as the scan cell. Within the scan_definition section of the model, the scan_out attribute defines which pin is used as the scan output pin. During the scan stitching process, the tool selects the output pin based on the lowest fanout count of each of the possible pins. If you have a preference as to which pin to use for a particular model or instance, you can also issue the [set_cell_model_mapping](#) command to define that pin.

For example, if you want to use “qn” instead of “q” for all occurrences of the fd1s scan model in the design, enter:

```
set_cell_model_mapping -new_model fd1s -output qn
```

Test Logic Insertion

Test logic is circuitry that the tool adds to improve the testability of a design. If so enabled, the tool inserts test logic during scan insertion based on the analysis performed during the design rules and scannability checking processes.

Test logic provides a useful solution to a variety of common problems. First, some designs contain uncontrollable clock circuitry; that is, internally-generated signals that can clock, set, or reset flip-flops. If these signals remain uncontrollable, the tool will not consider the sequential elements controlled by these signals scannable. Second, you might want to prevent bus contention caused by tri-state devices during scan shifting.

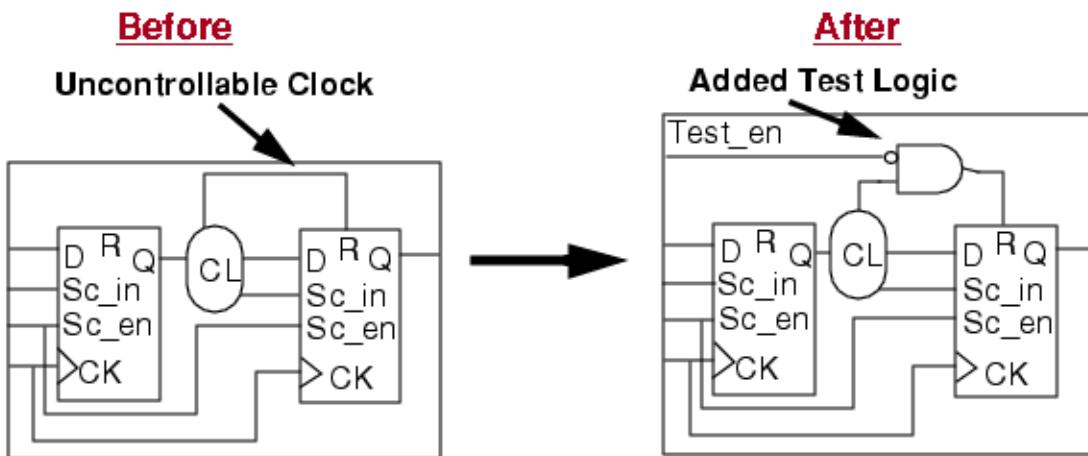
Tessent Scan can assist you in modifying your circuit for maximum controllability (and thus, maximum scannability of sequential elements) and bus contention prevention by inserting test logic circuitry at these nodes when necessary.

Note

 Tessent Scan does not attempt to add test logic to user-defined non-scan instances or models; that is, those specified by the [add_nonscan_instances](#) command.

Tessent Scan typically gates the uncontrollable circuitry with a chip-level test pin. [Figure 5-8](#) shows an example of test logic circuitry.

Figure 5-8. Test Logic Insertion



You can specify which signals to insert test logic on with [set_test_logic](#).

You can add test logic to all uncontrollable (set, reset, clock, or RAM write control) signals during the scan insertion process. By default, Tessent Scan does not add test logic. You must explicitly enable the use of test logic with this command.

In adding the test logic circuitry, the tool performs some basic optimizations in order to reduce the overall amount of test logic needed. For example, if the reset line to several flip-flops is a common internally-generated signal, the tool gates it at its source before it fans out to all the flip-flops.

Note

 You must turn the appropriate test logic on if you want the tool to consider latches as scan candidates. Refer to “[D6](#)” in the *Tessent Shell Reference Manual* for more information on scan insertion with latches.

If your design uses bidirectional pins as scan I/Os, the tool controls the scan direction for the bidirectional pins for correct shift operation.

This can be specified with [set_bidi_gating](#). If the enable signal of the bidirectional pin is controlled by a primary input pin, then the tool adds a “force” statement for the enable pin in the new load_unload procedure to enable/disable the correct direction. Otherwise, the tool inserts gating logic to control the enable line. The gate added to the bidirectional enable line is either a 2-input AND or OR. By default, no bidirectional gating is inserted and you must make sure that the inserted scan chains function properly by sensitizing the enable lines of any bidirectional ports in the scan path.

There are four possible cases between the scan direction and the active values of a tri-state driver, as shown in [Table 5-1](#). The second input of the gate is controlled from the scan_enable

signal, which might be inverted. You will need to specify AND and OR models through the `cell_type` keyword in the ATPG library or use the [add_cell_models](#) command.

Table 5-1. Scan Direction and Active Values

Driver	Scan Direction	Gate Type
active high	input	AND
active high	output	OR
active low	input	OR
active low	output	AND

If you enable the gating of bidirectional pins, the tool controls all bidirectional pins. The bidirectional pins not used as scan I/Os are put into input mode (Z state) during scan shifting by either “force” statements in the new `load_unload` procedure or by using gating logic.

The tool adds a “force Z” statement in the test procedure file for the output of the bidirectional pin if it is used as scan output pin. This ensures that the bus is not driven by the tristate drivers of both bidirectional pin and the tester at the same time.

How to Specify the Models to use for Test Logic

When adding test logic circuitry, the tool uses a number of gates from the library. The `cell_type` attribute in the library model descriptions tells the tool which components are available for use as test logic.

If the library does not contain this information, you can instead specify which library models to use with the [add_cell_models](#) command.

Alternately, you can use the `dft_cell_selection` keyword to specify the library models. For more information about `dft_cell_selection`, see “Cell Selection” in the “How to Define a Cell Library” section of the Tesson Cell Library Manual.

Note

 Tesson Scan treats any Verilog module enclosed in `'celldesign` / `'endcelldesign` directives as a library cell and prevents any logic changes to these modules.

Issues Concerning Test Logic Insertion and Test Clocks

Because inserting test logic actually adds circuitry to the design, you should first try to increase circuit controllability using other options. These options might include such things as performing proper circuit setup or, potentially, adding test points to the circuit prior to scan. Additionally, you should re-optimize a design to ensure that fanout resulting from test logic is correctly compensated and passes electrical rules checks.

In some cases, inserting test logic requires the addition of multiple test clocks. Analysis run during DRC determines how many test clocks the tool needs to insert. The [report_scan_chains](#) command reports the test clock pins used in the scan chains.

Related Test Logic Commands

Use these commands to delete and report cell models and report test added logic:

- [delete_cell_models](#) — Deletes the information specified by the [add_cell_models](#) command.
- [report_cell_models](#) — Displays a list of library cell models to be used for adding test logic circuitry.
- [report_test_logic](#) — Displays a list of test logic added during scan insertion.

User Clock Signals

Tessent Scan must be aware of the circuit clocks to determine which sequential elements are eligible for scan. The tool considers clocks to be any signals that have the ability to alter the state of a sequential device (such as system clocks, sets, and resets).

Therefore, you need to tell the tool about these “clock signals” by adding them to the clock list with the [add_clocks](#) command.

With this command, you must specify the off-state for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off state is the clock value prior to the clock’s capturing transition.

For example, you might have two system clocks, called “clk1” and “clk2”, whose off-states are 0 and a global reset line called “rst_1” whose off-state is 1 in your circuit. You can specify these as clock lines as follows:

```
SETUP> add_clocks 0 clk1 clk2
SETUP> add_clocks 1 rst_1
```

You can specify multiple clock pins with the same command if they have the same off-state. You must define clock pins prior to entering analysis mode. Otherwise, none of the non-scan sequential elements will successfully pass through scannability checks. Although you can still enter analysis mode without specifying the clocks, the tool will not be able to convert elements that the unspecified clocks control.

Note

 If you are unsure of the clocks within a design, you can use the [analyze_control_signals](#) command to identify and then define all the clocks. It also defines the other control signals in the design.

- [delete_clocks](#) — Deletes primary input pins from the clock list.
- [report_clocks](#) — Displays a list of all clocks.
- [report_primary_inputs](#) — Displays a list of primary inputs.
- [write_primary_inputs](#) — Writes a list of primary inputs to a file.

How to Specify Existing Scan Information

You may have a design that already contains some existing internal scan circuitry. For example, one block of your design may be reused from another design, and thus, may already contain its own scan chain. You may also have used a third-party tool to insert scan before invoking the tool. If either of these is your situation, there are several ways in which you may want to handle the existing scan data, including, leaving the existing scan alone or adding additional scan circuitry.

If your design contains existing scan chains that you want to use, you must specify this information to the tool while you are in setup mode; that is, before design rules checking. If you do not specify existing scan circuitry, the tool treats all the scan cells as non-scan cells and performs non-scan cell checks on them to determine if they are scan candidates.

Common methodologies for handling existing scan circuitry include:

- Use [Pre-Existing Scan Segments](#).
- Use [Pre-Existing Scan Chains](#).

The remainder of this section includes details related to these methodologies.

How to Specify Existing Scan Groups

A scan chain group consists of a set of scan chains that are controlled through the same procedures; that is, the same test procedure file controls the operation of all chains in the group.

If your design contains existing scan chains, you must specify the scan group to which they belong, as well as the test procedure file that controls the group. To specify an existing scan group, use the [add_scan_groups](#) command.

For example, you can specify a group name of “group1” controlled by the test procedure file “group1.test_proc” as follows:

SETUP> add_scan_groups group1 group1.test_proc

For information on creating test procedure files, refer to “[Test Procedure Files](#)” on page 96.

How to Specify Existing Scan Chains

After specifying the existing scan group, you need to communicate to the tool which scan chains belong to this group. To specify existing scan chains, use the [add_scan_chains](#) command.

You need to specify the scan chain name, the scan group to which it belongs, and the primary input and output pins of the scan chain. For example, assume your design has two existing scan chains, “chain1” and “chain2”, that are part of “group1”. The scan input and output pins of chain1 are “sc_in1” and “sc_out1”, and the scan input and output pins of chain2 are “sc_in2” and “sc_out2”, respectively. You can specify this information as follows:

```
SETUP> add_scan_chains chain1 group1 sc_in1 sc_out1
```

```
SETUP> add_scan_chains chain2 group1 sc_in2 sc_out2
```

How to Specify Existing Scan Cells

If the design has existing scan cells that are not stitched together in a scan chain, you need to identify these cells for Tessent Scan. (You cannot define scan chains if the scan cells are not stitched together.) This situation can occur if scan cells are used in the functional design to provide actual timing.

Additionally, defining these existing scan cells prevents the tool from performing possibly undesirable default actions, such as scan cell mapping and generation of unnecessary mux gates.

New Scan Cell Mapping

If you have existing scan cells, you must identify them as such to prevent the tool from classifying them as replaceable by new scan cells. One or the other of the following criteria is necessary for the tool to identify existing scan cells and not map them to new scan cells:

- **Criteria (a)** — Declare the “data_in = <port_name>” in the scan_definition section of the scan cell’s model in the ATPG library.

If you have a hierarchy of scan cell definitions, where one library cell can have another library cell as its scan version, using the data_in declaration in a model causes the tool to consider that model as the end of the scan definition hierarchy, so that no mapping of instances of that model will occur.

Note

 It is not recommended that you create a hierarchy of scan cell model definitions. If, for instance, your data_in declaration is in the scan_definitions section of the third model in the definitions hierarchy, but the tool encounters an instance of the first model in the hierarchy, it will replace the first model with the second model in the hierarchy, not the desired third model. If you have such a hierarchy, you can use the [set_cell_model_mapping](#) command to point to the desired model. [set_cell_model_mapping](#) overrides the mapping defined in the library model.

- **Criteria (b)** — The scan enable port of the instance of the cell model must be either dangling or tied (0 or 1) or pre-connected to a global scan enable pin(s). In addition, the scan input port must be dangling or tied or connected to the cell’s scan output port as a self loop or a self loop with (multiple) buffers or inverters.

Dangling implies that there are no connected fan-ins from other pins except tied pins or tied nets. To identify an existing (global) scan enable, use the [set_scan_signals](#) command.

Issue the [set_scan_signals](#) command before the [insert_test_logic](#) command.

Additional Mux Gates

Another consequence of not specifying existing scan cells is the addition of unnecessary multiplexers, creating an undesirable area and routing overhead.

If you use criteria (a) as the means of preventing scan cell mapping, the tool also checks the scan enable and scan in ports. If either one is driven by system logic, then the tool inserts a new mux gate before the data input and uses it as a mux in front of the preexisting scan cell. (This is only for mux-DFF scan; this mux is not inserted for LSSD or clocked_scan types of scan.)

If you use a combination of criteria (a) and (b), or just criteria (b), as the means of preventing scan cell mapping, the tool will not insert a mux gate before the data input.

Once the tool can identify existing scan cells, they can be stitched into scan chains in the normal scan insertion process.

How to Handle Existing Boundary Scan Circuitry

If your design contains boundary scan circuitry and existing internal scan circuitry, you must integrate the boundary scan circuitry with the internal test circuitry. You must ensure proper connection of the scan chains’ `scan_in` and `scan_out` ports to the TAP controller.

How to Run Rules Checking

Tessent Scan performs model flattening, learning analysis, rules checking, and scannability checking when you try to exit the setup system mode.

“[Common Tool Terminology and Concepts](#)” on page 87 explains these processes in detail. If you are finished with all the setup you need to perform, you can change the system mode by entering the [set_system_mode](#) command as follows:

SETUP> set_system_mode analysis

You can also do change the system mode by entering the [check_design_rules](#) command as follows:

SETUP> check_design_rules

If an error occurs during the rules checking process, the application remains in setup mode, where you must correct the error. You can clearly identify and easily resolve the cause of many errors. Other errors, such as those associated with proper clock definitions and test procedure files, can be more complex. “[How to Troubleshoot Rules Violations](#)” in the *Tessent Shell Reference Manual* discusses the procedure for debugging rules violations. You can also use DFTVisualizer to visually investigate the causes of DRC violations. For more information, refer to “[DFTVisualizer](#)” in the *Tessent Shell User’s Manual*.

Setup for a Basic Scan Insertion Run

Tessent Scan’s default mode of operation is to replace all candidate non-scan cells with scan cells and then stitch them into scan chains based on a number of internal criteria including minimizing the number of clock domain crossings in the scan path, and also minimizing the routing overhead.

The actual netlist processing is triggered by the [insert_test_logic](#) command. The default operation of the [insert_test_logic](#) command is to perform scan substitution and stitching.

Setup for Wrapper Chain Identification

Tessent Scan provides the ability to improve performance and test coverage of hierarchical designs by reducing the visibility of the internal core scan chains of the design submodules. The only requirements for testing submodules at the top-level is that the logic inputs must be controllable using scan chains (that is, must be driven by registers) and the logic outputs must be observable through scan chains (that is, must be registered).

These requirements can be met using *wrapper scan chains* which are scan chains around the periphery of the submodules that connect to each input and output (except user-defined clocks, scan-related I/O pins, and user-excluded pins) of the submodules to be tested. Wrapper chains comprising the wrapper cells reachable from a submodule's inputs are treated as the Input type wrapper chains and wrapper chains comprising the wrapper cells reachable from a submodule's outputs are treated as the Output type wrapper chains. The Input and Output wrapper chains are used to provide proper test coverage of hierarchical designs during the INTEST and EXTEST modes.

In INTEST mode, all inputs to submodules are controllable using the Input wrapper scan chains and all outputs are observable through the Output wrapper scan chains. This provides the ability to independently generate a complete set of ATPG patterns for submodules.

In EXTEST mode, all outputs from submodules are controllable using the Output wrapper scan chains and all inputs are observable through the Input wrapper scan chains. This provides the ability to test/optimize the logic surrounding the submodules at the top-level without requiring internal visibility into the submodules (that is, core scan chains are not needed).

Wrapper Chains Generation	168
Wrapper Cells Analysis.....	170
Clock Selection for Dedicated Wrapper Cells.....	174
How to Constrain Input Partition Pins	175
How to Mask Output Partition Pins	175

Wrapper Chains Generation

Tessent Scan generates separate Input and Output wrapper chains as described in this section.

The Input and Output wrapper chains are generated based on the following conditions:

- The scan cells comprising the wrapper scan chains are identified as the Input and Output wrapper cells using the `set_wrapper_analysis_options` and `analyze_wrapper_cells` commands and are stitched into the separate Input and Output wrapper chains using the `insert_test_logic` command.

- When you issue the `analyze_wrapper_cells` command, Tessent Scan performs the wrapper cells analysis and identifies the Input and Output wrapper cells, which will be stitched into the separate Input and Output wrapper chains.
- If the `set_wrapper_chains` command was not issued, the `analyze_wrapper_cells` command will perform the default wrapper cells identification similar to issuing the `set_wrapper_chains` command with no arguments.
- If the `analyze_wrapper_cells` command is issued in conjunction with the `set_dedicated_wrapper_cell_options` command, the dedicated wrapper cells added to the primary Inputs will be stitched into the appropriate Input wrapper chains, while the dedicated wrapper cells added to the primary Outputs will be stitched into the appropriate Output wrapper chains.
- When the `set_dedicated_wrapper_cell_options` command is issued and the `analyze_wrapper_cells` command is not issued, the dedicated wrapper cells added to the primary I/Os are treated as Core cells and are stitched into the appropriate Core scan chains.
 - Dedicated wrapper cells are normally analyzed and inserted into scan chains when you issue the `insert_test_logic` command.
 - For backward compatibility, the `report_wrapper_cells` command will trigger the dedicated wrapper cells analysis when the `set_dedicated_wrapper_cell_options` command was issued, but the `analyze_wrapper_cells` command was not.
- If you issue the `insert_test_logic` command after the `set_wrapper_chains` command, but have not issued the `analyze_wrapper_cells` command, you will get an error message stating that the wrapper chains analysis is required prior to insertion.

Error: Wrapper cells information is out of date:

```
use "analyze_wrapper_cells" to re-analyze wrapper cells prior to
insertion.
```

- The following commands can affect the I/O registration of dedicated wrapper cells and the identification of shared wrapper cells:
 - `add_nonscan_instances`, `delete_nonscan_instances`
 - `add_scan_instances`, `delete_scan_instances`

If you use any of these commands in conjunction with the `set_dedicated_wrapper_cell_options` and `set_wrapper_chains` commands, you will get the following warning message:

```
Warning: Wrapper cells information is out of date:
use "analyze_wrapper_cells" to re-analyze wrapper cells prior to
insertion.
```

Wrapper Cells Analysis

The analyze_wrapper_cells command is used to identify a combination of existing (shared) scan cells, and dedicated wrapper cells that provide complete observability of primary inputs, and complete controllability of primary outputs during top level testing (extest mode).

In addition, the input wrapper cells completely isolate the core from the primary inputs during internal testing (intest mode), and the output wrapper cells provide observability of the core's internal nodes.

The scan enable connection for the scan cells that are identified as input wrapper cells will be combined with the intest enable signal to prevent these cells from capturing any values coming from the primary inputs during intest. The scan_enable connection for the scan cells that are identified as output wrapper cells will be combined with the extest enable signal to prevent these cells from capturing data in extest mode. The behavior during each mode is shown below:

mode	core_cells	input_wrapper_cells	output_wrapper_cells
unwrapped	allowed to capture	allowed to capture	allowed to capture
intest mode	allowed to capture	forced to stay in shift	allowed to capture
extest mode	N/A	allowed to capture	forced to stay in shift

The following paragraphs provide a high level description of the wrapper analysis algorithm in order to better understand when the tool will choose to use dedicated wrapper cells versus shared wrapper cells for primary inputs as well as primary outputs.

The tool begins by tracing backward from the primary output pins. If the number of scan cells that are encountered during this back-trace exceeds a specific threshold, then a dedicated wrapper cell is inserted. In addition, the set and reset pins of an output wrapper cell must be controllable during external test. This means the tool needs to trace back from the set and reset ports of the potential shared output wrapper cells to see if there are any more scan cells that could impact these ports, and consider them as additional output wrapper cells. If the number of combinationally connected scan cells, plus the scan cells in the feedback path of the set and reset ports exceeds some threshold, then it will once again resort to inserting a dedicated wrapper cell.

Finally, a similar analysis is performed for each primary input. If the number of scan cells that is reached by a specific primary input exceeds the user specified threshold, then a dedicated input wrapper cell is used. For input wrapper cells, the tool must have known values at all of the ports of the scan cell. This means it must trace back from clock and data ports in addition to the set and reset ports looking for any scan cells that must also become wrapper cells. If the number of combinationally connected scan cells, and the number of scan cells in the feedback path of the set, reset, and clock line exceeds the user specified threshold, then a dedicated wrapper cell will be inserted. Note that the cells in the feedback path must remain fully controllable during extest, and they most likely capture values from non-wrapper (core) scan cells. This means the tool must mark these cells as output wrapper cells so they will be connected to an internally generated scan enable signal that remains high during capture.

All of the tracing that occurs is based on tracing through unblocked paths. This means that pin constraints can have a huge impact on this analysis. For example, if the set or reset port of an output wrapper cell maintains a constant 0 because of a pin constraint, then no backward tracing is necessary.

In order to facilitate debugging, a simulation context called "wrapper_analysis" is available. You can use the [set_current_simulation_context](#) command to view these values with [report_gates](#) (after doing "set_gate_report simulation_context"), or using the Visualizer (also after setting the current simulation context as the data source).

You can also force the insertion of a dedicated wrapper cell at any port using the [set_dedicated_wrapper_cell_options](#) command. In addition, there are several parameters that control the forward and backward tracing that can be modified using the [set_wrapper_analysis_options](#) command.

There are several other conditions (besides exceeding the threshold) that can trigger the insertion of a dedicated wrapper cell. Some of these are:

- The identified sequential cell is part of an existing scan chain.
- “blackbox” is encountered during the wrapper cells identification for an I/O.
- There were no sequential cells identified during the wrapper cells identification for an I/O. (Note: wrapper cells are not inserted for a straight pass-through connection).
- The currently identified sequential cell was already identified as wrapper cell for an I/O of the opposite direction than the current I/O.

The [set_dedicated_wrapper_cell_options](#) command can be used to specify the library models, other than the default, to be used for the dedicated wrapper cells and to provide an additional info regarding which I/Os should be registered with the dedicated wrapper cells. The I/Os specified in the “on -ports list” of the [set_dedicated_wrapper_cell_options](#) command will be automatically registered with the dedicated wrapper cells, while the I/Os specified in the “off -ports list” will be excluded from the automatic registration with the dedicated wrapper cells when they have failed the identification with the shared wrapper cells (in this case the failed I/Os will be neither identified with shared wrapper cells nor registered with the dedicated wrapper cells).

Examples

Following are examples showing automatic registration of the failed I/Os, and controlling the I/Os to be registered with the “on/off -ports lists” of [set_dedicated_wrapper_cell_options](#).

Example 1

```
set_wrapper_analysis_options -input_fanout_libcell_levels_threshold 1 \
                             -output_fanin_libcell_levels_threshold 1
set_system_mode analysis
analyze_wrapper_cells
report_wrapper_cells -verbose
```

Primary I/O	MAX Logic Level {1/1}	# Wrapper Cells Identified {256/256}	# Internal Feedback Gates	Wrapper Cells Identified	Wrapper Chain Type	New Registration	Reason For Failed Cell Added
out4 (O)	0	1		flop5	Output clk	No	--
out1 (O)	0	1		flop4	Output clk	No	--
out2 (O)	0	0		new cell	Output clk	Yes	Combinational Logic Only
in1 (I)	0	2/0		flop1	Input clk	No	--
				flop2	Input clk		
in2 (I)	0	0		new cell	Input clk	Yes	Output Wrapper Cell
in3 (I)	0	0		new cell	Input clk	Yes	MAX Logic Level
in4 (I)	0	0		new cell	Input clk	Yes	Combinational Logic Only
in5 (I)	0	0		new cell	Input test_clk	Yes	MAX Logic Level
in6 (I)	0	0		new cell	Input test_clk	Yes	Combinational Logic Only

The wrapper cells identification has failed for the following PIs and POs.
These PIs and POs will be I/O Registered unless excluded from the registration.

in2	Input	Encountered a sequential cell that was already identified as an output wrapper cell: flop4
in3	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells
in4	Input	Encountered combinational logic only before reaching PO: out2
in5	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells
in6	Input	Encountered combinational logic only before reaching PO: out2
out2	Output	Encountered combinational logic only before reaching PIs: in1

These PIs and POs will be excluded from the I/O Registration unless they were explicitly included.

in7	Input	Encountered no sequential cells and combinational logic other than buffers/inverters before reaching PO: out3
in8	Input	Encountered no combinational logic and just 1 sequential cell before reaching PO: out4
out3	Output	Encountered no sequential cells and combinational logic other than buffers/inverters before reaching PI: in7

Example 2

Pins in2 and in3 are excluded from the automatic registration.

```
set_wrapper_analysis_options -input_fanout_libcell_levels_threshold 1 \
-output_fanin_libcell_levels_threshold 1
set_dedicated_wrapper_cell_options off -ports in2
set_dedicated_wrapper_cell_options off -ports in3
set_system_mode analysis
analyze_wrapper_cells
report_wrapper_cells -verbose
```

Primary I/O	Max Logic Level	# Wrapper Cells Identified [256/256]	# Internal Gates	Wrapper Cells Identified	Wrapper Chain Type	Clock	New Cell Added	Reason For Failed Identification
out4 (O)	0	1		flop5	Output	clk	No	--
out1 (O)	0	1		flop4	Output	clk	No	--
out2 (O)	0	0		new cell	Output	clk	Yes	Combinational Logic Only
in1 (I)	0	2/0		flop1	Input	clk	No	--
				flop2	Input	clk		
in4 (I)	0	0		new cell	Input	clk	Yes	Combinational Logic Only
in5 (I)	0	0		new cell	Input	test_clk	Yes	Max Logic Level
in6 (I)	0	0		new cell	Input	test_clk	Yes	Combinational Logic Only

The wrapper cells identification has failed for the following PIs and POs.

These PIs and POs will be I/O Registered unless excluded from the registration.

in2	Input	Encountered a sequential cell that was already identified as an output wrapper cell: flop4 This PI was explicitly excluded from the I/O Registration.
in3	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells This PI was explicitly excluded from the I/O Registration.
in4	Input	Encountered combinational logic only before reaching PO: out2
in5	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells
in6	Input	Encountered combinational logic only before reaching PO: out2
out2	Output	Encountered combinational logic only before reaching PI: in4

These PIs and POs will be excluded from the I/O Registration unless they were explicitly included.

in7	Input	Encountered no sequential cells and combinational logic other than buffers/inverters before reaching PO: out3
in8	Input	Encountered no combinational logic and just 1 sequential cell before reaching PO: out4
out3	Output	Encountered no sequential cells and combinational logic other than buffers/inverters before reaching PI: in7

Example 3

Pins in7 and out4 are explicitly specified to be registered.

```
set_wrapper_analysis_options -input_fanout_libcell_levels_threshold 1 \
                             -output_fanin_libcell_levels_threshold 1
set_dedicated_wrapper_cell_options on -ports in7
set_dedicated_wrapper_cell_options on -ports out4
set_system_mode analysis
analyze_wrapper_cells
report_wrapper_cells -verbose
```

Primary I/O Post	Max Level [L/H]	# Wrapper Cells Identified (256/256)	# Internal Feedback Gates	Wrapper Cells Identified	Wrapper Chain Type	New Cell Added	Reason for Failed Identification
out1 (O)	0	1		flop4	Output clk	No	--
out4 (O)	0	0		new cell	Output clk	Yes	--
out2 (O)	0	0		new cell	Output clk	Yes	Combinational Logic Only
in8 (I)	0	1/0		flop5	Input clk	No	--
in1 (I)	0	2/0		flop1	Input clk	No	--
				flop2	Input clk	No	--
in7 (I)	0	0		new cell	Input test_clk	Yes	--
in2 (I)	0	0		new cell	Input clk	Yes	Output Wrapper Cell
in3 (I)	0	0		new cell	Input clk	Yes	MAX Logic Level
in4 (I)	0	0		new cell	Input clk	Yes	Combinational Logic Only
in5 (I)	0	0		new cell	Input test_clk	Yes	MAX Logic Level
in6 (I)	0	0		new cell	Input test_clk	Yes	Combinational Logic Only

The wrapper cells identification has failed for the following PIs and POs.
 These PIs and POs will be I/O Registered unless excluded from the registration.

in2	Input	Encountered a sequential cell that was already identified as an output wrapper cell: flop4
in3	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells
in4	Input	Encountered combinational logic only before reaching PO: out2
in5	Input	Exceeded the limit, 1, on the number of combinational logic levels between this PI and the first level of sequential cells
in6	Input	Encountered combinational logic only before reaching PO: out2
out2	Output	Encountered combinational logic only before reaching PI: in7

These PIs and POs will be excluded from the I/O Registration unless they were explicitly included.

out3	Output	Encountered no sequential cells and combinational logic other than buffers/inverters before reaching PI: in7
------	--------	--

Clock Selection for Dedicated Wrapper Cells

If you do not explicitly specify the clock for a dedicated wrapper cell, the tool first selects the appropriate clock domain based on the top level clocks that control the flops in the fanin and/or fanout.

During the clock domain selection, the tool will ignore any flops that are not either already existing scan cells, or targeted for scan insertion. In addition, the initial search will be limited to flops that are in the same power_domain_island as the test point location. When only a single clock domain is found in the fanout or fanin, then that domain will be used. Otherwise, the most referenced clock domain in the fanout of the control point (or fanin of the observe point) will be used.

Next, it chooses a connection point as described below:

- It connects to the clock port of one of the flops in the fanout (for an input wrapper cell), or fanin (for an output wrapper cell). If the clock port of the connected scan cell is not

directly accessible, then it will trace through sensitized paths in the clock network to find the closest possible connection point.

- If no valid connection point is found as described above, it tries to connect to the identified clock at the closest module input port moving up the hierarchy.
- Otherwise, it connects to the clock source.

How to Constrain Input Partition Pins

Input partition pins are block input pins that you cannot directly control from chip-level primary inputs.

Referring to [Figure 2-3](#) on page 36 and [Figure 2-4](#) on page 37 , the input partition pins are those inputs that come into Block A from Block B. Because these are uncontrollable inputs, you must constrain them to an X value using the [add_input_constraints](#) command.

How to Mask Output Partition Pins

Output partition pins are block output pins that you cannot directly observe from chip-level primary outputs.

Referring to [Figure 2-3](#) on page 36 and [Figure 2-4](#) on page 37, the output partition pins are those outputs that go to Block B and Block C. Because these are unobservable outputs, you must mask them with the [add_output_masks](#) command.

To ensure that masked primary outputs drive inactive values during the testing of other partitions, you can specify that the primary outputs hold a 0 or 1 value during test mode. Special cells called output hold-0 or output hold-1 wrapper cells serve this purpose. By default, the tool uses regular output wrapper cells.

Child OCC Schematic

The Child OCC is one of the three OCC types. It is used for clock chopping control. It also provides clock gating by creating a clock gate enable signal instead of a single clock gater. It enables layout tools to replicate clock gaters during layout as needed.

The child OCC allows the clock to go through when scan_en is high. When scan_en is low, it allows a programmable constant set of clock pulses to go through. A parent OCC typically feeds the clock input of the child OCC.

Two schematics for the child OCC are shown. One is with clock gater and the other without. [Figure 5-9](#) shows the child on-chip controller with the clock gater.

Figure 5-9. Child On-Chip Controller Logic Schematic

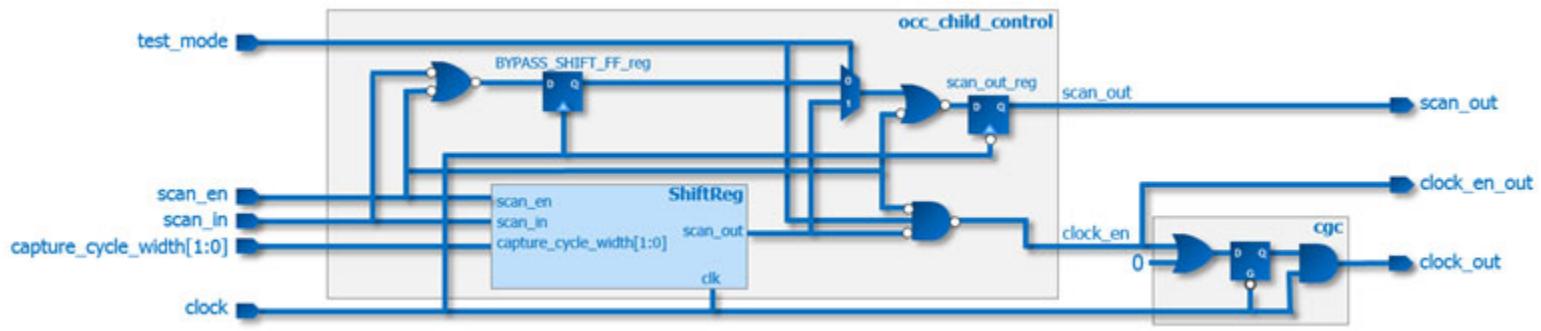
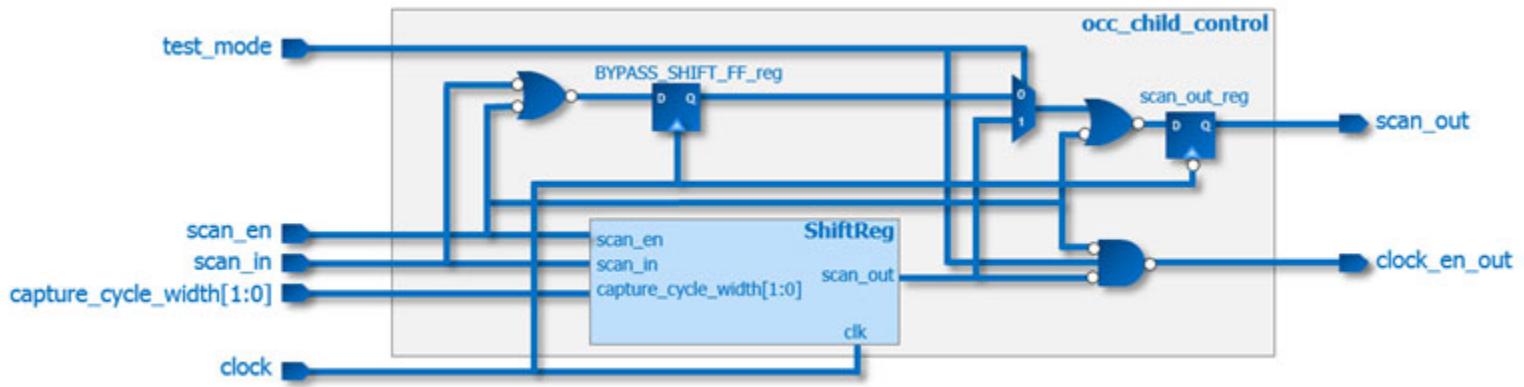


Figure 5-10 shows the child on-chip controller without the clock gater.

Figure 5-10. Child On-Chip Controller Logic Schematic, no Clock-Gater



How to Manually Include and Exclude Cells for Scan

Regardless of what type of scan you want to insert, you can manually specify instances or models to either convert or not convert to scan. Tessent Scan uses lists of scan cell candidates and non-scan cells when it selects which sequential elements to convert to scan. You can add specific instances or models to either of these lists. When you manually specify instances or models to be in these lists, these instances are called *user-class* instances. *System-class* instances are those Tessent Scan selects. The following subsections describe how you accomplish this.

Cells Without Scan Replacements

When Tessent Scan switches from setup to analysis mode, it issues warnings when it encounters sequential elements that have no corresponding scan equivalents. Tessent Scan treats elements without scan replacements as non-scan models and automatically adds them as system-class elements to the non-scan model list.

You can display the non-scan model list using the [report_nonscan_models](#) or [report_scan_elements](#) command.

In many cases, a sequential element may not have a scan equivalent of the currently selected scan type. For example, a cell may have an equivalent mux-DFF scan cell but not an equivalent LSSD scan cell. If you set the scan type to LSSD, Tesson Scan places these models in the non-scan model list. However, if you change the scan type to mux-DFF, Tesson Scan updates the non-scan model list, in this case removing the models from the non-scan model list.

How to Specify Non-Scan Components

Tesson Scan keeps a list of which components it must exclude from scan identification and replacement.

To exclude particular instances from the scan identification process, you use the [add_nonscan_instances](#) command.

For example, you can specify that I\$155/I\$117 and /I\$155/I\$37 are sequential instances you do not want converted to scan cells as follows:

```
SETUP> add_nonscan_instances /I$155/I$117 /I$155/I$37
```

Another method of eliminating some components from consideration for scan cell conversion is to specify that certain models should not be converted to scan. To exclude all instances of a particular model type, you can use the [add_nonscan_instances](#) command.

For example, the following command would exclude all instances of the dff_3 and dff_4 components from scan cell conversion.

```
SETUP> add_nonscan_instances -instances dff_3 dff_4
```

Note

 Tesson Scan automatically treats sequential models without scan equivalents as non-scan models, adding them to the non-scan model list.

How to Specify Scan Components

After you decide which specific instances or models you do not want included in the scan conversion process, you are ready to identify those sequential elements you do want converted to scan. The instances you add to the scan instance list are called user-class instances.

To include particular instances in the scan identification process, use the [add_scan_instances](#) command. This command lets you specify individual instances, hierarchical instances (for which all lower-level instances are converted to scan), or control signals (for which all instances controlled by the signals are converted to scan).

For example, the following command ensures the conversion of instances /I\$145/I\$116 and /I\$145/I\$138 to scan cells when Tesson Scan inserts scan circuitry.

SETUP> add_scan_instances /I\$145/I\$116 /I\$145/I\$138

To include all instances of a particular model type for conversion to scan, use the [add_scan_instances](#) command. For example, the following command ensures the conversion of all instances of the component models dff_1 and dff_2 to scan cells when Tesson Scan inserts scan circuitry.

SETUP> add_scan_instances -instances dff_1 dff_2

Related Scan and Nonscan Commands

In addition to commands mentioned in previous sections, the following commands are useful when working with scan and nonscan instances:

- [delete_nonscan_instances](#) — Deletes instances from the non-scan instance list.
- [delete_scan_instances](#) — Deletes instances from the scan instance list.
- [report_nonscan_models](#) — Displays the models in the non-scan instance list.
- [report_scan_elements](#) — Displays information and testability data for sequential instances.
- [report_scan_models](#) — Displays models in the scan model list.

How to Report Scannability Information

Scannability checking is a modified version of clock rules checking that determines which non-scan sequential instances to consider for scan. You may want to examine information regarding the scannability status of all the non-scan sequential instances in your design.

To display this information, you use the [report_scan_elements](#) command. This command displays the results of scannability checking for the specified non-scan instances, for either the entire design or the specified (potentially hierarchical) instance.

The following command will report a list of scan elements that are non-scannable because of failed S rules:

```
report_scan_elements [get_scan_elements -filter \
    is_non_scannable_reason==inferred_from_drc]
```

Use the following command to report a list of elements that are non-scannable (for any reason), as shown in the sample report:

```
report_scan_elements [get_scan_elements -filter is_non_scannable==true]
```

name	type	length	si_so_clocks	state	is_non_scannable_reason
/f	leaf_cell	1	+test_clk	ignored	inferred_from_drc
/f2	leaf_cell	1	+test_clk	ignored	user_specified

The report_control_signals, report_statistics, and report_scan_elements commands are useful when reporting scannability.

- [report_control_signals](#) — Displays control signal information.
- [report_statistics](#) — Displays a statistics report.
- [report_scan_elements](#) — Displays information and testability data for sequential instances.

Automatic Recognition of Existing Scan Chains In Cell Library Models

Tessent Scan automatically declares existing scan segments (subchains) in library models if they are instantiated in the design netlist and you do not define these subchains manually via the add_scan_segments command.

You must define the subchains in a library model in a specific vector format for the parser to identify them before they can be auto-declared. The format is illustrated in the following example. For more information, see “[Example Scan Definitions](#)” in the *Tessent Cell Library Manual*.

```
model lff2
(CLK, D1, D2,
D3, D4, Q1, Q2,
Q3, Q4, SI1, SI2,
SE1, SE2, SO1, SO2)
(
    input (CLK) (posedge_clock; )
    input (D1) ( )
    input (D2) ( )
    input (D3) ( )
    input (D4) ( )
    input (SI1) ( scan_in[0]; )
    input (SI2) ( scan_in[1]; )
    input (SE1) ( scan_enable[0]; )
    input (SE2) ( scan_enable[1]; )
    output (Q1) ( )
    output (Q2) ( )
    output (Q3) ( )      output (Q4) ( )
    output (SO1) ( scan_out[0]; )
    output (SO2) ( scan_out[1]; )
(
    // subchain-1
    primitive = _mux (D1, SI1, SE1, n1);
    primitive = _dff ( , , CLK, n1, Q1, n2);
    primitive = _mux (D2, n2, SE1, n3);
    primitive = _dff ( , , CLK, n3, Q2, SO1);

    // subchain-2
    primitive = _mux (D3, SI2, SE2, n4);
    primitive = _dff ( , , CLK, n4, Q3, n5);
    primitive = _dff ( , , CLK, n5, Q4, SO2);
)
)
```

Any modeling problems that will prevent shifting data in the scan path are reported during library parsing (upon executing the set_current_design command). For example:

```
// command: set_current_design
// Warning: Input pin #2 (1st is 1) of primitive instance 'm2' (primitive
// type '_mux') inside model 'lff' is floating. TieXing that input.
```

The subchains that are successfully identified by the parser, and that are not defined via the add_scan_segments command, are automatically declared as existing scan segments by the tool during SMT from SETUP to ANALYSIS. A note will be printed as below.

```
// command: set_system_mode analysis
// Note: Auto-declared an existing scan segment between the scan I/O pins
// 'SI1' and 'SO1' on the library model 'lff2'.
```

The tool does not auto-declare the subchains of a library model if you set the library model or the instances of the model nonscan (as it will be an unverified scan cell candidate).

If the library parser detects problems with the subchains of a library model, and you do not define them manually with the `add_scan_segments` command, the tool sets the library model `nonscan`. The following warning message is then printed.

```
// Warning: The tool cannot auto-declare the existing scan segments
// implied by the scan I/O pins on the library model 'lff3' due to model
// errors. The instances of this library model will be excluded from scan
// insertion. Alternatively, you can define the existing scan segments on
// this library model using the add_scan_segments command in setup mode.
```

An important fact about the auto-declared subchains is that they are not traced during DRC rule checks because their shifting ability is verified during library parsing. The subchains defined by the `add_scan_segments` command, on the other hand, are always traced during DRC to verify their shifting ability.

If you choose to override the auto-declaration of the subchains of a library model by means of the `add_scan_segments` command, you must cover all the subchains on the library model because the tool will not auto-declare subchains for the uncovered ones.

The tool compares the user-specified subchain declarations to those identified by the library parser and reports any discrepancies in the specifications as warning messages. Examples are:

```
// Warning: The existing scan segment between the scan-input pin 'SI1'
// and scan-output pin 'SO1' on library model 'lff2' may need to be
// declared via add_scan_segments command.

// Warning: The tool-identified length, 2, of the existing scan segment
// with scan-out pin 'SO2' on library model 'lff2' differs from the user-
// specified length, '7', for the existing scan segment
// 'sch1/lff2_u1/SO2'.

// Warning: The tool-identified scan-in pin, SI2, of the existing scan
// segment with scan-out pin 'SO2' on library model 'lff2' differs from
// the user-specified scan-in pin, 'SI1', for the existing scan segment
// 'sch1/lff2_u1/SO2'.
```

The requirement for the tool to auto-declare subchains on a library model is that there should be at least two subchains identified by the parser. A multi-bit scan cell is identified as a single-subchain library model by the library parser, but the subchain is not auto-declared. This allows the tool to map the multi-bit scan cell back to its non-scan equivalent counterpart during scan chain stitching, as in the case of a shift register composed of multi-bit scan cells. However, you can still define a library model subchain on a multi-bit scan cell if needed.

Currently, the tool can only identify subchains in a library model when you define a single clock (`posedge_clock` or `negedge_clock`) for all the subchains of the model. If the subchains require multiple clocks, you need to declare them manually using `add_scan_segments` command.

The auto-declared subchains are deleted when you switch from ANALYSIS mode to SETUP mode. This will also give you the opportunity to alternatively define the subchains using the

add_scan_segments command before switching back to ANALYSIS mode to allow the tool to auto-declare them.

Automatic Recognition of Existing Shift Registers

Tessent Scan automatically identifies the shift register structures in the input netlist and tries to preserve their original connections (shift orders) when they are stitched into scan chains. In this attempt, Tessent Scan converts only the first flip-flop of a shift register into a scan cell (if originally a nonscan cell), and preserves the remaining flip-flops in the shift register as nonscan (or replaces them with nonscan cells if originally scan cells).

This approach has several potential benefits such as the reduction of scan cells in the design, and therefore fewer scan path muxes, and the better locality of scan path connections due to the preservation of functional connections.

The following sections describe the process by which Tessent Scan identifies and converts scan cells.

Shift Registers Identification

Tessent Scan identifies shift registers by performing a structural backward tracing in the flattened model of the design.

The tracing starts from the data pin of the _dff primitive of a sequential cell and ends at the output pin of the _dff primitive of another sequential cell. The tracing continues only on a single sensitized path. The sensitization path of a combinational logic gate in the tracing path is checked based on the state-stability values calculated by the tool during DRC rule checking. When “-ALLOW_COMBINATIONAL_logic_between_registers ON or Off” is specified with the set_shift_register_identification command, the state-stability values are ignored during tracing. As a result, multiple-input gates (AND, OR, MUX, and so on) between sequential cells cannot be traced during shift register identification. Single-input gates (BUF and INV), on the other hand, can always be traced. This switch is Off, by default.

The identification occurs when switching to analysis mode, after DRC rule checking is completed, as indicated in the following transcript.

```
// -----
// Begin shift register identification process for 9971 sequential
// instances.
// -----
// Number of shift register flops recorded for scan insertion: 3798
// (38.09%)
// Number of shift registers recorded for scan insertion: 696
// Longest shift register has 15 flops.
// Shortest shift register has 2 flops.
// Potential number of nonscan flops to be converted to scan cells: 696
// Potential number of scan cells to be converted to nonscan flops: 25
```

Shift register identification assumes the following:

1. Flip-flops that constitute a shift register reside under the same hierarchical instance.
2. Flip-flops that constitute a shift register use the same clock signal.
3. Multiple clock edges are allowed in the shift register structure as long as no lockup cells are required (no TE-LE transitions occur). When lockup cells are required (as in LE-TE transitions), the tool breaks the shift register at this location.
4. Both nonscan and scan flip-flops are considered for identification. However, every nonscan flip-flop should have a mapping scan flip-flop model in the ATPG library and every scan flip-flop should have a mapping nonscan flip-flop model in the ATPG library. In addition, a scan flip-flop should satisfy the following requirements:
 - Its scan input pin is not functionally driven (either dangling or tied to a constant, or looped back from Q/QB output).
 - Its scan enable pin is not functionally driven, and is tied to a constant signal to sensitize the data input pin of the sequential cell such that this input is preserved as the shift path. The scan enable pin is not considered functionally driven if a global scan enable pin (defined using set_scan_signals -SEN) is the driver.
5. Shift registers with multiple branches are identified such that each branch is a separate shift register. The flip-flops on the trunk are included in one of the branches.
6. Shift registers with sequential feedback loops are identified such that the first cell of the shift register is determined by the tool either randomly or based on the availability of the scan cell in the loop.

Scan Chain Stitching

Non-scan flip-flops are replaced with their scan-equivalent flip-flops before stitching them into scan chain structures. The identified shift register flip-flops are handled as follows.

1. The first flip-flop is replaced with the equivalent scan flip-flop. If the first flip-flop is originally a scan flip-flop, it is preserved as is.
2. All remaining flip-flops are preserved as nonscan. If they are originally scan flip-flops, they are converted into nonscan flip-flops.

After performing the scan chain insertion, the report_shift_registers command may list fewer and shorter shift registers than what were originally identified by switching to analysis mode. The following lists the main reasons why originally identified shift registers can be modified by the scan chain stitching.

1. All scan cell candidates are sorted based on clock, edge, and hierarchy before stitching. The sorting process preserves the cells of the shift registers in exactly the same order as they are functionally connected. Sorted scan cell candidates are then distributed over

scan chains. When placing a shift register into a scan chain, it may be cut to a desired length to satisfy the specified/calculated scan chain length.

2. The first or the last cell in a shift register may be removed from the shift register if it is positioned to be at the clock transition between the cells in the scan chain, or positioned to be the first or last cell in the scan chain. These modifications are performed to move shift register cells from the START and STOP declarations in a scanDEF file that will be generated with a write_scan_order command. It is unlikely that the first or the last cell in a shift register will be positioned to be the first or last cell in the scan chain, as the tool when distributing scan cells to chains tries to avoid placing shift registers at the beginning or tail of a scan chain. This is likely to happen only when the percentage of shift registers are very high in the design compared to floating flops, and the tool does not have a choice to place any other potential scan cell other than a shift register at the beginning/end of a chain.

The stitching of the flip-flops inside a shift register structure is skipped but the following connections are performed:

1. The scan input pin of the first flip-flop.
2. The scan enable pin of the first flip-flop.
3. The scan output pin of the last flip-flop. The scan output pin on the last flip-flop will be determined by checking the load on the Q and QN ports.

How to Report Shift Registers

You may want to report shift registers in the design. To display this information, you use the [report_shift_registers](#) command. This command reports the identified shift registers in the design after switching to analysis mode. The tool tries to preserve the original connections inside the identified shift. For each identified shift register, this command reports the following information:

- Length.
- Hierarchical path where the shift register flip-flops reside.
- First and last flip-flop instance name unless the -verbose switch is specified in which case all flip-flops in the shift registers are reported.

Scan cells can also be reported by the [report_scan_cells](#) command after the [insert_test_logic](#) command is executed. If any shift registers are identified in the netlist, a column is added to the report. The column contains a tool-assigned shift register ID number and a cell number that indicates the order in which the flip-flops are originally connected in the shift register structures.

Scan Cell Identification and Reporting

Once you complete the proper setup, the identification process for any of the test structures is done automatically when you switch to analysis mode.

During the identification process, a number of messages may be issued about the identified structures.

To identify the dedicated and shared wrapper cells, you can use the [analyze_wrapper_cells](#) command.

Note

 If you want to start the selection process anew each time, you must use the [reset_state](#) command to clear the existing scan candidate list.

If you want a statistical report on all aspects of scan cell identification, you can enter the [report_statistics](#) command. The [report_statistics](#) command lists the total number of sequential instances, user-defined non-scan instances, user-defined scan instances, system-identified scan instances, scannable instances with test logic, and the scan instances in preexisting chains identified by the rules checker.

The [report_scan_elements](#) command displays information and testability data for sequential instances.

Multi-Mode Chains

This section provides a high-level overview of how the tool handles multi-mode chains.

Introduction to Multi-Mode Chains.....	186
Multi-Mode Gating Logic	187
Multi-Mode Scan Insertion.....	188
Chain Port Sharing	189
Population Control	190

Introduction to Multi-Mode Chains

This section describes how you can use multi-mode for chain allocation.

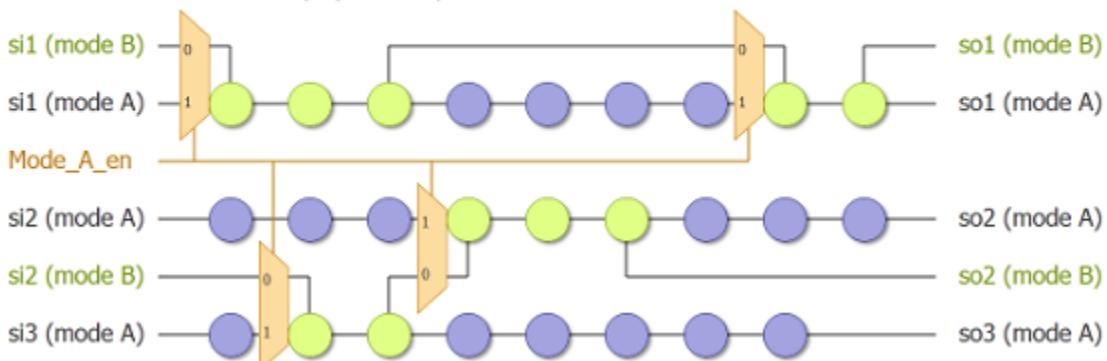
Multi-Mode scan chains are used for the following:

1. To build bypass (single chain bypass / multi chain bypass) if EDT IP is not built with one.
2. To create multiple configurations for EDT mode.
3. Are required if using Hierarchical DFT where internal and external mode scan chains are needed for wrapped cores.

Here is a simple example. The design comprises of 26 scan elements of types purple and green scan cells. In Mode A, the entire full population of scan cells are used and connected to chain_length of 9 to form 3 chains. In Mode B, only the Green type of scan elements are used to form a chain_count of 2. In [Figure 5-11](#), Mode A uses the black si1, si2 and si3 as the scan chain inputs, and its corresponding scan output ports are so1, so2 and so3. In Mode B, has just 2 chains and has the green ports labeled so1 and so2.

Figure 5-11. Simple Example of Two Modes

- 26 scan elements of types purple & green
 - Mode A: Full population, chain length 9
 - Mode B: Green population, chain count 2



Multi-Mode Gating Logic

Once the distribution of scan elements has been completed for each mode, it is possible to determine where the scan paths from each mode diverge. These scan path positions are called inflection points.

Inflection points are identified when more than one scan path source exists for a given scan element. In [Figure 5-11](#) on page 186, these are the points where the muxes are added in.

Multi-Mode Scan Insertion

One of the major advantages of this new scan insertion infrastructure is the ability to distribute scan elements and allocate scan chains for more than one mode. When more than one mode is specified, the scan insertion tool will automatically insert gating logic capable of reconfiguring the scan chains based on a selected mode.

If multi-modes are used, then you have the capability to see how the scan chains are to be stitched up in all the different modes and you can alter the specification for specific and/or all modes. You also have the ability to introspect and see how the scan chains are balanced.

In the following example, first two modes mode1 and mode 2 are added. In mode 1, chain_length of 200 is specified, so the tool will balance the scan chains, each scan chain containing about 200 flops. In mode 2, the total number of flops are divided into 20 chains.

Example:

```
>add_scan_mode mode1 -type unwrapped \
    -single_clock_domain_chains on \
    -single_clock_edge_chains on -chain_length 200

>add_scan_mode mode2 -type unwrapped \
    -chain_count 20

# Before scan gets inserted, you can analyze the different scan modes and scan chains

>analyze_scan_chains
>report_scan_chains
```

The `report_scan_chains` command here tells you how many chains are going to be in mode 1 and how many flops are going to be in each of the 20 scan chains in mode 2. If you decide that there are far too many flops in a chain and want to see how many flops get identified in a chain if 30 chains are allocated, then you can delete the specific scan mode, re-specify with the modified constraints and then `analyze_scan_chains` and `report_scan_chains`.

```
>delete_scan_modes mode2
>add_scan_mode mode2 -type unwrapped -chain_count 30

>analyze_scan_chains
>report_scan_chains
```

Now if you decide the number of flops per chain is acceptable, then stitch up the scan chains using `insert_test_logic`.

```
>insert_test_logic -write_in_tsdb On
```

The “`insert_test_logic -write_in_tsdb On`” command writes the design and TCD file into a `tsdb_outdir` directory structure so Automatic Test Pattern Generation can make use of it. This is described in detail in “[Internal Scan and Test Circuitry Insertion](#)” on page 143.

Chain Port Sharing	189
Population Control	190

Chain Port Sharing

It is possible to share the scanIn and scanOut connections among chains of different modes. Extra-gating logic is inserted on the scanOut side (after the chain's last elements) to enable each chain path during its active mode only, using a 2x1 MUX (2-mode sharing) or AND/OR logic (multi-mode sharing).

Either existing ports/pins or generated ports can be shared.

Existing Ports/Pins

Existing design ports/pins can be shared across modes by specifying them explicitly using the -si_connections and -so_connections options in the [add_scan_mode](#) or [create_scan_chain_family](#) command. For example, the following commands would cause ports MY_SCAN_IN and MY_SCAN_OUT to be shared across the modes Core and Wrapper. The core and wrapper are built-in switches for type class.

```
> add_scan_mode Core \
    -include_elements [get_scan_elements -class core] \
    -si_connections MY_SCAN_IN \
    -so_connections MY_SCAN_OUT \
> add_scan_mode Wrapper \
    -include_elements [get_scan_elements -class wrapper] \
    -si_connections MY_SCAN_IN \
    -so_connections MY_SCAN_OUT
```

Note

 Typically, you would use “add_scan_mode -type internal” or “-type external” mode to choose between the “-class core” and “-class wrapper” type scan elements.

Functional Existing Ports/Pins

Existing design ports/pins that are used functionally and specified as chain connection points automatically receive a dedicated 2x1 MUX controlled by scanEnable logic.

Generated Ports

Ports generated by the tool can be shared across modes by specifying identical formatting rules that omit the %s mode name field using the -si_port_format and -so_port_format options. For example, the following commands would cause ports EXT_SCAN_IN1 to EXT_SCAN_IN10

and EXT_SCAN_OUT1 to EXT_SCAN_OUT10 to be shared across modes Basic and Wrapper:

```
> create_scan_chain_family ext_chains -chain_count 10 \
    -si_port_format "EXT_SCAN_IN%d" \
    -so_port_format "EXT_SCAN_OUT%d" \
    -include_elements [get_scan_elements -class wrapper]
> add_scan_mode Basic -chain_count 100 \
    -include_elements [get_scan_elements] \
    -include_chain_families { ext_chains }
> add_scan_mode Wrapper -include_chain_families { ext_chains }
```

Population Control

The scan cell set considered for distribution in each scan mode may be specified explicitly.

This can be done using the add_scan_mode options “-include_elements” and/or “-include_chain_families”. Only scan_element objects with a state “usable” get added to a population; accordingly, it may be necessary to set the “active_child_scan_mode” attribute to a specific child mode value in order to include its corresponding existing segments in a population.

Default Population

As a convenience for mainstream users, if no “-include_...” option is invoked with the add_scan_mode command, then the command is implicitly called with every object defined at that point, that is:

```
> add_scan_mode Implicit_population
```

really maps to:

```
> add_scan_mode Implicit_population \
    -include_chain_families [get_scan_chain_families] \
    -include_elements [get_scan_elements]
```

This might not necessarily be the desired population for the mode, so you should be cautious when using this feature.

Population prioritization

If the same scan element is included by more than one “-include_...” option, then the following prioritization rules dictate how it gets used:

1. -include_chain_families
2. -include_elements

Scan elements included by both a scan_chain_family inclusion and by the -include_elements option will be allocated to the chains specified by the scan_chain_family object. It should be

noted that existing chains are automatically excluded from the populations considered for distribution, and hence never get concatenated to other scan elements.

Mode population

As previously implied, the population of a scan mode is defined as the union of the sub-populations specified by the options `-include_chain_families` and `-include_elements`.

Scan Insertion Flows

This section describes and discusses in detail the different flows that can be used for Scan Insertion.

Scan Insertion Flow Steps	192
Controlling Scan Insertion	193
Scan Insertion for Unwrapped Core	193
Scan Insertion for Wrapped Core	197
Scan Insertion at the Parent Level	202
Dedicated Wrapper Cell Placement After Loading UPF/CPF File	204

Scan Insertion Flow Steps

The new scan insertion capability provides better control of scan element grouping/ordering when the new scan chains are created.

There are three major steps in the flow. First, during SETUP, along with loading the design and Tesson Cell Libraries (ATPG library files are also accepted), any existing scan segments (also called subchains) and existing scan chains (from previous scan insertion pass) must also be described. The `add_scan_segments` and `add_scan_chains` commands can be used respectively for this purpose. Refer to chapter 2 for more details.

Second, during the system mode transition from SETUP to ANALYSIS, the tool will run DRC and extract all the usable library cell/leaf cell scan elements from the design at hand. At that point, all existing scan elements are defined and may be introspected using the `get_scan_elements` command.

Third, during ANALYSIS, virtual scan elements might be created as a result of some commands; for example, virtual scan elements will be created if you request the insertion of dedicated wrapper cells. Once again, all created virtual scan elements may be introspected using the `get_scan_elements` command.

Controlling Scan Insertion

The command `set_scan_insertion_options` allows you to control some parameters that affect scan insertion.

For instance, if you want all chains to have max chain length 1024, use the following command:

```
> set_scan_insertion_options -chain_length 1024
```

When adding a scan mode using the `add_scan_mode` command, you have the ability to override most of the general scan insertion settings by simply re-specifying them with a new value. The command basically takes a snapshot of the active scan insertion settings and overrides any options explicitly specified in `add_scan_mode`. If you want to create an external mode with shorter chains of length 512 for mode `ext_mode`, you can do so as follows:

```
> add_scan_mode ext_mode -type external -chain_length 512
```

Alternatively, if you want to create a single OCC chain with no length limit in `full_mode` which has a chain length of 256 for the rest of the scan element population, you can do so as follows:

```
> create_scan_chain_family occ \
    -chain_count 1 \
    -chain_length unlimited \ // overrides to no length limit!
    -include_elements [get_scan_elements -filter is_occ]

> add_scan_mode full_mode -chain_length 256 \
    -include_elements [get_scan_elements] \
    -include_chain_families occ
```

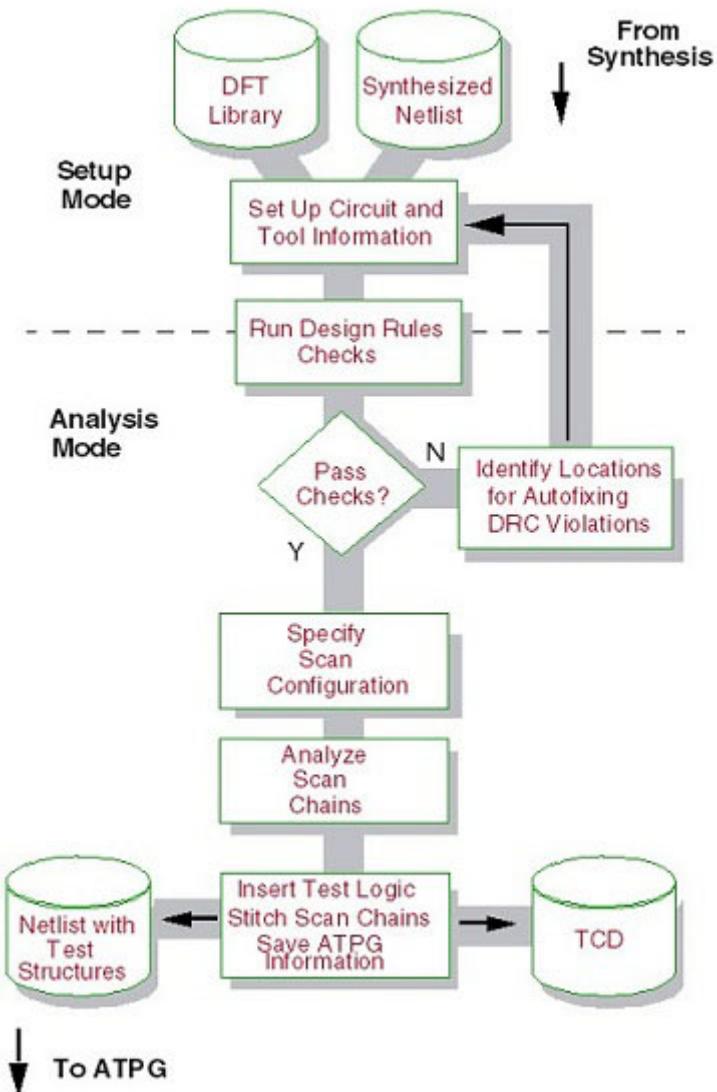
Scan Insertion for Unwrapped Core	193
Scan Insertion for Wrapped Core	197
Scan Insertion at the Parent Level.....	202
Dedicated Wrapper Cell Placement After Loading UPF/CPF File.....	204

Scan Insertion for Unwrapped Core

This section describes scan insertion for Unwrapped core.

For unwrapped core, there are no wrapper cells created thus the scan insertion flow uses the Generic scan insertion steps described in Chapter 1.

Figure 5-12. Scan Insertion Flow for Unwrapped Core



In this scan insertion flow, first the scan elements that are present in the design are identified. Next, optionally you can create `scan_chain_family` objects that instruct the tool how to allocate specific scan element sub-populations. During this phase, one or more scan modes get defined.

The population of each scan mode is defined by including `scan_element` and `scan_chain_family` objects. If you do not explicitly add at least one scan mode, the tool will infer a default mode which includes every `scan_element` and `scan_chain_family` objects.

Once the pre-scan stitched netlist has been read in, along with the required libraries, then the design gets elaborated (with `set_current_design`). If EDT IP was already inserted using `DftSpecification` and `DFTSignals` are used, then these signals are available here during scan insertion and the tool knows and uses them. For instance, if `scan enable` was declared as `DFTSignal`, then you do not need to declare the `scan enable` again. The tool knows which port

was declared as scan enable and uses it without you providing it using the set_scan_enable property. For more information about DFTSignals, refer to the [add_dft_signals](#) command in the *Tessent Shell Reference Manual*. If EDT IP was already inserted and DFTSignals are used to specify different modes then for edt_mode you can use the -edt_instance to connect to the EDT IP when specifying [add_scan_mode](#). The enable decode for this mode is automatically understood by the tool.

Note

-  DFTSignals are not mandated to be used as ScanEnable signal to use Hierarchical Scan Insertion for scan chain stitching.
-

Example 1: If EDT IP was inserted and an ICL for EDT IP exists then use:

```
>set edt_instance [get_instances -of_icl_instances \
    [get_icl_instances -filter tessent_instrument_type==mentor::edt]]
>add_scan_mode edt_mode -type unwrapped -edt_instance $edt_instance
```

Example 2: If EDT IP was inserted and an ICL for EDT IP does not exist, then use:

```
>set edt_instance [get_instance -of_modules *_edt_c1 ]
>add_scan_mode edt_mode -type unwrapped -edt_instance $edt_instance
```

During check_design_rules DRCs are run. If there are any pre-existing scan segments, then refer to “[Pre-Existing Scan Segments](#)” on page 156 on how to handle them.

If multiple modes are requested to be included and DFTSignals are used then the following example shows how to do this.

Example 3: EDT IP is not built in with bypass. DFTSignals edt_mode, multi_mode and single_mode are available and can be used. The default -type is unwrapped and hence it is not necessary to specify them.

```
>set edt_instance [get_instance -of_modules *_edt_c1 ]
>add_scan_mode int_edt_mode -edt_instance $edt_instance
>add_scan_mode multi_mode -chain_count 6
>add_scan_mode single_mode -chain_count 1
```

Full Example: The example below is broken into three major sections. The first section - Design, Library and Design Elaboration, shows how you read the Tessent Cell Library and Synthesized Verilog Gate-level design. If EDT IP was already inserted then use the [set_tsdb_output_directory](#) to point to where the tsdb database is located. Then use [read_design](#) to read all the other supporting files like ICL, PDL, TCD from the last insertion pass, and elaborate the design.

```
# Design, Library loading and design elaboration.
>set_context dft -scan -hierarchical_scan_insertion

# Sets and opens the tsdb_output directory.
>set_tsdb_output_directory ..//tsdb_outdir
>read_cell_library ../../library/tessent/adk.tcelllib
```

```
# Read synthesized netlist
>read_verilog ./3.synthesis/processor_core_synthesized.vg

# Use read_design to read in information(DFT signals etc) performed in previous pass.
>read_design processor_core -design_identifier rti2 -icl_only
>set_current_design processor_core
```

The second section of this example defines clocks and design constraints if present that need to be declared.

```
# Define clocks and design constraints if any
>add_clock clock1
>add_clock clock2

# Run DRCs, specify different scan insertion options, and write scan stitched design
>check_design_rules
>report_clocks
>report_dft_signals
```

The last section is where DRCs are run and if DFTSignals were used, then the add_scan_mode command can be utilized to specify them. If a tsdb_outdir was not specified, it can be specified with insert_test_logic -write_in_tsdb On to write the design and adjoining files into a tsdb directory database.

```
# Find edt_instance
>set edt_instance [get_instances -of_icl_instances [get_icl_instances \
    -filter tessent_instrument_type==mentor::edt]]

# Specify different modes the chains need to be stitched
# EDT is built-in with Bypass

>add_scan_mode edt_mode \
    -edt_instance $edt_instance

# Before scan gets inserted can analyze the different scan modes and scan chains
>analyze_scan_chains
>report_scan_chains

# Insert scan chains and writes the scan inserted design into tsdb_outdir directory
>insert_test_logic
>report_scan_chains
>exit
```

The following are some more examples of Scan Insertion for the unwrapped core.

Example 1: 1 mode: 32 flat scan chains

In this very basic example, all scan elements are allocated (unwrapped/flat) across 32 scan chains using the default distribution constraints (multiple clock domain, multiple clock edge, power domain, and single cluster per chain). The tool will connect the chains to new top-level ports called ts_si[31:0] and ts_so[31:0].

```
> add_scan_mode unwrapped_mode -chain_count 32
> analyze_scan_chains
> insert_test_logic
```

Example 2: 2 modes: edt, multi bypass (EDT controller already present)

In this example, for the edt mode, 200 chains are allocated from all scan elements and connected to the specified EDT controller si/so pins.

For the **multi bypass** mode, all scan elements are distributed to 12 multi-bypass chains and connected to new top-level ports called `ts_multi_bypass_si[11:0]` and `ts_multi_bypass_so[11:0]`. Default top-level ScanTestMode ports called `ts_stm0s0` and `ts_stm1s0` also get created to control the activation of the 2 modes.

```
> add_scan_mode edt \
    -si_connections edt_inst/to_si[199:0] \
    -so_connections edt_inst/from_so[199:0]
> add_scan_mode multi_bypass -chain_count 12
> analyze_scan_chains
> insert_test_logic
```

Example 3: 3 modes: edt, multi bypass, single chain (EDT controller already present)

This example is a variation of the previous one. The edt mode consists of 200 chains, during multi-mode bypass it has 12 chains connected to top-level ports and during single chain it has just one scan in and one scan out. Top-level ScanTestMode ports called `edt_mode_enable`, `multi_bypass_mode_enable`, and `single_mode_enable` get created to control the activation of the 3 modes. For the multi bypass mode, all scan elements are distributed to 12 multi-bypass chains and connected to new top-level ports called `ts_multi_bypass_si[11:0]` and `ts_multi_bypass_so[11:0]`. For single chain mode, all the scan elements are distributed to 1 single chain and connected to new top-level ports called `ts_single_chain_si` and `ts_single_chain_so`.

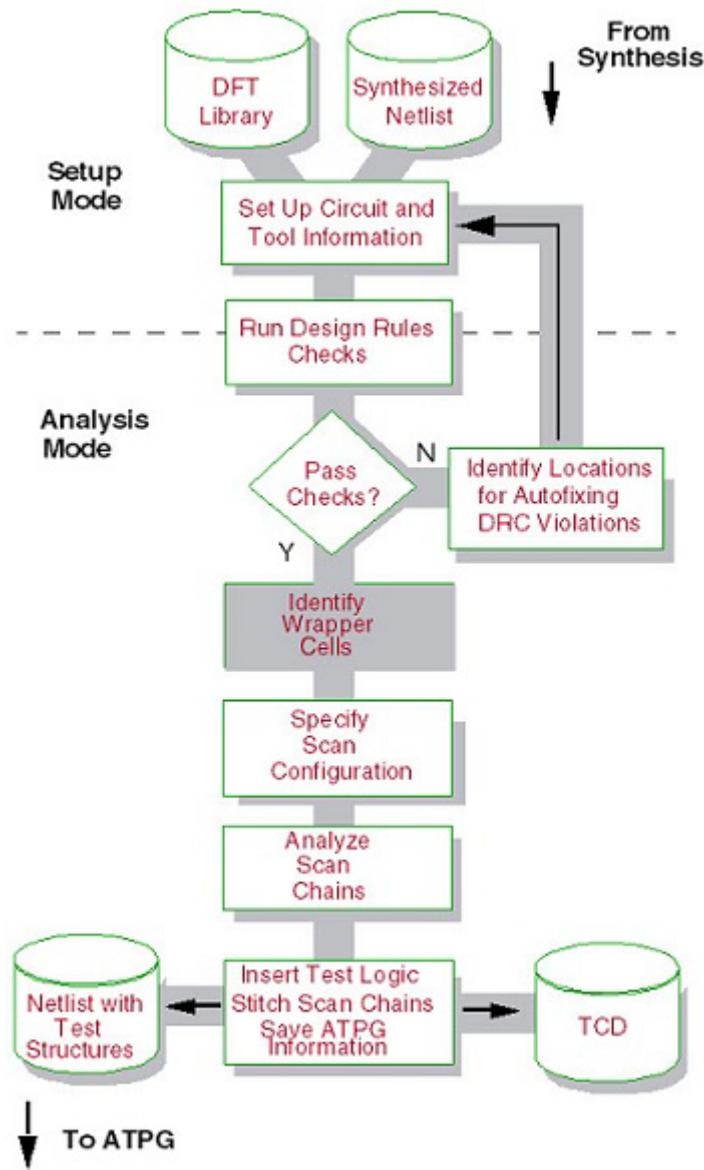
```
> add_scan_mode edt -si_connections edt_inst/to_si[199:0] \
    -so_connections edt_inst/from_so[199:0] -enable_connections edt_mode_enable
> add_scan_mode multi_bypass -chain_count 12 -enable_connections \
    multi_bypass_mode_enable
> add_scan_mode single_chain -single_class_chains off \
    -single_power_domain_chains off -single_cluster_chains off -chain_count 1 \
    -enable_connections single_mode_enable
> analyze_scan_chains
> insert_test_logic
```

Scan Insertion for Wrapped Core

This section describes scan insertion for Wrapped core.

A wrapped core is used for hierarchical DFT as described in “[Tessent Shell Flow for Hierarchical Designs](#)” in the *Tessent Shell User’s Manual*. For hierarchical DFT, wrapper cells need to be identified and inserted. Wrapper cells can be of dedicated or shared cells. A dedicated wrapper cell is one that does not exist in the design and is typically inserted on high fanin or fanout logic ports of the core that is targeted for hierarchical DFT. A shared wrapper cell is one where isolation of the core functionality is shared with an existing functional flop.

Figure 5-13. Scan Insertion Flow for Wrapped Core



In this scan insertion flow, first the scan elements that are present in the design are identified and any virtual scan elements resulting directly or indirectly from user commands get created. All these scan elements can be introspected and attributed.

Second, optionally you can create `scan_chain_family` objects that instruct the tool how to allocate specific scan element sub-populations. During this phase, one or more scan modes get defined. The population of each scan mode is defined by including `scan_element` and `scan_chain_family` objects. If you do not explicitly add at least one scan mode, the tool will infer a default mode which includes every `scan_element` and `scan_chain_family` object.

The pre-scan stitched netlist gets read in along with the required libraries, then the design gets elaborated (when set_current_design is specified). If EDT hardware was already inserted using DftSpecification and DFTSignals are used, then these signals are available here and the tool knows and uses them. For instance, if scan enable was declared as DFTSignal, then you do not need to declare the scan enable again. The tool knows which port was declared as scan enable and uses it without the you providing it using the set_scan_enable property. To know more about DFTSignals refer to the [add_dft_signals](#) command the *Tessent Shell Reference Manual*. IF EDT hardware was already inserted and DFTSignals are used to specify different modes then for edt_mode can use the -edt_instance to connect to the EDT hardware when specifying [add_scan_mode](#). The enable decode for this mode is automatically understood by the tool.

For hierarchical DFT, there are at least two modes: internal mode and external mode.

Example 1: If EDT hardware is built in with bypass, internal mode EDT hardware is present and for external mode there is no EDT hardware built inside the core. So, for ext_mode (this is a DFTSignal that was declared while EDT hardware was inserted) just scan chains are stitched up. The enable for both int_mode and ext_mode is automatically decoded from these DFTSignals.

```
>set edt_instance [get_instances -of_icl_instances [get_icl_instances \
    -filter tessent_instrument_type==mentor::edt]]
>add_scan_mode int_mode -type internal -edt_instances $edt_instance
>add_scan_mode ext_mode -type external -chain_count 4
```

Example 2: If EDT hardware is not built in with bypass, internal mode has three scan modes — when EDT is used (int_edt_mode), when EDT is bypassed with multiple scan chains (int_multi_mode) and when EDT is bypassed with single scan chain (int_single_mode). In External mode there is no EDT hardware and so scan chains are just stitched up to form three scan chains.

```
>set edt_instance [get_instance -of_modules *_edt_c1 ]
>add_scan_mode int_edt_mode -type internal -edt_instance $edt_instance
>add_scan_mode int_multi_mode -type internal -chain_count 6
>add_scan_mode int_single_mode -type internal -chain_count 1
>add_scan_mode ext_multi_mode -type external -chain_count 3
```

Example 3: If EDT Hardware is built in with bypass for internal mode of the wrapped core, and there are 2 scan configurations for external mode (ext_multi_mode) and (ext_single_mode). In ext_multi_mode the external chains are stitched into 6 scan chains, where as in ext_single_mode the external scan chains are stitched into 1 scan chain.

```
>set edt_instance [get_instances -of_icl_instances [get_icl_instances \
    -filter tessent_instrument_type==mentor::edt]]
>add_scan_mode int_mode -edt_instances $edt_instance
>add_scan_mode ext_multi_mode -chain_count 6
>add_scan_mode ext_single_mode -chain_count 1
```

Full Example: The example below is broken into four major sections. The first section - Design, Library and design elaboration shows how you read the Tessent Cell Library and Synthesized Verilog Gate-level design. If EDT hardware was already inserted then use the

`set_tsdb_output_directory` to point to where the tsdb data base is located. Then use `read_design` to read all the other supporting files like ICL, PDL, TCD from the last insertion pass and elaborate the design.

```
# Design, Library loading and design elaboration.  
>set_context dft -scan  
  
# Sets and opens the tsdb_output directory.  
>set_tsdb_output_directory ..//tsdb_outdir  
>read_cell_library ../../library/tessent/adk.tcelllib  
  
# Reading synthesized netlist  
>read_verilog ./3.synthesis/processor_core_synthesized.vg  
  
# Use read_design to read in information(DFT signals etc) performed in previous pass.  
>read_design processor_core -design_identifier rtl2 -icl_only  
>set_current_design processor_core
```

The second section of this example defines clocks and design constraints if present that need to be declared.

```
# Define clocks and design constraints if any  
>add_clock clock1  
>add_clock clock2  
  
# Run DRCs, and check_design_rules  
>report_clocks  
>report_dft_signals
```

The third section of this example identifies wrapper cells by providing input on what ports to exclude from wrapper cell analysis, and asynchronous set/reset ports to insert dedicated isolation cells. Then you get the tool to analyze wrapper cells and report it to make sure all ports have wrapper cells.

```
# Exclude the edt_channel in and out ports from wrapper chain analysis.  
# The ijttag_* edt_update ports are automatically excluded  
>set_wrapper_analysis_options -exclude_ports [get_ports {*_edt_channels_*}]  
  
# Added a new wrapper dedicated cell on reset  
>set_dedicated_wrapper_cell_options on -ports reset_n  
  
# Performs wrapper cell analysis  
>analyze_wrapper_cells  
>report_wrapper_cells -Verbose
```

The last section is where DRCs are run and if DFTSignals were used, then the `add_scan_mode` command can be utilized to specify them. If a `tsdb_outdir` was not specified, it can be specified with `insert_test_logic -write_in_tsdb On` to write the design and TCD files into a tsdb directory database.

```
# Specify different scan insertion options, and write scan stitched design  
  
# Find edt_instance  
>set edt_instance [get_instances -of_icl_instances [get_icl_instances \  
-filter tessent_instrument_type==mentor::edt]]
```

```
# Nice to specify different modes the chains need to be stitched
>add_scan_mode int_mode -type internal \
    -single_clock_domain_chains off \
    -single_clock_edge_chains off \
    -edt_instances $edt_instance
>add_scan_mode ext_mode -type external \
    -chain_count 2
>analyze_scan_chains
>report_scan_chains
```

The following are some more examples of Scan Insertion for the wrapped core.

Example 1: 3 modes: edt, ext, multi bypass (EDT controller already present)

In this example, for the **edt** mode, 200 chains are allocated from all scan elements and connected to the specified EDT controller si/so pins. For the **ext** mode, the wrapper scan elements are distributed into chains of 800 scan elements or less and connected to new top-level ports called `ts_ext_si[N-1:0]` and `ts_ext_so[N-1:0]`. For the **multi-bypass** mode, all scan elements are also distributed to 12 multi-bypass chains and connected to new top-level ports called `ts_multi_bypass_si[11:0]` and `ts_multi_bypass_so[11:0]`. Top-level ScanTestMode ports called `ts_stm0s0`, `ts_stm1s0`, and `ts_stm2s0` get created to control the activation of the three modes.

```
> analyze_wrapper_cells
> add_scan_mode edt \
    -si_connections edt_inst/to_si[199:0] \
    -so_connections edt_inst/from_so[199:0]
> add_scan_mode ext -type external -chain_length 800
> add_scan_mode multi_bypass -chain_count 12
> analyze_scan_chains
> insert_test_logic
```

Example 2: 3 modes: edt, ext, multi bypass (EDT controller already present, OCC chains)

This example is a variation of the previous one, adding existing OCC uncompressed chains. These existing segments are defined in setup mode and are attributed in order to allow the desired distribution during analysis. The **edt** mode consists of 200 chains plus one extra chain connected to `occ_edt_si/occ_edt_so` ports and which contains all existing OCC scan segments. The ext mode contains the external/wrapper scan elements plus one OCC segment used externally.

```
# Add & attribute all instances of the occ scan segments during setup mode
> add_scan_segments occ_chain \
    -on_module occ \
    -si_connections si -so_connections so
> register_attribute -name is_occ \
    -obj_type scan_element -value_type boolean
> register_attribute -name is_ext_occ \
    -obj_type scan_element -value_type boolean
> set_attribute_value [get_scan_elements -filter "name =~ *occ_chain"] \
    -name is_occ
> set_attribute_value [get_scan_elements -filter "name =~ u2/*occ_chain"] \
    -name ext_used_occ
> set_system_mode analysis
> analyze_wrapper_cells
> create_scan_chain_family compressed \
    -si_connections edt_inst/si[199:0] -so_connections edt_inst/so[199:0] \
    -include_elements [get_scan_element -filter !is_occ]
> create_scan_chain_family occ \
    -si_connections occ_edt_si -so_connections occ_edt_so \
    -include_elements [get_scan_elements -filter is_occ]
> add_scan_mode edt -include_chain_families{ compressed occ }
> add_scan_mode ext -type external -chain_length 800 \
    -include_elements [get_scan_elements -filter "ext_used_occ || class == wrapper"]
> add_scan_mode multi_bypass -chain_count 12 \
    -include_elements [get_scan_elements]
> analyze_scan_chains
> insert_test_logic
```

Scan Insertion at the Parent Level

Once the lower level cores are scan inserted and stitched, they need to be integrated at the next parent level.

How these lower level cores are integrated depends on whether the lower level core was an unwrapped core or a wrapped core.

If the lower level was an unwrapped core, then all the scan chains are visible at the parent level either via the EDT hardware present inside the core or via bypassed scan chains to parent-level pins or ports.

If the lower level was a wrapped core, then only the external mode scan chains are visible at the parent level that need to be connected to primary pins as regular scan chains. Refer to “[Tessent Shell Flow for Flat Designs](#)” in the *Tessent Shell User’s Manual*.

Alternatively, the external mode scan chains from various other wrapped cores can be connected to EDT hardware at the parent level so they become compressed scan chains to the EDT hardware at the parent level.

The internal modes of the wrapped cores need to be connected to the parent level by either broadcasting if they are identical cores, channel sharing techniques (data channels are shared but control channels are separate) or muxed into available pins that can be used as channel pins at the parent level.

If you are reading lower level wrapped cores, then you need to open the TSDB of these lower level cores. From the TSDB, the TCD file from the last pass that has dft_inserted_design directory is read in. The TCD file contains information in the Scan section about how many modes are present, if they are of type internal or external and how many scan chains are in each mode. The clock and scan_enable information are also contained with each mode inside this TCD file.

From reading this TCD file, the external scan chains are then stitched at the parent level of the child cores based on scan stitching criteria provided at the parent level.

The example below shows two lower level wrapped cores, corea and coreb being read in at the parent level "top" which is the chip. Graybox models for the corea and coreb blocks are read in after the top level verilog is read in. In this example EDT was inserted in RTL as a second pass. In the first pass, TAP controller and Boundary scan cells were inserted.

Example 1:

```
# If no -design_id is provided, then the default design_id is gate
>set_context dft -scan -design_id gate1

# Set the TSDB outdir where the parent needs to write
>set_tsdb_output_directory ../../tsdb_outdir

# Open the TSDB directory of lower level wrapped cores
>open_tsdb ../../corea/tsdb_outdir
>open_tsdb ../../coreb/tsdb_outdir

# Read the Tessent Cell Library
>read_cell_library ../../adk.tcell_library

# Read the parent-level verilog using read_verilog or read_design
>read_design top -design_id rtl2 -verbose

# Read the graybox of lower level wrapped cores corea and coreb
>read_design corea -design_id gate1 -view graybox -verbose
>read_design coreb -design_id gate1 -view graybox -verbose
>set_current_design top

# Set the design level
>set_design_level chip

# The clocks were added in during EDT/OCC insertion in rtl2 pass
# Pass 1 was TAP and boundary scan insertion.

>set_system_mode analysis
>report_clocks
>set edt_instance [get_name_list [get_instance -of_module [get_name \
    [get_icl_module -of_instances top* -filter tessent_instrument_type==mentor::edt]]] ]]

>add_scan_mode edt_mode -edt_instance $edt_instance

>analyze_scan_chains
>report_scan_chains
```

```
# Insert scan chains
>insert_test_logic
>report_scan_chains
>exit
```

Example 2: In this example there are 2 external mode scan chains for each of the lower level wrapped cores - core1 and core2. To see how the lower level scan chain is set up, refer to Example 3 in the section "[Scan Insertion for Wrapped Core](#)".

Now at the parent level, one of these external mode scan configuration needs to be identified to be used during scan insertion by setting the attribute "active_child_scan_mode" just before issuing the command set_system_mode analysis as below. In this example the ext_multi_mode of the wrapped cores (core1 and core2) is identified to be included with the edt_mode for the top-level. Then the other external mode ext_single_mode of the wrapped cores (core1 and core2) is included with the single_mode scan configuration at the top-level by using the attribute "active_child_scan_mode" to ext_single_mode.

```
##Specify which of the external mode scan configuration to use for top-level
set_attribute_value {core1_inst} -name active_child_scan_mode -value ext_multi_mode
set_attribute_value {core2_inst1 core2_inst2} -name active_child_scan_mode \
    -value ext_multi_mode

set_system_mode analysis

set edt_instance [get_name_list [get_instance -of_module \
    [get_name [get_icl_module -of_instances chip_top* -filter \
        tesseract_instrument_type==mentor::edt]]] ]
add_scan_mode edt_mode $edt_instance

## Specify to use the external mode single chain configuration for lower level cores \
## using "single_mode" at top-level
set_attribute_value {core1_inst} -name active_child_scan_mode -value ext_single_mode
set_attribute_value {core2_inst1 core2_inst2} -name active_child_scan_mode \
    -value ext_single_mode

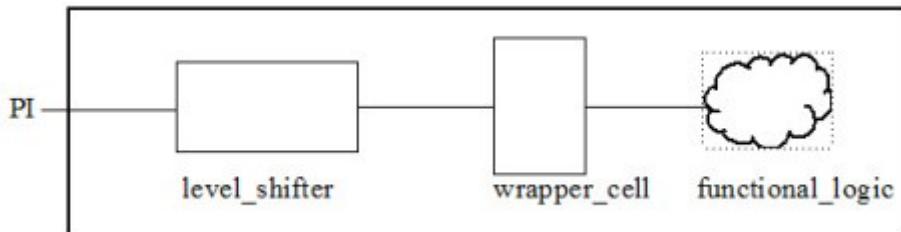
add_scan_mode single_mode -chain_count 4
analyze_scan_chains
report_scan_chains
insert_test_logic
```

Dedicated Wrapper Cell Placement After Loading UPF/CPF File

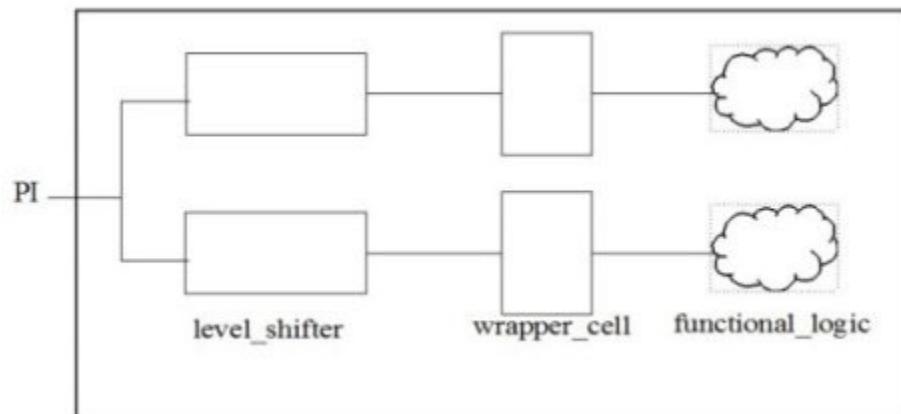
The read_cpf and read_upf commands can be used to read power data into Tessent Shell. The UPF/CPF files define different power-domains within the netlist as well as defining whether library cells will be used as isolation cells and as level shifters. This includes defining which specific instances of those cells are serving in that capacity.

The presence of power data impacts the placement of dedicated wrapper cells. Input wrapper cells should be placed on the output side of isolation cells, and at the output of level shifters that are at the input of a block, and at the input of a level shifter that is driving the output of a block.

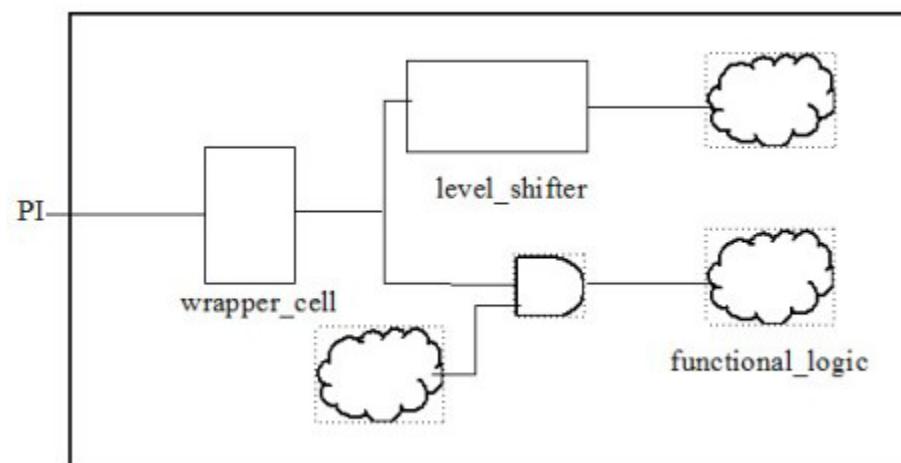
In general, this means moving the dedicated wrapper cells away from the primary input and/or primary output toward the “inside” of the core.



If the primary input fans out to multiple level shifters or isolation cells, then multiple dedicated wrapper cells are inserted to register a single primary input.



One exception to this general approach occurs when a primary input fans out to a mixture of level shifters and/or isolation cells as well as library cells that are using to implement functional logic. In this case, the dedicated wrapper cell is placed at the driver of the net and no wrapper cells are placed at the outputs of the wrapper cells.



Note

-  When defining level shifters in a CPF file, you must define the level shifter with the “set_level_shifter” command, and must also specify which power domains will be using the cell as a level shifter with the “map_level_shifter” command.
-

Note

-  The UPF/CPF file must define at least one power domain in order to trigger this analysis.
-

Chapter 6

Test Point Analysis and Insertion

Test points are an important feature of the Tessent product portfolio. This chapter introduces VersaPoint™ test point technology and explains how to use Tessent ScanPro or Tessent LogicBIST to insert test points in your design.

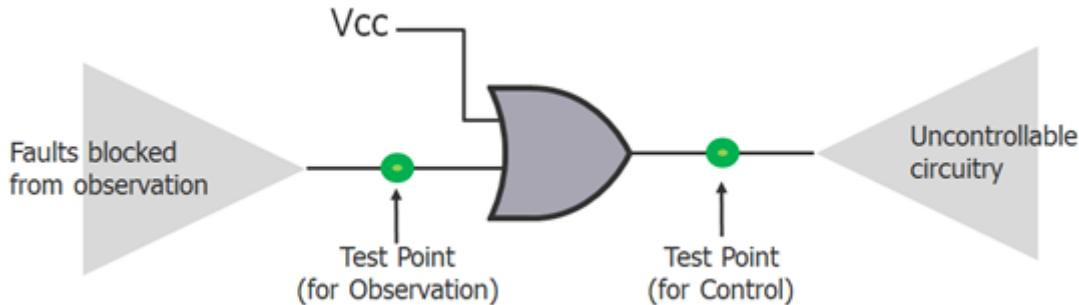
What are Test Points?	208
Why You Use Test Points.	208
Control Points and Observe Points	209
Test Point Insertion Flows	212
Analyze and Insert Test Points in a Pre-Scan Design.	213
Analyze and Insert Test Points in a Post-Scan Design.	215
How to Insert Test Points and Perform Scan Stitching Using Third-Party Tools	220
Test Point Usage Scenarios	224
Test Points for ATPG Pattern Count Reduction.	225
Test Points for LBIST Test Coverage Improvement	229
Test Points for Hybrid TK/LBIST Design	234
Test Points for ATPG Test Coverage Improvement.	236
Test Points Special Topics.	240
Control Point and Observe Point Enable Signal Handling.	240
Test Point Analysis With Multiple Power Domains.	241
Test Point Analysis Multicycle and False Path Handling.	242
Test Point Analysis Critical Path Handling	244
User-Defined Test Points Handling	247
Test Point Deletion	249
Back to Back Command Handling	250
Static Timing Analysis for Test Points.	250
Test Points and At-Speed Testing.	251
Clock Selection for Test Point Flops	252
Example of Test Point Insertion With Custom Prefix for Inserted Logic	252

What are Test Points?

Designs contain a number of internal modes which are difficult to control and/or observe. This is true even in designs which are fully scan inserted. By adding special test circuitry called test points to these locations, it is possible to increase the testability of the design.

For example, [Figure 6-1](#) shows a portion of circuitry with a controllability and observability problem.

Figure 6-1. Uncontrollable and Unobservable Circuitry



In this example, one input of an OR gate is tied to a 1 - this value may come from a tied primary input or it may be a learned static value internal to the circuit. This static value blocks the ability to propagate the faults effects of the second path through this OR gate. Thus, an observe test point is needed on the second input of the OR gate to improve observation. The tied input also causes a constant 1 at the output of the OR gate. This means any circuitry downstream from that output is uncontrollable. The pin at the output of the gate becomes a test point to improve controllability. Once identification of these points occurs, added circuitry can improve the controllability and observability problems.

Why You Use Test Points	208
Control Points and Observe Points	209

Why You Use Test Points

There are three distinct purposes for using test points, all of which are addressed by Tessent's VersaPoint test point technology.

- To reduce deterministic pattern counts. In some cases test coverage may also improve, however improvements are generally minimal.
- To improve test coverage for random pattern resistant faults. Test points are inserted to improve the probability of random patterns to control and/or observe a given fault site.
- To improve coverage of undetected faults during ATPG. ATPG coverage loss may be due to various factors, including untestable circuitry or aborted faults.

Control Points and Observe Points

The inserted test points consist of control points and observe points.

Refer to the [Control Points](#) and [Observe Points](#) sections for details. During test point analysis and insertion, the tool inserts these test points at gate inputs or outputs—see also "[User-Defined Test Points Handling](#)."

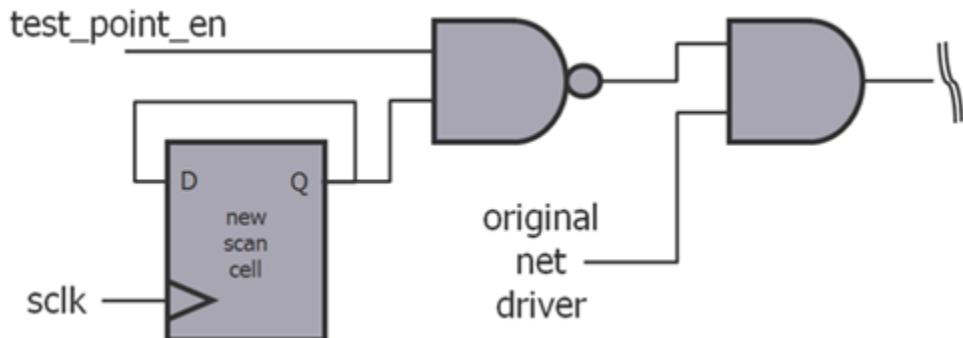
Control Points	209
Observe Points	210

Control Points

You can insert two types of control points, AND Control Points or OR Control Points.

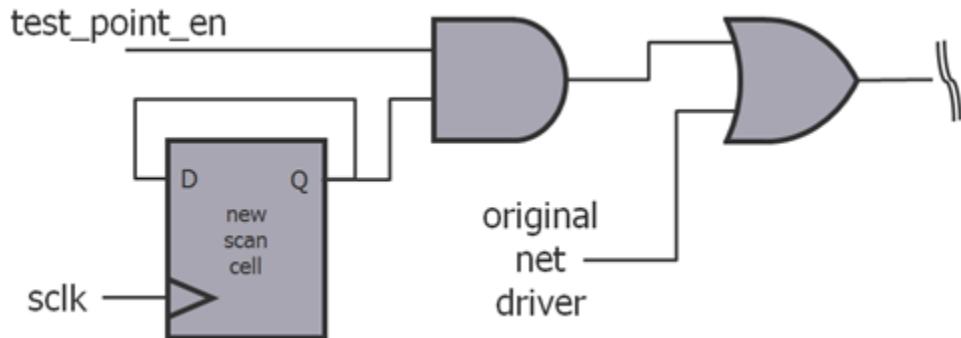
- **AND Control Points**

Figure 6-2. AND Control Point



- **OR Control Points**

Figure 6-3. OR Control Point



Control points do not change during capture cycles. The AND and OR control points are mutually exclusive: specifically, the tool does not insert both an AND and an OR control point on the same net.

Control Point Clocking

The tool looks at all the flops in the fan-out cone of the test point and picks the clock that drives the largest number of scan flops in the fan-out cone. If no scan flops are found in the fan-out cone, it picks the clock that drives the largest number of scan flops in the fan-in cone.

Once the target clock is identified, the tool again processes the fan-out cone and taps the target clock from the clock port of the scan flop that requires crossing the least number of levels in the Verilog design hierarchy and is in the same power domain as the test point. Once again, if it cannot find any scan flops in the fan-out cone, it picks the closest one in the fan-in cone.

In the event that there are no scan flops in either cone, the tool will perform a hierarchical search for a scan cell that is driven by the target clock and is in the same power domain as the test point.

If this fails, the tool checks to see if the clock is an input of the instance where the test point flop will be located. If this fails, it performs the same check for parent instances as long as the parent instance is still in the correct power domain.

Finally, if everything else fails, it uses test clock by tapping the source of the test clock signal (either at the primary input pin, or at the source of the clock signal defined using the [add_clocks](#) command).

Note

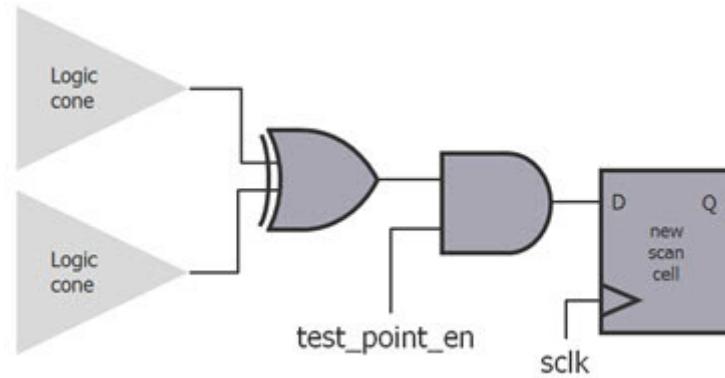
 Potentially scannable flops are also considered as “scan flops” during the clock selection analysis. Non-scan flops are ignored during the test point clock selection processing.

Observe Points

The tool inserts observe points in the form of new scan cells at gate outputs to improve the observability of the netlist. The tool supports both a dedicated scan cell per observation point or a scan cell shared by multiple observation points. When sharing the new scan cell among several observation points, the tool inserts an XOR tree.

[Figure 6-4](#) shows an example of two stems connected through an XOR gate and observed with the same observation point.

Figure 6-4. Observe Point Sharing



Observe Point Clocking

The tool looks at all the flops in the fan-in cone of the test point and picks the clock that drives the largest number of scan flops in the fan-in cone. If no scan flops are found in the fan-in cone, it picks the clock that drives the largest number of scan flops in the fan-out cone.

Once the target clock is identified, the tool again processes the fan-in cone and taps the target clock from the clock port of the scan flop that requires crossing the least number of levels in the Verilog design hierarchy and is in the same power domain as the test point. Once again, if it cannot find any scan flops in the fan-in cone, it picks the closest one in the fan-out cone.

In the event that there are no scan flops in either cone, the tool will perform a hierarchical search for a scan cell that is driven by the target clock and is in the same power domain as the test point.

If this fails, the tool checks to see if the clock is an input of the instance where the test point flop will be located. If this fails, it performs the same check for parent instances as long as the parent instance is still in the correct power domain.

Finally, if everything else fails, it uses test clock by tapping the source of the test clock signal (either at the primary input pin, or at the source of the clock signal defined via the `add_clocks` command).

Note

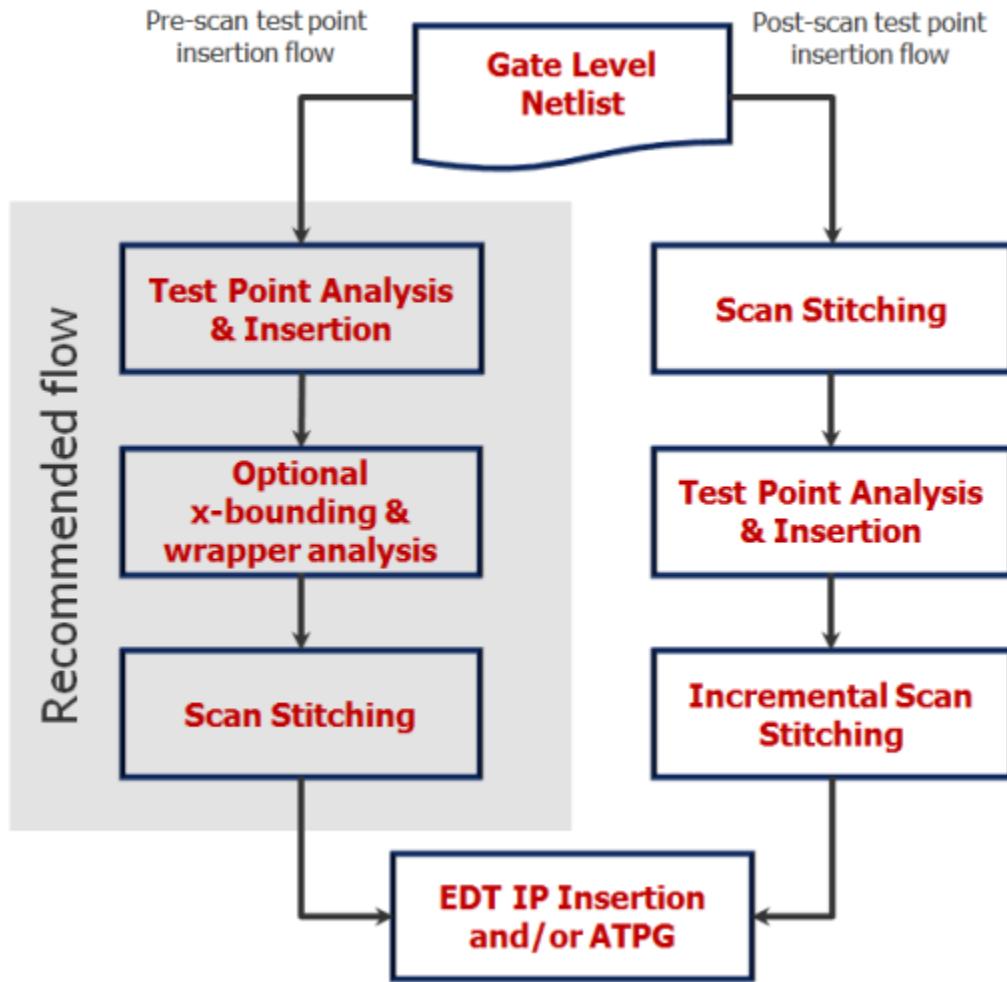
 Potentially scannable flops are also considered as “scan flops” during the clock selection analysis. Non-scan flops are ignored during the test point clock selection processing.

Test Point Insertion Flows

This section describes the VersaPoint test point insertion flows for pre-scan and post scan designs.

Figure 6-5 shows the typical design flows for inserting test points in a Pre-scan or Post-scan gate level netlist.

Figure 6-5. Test Point Analysis & Insertion Starting With a Gate-Level Netlist



Note

- Test point analysis and insertion done on a wrapped core may place test points outside of core isolation, if present.

Analyze and Insert Test Points in a Pre-Scan Design.....	213
Analyze and Insert Test Points in a Post-Scan Design.....	215
How to Insert Test Points and Perform Scan Stitching Using Third-Party Tools	220
How to Insert Test Points for Design Compiler or Genus™ Synthesis Solution.....	220

Analyze and Insert Test Points in a Pre-Scan Design

You can analyze and insert test points into a pre-scan design using Tesson Shell.

Procedure

1. From a shell, invoke Tesson Shell using the following syntax:

```
tesson -shell
```

After invocation, the tool is in unspecified setup mode. You must set the context before you can invoke the test point insertion commands.

2. Set the tool context to test point analysis using the [set_context](#) command as follows:

```
SETUP> set_context dft -test_points -no_rtl
```

3. Load the pre-scan gate-level Verilog netlist using the [read_verilog](#) command.

```
SETUP> read_verilog my_netlist.v
```

4. Load one or more cell libraries into the tool using the [read_cell_library](#) command.

```
SETUP> read_cell_library tesson_cell.lib
```

5. Set the top level of the design using the [set_current_design](#) command.

```
SETUP> set_current_design
```

6. Set additional parameters as necessary. Examples include: black boxes, pin constraints, defining clocks, and declaring any additional settings which are required to pass DRCs to allow the transition to Analysis system mode. For this example, it is assumed these additional setup constraints are made available in a separate dofile which is read in this step.

```
SETUP> dofile set_up_constraints.do
```

Note

 To continue with test point analysis and insertion, the design must pass scan DRC rules, including S1 and S2 rules. If it is not possible to setup the design to pass all DRC rules for flattening, it is recommended as part of this setup step that any cells which produce Errors during the system mode transition, are defined as non-scan instances to the tool. This can be done using either of the following two methods:

- If only S1 and/or S2 rule violations are present:

```
set_drc_handling S1 Warning  
set_drc_handling S2 Warning
```
 - If there are other additional cells preventing the system-mode transition, define those as non-scan by using the [add_nonscan_instances](#) command.
-

7. If required, read in the SDC file containing the multicycle and false path information using the [read_sdc](#) command—see [Test Point Analysis Multicycle and False Path Handling](#).

SETUP> read_sdc sdc_file_name

8. Set the test point type using the [set_test_point_type](#) command. This example demonstrates the usage of the [edt_pattern_count](#) option. For other test point types, please see the [set_test_point_types](#) command.

SETUP> set_test_point_type edt_pattern_count

9. In the following example, the flop type “dflop2” from the Tessent Cell Library is specified. This ensures this specific flop type will be used as test point flop. If this is not done, the Tessent software will use the first defined flop available in the Tessent Cell Library as the test point flop. Please see the command [add_cell_models](#) for further usage information.

SETUP> add_cell_models dflop2 -type DFF CK D

10. Change the tool’s system mode to analysis using the [set_system_mode](#) command.

SETUP> set_system_mode analysis

During the transition from setup to analysis modes, the tool flattens the netlist and performs Scannability Rules (S Rules) checking. You must resolve any DRC rule violations that result in an error before you can move to analysis mode. The [set_test_logic](#) command determines if cells with S-rule DRC violations are converted to scan cells or remain non-scan. The DRC performed when going to analysis mode helps the tool to identify potential scan cells that are necessary during test point analysis. The DRC warnings and errors should be addressed to ensure that the tool identifies all the potential scan cells in the netlist.

11. This step may be performed in setup or analysis system mode, up to the time [analyze_test_points](#) is called. Set up the test point analysis options using the [set_test_point_analysis_options](#) command. This usage example sets one of the options which pertains to the above test point type. For an explanation of other test point analysis options and how they apply to the different test point types, please refer to the command documentation.

ANALYSIS> set_test_point_analysis_options -total_number 1000

In analysis mode, the tool retains the enabled commands that you set to characterize the analysis of test points.

12. Display the analysis options you have set. For example, specifying the command without any arguments shows the default settings within the tool.

ANALYSIS> set_test_point_analysis_options

```
// command: set_test_point_analysis_options -total_number 1000
// command: set_test_point_analysis_options
// Maximum Number of Test Points      :      1000
// Maximum Number of Control Points   :      1000
// Maximum Number of Observe Points  :      1000
// Maximum Control Points per Path   :      5
// Exclude Cross Domain Paths       :      off
```

13. This step may be performed in setup or analysis system mode, up to the time `insert_test_logic` is called. Specify the test point insertion options using the `set_test_point_insertion_options` command. For example, this step shows how you can specify to the tool to generate separate enable signals for control and observe points.

```
ANALYSIS> set_test_point_insertion_options \
-control_point_enable cp_en -observe_point_enable op_en
```

In this example, `cp_en` is specified to be used as the `control_point_enable` port. Similarly, `op_en` is specific to be used as the `observe_point_enable` port. If either of these ports do not already exist in the design, then the respective port will be created at the root of the design.

14. Perform the test point analysis using the `analyze_test_points` command:

```
ANALYSIS> analyze_test_points
```

You have now generated the test points.

15. Add test points to your design using the `insert_test_logic` command. Executing this command will transition you to INSERTION mode.

```
ANALYSIS> insert_test_logic
```

16. If needed, report the test points that have been analyzed and inserted into your design using the `report_test_points` command. The file `testpoints.txt` provides the ordered list of observe and control test points analyzed for the design.

```
INSERTION> report_test_points > testpoints.txt
```

17. Write out the modified design netlist containing the test points using the `write_design` command as in the following example:

```
INSERTION> write_design -output_file my_modified_design.v
```

18. Write out the dofile containing all the necessary steps for scan insertion using the `write_scan_setup` command.

```
INSERTION> write_scan_setup -prefix scan_setup
```

Analyze and Insert Test Points in a Post-Scan Design

You can analyze and insert test points into a post-scan design using Tessent Shell, however, this flow is not recommended if you are attempting to insert test points into a design with

pre-existing wrapper chains. Test point flops may be inserted outside of core isolation logic and will be stitched into separate scan chains as part of the incremental scan insertion step.

Procedure

1. From a shell, invoke Tesson Shell using the following syntax:

```
% tessent -shell
```

After invocation, the tool is in unspecified setup mode. You must set the context before you can invoke the test point insertion commands.

2. Set the tool context to test point analysis using the [set_context](#) command as follows:

```
SETUP> set_context dft -test_points -no_rtl
```

3. Load the post-scan gate-level Verilog netlist using the [read_verilog](#) command.

```
SETUP> read_verilog my_netlist.v
```

4. Load one or more cell libraries into the tool using the [read_cell_library](#) command.

```
SETUP> read_cell_library tessent_cell.lib
```

5. Set the top level of the design using the [set_current_design](#) command.

```
SETUP> set_current_design
```

6. Load the *setup_constraints.dofile* which has all the clock definitions, input constraints, and other parameters affecting the setup of the design in test mode.

```
SETUP> dofile set_up_constraints.do
```

7. If required, read in the SDC file containing the multicycle and false path information using the [read_sdc](#) command — see [Test Point Analysis Multicycle and False Path Handling](#).

```
SETUP> read_sdc sdc_file_name
```

8. Read in the *scan_chains.do* file that describes the scan chains that are already stitched in the design.

```
SETUP> dofile scan_chains.do
```

Note



This is a design with pre-existing scan chains. As such, those chains must be defined to the tool along with associated setup procedures in order to pass scan DRC rules. The scannability rules S1 and S2 must be error-free in order to successfully make the system-mode transition into analysis mode. There are cases where a scanned design may have had specific instances defined as non-scan. Those instances will likely fail the S1 and S2 DRC rules. Those rules can be set to warning using the [set_drc_handling](#) command, and by doing so, those instances will be, internally, defined as non-scan.

9. Set the test point type using the `set_test_point_type` command. This example demonstrates the usage of the `lbist_test_coverage` option. For other test point types, please see the `set_test_point_types` command.

SETUP> set_test_point_type lbist_test_coverage

10. In the following example, the flop type “dflopX2” from the Tessent Cell Library is specified. This ensures this specific flop type will be used as test point flop. If this is not done, the Tessent software will use the first defined flop available in the Tessent Cell Library as the test point flop. Please see the command `add_cell_models` for further usage information.
11. This step may be performed in setup or analysis system mode, up to the time `analyze_test_points` is called. Set up the test point analysis options using the `set_test_point_analysis_options` command. This usage example sets options which pertains to the above test point type. For an explanation of other test point analysis options and how they apply to the different test point types, please refer to the command documentation.

SETUP> set_test_point_analysis_options -total_number 2500 \ -test_coverage_target 99.90 -pattern_count_target 10000

12. This step may be performed in setup or analysis system mode, up to the time `insert_test_logic` is called. Specify the test point analysis options using the `set_test_point_insertion_options` command. For example:

ANALYSIS> set_test_point_insertion_options \ -control_point_enable cp_en -observe_point_enable op_en

In this example, `cp_en` is specified as the `control_point_enable` port. Similarly, `op_en` is specified to be used as the `observe_point_enable` port. If either of these ports do not already exist in the design, then the respective port will be created at the root of the design.

13. Change the tool’s system mode to analysis using the `set_system_mode` command.

SETUP> set_system_mode analysis

In this case, the tool runs drc to validate that the scan chains specified can be traced properly.

14. Perform the test point analysis using the `analyze_test_points` command:

ANALYSIS> analyze_test_points

You have no generated the test points.

15. Add test points to your design using the `insert_test_logic` command. Executing this command will transition you to INSERTION mode.

ANALYSIS> insert_test_logic

16. Report the test points that have been analyzed and inserted into your design using the `report_test_points` command. The file `testpoints.txt` provides the list of observe and control test points analyzed for the design.

INSERTION> report_test_points > testpoints.txt

17. Write out the modified design netlist containing the test points using the `write_design` command as in the following example:

INSERTION> write_design -output scan_design_name_tp_inserted.v

18. Write out a test procedure file and the dofile file for the existing scan chains.

INSERTION> write_scan_setup -prefix scan_top

19. Perform incremental scan chain stitching for just the test point flops. The test point flops must be stitched into scan chains by transitioning into dft -scan context.

INSERTION>set_context dft -scan

20. Start with deleting the current design.

SETUP> delete_design

21. Read in the dofile written out in the previous steps when `write_scan_setup` was run. This dofile has the clocks, sets, resets, input constraints and pre-existing scan chains information from the `scan_tp.do` file. All the flops in the design are made non-scan and only the test point flops that need to be stitched into scan chains will be added as scannable elements.

SETUP> dofile scan_top.dofile

22. Change the tool's system mode to analysis using the `set_system_mode` command.

SETUP> set_system_mode analysis

23. Incremental scan insertion – adding these new elements into pre-existing chains in the design – is not supported in Tesson Scan. New chains must be added for the test points and these chains will be added to a new scan mode, separate from other, pre-defined scan modes. The following set of commands are used to specify the new mode, analyze the scan chains, and finally insert the new scan chains.

**ANALYSIS> add_scan_mode unwrapped_test_points **
**-type unwrapped **
**-chain_length 500 **
-enable_connections unwrapped_tp_chain_mode

The `add_scan_mode` command above will perform the following tasks: define a new scan mode named “unwrapped_test_points” which will be independent from all other scan modes, set the maximum chain length to “500” and define a new top-level enable signal for the enabling of this scan mode.

ANALYSIS> analyze_scan_chains

The `analyze_scan_chains` command above analyzes the available, scannable cells and organizes them into scan chains for insertion. The defined setup may be viewed using `report_scan_chains` and may be changed until `insert_test_logic` is issued, by deleting and re-adding the scan mode and re-analyzing the scan chains.

ANALYSIS> `insert_test_logic`

24. Write out an updated netlist along with the `atpg_tp` test procedure file and dofile that will be used during pattern generation.

INSERTION> `write_design -output scan_design_name_tp.v`

INSERTION> `write_design basename -replace`

INSERTION> `exit`

How to Insert Test Points and Perform Scan Stitching Using Third-Party Tools

You can perform test point analysis with Tesson Shell. The insertion as well as scan chain stitching of these test points can be done using third-party tools.

After test points are analyzed, write out a report using `report_test_points`.

```
ANALYSIS> analyze_test_points
ANALYSIS> report_test_points > test_points.list
```

An example sample report is shown below.

TestPoint Location	TestPoint Type	UserAdded ControlPoint Type	ScanCell	ScanCell Clock	EnableSignal
dramct10/i_1161/ZN	Observe		dramct10/ts_op_182#ffpl_i/D	dram_gclk	obs_en
dramct11/i_1161/ZN	Observe		dramct11/ts_op_264#ffpl_i/D	dram_gclk	obs_en
dramct10/i_12998/ZN	Observe		dramct10/ts_op_184#ffpl_i/D	dram_gclk	obs_en
dramct10/i_1119/ZN	Control OR		dramct10/ts_cp_2#ffpl_i/Q	dram_gclk	cntrl_en
dramct11/i_1119/ZN	Control OR		dramct11/ts_cp_82#ffpl_i/Q	dram_gclk	cntrl_en
dramct11i_8525/ZN	Control AND		ramct11/ts_cp_148#ffpl_i/Qd	ram_gclk	cntrl_en
dramct10/i_8525/ZN	Control AND		dramct10/ts_cp_70#ffpl_i/Q	dram_gclk	cntrl_en

The TestPoint Location column specifies where the functional net needs to be intercepted. The ScanCell Clock column specifies the instance name of the test point flop to be inserted. The TestPoint Type column specifies what type of test point it is. The clock needed to be connected to the test point flop is specified in the ScanCell Clock column.

Refer to “[Control Points and Observe Points](#)” for information on how to implement AND control points, OR control points and observe points.

The test points can be inserted as compressed scan chains connected to the Decompressor and the Compactor. Mentor Graphics recommends that you stitch the test points into their own separate scan chains for more efficient low power EDT pattern generation.

How to Insert Test Points for Design Compiler or Genus™ Synthesis Solution 220

How to Insert Test Points for Design Compiler or Genus™ Synthesis Solution

You can use the Tesson Shell script generation feature to generate stitching scripts for test point insertion for Design Compiler or Genus Synthesis Solution.

The “`insert_test_logic -write_insertion_script`” command generates a TCL stitching script that will mirror the Tesson inserted test points into the DC/Genus environment.

Example

```
> set_system_mode analysis  
> set_test_point analysis_options -total_number 200  
> analyze_test_point  
> write_test_point_dofile -output_file generated/output -200.dofile -replace  
> insert_test_logic -write_insertion_script dc_stitch_script.tcl -replace
```

The `insert_test_logic` command inserts the Tesson analyzed test points into the current design and generates an equivalent TCL stitch script to use in a DC shell. You can take the TCL script and source it from a DC shell. For example:

```
dc> source dc_stitch_script.tcl
```

Note that the DC insertion script contains `create_cell` DC operations for the instantiation of cell instances. The `create_cell` command requires a library name specification which is not known during script generation time. As such, the generated DC script contains library query operations to attempt to resolve the cell library name currently loaded in DC shell. If the library query fails or you wish to define the library to use, you can do so using a TCL variable specification prior to sourcing the script. For example:

```
dc> set insertion_cell_library_name myCellLib  
dc> source dc_stitch_script.tcl
```

If your target insertion tool is the Cadence Genus Synthesis Solution, use the following command:

```
> insert_test_logic -write_insertion_script genus_stitch_script.tcl -replace -insertion genus
```

Then, source the TCL stitch from a Genus shell. For example:

```
genus> source genus_stitch_script.tcl
```

Example of a Generated DC Insertion Script output File

```
#-----
#  File created by: Tessent Shell
#      Version: 2019.2
#      Created on: Tue May  7 10:38:40 PDT 2019
#-----

#-----
# Variables
#-----

# Use insertion_cell_library_name variable if specified
set ts_library_name ""
if { [info exists insertion_cell_library_name] } {
    set ts_library_name $insertion_cell_library_name
}

set ts_path_prefix ""
#-----
# Utilities Procs
#-----


# Procedures sourced from
# '<tessent_home>/dft_insertion_dc_utility_procedures.tcl':


# Prepares a destination pin for connection.
# Remove existing net connection if found.
proc ts_prepDestPin { destPin } {
    set destPins [get_pins $destPin]
    if { [sizeof_collection $destPins] } {
        set destNets [get_nets -of_objects $destPins]
        if { [sizeof_collection $destNets] } {
            disconnect_net $destNets $destPins
        }
    }
}

...


...
```

```

#-----
# Insertion Command Mapping Procs
#-----
proc ts_create_port { path port dir } {
    set dirVal ""
    switch -exact -- $dir {
        "input" { set dirVal "in" }
        "output" { set dirVal "out" }
        "inout" { set dirVal "inout" }
    }   create_port [ts_hier_path $path $port] -direction $dirVal}
proc ts_delete_port { path port } {
    remove_port [ts_hier_path $path $port]
}
proc ts_insert_instance { path libName cellName instName } {
    global ts_library_name
    if { $ts_library_name eq "" } {
        set cell_libs [get_libs]      foreach_in_collection cell_lib $cell_libs {
            set cell_lib_name [get_object_name $cell_lib]
            set lib_cells [get_lib_cells -quiet $cell_lib_name/$cellName]
            if { [sizeof_collection $lib_cells] } {
                create_cell [ts_hier_path $path $instName] $cell_lib_name/$cellName
                break
            }
        }
    } else {
        create_cell [ts_hier_path $path $instName] $ts_library_name/$cellName
    }
}
...
#-----
# Generated Insertion Operations for Design "sparc"
#-----
ts_insert_instance {test_stub/scan_ctls} work SDFF_X1 ts_cp_0sffp1_i
ts_create_port {test_stub/scan_ctls} {ts_clkc0} input
ts_connect_port_to_pin {test_stub/scan_ctls} {ts_clkc0} \
    {test_stub/scan_ctls} {ts_cp_0sffp1_i/CK}
ts_connect_constant_net {test_stub/scan_ctls} {1'b0} {ts_cp_0sffp1_i/SE}
ts_connect_constant_net {test_stub/scan_ctls} {1'b0} {ts_cp_0sffp1_i/SI}

```

Test Point Usage Scenarios

There are four distinct purposes for using test points which are described in this section.

Test Points for ATPG Pattern Count Reduction	225
Test Points for Reducing Pattern Count	226
Requirements for Test Point Analysis and Insertion	226
Specify Test Point Type	226
Inserting Test Points When Using the EDT Skeleton IP Flow.....	228
Test Points for LBIST Test Coverage Improvement	229
Test Point Analysis and Insertion Step Overview	229
Test Point Analysis With Small LBIST Pattern Counts.....	230
Test Coverage Reporting During Test Point Analysis for LogicBIST Coverage Improvement	
231	
Test Points for Hybrid TK/LBIST Design.....	234
Improving Test Coverage and Pattern Count Reduction	234
Test Points for ATPG Test Coverage Improvement	236
Test Points for Improving the Test Coverage of Deterministic Patterns	236

Test Points for ATPG Pattern Count Reduction

You can use test points in your design to improve compression and test coverage.

[Figure 6-5](#) shows the typical design flows for inserting test points in a Pre-scan or Post-scan gate level netlist.

Test Points for Reducing Pattern Count	226
Requirements for Test Point Analysis and Insertion	226
Specify Test Point Type.....	226
Inserting Test Points When Using the EDT Skeleton IP Flow	228

Test Points for Reducing Pattern Count

The test points are targeted primarily to reduce the EDT pattern counts. The test points may improve test coverage as well but the impact may be minimal.

You must first analyze and insert test points into the netlist to achieve reduction in pattern count, hence improving compression. Either a pre-scan design or post-scan stitched netlist can be used.

Note

 You generate and insert the test points as a separate step from scan insertion — you cannot perform the two operations at the same time, although they can be done sequentially with a single invocation of the tool.

After test point analysis has been completed, the desired number of test points can be reported and can also be inserted into the design. The modified netlist with the test points already inserted and the dofile containing all the necessary information for the next steps of the design flow can be written out.

Requirements for Test Point Analysis and Insertion

Performing test point analysis and insertion has certain requirements.

You must adhere to the following:

- Test point insertion needs a gate-level Verilog netlist and a Tessent Cell Library.
- You can read in functional SDC so the tool omits adding any test points to multicycle paths or false paths—see the [read_sdc](#) command.
- The tool identifies potential scan candidates for correct controllability/observability analysis. To ensure that eventual non-scan cells are not used as the destination for observe points or source for control points, you should declare all the non-scan memory elements during test point insertion using the [add_nonscan_instances](#) command.
- You should define black boxes using the [add_black_boxes](#) command so that test point analysis can incorporate this information.
- If you are performing test point analysis on a pre-scan netlist that has unconnected clock gaters, you should add the “set_clock_gating on” command to your dofile.

Specify Test Point Type

By default the tool generates test points that are specifically targeted to reducing deterministic pattern count. You can specify the test point type.

Prerequisites

- The tool must be in SETUP mode to specify the test point type.

Procedure

```
set_test_point_type edt_pattern_count
```

Results

The tool inserts test points to improve EDT pattern count. For more information regarding the command, refer to the command description for [set_test_point_types](#).

Inserting Test Points When Using the EDT Skeleton IP Flow

The EDT skeleton flow is unique in that the EDT IP is created before having a proper netlist. This means that certain estimates are made including the chain length and chain count.

When creating the EDT skeleton IP, it is important to account for test points if they are to be used. That means that you must account for the total number of test points in the estimates for chain length and chain count when creating the skeleton netlist.

After skeleton IP is created and inserted into the netlist, either the pre- or post-scan test point insertion flow will be followed. See [Analyze and Insert Test Points in a Pre-Scan Design](#) or [Analyze and Insert Test Points in a Post-Scan Design](#). The most important aspect to having EDT IP pre-existing in the netlist is that you must ensure that the test points are not inserted into the EDT IP. You accomplish this by issuing a command similar to the following in Analysis system mode:

```
“catch_output { add_notest_points [get_instances -of_module [get_module *_edt_*]] }”
```

Once the Verilog gate-level netlist is available, you can insert test points on the pre-scan inserted netlist or post-scan inserted netlist as shown in [Figure 6-5](#).

Test Points for LBIST Test Coverage Improvement

You can use test points to improve test coverage for random pattern resistant faults.

Test Point Analysis and Insertion Step Overview	229
Test Point Analysis With Small LBIST Pattern Counts	230
Test Coverage Reporting During Test Point Analysis for LogicBIST Coverage Improvement	231
Incremental Relevant Test Coverage Report	231
False Paths and Test Coverage	231
Fault Classes and Fault Analysis	232
“Blocked by xbounding” Faults	232
Uncontrollable/Unobservable Faults	232
Example Test Report	232

Test Point Analysis and Insertion Step Overview

In the test point analysis and insertion step of the hybrid TK/LBIST flow, you generate and insert test points into the netlist to achieve high test coverage.

During test point analysis and insertion, you add random pattern test points to certain locations in your design. By adding these test points, you can increase the testability of the design by improving controllability or observability.

[Figure 6-5](#) illustrates this step.

Note

 You generate and insert the test points as a separate step from scan insertion—you cannot perform the two operations together.

At the conclusion of test point analysis and insertion, you write out both the modified design netlist containing the test points and a dofile containing all the necessary steps for the next step of the flow, Scan Insertion and X-Bounding.

Requirements

Performing test point analysis and insertion has certain requirements. You must adhere to the following:

- Test point insertion needs a gate-level Verilog netlist and a Tessent Cell Library.
- You can read in functional SDC so the tool omits adding any test points to multicycle paths or false paths—see the [read_sdc](#) command.
- The tool identifies potential scan candidates for correct controllability/observability analysis. To ensure that eventual non-scan cells are not used as the destination for

observe points or source for control points, you should declare all the non-scan memory elements during test point insertion using the [add_nonscan_instances](#) command.

- You should define black boxes using the [add_black_boxes](#) command so that test point analysis can incorporate this information.

Specify Test Point Type

By default the tool generates test points that are specifically targeted to reduce the deterministic pattern counts. You can specify the test point type.

Prerequisite

The tool must be in SETUP mode to specify the test point type.

Procedure

```
set_test_point_type lbist_test_coverage
```

Results

The tool inserts test points to improve random pattern testability of a design. For more information regarding the command, refer to the command description for [set_test_point_types](#).

For more information, refer to [What are Test Points?](#)

Test Point Analysis With Small LBIST Pattern Counts

The estimated test coverage when inserting test points for LBIST coverage is based on detection probabilities of faults for the specified number of patterns (using the “`set_test_point_analysis_options -pattern_count_target`” command option).

Just as for statistical samples, the margin of error increases when the number of patterns is smaller. If the target pattern set is less than 20k patterns and an accurate figure for the test coverage is needed, it is recommended to fault simulate the netlist with test points, rather than rely on the coverage estimate of the test point analysis run.

Test Coverage Reporting During Test Point Analysis for LogicBIST Coverage Improvement

Test point analysis uses a fault population of all possible collapsed stuck-at faults. The test coverage is calculated by assessing, for each fault, the probability that the fault will be detected by a set of random patterns as specified by the -pattern_count_target option of the set_test_point_analysis_options command.

Common terms associated with test coverage are:

- Total Number of Faults — The sum total of all possible stuck-at faults.
- Testable Faults — Faults that can be detected by externally applied patterns.
- Logic BIST Testable Faults — Testable Faults that can be detected by Logic BIST patterns. Testable Faults that are not “Logic Bist Testable” are either faults that are “Blocked by xbounding” or Uncontrollable/Unobservable faults.
- Estimated Test Coverage — The (estimated) coverage of the “Testable Faults.”
- Estimated Maximum Test Coverage — The upper limit of the test coverage that might be achieved if enough test points were inserted.
- Estimated Relevant Test Coverage — The (estimated) coverage of the “Logic Bist Testable” faults.

Incremental Relevant Test Coverage Report	231
False Paths and Test Coverage.....	231
Fault Classes and Fault Analysis	232
“Blocked by xbounding” Faults	232
Uncontrollable/Unobservable Faults	232
Example Test Report.....	232

Incremental Relevant Test Coverage Report

The Incremental Relevant Test Coverage Report shows the progress that the tool makes while selecting test points. This report is updated after every 50 test points. The report shows the total number of test points that have been selected (TPs), and the breakdown in number of control points (CP) and observe points (OP), as well as the relevant test coverage for that number of test points (TC).

False Paths and Test Coverage

A false path is considered an X-source and is bound when it reaches a scan flop. Therefore, false paths can increase the number of faults reported as “Blocked by xbounding.”

The tool does not insert a test point on a false path. This may or may not impact the coverage after test point analysis. The tool tries to find other locations for test points that may not be at the most optimal locations. This would result in a smaller increase in the test coverage, and the tool may insert more test points, if necessary, to reach the requested target coverage.

Fault Classes and Fault Analysis

Test point analysis does not perform fault analysis in the same way as is done for ATPG. Test point analysis does not classify individual faults or use fault classes. Rather, test point analysis calculates the probability that each fault is detected by a set of random patterns, and from these probabilities, calculates a test coverage estimate.

“Blocked by xbounding” Faults

An X-source will be bound by an X-bounding mux. The faults on gates that are blocked by the X-bounding mux, as well as the faults at the “blocked” input of the X-bounding mux cannot be detected by Logic BIST patterns and are reported as “Blocked by xbounding.”

Uncontrollable/Unobservable Faults

Logic gates that are constant due to constraints do not toggle during Logic BIST. Therefore, many faults on these gates cannot be detected because the gate cannot be controlled to the opposite value. These constant gates may also block fault propagation. Faults that cannot be observed because they are blocked by gates that are constant are reported as “Uncontrollable/Unobservable.”

Example Test Report

The number of “Testable Faults” and the number of “Logic Bist Testable” faults is larger after test point insertion as these numbers include the (testable) faults at the new test point logic.

The following is a snippet of a log file showing the test coverage reporting before and after selecting test points.

```

// Test Coverage Report before Test Point Analysis
// -----
// Target number of random patterns          10000
//
// Total Number of Faults                  1876994
//   Testable Faults                      1850623  ( 98.60%)
//     Logic Bist Testable                1788912  ( 95.31%)
//     Blocked by xbounding               44      ( 0.00%)
//     Uncontrollable/Unobservable       61667   ( 3.29%)
//
// Estimated Maximum Test Coverage        96.67%
// Estimated Test Coverage (pre test points) 89.13%
// Estimated Relevant Test Coverage (pre test points) 87.98%
//
//
// Incremental Test Point Analysis
// -----
// TPs 100 = 43 (CP) + 57 (OP), Est_TC 92.21
// TPs 200 = 55 (CP) + 145 (OP), Est_TC 92.98
// TPs 300 = 79 (CP) + 221 (OP), Est_TC 93.39
// TPs 400 = 105 (CP) + 295 (OP), Est_TC 93.60
// TPs 500 = 139 (CP) + 361 (OP), Est_TC 94.01
// TPs 600 = 149 (CP) + 451 (OP), Est_TC 94.25
// TPs 700 = 168 (CP) + 532 (OP), Est_TC 94.46
// TPs 800 = 205 (CP) + 595 (OP), Est_TC 94.61
// TPs 900 = 235 (CP) + 665 (OP), Est_TC 94.70
// TPs 1000 = 252 (CP) + 748 (OP), Est_TC 94.79
// TPs 1100 = 282 (CP) + 818 (OP), Est_TC 94.85
// TPs 1200 = 306 (CP) + 894 (OP), Est_TC 94.87
// TPs 1300 = 332 (CP) + 968 (OP), Est_TC 94.94
// TPs 1400 = 353 (CP) + 1047 (OP), Est_TC 95.02
// TPs 1500 = 437 (CP) + 1063 (OP), Est_TC 95.07
//
// Incremental optimization to find more effective test points is in
// progress. The final distribution of control and observe points may
// change.
//
// Test Coverage Report after Test Point Analysis
// -----
// Target number of random patterns          10000
//
// Total Number of Faults                  1876994
//   Testable Faults                      1850623  ( 98.60%)
//     Logic Bist Testable                1788912  ( 95.31%)
//     Blocked by xbounding               44      ( 0.00%)
//     Uncontrollable/Unobservable       61667   ( 3.29%)
//
// Estimated Test Coverage (post test points) 95.31%
// Estimated Relevant Test Coverage (post test points) 97.84%

```

Test Points for Hybrid TK/LBIST Design

Improving Test Coverage and Pattern Count Reduction 234

Improving Test Coverage and Pattern Count Reduction

For some designs, you may require test coverage improvement as well as pattern count reduction. For these designs, you need test points which target both requirements. VersaPoint test points should be enabled to target both `edt_pattern_count` and `lbist_test_coverage`.

This is explained below.

Test points may be inserted in either a pre- or post-scan netlist, as described in [Figure 6-5](#).

Requirements

Performing test point analysis and insertion has certain requirements. You must adhere to the following:

- Test point insertion needs a gate-level Verilog netlist and a Tesson Cell Library.
- You can read in functional SDC so the tool omits adding any test points to multicycle paths or false paths—see the [read_sdc](#) command.
- The tool identifies potential scan candidates for correct controllability/observability analysis. To ensure that eventual non-scan cells are not used as the destination for observe points or source for control points, you should declare all the non-scan memory elements during test point insertion using the [add_nonscan_instances](#) command.
- You should define black boxes using the [add_black_boxes](#) command so that test point analysis can incorporate this information.

Specify Test Point Type

By default the tool generates test points that are specifically targeted to reduce the deterministic pattern counts. You can specify the test point type.

Prerequisite

The tool must be in SETUP mode to specify the test point type.

Procedure

```
set_test_point_type edt_pattern_count lbist_test_coverage
```

Results

The tool inserts test points to both reduce both pattern count and improve random pattern testability. For more information regarding the command, refer to the command description for [set_test_point_types](#).

For more information, refer to [What are Test Points?](#)

Test Points for ATPG Test Coverage Improvement

This section describes how you can insert test points to improve ATPG test coverage.

Test Points for Improving the Test Coverage of Deterministic Patterns 236

Test Points for Improving the Test Coverage of Deterministic Patterns

Test points for BIST coverage improvements are typically used for improving random pattern testability of a design, however they may also be used to improve ATPG coverage of undetected faults after ATPG has been run. They are based on algorithms that use gate-level testability measures to identify random pattern resistant faults and insert control and observe points to improve the overall testability of a design.

Test points can be also used to improve coverage of certain hard-to-detect, or ATPG-untestable faults targeted by deterministic patterns. This section describes how to use test points to improve the overall deterministic test coverage.

The example below describes how to perform test point analysis for a set of target faults that are left undetected (AU, UC, UO, PT or PU) after performing ATPG. In some cases, if the major source of undetected faults is ATPG aborted faults (UD.AAB), test points may help without the need to read in the fault list to target test points for coverage. The following steps illustrate the process:

1. When atpg has completed, write out the fault list. For example:

ANALYSIS> write_faults fault_list

2. In the dft -test_points context, in analysis mode, read this fault list and perform test point analysis. The fault list from ATPG must be read with the –retain switch to target coverage of undetected faults, otherwise test points will default to all faults, which will mean targeting the specific test points type selected (not for coverage improvement). The test point type must be set to only “set_test_point_type lbist_test_coverage” to target undetected faults from the retained fault list, otherwise test points will default to all faults for the specified test point type (for example if edt_pattern_count is used or combined with lbist_test_coverage). Please see the following example:

```
SETUP> set_context dft -test_points -no_rtl
SETUP> set_test_point_type lbist_test_coverage
SETUP> set_system_mode analysis
ANALYSIS> read_faults fault_list -retain
ANALYSIS> analyze_test_points
```

Test point analysis will only consider the undetected faults in the fault population and generate test points specifically for those faults.

See [Analyze and Insert Test Points in a Pre-Scan Design](#) or [Analyze and Insert Test Points in a Post-Scan Design](#) for full information.

3. After inserting the test points in the netlist, run atpg again. Because of the test points, the atpg will likely be able to generate patterns for many of the previously undetected faults.

By default, test point analysis uses a fault population of all possible stuck-at faults. You can create a custom set of target faults in analysis mode with the commands “add_faults,” “delete_faults,” and/or “read_faults.” When you use a custom set of target faults then the fault coverage that is reported by “analyze_test_points” is based on that fault set. Test coverage is calculated by assessing, for each fault, the probability that it will be detected by a random pattern set as specified by the “-pattern_count_target” option of the “set_test_point_analysis_options” command.

Only stuck-at fault sets will be supported for test point analysis. Faults in a fault list that is read with the “read_faults” command will be interpreted as stuck-at faults, if possible. When the fault set contains faults that cannot be interpreted as stuck-at faults (such as bridging faults or UDFM faults) then an error will be given. The “set_fault_type” command is not supported in the “dft -testpoints” context. The default fault type in this context is “stuck.”

The fault set that you create in the analysis system mode of the “dft -test_points” context will not be modified by the “analyze_test_points” command. Specifically, this command will not modify the class of any of the faults in the fault set. The fault set will be cleared when you transition out of the analysis system mode. That is, when the “insert_test_logic” command is executed then the system mode is transitioned to insertion mode and the fault set is cleared. Also, if you transition back to the setup system mode from the analysis mode the fault set is cleared. The commands “report_faults” and “write_faults” can be used to print out the target fault set before transitioning out of analysis mode.

Targeted Test Faults Example

In the following example, the faults in the fault file *targetFault.list* are read in and the fault class for each of these faults is retained.

The command analyze_test_points identifies test points for improving the testability of the undetected faults in the fault set and tries to achieve 100 percent coverage of these faults with a maximum of 700 test points.

```
SETUP> set_test_point_type lbist_test_coverage
SETUP> set_system_mode analysis
ANALYSIS> read_faults targetFault.list -retain
ANALYSIS> set_test_point_analysis_options -total 700 -test_coverage_target 100
ANALYSIS> analyze_test_points
```

Note that because the fault class of each fault in the file is retained, only undetected faults will be targeted by test point analysis. Faults that have been detected already are ignored by test point analysis.

The following is a snippet from the log file of a run with targeted faults after the `analyze_test_points` command is issued:

```
// Test Coverage Report before Test Point Analysis
// -----
// Target number of random patterns      1000
//
// Total Number of Targeted Faults      10682
//   Testable Faults                  10682 (100.00%)
//     Logic Bist Testable           9995 ( 93.57%)
//     Blocked by xbounding          0 ( 0.00%)
//     Uncontrollable/Unobservable  687 ( 6.43%)
//
// Estimated Maximum Test Coverage      93.57%
// Estimated Test Coverage (pre test points) 36.31%
// Estimated Relevant Test Coverage (pre test points) 38.81%
//
//
// Inserted 6 observe points for unobserved gates.
//
.....
.....
.....
.....
// Test point analysis completed: maximum number of test points has been
// reached.
//
// Inserted Test Points                700
//   Control Points                   347
//   Observe Points                  353
//
//
// Test Coverage Report after Test Point Analysis
// -----
// Target number of random patterns      1000
//
// Total Number of Faults              10682
//   Testable Faults                  10682 ( 100.00%)
//     Logic Bist Testable           10169 ( 95.20%)
//     Blocked by xbounding          0 ( 0.00%)
//     Uncontrollable/Unobservable  422 ( 3.95%)
//
// Estimated Test Coverage (post test points) 95.12%
// Estimated Relevant Test Coverage (post test points) 99.91%
```

Note

 The Test Coverage Report only includes the target faults and not other faults such as detected faults (DI, DS). Also note that the “before” and “after” test coverage reports show that the (random pattern) “Estimated Test Coverage” has dramatically been improved by inserting test points (from 35.88 percent to 94.04 percent).

Note

 The number of “Uncontrollable/Unobservable” faults (422) in the report after test point analysis is lower than the number of these faults before test point analysis (687). This is due to the “6 observe points for unobserved gates”. These six observe points are not counted in the total of 700 test points, but are over and above the 353 observe points that together with 347 control points make up the total of 700 test points.

Test Points Special Topics

This section describes additional topics related to test points.

Control Point and Observe Point Enable Signal Handling	240
Test Point Analysis With Multiple Power Domains	241
Test Point Analysis Multicycle and False Path Handling	242
Test Point Analysis Critical Path Handling	244
PrimeTime Script for Preventing Test Points on Critical Paths	245
User-Defined Test Points Handling	247
Test Point Deletion.....	249
delete_test_points Command	249
Modifying the Test Point Dofile	249
Back to Back Command Handling.....	250
Static Timing Analysis for Test Points.....	250
Test Points and At-Speed Testing.....	251
Clock Selection for Test Point Flops	252
Example of Test Point Insertion With Custom Prefix for Inserted Logic	252

Control Point and Observe Point Enable Signal Handling

By default, the tool uses the *test_point_en* signal for enabling the control and observe points.

It is recommended that you use separate enable signals for control and observe points just to keep their controls independent and flexible across pattern sets.

Note

 If the `add_dft_signals` command was used in a previous step to define the `control_test_point_en` and `observe_test_point_en` signals, then the specified primary input or internal connection point will be used instead of the default which is to create a new primary input called "`test_point_en`".

You can use the `set_test_point_insertion_options` command to specify different enable signals for the control and observe points as follows:

- The following example uses separate control point enable and observe point enable signals:

```
set_test_point_insertion_options -control_point_enable control_tp_enable
set_test_point_insertion_options -observe_point_enable obs_tp_enable
```

- The following example uses separate control point enable and observe point enable signals coming from different internal register outputs:

```
set_test_point_insertion_options -control_point_enable CR_reg/Q \
                                -observe_point_enable TR_reg[0]/Q
```

Test Point Analysis With Multiple Power Domains

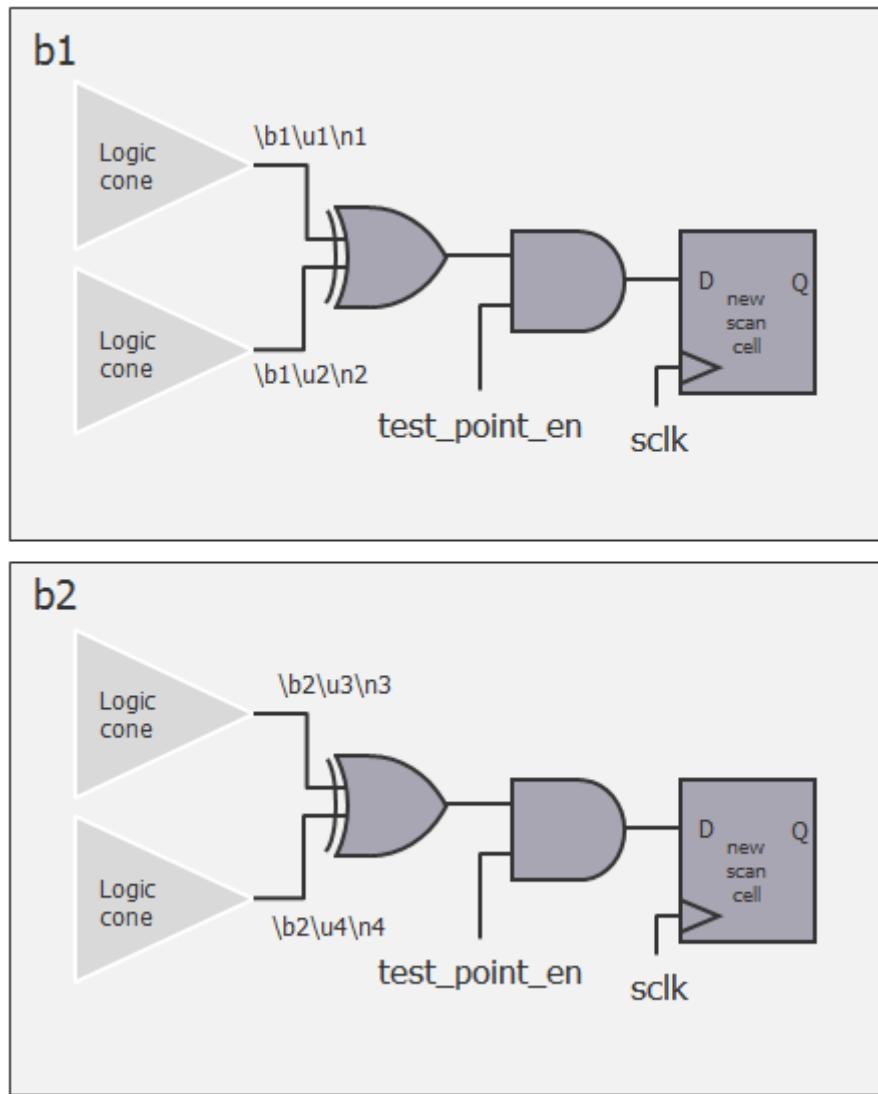
If your design contains multiple power domains, you should not share test points across power domains.

By loading power data with the [read_cpf](#) or [read_upf](#) file, you prevent any test point sharing across power domains.

Here is an example of a (partial) CPF file:

```
set_design design_top
create_power_domain -name PD1 -default
create_power_domain -name PD2 -instances {b1}
create_power_domain -name PD3 -instances {b2}
```

[Figure 6-6](#) shows an example of two blocks b1 and b2 which are in different power domains. In this case, the test point at \b1\u1\z will never be merged with any of the test points in the \b2 block.

Figure 6-6. Test Point Sharing Across Power Domains

Test Point Analysis Multicycle and False Path Handling

During test point analysis and insertion, you identify multicycle paths and false paths in a functional SDC file that you read into the tool using the `read_sdc` command. The tool does not insert observe points or control points on these paths.

For more information about the `read_sdc` command, refer to the command descriptions in the *Tessent Shell Reference Manual*.

If you want to add test points in the MCP (Multicycle Path)/false paths for slow-speed tests, then do not read in the functional SDC in the tool.

On the other hand, if you do not have a functional SDC and would like to exclude test points from MCP/FP, then you use the following command,

```
set_test_point_analysis_options -exclude_cross_domain_paths on
```

If a large portion of the design is excluded from test point insertion due to the MCP/false path timing exceptions listed in the SDC file, the tool will generate a warning message when the `analyze_test_points` command is issued. For example:

```
// command: analyze_test_points
// Analyzing false and multicycle paths ...
// False and multicycle path summary : 112 false paths, 25 multicycle
// paths
// Warning: The paths have 5 errors.
//           Use the commands "report_false_paths -debug_error" and
//           "report_multicycle_paths -debug_error" to
//           report the causes of the errors/warnings.
// Warning: Test points cannot be inserted at 1534865 (20.6%) gate-pins.
//           This may increase the number of test points needed to reduce
//           pattern counts.
//           1302614 (17.5%) gate-pins are located on clock lines or on the
//           scan path.
```

If you see the above warning message, and the portion of the design excluded from insertion of test points is more than about 20 percent, this may impact the ability to insert test points that improve test coverage during slow-speed tests. To avoid this, do not read the SDC file during test point analysis. During at-speed transition tests, however, you should read the SDC file for ATPG so that the MCP/false paths are excluded.

During transition tests, the control points capture themselves during the capture cycle and will not create transitions, but the observe points in the timing exceptions paths may capture invalid values. To prevent this, turn off the observe points by setting the observe control enable to 0 during transition at-speed tests. For stuck-at tests, both the observe and control points can be used. Make sure you have separate enables for the observe and control points that are inserted by the tool so you can independently manage the control and observe points.

Test Point Analysis Critical Path Handling

During test point analysis, a critical path which was not specified through the reading of a functional SDC file may be excluded such that no observe or control test point is added along such a path.

- To exclude specific instances or specific paths from test point analysis and insertion use the add_notest_points command.

```
add_notest_points {pin.pathname...
| instance.pathname... | instance_expression \ [-Observe_scan_cell]}
| -Path filename
```

For more information about this command, refer to [add_notest_points](#) in the *Tessent Shell Reference Manual*.

The command report_notest_points can be used to report paths and instances that are excluded from test point analysis and insertion. For more information about this command, refer to [report_notest_points](#) in the *Tessent Shell Reference Manual*.

PrimeTime Script for Preventing Test Points on Critical Paths..... 245

PrimeTime Script for Preventing Test Points on Critical Paths

The script “*tessent_write_no_tpi_paths.tcl*” is a tcl script that reads a list of critical paths extracted by PrimeTime and marks them as not valid test point locations.

One way to minimize problems during timing closure is to identify critical paths prior to test point analysis and insertion. This approach depends on accurately identifying the critical paths, and this typically requires placement and global routing information. The provided PrimeTime script can be used to extract a list of critical paths based on the current PrimeTime environment (preferably with placement information) and generate Tesson Shell commands that mark these paths with the appropriate attributes to avoid inserting test points on them.

PrimeTime Script Usage **246**

PrimeTime Script Usage

The PrimeTime script translates a list of critical paths extracted by PrimeTime into a list of set_attribute_value commands that mark all the pins on the critical paths with the appropriate attributes.

Usage

`tessent_write_no_tpi_paths $paths filename`

The script is located at

`<Tessent_Tree_Path>/share/TestPoints/tessent_write_no_tpi_paths.tcl`

Description

The “`tessent_write_no_tpi_paths.tcl`” script translates a list of critical paths extracted by PrimeTime into a list of `set_attribute_options` commands that mark all the pins on the critical paths with the appropriate attributes.

You would typically do this before Test Point Analysis and Insertion.

Arguments

- **\$paths**
Environmental variable that points to the critical paths extracted by PrimeTime.
- **filename**
Specifies the file name for the Tessent Shell dofile generated by the script.

Examples

Below is an example of how to use this script.

Step 1: From PrimeTime:

```
set paths [get_timing_path -delay_type max -max_paths 10 -nworst 1 \ -slack_lesser_than 4]
tessent_write_no_tpi_paths $paths critical_paths.dofile
```

First, use the PrimeTime “`get_timing_path`” command to extract the critical paths. This command requires a number of important parameters described below which should be tuned for your design.

Next, invoke the “`tessent_write_no_tpi_paths`” script to read in the critical paths extracted by PrimeTime and write out a Tessent Shell dofile.

Step 2: From Tessent Shell:

```
dofile critical_paths.dofile
```

The dofile adds appropriate attributes to all the pins on the critical paths to prevent test points from being placed there.

Parameters for the “get_timing_paths” command

- **-delay_type max**

Specify “-delay_type max” to target setup violations (not hold violations).

- **-max_paths number**

The parameter “-max_paths *number*” specifies the maximum number of paths that will be reported per domain. For example, if your design has two clock domains (clk1 and clk2), and there are no false-path statements, you might have four domains (clk1, clk2, clk1->clk2, and clk2->clk1).

- **-nworst**

The -nworst parameter avoids listing many “similar” paths. For example, if there are many relatively long paths between <src> and <dest>, then setting “-nworst 1” means only the path with the least slack will be reported. Set “-nworst 9999” to avoid adding test points to any path that does not have enough slack to absorb it.

- **-slack_lesser_than number**

The “-slack_lesser_than *number*” parameter controls which paths get reported. Tune this parameter based on the slack in your design.

User-Defined Test Points Handling

You can specify user-defined test points using the add_control_points and add_observe_points commands.

For more information, see the [add_control_points](#) and [add_observe_points](#) command descriptions in the *Tessent Shell Reference Manual*.

How the tool handles your user-defined test points depends on whether you have already inserted tool-generated test points as follows:

- **Adding Test Points Before Test Point Analysis**— The following command sequence demonstrates this:

```
ANALYSIS> add_control_point -location OY4 -type and
// Adds a user-defined control point
```

```
ANALYSIS> analyze_test_points
// Performs the analysis and generates test points
```

```
ANALYSIS> insert_test_logic
```

```
...
```

When you issue the [analyze_test_points](#) command, then the tool takes the user-defined test points in account.

- **Adding Test Points After Test Point Analysis** — The following command sequence demonstrates this:

```
ANALYSIS> analyze_test_points
// Performs the analysis and generates test points

ANALYSIS> add_control_point -location OY4 -type and
// Tool gives preference to the user-defined test points

ANALYSIS> insert_test_logic

...
```

If you specify user-defined test points using the `add_control_points` or `add_observe_points` commands *after* analysis but *prior* to insertion, then the tool inserts these test points into the design. If the user-defined test points are in the same location as the tool-generated test points then the user-defined test points are given preference.

Note

 If you define test points in a restricted `notest_point` region, the tool will ignore them and will issue a warning.

You can use the `add_control_points` and `add_observe_points` commands to specify individual control points and observe points with different enable signals as follows:

```
add_control_points -enable enable_pin/port
add_observe_points -enable enable_pin/port
```

Test Point Deletion

This section explains the methods for deleting test points.

After you have issued the [analyze_test_points](#) command, you cannot reduce the number of test points with the [set_test_point_analysis_options](#) command. However, you can delete the test points using one of the following methods: [delete_test_points Command](#) or [Modifying the Test Point Dofile](#).

delete_test_points Command	249
Modifying the Test Point Dofile	249

delete_test_points Command

You can delete a subset of the tool-generated test points.

Use the [delete_test_points](#) command to specify the locations that you want to remove from the list of identified test points.

You should use this method if you only have a small number of test points to delete.

Modifying the Test Point Dofile

If you have a large number of test points you need to delete, then you can do so by modifying the dofile.

Procedure

1. Use the [write_test_point_dofile](#) command to write out the dofile that contains the test points. For example:

```
ANALYSIS> write_test_point_dofile -output_file my_test_points.dofile
```

2. Edit this dofile and remove the test points you do not want.
3. From within Tesson Shell, delete all of the test points using the [delete_test_points](#) command as follows:

```
ANALYSIS> delete_test_points -all
```

4. Use the [dofile](#) command to read the modified test point dofile back into the tool. For example:

```
ANALYSIS> dofile my_test_points_modified.dofile
```

Back to Back Command Handling

The `analyze_test_points` command generates test points based on the default settings or any analysis options you specify. If you issue the `analyze_test_points` command a second time, no new test points will be generated.

```
ANALYSIS> analyze_test_points // Performs the analysis and generates test points
```

```
ANALYSIS> analyze_test_points // No new test points are generated
```

If you change any of the analysis options *before* issuing the `analyze_test_points` command a second time, the tool will generate new test points based on the current options. For example:

```
ANALYSIS> analyze_test_points // Performs the analysis and generates test points
```

```
ANALYSIS> set_test_point_analysis_options -total_number 2000
```

```
ANALYSIS> analyze_test_points // New test points are generated
```

You can use the `delete_test_points` command to delete all existing test points and generate new test points with the `analyze_test_points` command.

```
ANALYSIS> delete_test_points -all // Deletes all existing test points
```

```
ANALYSIS> analyze_test_points // Generates new test points
```

If you issue any command between back-to-back `analyze_test_points` commands that does not impact test point analysis, the tool will report the same test points again.

Static Timing Analysis for Test Points

This section describes how to run Static Timing Analysis (STA) with Test Points.

For control points there is no timing impact as the control point flop captures itself during capture cycle.

For the “control_point_en” signal that reaches the control test points, use a “set_case_analysis” which will ensure that:

- The source flop’s holding path will not be relaxed.
- The scan path from that source flop to the next SI pin will not be relaxed either.

```
> set_case_analysis control_point_en 0
```

If observe points are to be used during at-speed tests, then they must meet single-cycle timing from the point where the data gets sourced. During STA, the signal used as “observe_point_en” will be left unconstrained to allow the STA tool to check timing for both the capture and shift paths of these cells.

If timing is not closed to meet single-cycle timing on the observe points during at-speed test, you must disable that path during at-speed ATPG, by using “add_input_constraints observe_point_en -C0”.

Static Timing Analysis and Place and Route tools operate differently. It is for this reason that the “set_case_analysis” used during STA to prevent checking timing on a given path would prevent a Place and Route tool from fixing timing on the same path. During slow-speed tests, the hold-path for these cells must meet timing, so instead, a command such as “set_false_path – setup –to <all observe point flops>” should be used.

Test Points and At-Speed Testing

This section describes how to use test points during at-speed testing. Mentor Graphics recommends to close timing on observe points during timing closure.

Impact on Transition-Delay Fault

Test points can safely be used for at-speed testing for transition patterns that target gross delay defects. By definition, transition patterns use the transition fault model that targets a “gross” delay at every fault site. A transition pattern will detect faults by triggering a transition from a scan flop and capturing the result at a downstream scan flop. The fault model does not care about the path used. It is for this reason that test points used during transition patterns will not interfere with the detection of transition faults.

Control points remain static once the circuit is placed in functional mode for capture. This means that control points only help to sensitize a path and will not shorten a functional path to be smaller than the function of the circuit. The one exception is if LoS (launch off shift) patterns are used. In such a case, an at-speed transition from the control test point can be launched as scan enable drops. This is easily prevented either by defining a false path from the test point or the test point can use a non-pipelined scan enable signal while the functional logic uses a pipelined scan enable.

Impact on Timing-Aware ATPG

Unlike the transition-delay fault model, timing aware ATPG, used to detect small delay defects, considers the actual paths used during the tests. During timing-aware ATPG, the timing of the circuit is read in and ATPG automatically finds and uses the longest sensitizable paths around a fault site when creating patterns. Therefore, the addition of test points will not hinder the ability to target and detect timing-aware ATPG tests.

In cases where timing was not closed on observe points during place-and-route timing closure, it may be necessary to turn off the observe points. Set the observe-point enable signal to the off-state and continue with timing-aware ATPG as before.

Impact on Path-Delay ATPG

Path-delay testing targets specific paths, extracted from a static-timing analysis program, with specific fault sites read in to ATPG for targeting. Stimulus and response is launched down a specific path, often in a specific manner. It is for this reason that observe points should have no impact on path delay tests. When inserting test points, false and multicycle paths can be ignored during analysis by calling `read_sdc` or `add_false_path` commands during the test point insertion process.

Other Considerations

All test points inserted by Tesson software have enable signals. For more conservative results test points may be disabled to ensure only functional paths are tested during at-speed test. Even for conservative cases, it makes sense to allow control points because these are only static signals that make the functional paths more easily sensitized. In such cases, you have the option to have separate control point and observe point enable signals when inserting the test points.

Clock Selection for Test Point Flops

If you do not explicitly specify the clock for a test point flop, the tool first selects the appropriate clock domain based on the top level clocks that control the flops in the fanin and/or fanout.

During the clock domain selection, the tool will ignore any flops that are not either already existing scan cells, or targeted for scan insertion. In addition, the initial search will be limited to flops that are in the same `power_domain_island` as the test point location. When only a single clock domain is found in the fanout or fanin, then that domain will be used. Otherwise, the most referenced clock domain in the fanout of the control point (or fanin of the observe point) will be used.

Next, it chooses a connection point as described below:

- It connects to the clock port of one of the flops in the fanout (for a control point), or fanin (for an observe point). If the clock port of the connected scan cell is not directly accessible, then it will trace through sensitized paths in the clock network to find the closest possible connection point.
- If no valid connection point is found as described above, it tries to connect to the identified clock at the closest module input port moving up the hierarchy.
- Otherwise, it connects to the clock source.

Example of Test Point Insertion With Custom Prefix for Inserted Logic

To define a custom prefix that will be used during test point insertion, overriding the default prefix `ts_`, use the `set_insert_test_logic_options` command.

The following example prefixes all inserted nets and instances in the output_lbist_ready.v netlist with *demo*_:

```
SETUP> set_context dft -test_points -no_rtl
SETUP> read_verilog non_scan_design_name
SETUP> read_cell_library library_name
SETUP> set_current_design
SETUP> set_system_mode analysis
ANALYSIS> analyze_test_points
ANALYSIS> set_insert_test_logic_options -inserted_object_prefix demo_
ANALYSIS> insert_test_logic
ANALYSIS> write_design output_netlist.v
ANALYSIS> exit
```


Chapter 7

Running ATPG Patterns

This section describes how to generate patterns for ATPG after scan insertion.

Running ATPG Patterns after Tessent Scan	255
Running ATPG Patterns without Tessent Scan	257

Running ATPG Patterns after Tessent Scan

This section describes how you can take advantage of Tessent Scan while generating ATPG patterns by using the new command `import_scan_mode`.

When you use the `import_scan_mode` command, the specific scan mode for which scan insertion was performed is passed as an argument. The tool automatically reads in the TCD and for that specific mode identifies which clocks need to be added, which scan chains to trace and if applicable which OCC instances to be read in and used.

The `import_scan_mode` command performs the following tasks:

- Adds scan chains used in the scan mode.
- Adds EDT instruments used in the scan mode.
- Adds OCC instruments used in the scan mode.
- Configures scan clocks used in the scan mode.
- Configures DFT signals used in the scan mode.
- Creates or updates load_unload and shift procedures.
- Adds scan clock pulses to shift procedure.
- Adds scan enable signal forces to shift procedure.

For the `import_scan_mode` command to work properly with EDT, during scan insertion the EDT IP needs to be connected using `-edt_instance` or using `-si_connections/-so_connections` pointing to the EDT instance with `add_scan_mode` command. Refer to [Scan Insertion Flow Steps](#) in Chapter 5 for more information.

Example 1: If you are using unwrapped core and have used `edt_mode` as the DFTSignal during scan insertion, then you are passing it as an argument to `import_scan_mode`:

```
>set_context patterns -scan
```

```
# Read the library
>read_cell_library ../../library/tesson/adk.tcelllib
>read_cell_library ../../library/mem_ver/memory.lib

# Open all the previous available tsdb_outdirs
>open_tsdb ./tsdb_outdir

# Read the netlist
>read_design cpu_top -design_id gate
>import_scan_mode edt_mode
>set_system_mode analysis
>report_clocks
>report_core_instances
>create_patterns
>write_tsdb_data -replace

# Write out patterns for simulation
>write_patterns ./generated/pat.v -verilog -serial -replace -Begin 0 -End 64
>write_patterns ./generated/pat_parallel.v -verilog -parallel -replace -scan
```

Example 2: In this example a Wrapped core is setup to be run at-speed transition patterns. The **import_scan_mode** is passed the argument **int_mode** (this is the scan configuration mode) that was used during scan insertion for internal mode:

```
>set_context patterns -scan
>set_tsdb_output_directory ./tsdb_outdir
>read_cell_library ../../library/tesson/adk.tcelllib
>read_design processor_core -design_id gate
>set_current_design processor_core

# Specify a different name than what was used during scan insertion with
# add_scan_mode command
>set_current_mode edt_transition -type internal
>report_dft_signals
>import_scan_mode int_mode -fast_capture_mode on
>set_system_mode analysis
>report_core_instances
>report_clocks
>set_fault_type transition
>set_external_capture_options -pll_cycles 5 [lindex [get_timeplate_list] 0]
>create_patterns
>write_tsdb_data -replace
>write_patterns generated/processor_core_transition_serial.v -verilog -serial -replace
>write_patterns generated/processor_core_transition_parallel.v -verilog -parallel -replace
```

ATPG Setup when not using import_scan_modes:

There is no change in how you setup to run ATPG if not using **import_scan_mode**. You can set up the EDT using the TCD IP mapping flow by using **add_core_instances** and pointing to the EDT IP instance and passing the desired parameters to it. The same is true if there are OCCs present in the design.

Running ATPG Patterns without Tessent Scan

This section describes how you can run ATPG if you have used Tessent Scan in the Legacy mode (without hierarchical scan insertion), or if you have used third party tools for scan insertion.

Core mapping for ATPG provides an ease-of-use flow by automating the mapping of DFT information from one level to the next so that you can generate patterns at a higher level.

You can use this functionality in flows in which DFT is implemented bottom-up but pattern generation is run at a higher level. You can use this functionality for modular EDT flows (in which EDT and scan is implemented within a core but pattern generation is run at the next higher level) to map EDT, scan, clock, and procedures from the core level to the level at which pattern generation is to be run; this can be at a higher level core or at the chip level.

Core mapping for ATPG extends the core description functionality and use model developed for pattern retargeting to top-level ATPG. Implementing core mapping for ATPG provides you with the following benefits:

- Clean transfer of core information to the top-level.
- A simple and less error-prone use model by relying on Tessent core description files for the transfer of information. This is consistent with the pattern retargeting use model.
- Support for all combinations of uncompressed and compressed scan chains.
- Mapping or verification of procedures and clocks.

Core mapping for ATPG requires a Tessent TestKompress or Tessent FastScan license. A TestKompress license is specifically needed if EDT logic is present at the top level of the design or in any of the cores.

Core Mapping for ATPG Process Overview	257
Core Mapping Examples	265
Limitations	271

Core Mapping for ATPG Process Overview

This section provides an overview of the core mapping for ATPG process. You can use core mapping to transfer the core-level information to the top level of the design and generate patterns for the whole chip. The tool uses the Tessent core description (TCD) file to transfer core-level information to the top level of the design.

An example of the core mapping process is presented in “[Core Mapping Examples](#)” on page 265.

Tessent Core Description File

The Tessent core description (TCD) file contains the information the tool needs to map the core to the top level, including information such as a description of the EDT hardware and scan chains.

Test Procedure Retargeting for Core Mapping

The automatic mapping (transfer) of test procedure information from the core level to the top level of the design occurs when the tool transitions to analysis mode.

load_unload and shift Procedures

When cores are mapped to the chip level, chip-level load_unload and shift test procedures are also needed. If chip-level load_unload and shift test procedures do not exist, the tool automatically generates them based on the information in the core-level TCD files. This step maps the core load_unload and shift procedures from the core level, merges all of the core load_unload and shift procedures (if there is more than one core), and then merges them with the top-level test procedures (if there are any and they do not already have the needed events to drive the core instances).

Any timeplates in the core-level load_unload and shift procedures are updated when they are mapped to the top level based on the following rules:

- If the timeplate does not have any pulse statements defined, the tool uses an offset and width of 25 percent of the period as the pulse template. If the template pulse occurs before measure_po time, the tool moves the pulse window by another 25 percent.
- If non-pulse-always clock pulses are defined in the timeplate, the specification of the clock that is pulsed as the first one is used as a pulse timeplate.
- If only pulse-always clock pulses are defined in the timeplate, the tool uses the first one as a pulse template.
- If the template clock has multiple pulses, the tool only uses the first pulse specification.

Clocks and Pin Constraints

By default, the tool maps all clocks and pin constraints that are specified at the core-level to the top. Clock types are mapped based on the core-level specification. Note, the tool maps any permanently tied constraints (CT) in those files as constraints that can be overridden (C).

Even though clocks and pin constraints are automatically mapped to the top level by default, you can still define them at the top, or drive the core-level input constraints through logic such as a TAP.

Any clocks that are mapped to the top are removed when the tool exits analysis mode.

test_setup and test_end Procedures

You will typically need to provide top-level test_setup procedures and, occasionally, test_end procedures to configure instruments such as PLLs, OCCs, EDT IPs, channel multiplexer access, and so on in preparation for scan test. These instruments may be at the top level and/or within the cores. The test_setup and test_end procedures can consist of a cyclized sequence of events, and/or IJTAG [iCall](#) statements.

If any of the instruments used for scan test are inside a core, a test_setup and/or test_end procedure at the core level must have been used to initialize them. If IJTAG was used within a core-level test_setup or test_end procedure, this information is retained in the TCD file. When an instance of this core is added for mapping, the iCalls that were used at the core level are automatically mapped to the top level and added to the top-level test_setup or test_end procedure. In other words, if you use IJTAG to set up the core at the core level, the tool automatically maps those sequences to the top-level test_setup and test_end procedures without you doing anything. Of course, the top-level test_setup and test_end procedures may have additional events to configure top-level instruments.

In order for the tool to perform automatic IJTAG mapping, you must have extracted the chip-level ICL using the [extract_icl](#) command. You must also have sourced the core-level PDL.

Several [R](#) DRCs validate the presence of both ICL and PDL. Note that the tool does not store ICL and PDL in the core-level TCD file; it only stores references to their usage in the test_setup or test_end procedure.

Note

 If you have not previously extracted ICL in a separate step, you can still execute the normal core mapping flow with one change. After entering the patterns -scan context, you need to read ICL for each of the cores, extract the ICL using the [extract_icl](#) command, read the PDL file for each module, and then proceed with the rest of the core mapping flow.

Make sure to call the [extract_icl](#) command before you issue the [set_procfile_name](#) command otherwise it will be understood that the test_setup proc located in the specified proc file is actually needed to extract the ICL. The test_setup proc will be simulated and used to establish the back ground simulation values when tracing and extracting the ICL network. Because this test_setup was simulated to do ICL extraction, its name will end up in the extracted ICL and the [process_patterns_specification](#) command will give you an error if you do not refer to it using the procfile_name property inside the [AdvancedOptions](#) wrapper.

For specific instructions on performing ICL extraction, refer to “[Performing ICL Extraction](#)” in the *Tessent IJTAG User’s Manual*.

Note

 You can disable load_unload and shift procedure retargeting by executing the “`set_procedure_retargeting_options -scan off`” command. If test procedure retargeting is disabled and chip-level load_unload and shift test procedures are missing, the tool generates an error.

You can disable the automated mapping of core-level iCalls stored in the TCD file to the top level by executing the “`set_procedure_retargeting_options -ijtag off`” command. (This will not have any impact on iCalls you explicitly added into the top-level setup procedures either explicitly or indirectly using the `set_test_setup_icall` command.) If IJTAG retargeting is disabled, you must provide the needed `test_setup` and `test_end` procedures.

For more information on using IJTAG to automate `test_setup` and `test_end` procedure creation, see “[IJTAG and ATPG in Tessent Shell](#)” in the *Tessent IJTAG User’s Manual*.

Core Mapping Process

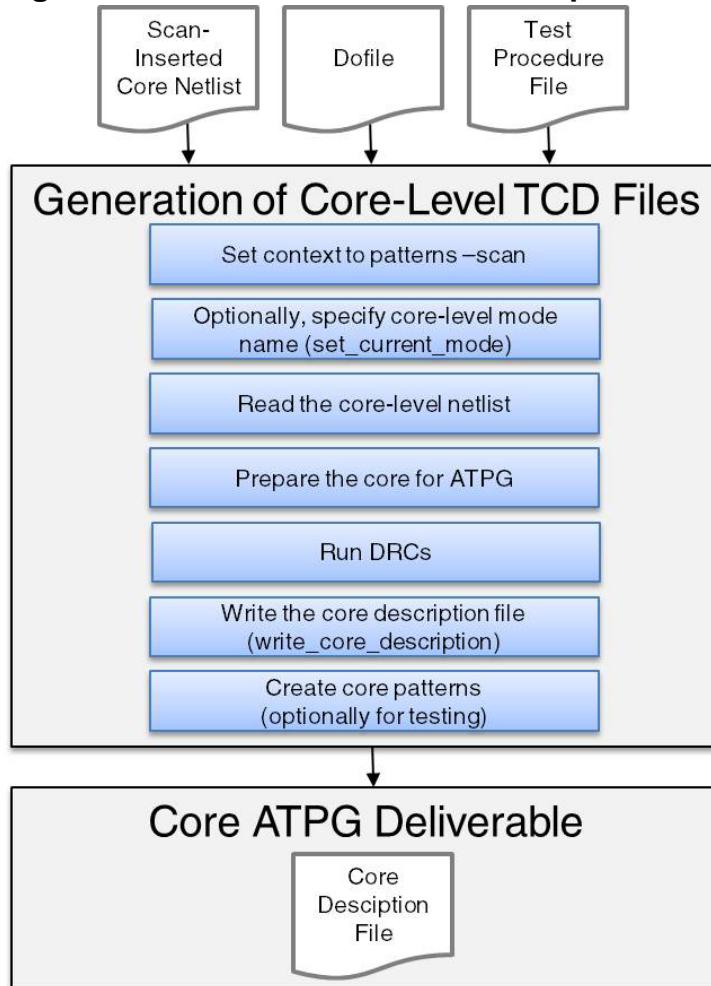
In addition to scan logic within the cores, you may also have top-level scan logic that will operate together with the core logic in the current mode when DRCs and ATPG are run. The standard core mapping process enables you to add the top-level scan logic using standard commands such as `add_scan_chains` and `add_scan_groups` commands (or using the `add_core_instances` command). The tool will merge the core-level `load_unload` and `shift` procedures with the top-level `load_unload` and `shift` procedures if you have read them in.

1. Generate a Tessent core description (TCD) file for each core that you want to map to the top. This process is illustrated in [Figure 7-1](#).

Note

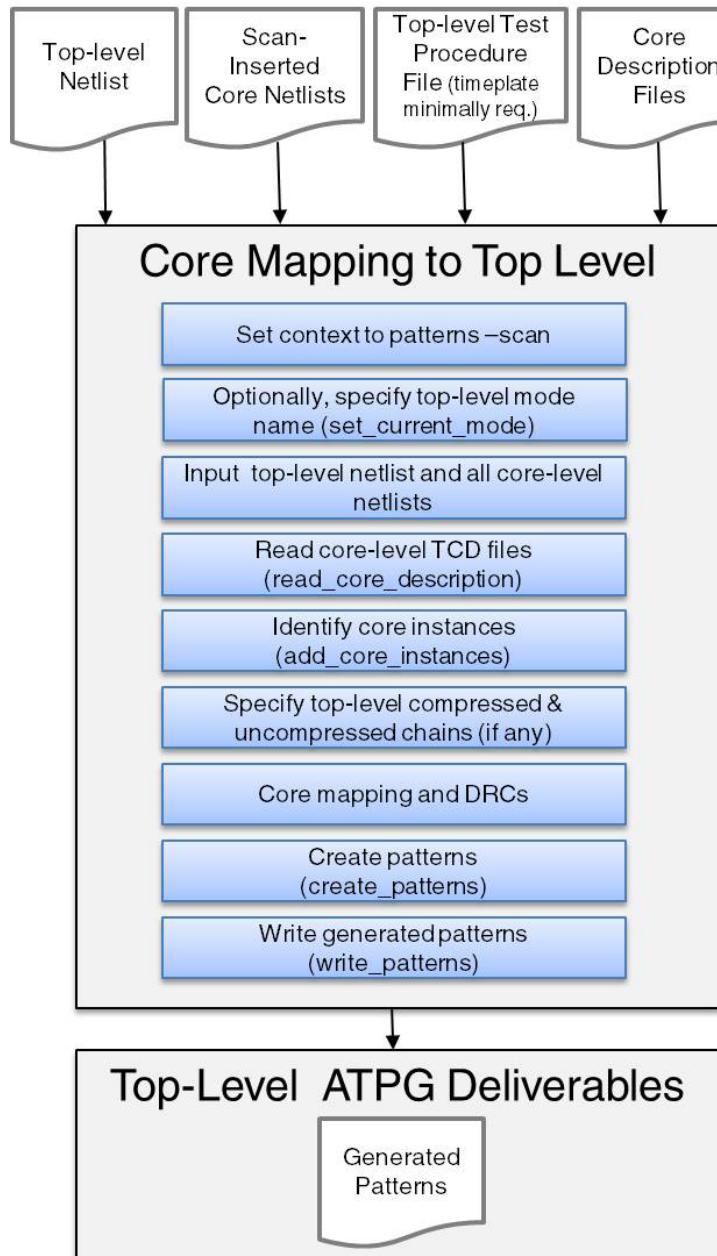
 This is the same process that you use to run DRC and ATPG at the core level, except that in this step you export the core description to the next higher level using the `write_core_description` command.

Figure 7-1. Generation of Core Description Files



2. Map the core-level TCD files to the top level. This process is illustrated in [Figure 7-2](#).

Figure 7-2. Core Mapping to Top Level



Alternative Core Mapping Process

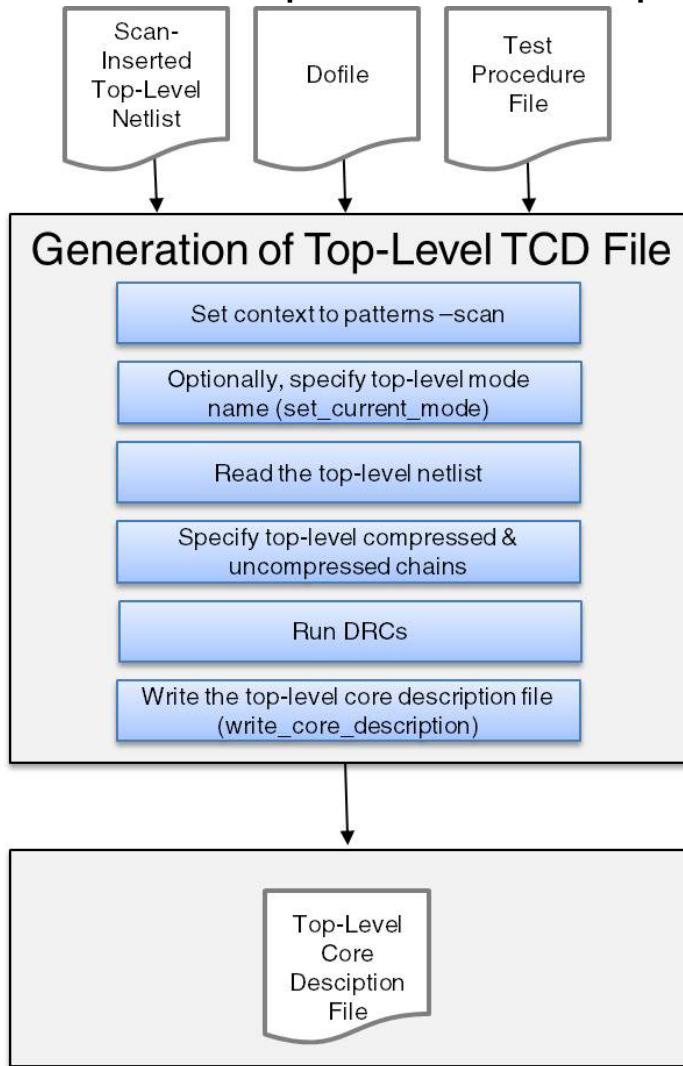
If you want to verify your top-level scan logic independently, you can use an alternative mapping process. The alternative mapping process requires that you create, validate, and then add the top-level scan logic as a core instance using the “`add_core_instances -current_design`” command.

1. Generate a Tessent core description (TCD) file for each core that you want to map to the top. This process is illustrated in [Figure 7-1](#).

2. Generate a TCD file for the scan logic present at the top level which excludes the scan logic in the cores that will be mapped and merged later. This process is illustrated in [Figure 7-3](#). This is identical to what was done at the core level in [Figure 7-1](#) except that this mode only defines the top-level logic in the design with the exception of the logic in the core that will be mapped in the next step.

The TCD file you generate in this step only describes the top-level scan logic (it is not the TCD file that describes everything including the top-level and mapped core information).

Figure 7-3. Generation of Top-Level TCD File for Top-Level Logic

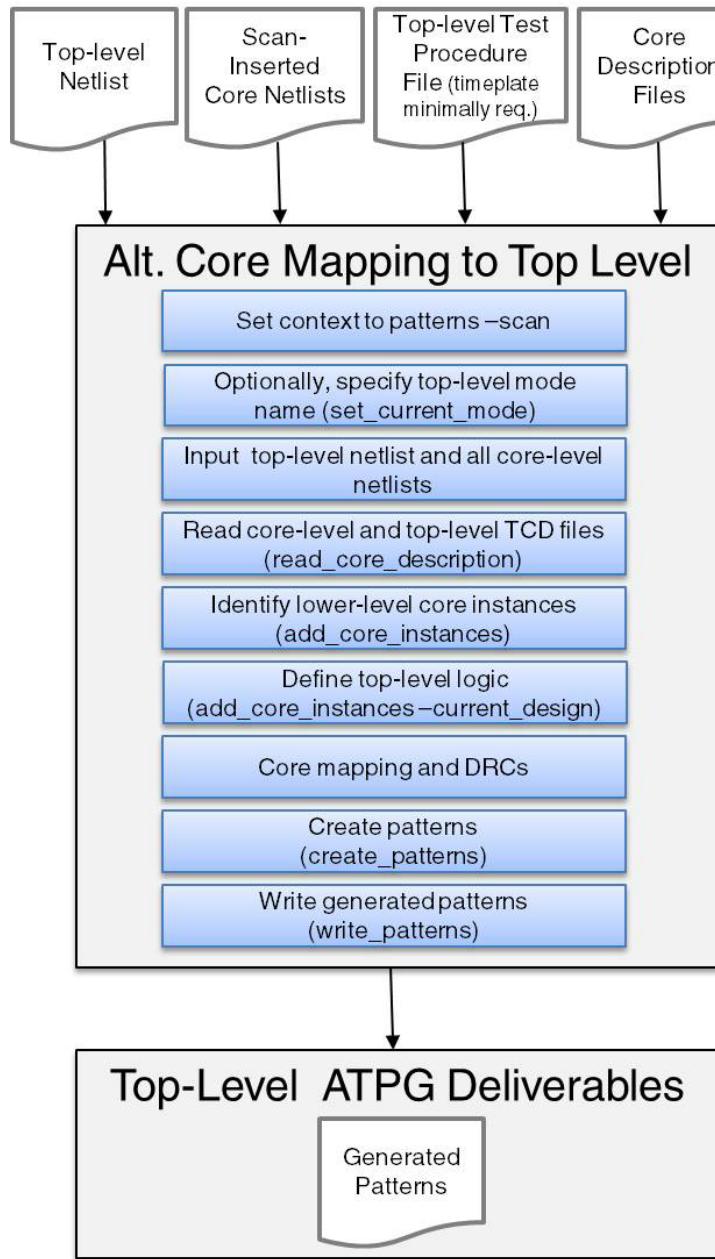


3. Map the core-level TCD files to the top level.

This process is illustrated in [Figure 7-4](#). In this alternative flow, if scan logic exists at the top level, you will have two different views (modes) of the top level: a view that only includes the top-level scan logic and the procedures that operate that logic (if there is any), and the merged top-level view created after core mapping which, in addition to the

top-level scan logic and procedures, also includes all of the core-level scan logic mapped to the top. If there are two views of the top level, you must specify a different top-level mode name in either this step or the next step, using the `set_current_mode` command.

Figure 7-4. Alternate Process for Core Mapping to Top Level



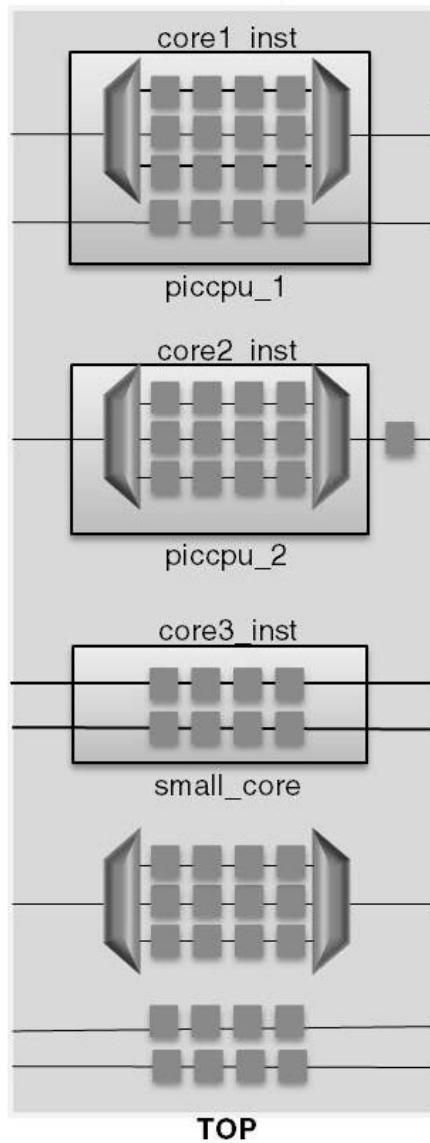
This flow matches the process shown in [Figure 7-2](#), except that in addition to adding core instances for the lower-level cores, you use the “`add_core_instances -current_design`” command to define the top-level scan logic that was stored in the TCD in step [2](#).

Core Mapping Examples

The following two examples demonstrate the core mapping use model and the alternative core mapping use model.

Both of these examples are based on the design shown in [Figure 7-5](#). The design has three cores: one instance of the piccpu_1 core containing both compressed and uncompressed scan chains, one instance of the piccpu_2 core containing only compressed chains, and one instance of the small_core core containing only uncompressed scan chains. TOP is the top design where both compressed and uncompressed chains exist.

Figure 7-5. Mapping of Cores to Chip Level



Used in core mapping beta doc—example_design.jpg

Core Mapping Example

As shown in this example, after you have generated a TCD file for each of the cores in your design, you map the cores to the chip level using those TCD files, add any additional scan logic, and then generate patterns for the entire design.

```

# Set the proper context for core mapping and subsequent ATPG
set_context pattern -scan

# Read cell library (library file)
read_cell_library technology.tcelllib

# Read the top-level netlist and all core-level netlists
read_verilog generated_1_edt_top_gate.vg generated_2_edt_top_gate.vg \
generated_top_edt_top_gate.vg

# Specify the top level of design for all subsequent commands and set mode
set_current_design
set_current_mode top_design_mode

# Read all core description files
read_core_descriptions piccpu_1.tcd
read_core_descriptions piccpu_2.tcd
read_core_descriptions small_core.tcd

# Bind core descriptions to cores
add_core_instances -instance core1_inst -core piccpu_1
add_core_instances -instance core2_inst -core piccpu_2
add_core_instances -instance core3_inst -core small_core

# Specify top-level compressed chains and EDT
dofile generated_top_edt.dofile

# Specify top-level uncompressed chains
add_scan_chains top_chain_1 grp1 top_scan_in_3 top_scan_out_3
add_scan_chains top_chain_2 grp1 top_scan_in_4 top_scan_out_4

# Report instance bindings
report_core_instances

# Change to analysis mode
set_system_mode analysis

# Create patterns
create_patterns

# Write patterns
write_patterns top_patts.stil -stil -replace

# Report procedures used to map the core to the top level (optional)
report_procedures

```

Alternative Core Mapping Example

As shown in this alternative example flow, after you have generated a TCD file for each of the cores in your design and for the top-level scan logic, you map the cores to the chip level using those TCD files, and then generate patterns for the entire design.

Notice that in this flow you must add the top-level logic as a core instance using the “`add_core_instances -current_design`” command. You must also specify a top-level mode name

using `set_current_mode` if you did not specify one during TCD file generation; the mode must have a different name than the mode in the TCD files to avoid overwriting the first one.

The differences between this flow and the standard core mapping flow are shown in **bold** font.

```
# Set the proper context for core mapping and subsequent ATPG
set_context pattern -scan

# Read cell library (library file)
read_cell_library technology.tcelllib

# Read the top-level netlist and all core-level netlists
read_verilog generated_1_edt_top_gate.vg generated_2_edt_top_gate.vg \
generated_top_edt_top_gate.vg

# Specify the top level of design for all subsequent commands and set mode
set_current_design
set_current_mode top_design_mode

# Read all core description files
read_core_descriptions core1.tcd
read_core_descriptions core2.tcd
read_core_descriptions core3.tcd
read_core_descriptions top_only.tcd

# Bind core descriptions to cores
add_core_instances -instance core1_inst -core piccpu_1
add_core_instances -instance core2_inst -core piccpu_2
add_core_instances -instance core3_inst -core small_core
add_core_instances -current_design -core my_design

# Report instance bindings
report_core_instances

# Change to analysis mode
set_system_mode analysis

# Create patterns
create_patterns

# Write patterns
write_patterns top_patts.stil -stil -replace

# Report procedures used to map the core to the top level (optional)
report_procedures
```

Core Mapping Example With IJTAG

If you used IJTAG at the core level, by default, the tool will try to map any core-level iCalls to the chip level as described in section “[test_setup and test_end Procedures](#).” As shown in this example, after you have generated a TCD file for each of the cores in your design, you need to extract the top-level ICL and source the necessary PDL. When you have done this, you can map the cores to the chip level using the previously generated TCD files, add any additional scan logic, and then generate patterns for the entire design.

The differences between this flow and the standard core mapping flow are shown in bold font.

```
# Set the proper context for core mapping and subsequent ATPG
set_context pattern -scan

# Read cell library (library file)
read_cell_library technology.tcelllib

# Read the top-level netlist and all core-level netlists
read_verilog generated_1_edt_top_gate.vg generated_2_edt_top_gate.vg \
generated_top_edt_top_gate.vg

# Read core-level ICL. ICL may be automatically loaded when the
# set_current_design command is issued as described in section
# "Top-Down and Bottom-Up ICL Extraction Flows" in the Tessonnt IJTAG
# User's Manual
read_icl piccpu_1.icl
read_icl piccpu_2.icl
read_icl small_core.icl

# Specify the top level of design for all subsequent commands and set mode
set_current_design
set_current_mode top_design_mode

# Extract top-level ICL and write the extracted ICL for later usage
extract_icl
write_icl -output_file top_design.icl

# Source core-level PDL
source generated_1_edt.pdl
source generated_2_edt.pdl
source generated_3_edt.pdl

# Read all core description files
read_core_descriptions piccpu_1.tcd
read_core_descriptions piccpu_2.tcd
read_core_descriptions small_core.tcd

# Bind core descriptions to cores
add_core_instances -instance core1_inst -core piccpu_1
add_core_instances -instance core2_inst -core piccpu_2
add_core_instances -instance core3_inst -core small_core

# Specify top-level compressed chains and EDT
dofile generated_top_edt.dofile

# Specify top-level uncompressed chains
add_scan_chains top_chain_1 grp1 top_scan_in_3 top_scan_out_3
add_scan_chains top_chain_2 grp1 top_scan_in_4 top_scan_out_4

# Report instance bindings
report_core_instances

# Change to analysis mode
set_system_mode analysis

# Create patterns
create_patterns

# Write patterns
```

```
write_patterns top_patts.stil -stil -replace
# Report procedures used to map the core to the top level (optional)
report_procedures
```

Limitations

The limitations of the Core Mapping for ATPG functionality are listed here.

- Core-level scan pins, whether compressed or uncompressed, must connect to the top level through optional pipeline stages. Uncompressed chains from different cores cannot be concatenated together before being connected to the top level.

Chapter 8

Test Pattern Generation

It is important to understand the overall process for generating test patterns for your design. You use the ATPG tool and possibly ModelSim, depending on your test strategy, to perform these tasks.

The sections that follow outline the steps required for test generation.

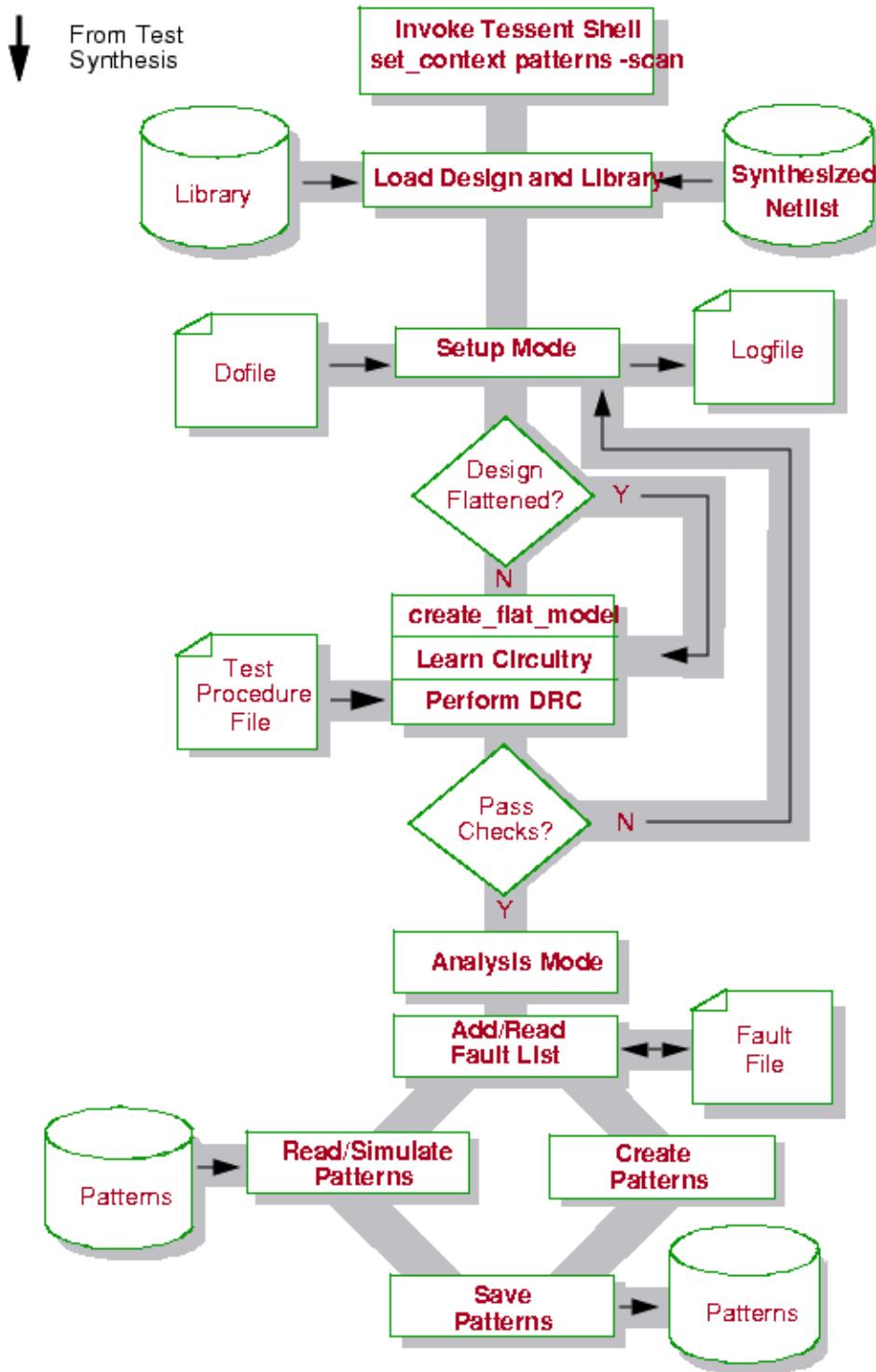
ATPG Basic Tool Flow	275
ATPG Tool Inputs and Outputs	277
ATPG Process Overview	279
ATPG Procedures	285
Tool Invocation	285
Circuit Behavior Setup	286
Tool Behavior Setup	292
Scan Data Definition	296
Rules Checks and Violations.....	298
Good/Fault Simulation with Existing Patterns.....	299
Fault Simulation	299
Good-Machine Simulation	301
Random Pattern Simulation.....	302
Tool Setup for Random Pattern Simulation	302
Faults List Addition	302
Running Random Pattern Simulation	302
Fault Information Setup for ATPG	303
Tool Setup for ATPG	303
Faults List Creation.....	304
Faults Addition to an Existing List.....	304
How to Load Faults from an External List.....	304
How to Write Faults to an External File.....	305
Fault Sampling Percentage Setup	305
Fault Mode Setup	305
Possible-Detect Credit Setup	305
ATPG Operations	306
ATPG Setup	307
Pattern Creation with Default Settings	314
Approaches for Improving ATPG Efficiency.....	316
How to Save the Test Patterns	318
Low-Power ATPG	319

Low-Power Capture	319
Low-Power Shift	319
Setting up Low-Power ATPG	320
IDDQ Test Set Creation	322
IDDQ Test Set Generation	323
Leakage Current Checks	324
Delay Test Set Creation	325
Transition Delay Test Set Creation	326
Transition Fault Detection	327
Generating a Transition Test Set	330
Timing for Transition Delay Tests	331
Transition Fault Detection and Multiple Clocks	334
Pattern Failures Due to Timing Exception Paths	337
SDC Timing Exception Effects	343
Debugging Test Coverage with Timing Exceptions	347
Path Delay Test Set Creation	349
At-Speed Test With Named Capture Procedures	359
Mux-DFF Example	367
Support for Internal Clock Control	373
Generating Test Patterns for Different Fault Models and Fault Grading	380
Timing-Aware ATPG	383
Bridge and Open, and Cell Neighborhood Defects UDFM Creation	396
Interconnect Bridge and Open Extraction and UDFM Creation	397
Cell Neighborhood Defects UDFM Creation	419
Pattern Generation for a Boundary Scan Circuit	426
About Dofile	426
TAP Controller State Machine	427
About the Test Procedure File	428
MacroTest Overview	434
The MacroTest Process Flow	435
Macro Qualification for MacroTest	437
When to Use MacroTest	438
Macro Boundary Definition	441
Test Values Definition	445
Recommendations for MacroTest Use	447
Macro Test Examples	449
Verifying Test Patterns	456
Design Simulation with Timing	457
Potential Causes of Simulation Mismatches	465

ATPG Basic Tool Flow

The following figure shows the basic process flow for the ATPG tool.

Figure 8-1. Overview of ATPG Tool Usage



The tasks required to complete the test pattern generation shown in [Figure 8-1](#) are described as follows:

1. Invoke Tesson Shell using the “tesson -shell” command. Set the context to “patterns -scan” using the `set_context` command, which allows you to access ATPG functionality.
2. The ATPG tool requires a structural (gate-level) design netlist and a DFT library, which you accomplish with the `read_cell_library` and `read_verilog` commands, respectively. “[ATPG Tool Inputs and Outputs](#)” on page 277 describes which netlist formats you can use with the ATPG tool. Every element in the netlist must have an equivalent description in the specified DFT library. The “[Design Library](#)” section in the *Tesson Cell Library Manual* gives information on the DFT library. The tool reads in the library and the netlist, parsing and checking each.
3. After reading the library and netlist, the tool goes into setup mode. Within setup mode, you perform several tasks, using commands either interactively or through the use of a dofile. You can set up information about the design and the design’s scan circuitry. “[Circuit Behavior Setup](#)” on page 286 documents this setup procedure. Within setup mode, you can also specify information that influences simulation model creation during the design flattening phase.
4. After performing all the desired setup, you can exit setup mode, which triggers a number of operations. If this is the first attempt to exit setup mode, the tool creates a flattened design model. This model may already exist if a previous attempt to exit setup mode failed or you used the `create_flat_model` command. “[Model Flattening](#)” on page 98 provides more details about design flattening.
5. Next, the tool performs extensive learning analysis on this model. “[Learning Analysis](#)” on page 104 explains learning analysis in more detail.
6. Once the tool creates a flattened model and learns its behavior, it begins design rules checking. The “[Design Rule Checking](#)” section in the *Tesson Shell Reference Manual* gives a full discussion of the design rules.
7. Once the design passes rules checking, the tool enters analysis mode, where you can perform simulation on a pattern set for the design. For more information, refer to “[Good-Machine Simulation](#)” on page 301 and “[Fault Simulation](#)” on page 299.
8. At this point, you may want to create patterns. You can also perform some additional setup steps, such as adding the fault list. “[Fault Information Setup for ATPG](#)” on page 303 details this procedure. You can then run ATPG on the fault list. During the ATPG run, the tool also performs fault simulation to verify that the generated patterns detect the targeted faults.

In either case (full or partial scan), you can run ATPG under different constraints, or augment the test vector set with additional test patterns, to achieve higher test coverage. See “[ATPG Operations](#)” on page 306 for details on configuring ATPG runs.

After generating a test set with the ATPG tool, you should apply timing information to the patterns and verify the design and patterns before handing them off to the vendor. “[Verifying Test Patterns](#)” on page 456 documents this operation.

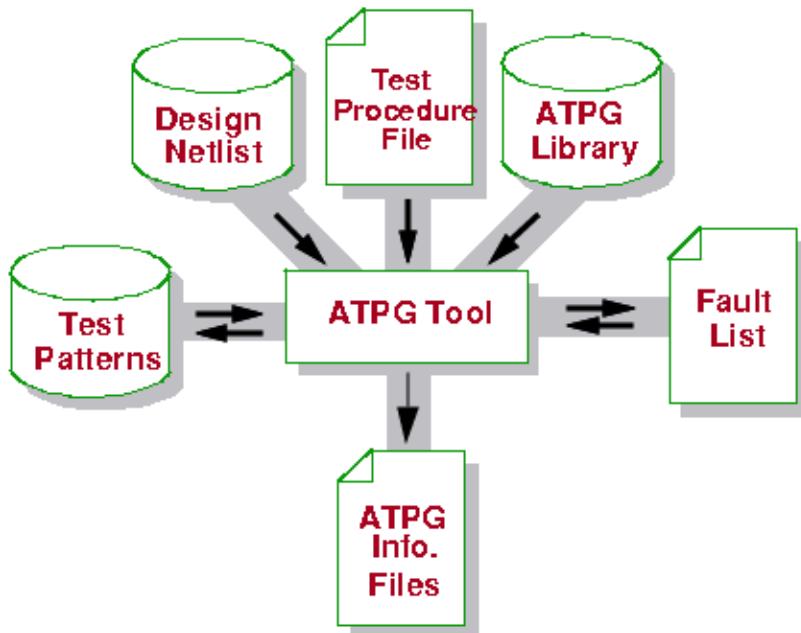
ATPG Tool Inputs and Outputs	277
ATPG Process Overview	279

ATPG Tool Inputs and Outputs

The ATPG tool uses multiple inputs to produce test patterns, a fault list, and ATPG information files.

[Figure 8-2](#) shows the inputs and outputs of the ATPG tool.

Figure 8-2. ATPG Tool Inputs and Outputs



The ATPG tool uses the inputs shown in [Table 8-1](#):

Table 8-1. ATPG Inputs

Design	The supported design data format is gate-level Verilog. Other inputs also include 1) a cell model from the design library and 2) a previously-saved, flattened model.
Test Procedure File	This file defines the operation of the scan circuitry in your design. You can generate this file by hand, or Tesson Scan can create this file automatically when you issue the command <code>write_atpg_setup</code> .

Table 8-1. ATPG Inputs (cont.)

Library	The design library contains descriptions of all the cells used in the design. The tool uses the library to translate the design data into a flat, gate-level simulation model for use by the fault simulator and test generator.
Fault List	The tool can read in an external fault list. The tool uses this list of faults and their current status as a starting point for test generation.
Test Patterns	The tool can read in an external fault list. The tool uses this list of faults and their current status as a starting point for test generation.

The ATPG tool produces the outputs described in [Table 8-2](#):

Table 8-2. ATPG Outputs

Test Patterns	The tool generates files containing test patterns. They can generate these patterns in a number of different simulator and ASIC vendor formats. “ Test Pattern Formatting and Timing ” on page 531 discusses the test pattern formats in more detail.
ATPG Information Files	These consist of a set of files containing information from the ATPG session. For example, you can specify creation of a log file for the session
Fault List	This is an ASCII-readable file that contains internal fault information in the standard Mentor Graphics fault format.

ATPG Process Overview

To understand how the ATPG tool operates, you should understand the basic ATPG process, timing model, and basic pattern types that the tool produces. The following subsections discuss these topics.

Basic ATPG Process	279
ATPG Tool Timing Model	280
ATPG Tool Pattern Types	280

Basic ATPG Process

The ATPG tool has default values set, so when you start ATPG for the first time (by issuing the create_patterns command), the tool performs an efficient combination of random pattern fault simulation and deterministic test generation on the target fault list.

[“The ATPG Process”](#) on page 40 discusses the basics of random and deterministic pattern generation.

Random Pattern Generation Using the ATPG Tool

The tool first performs random pattern fault simulation for each capture clock, stopping when a simulation pattern fails to detect at least 0.5 percent of the remaining faults. The tool then performs random pattern fault simulation for patterns without a capture clock, as well as those that measure the primary outputs connected to clock lines.

Note

 ATPG constraints and circuitry that can have bus contention are not optimal conditions for random pattern generation. If you specify ATPG constraints, the tool does not perform random pattern generation.

Deterministic Test Generation Using the ATPG Tool

Some faults have a very low chance of detection using a random pattern approach. Thus, after it completes the random pattern simulation, the tool performs deterministic test generation on selected faults from the current fault list. This process consists of creating test patterns for a set of (somewhat) randomly chosen faults from the fault list.

During this process, the tool identifies and removes redundant faults from the fault list. After it creates enough patterns for a fault simulation pass, it displays a message that indicates the number of redundant faults, the number of ATPG untestable faults, and the number of aborted faults that the test generator identifies. The tool then once again invokes the fault simulator, removing all detected faults from the fault list and placing the effective patterns in the test set. The tool then selects another set of patterns and iterates through this process until no faults

remain in the current fault list, except those aborted during test generation (that is, those in the UC or UO categories).

ATPG Tool Timing Model

The tool uses a cycle-based timing model, grouping the test pattern events into test cycles. The ATPG tool simulator uses the non-scan events: **force_pi**, **measure_po**, **capture_clock_on**, **capture_clock_off**, **ram_clock_on**, and **ram_clock_off**. The tool uses a fixed test cycle type for ATPG; that is, you cannot modify it.

The most commonly used test cycle contains the events: **force_pi**, **measure_po**, **capture_clock_on**, and **capture_clock_off**. The test vectors used to read or write into RAMs contain the events **force_pi**, **ram_clock_on**, and **ram_clock_off**. You can associate real times with each event via the timing file.

ATPG Tool Pattern Types

The ATPG tool has several different types of testing modes. That is, it can generate several different types of patterns depending on the style and circuitry of the design and the information you specify. By default, the tool generates basic scan patterns, which assume a full-scan design methodology. The following subsections describe basic scan patterns, as well as the other types of patterns that the tool can generate.

Basic Scan Patterns

As mentioned, the tool generates basic scan patterns by default. A scan pattern contains the events that force a single set of values to all scan cells and primary inputs (**force_pi**), followed by observation of the resulting responses at all primary outputs and scan cells (**measure_po**). The tool uses any defined scan clock to capture the data into the observable scan cells (**capture_clock_on**, **capture_clock_off**). Scan patterns reference the appropriate test procedures to define how to control and observe the scan cells. The tool requires that each scan pattern be independent of all other scan patterns. The basic scan pattern contains the following events:

1. Load values into scan chains.
2. Force values on all non-clock primary inputs (with clocks off and constrained pins at their constrained values).
3. Measure all primary outputs (except those connected to scan clocks).
4. Pulse a capture clock or apply selected clock procedure.
5. Unload values from scan chains.

While the list shows the loading and unloading of the scan chain as separate events, more typically the loading of a pattern occurs simultaneously with the unloading of the preceding

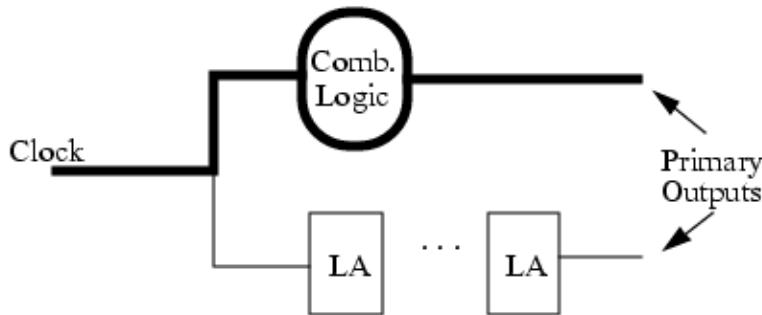
pattern. Thus, when applying the patterns at the tester, you have a single operation that loads in scan values for a new pattern while unloading the values captured into the scan chains for the previous pattern.

Because the ATPG tool is optimized for use with scan designs, the basic scan pattern contains the events from which the tool derives all other pattern types.

Clock PO Patterns

The following figure shows that in some designs, a clock signal may go to a primary output through some combinational logic.

Figure 8-3. Clock-PO Circuitry



The tool considers any pattern that measures a PO with connectivity to a clock, regardless of whether or not the clock is active, a clock PO pattern. A normal scan pattern has all clocks off during the force of the primary inputs and the measure of the primary outputs. However, in the clocked primary output situation, if the clock is off, a condition necessary to test a fault within this circuitry might not be met and the fault may go undetected. In this case, in order to detect the fault, the pattern must turn the clock on during the force and measure. This does not happen in the basic scan pattern. The tool allows this within a clock PO pattern, to observe primary outputs connected to clocks.

Clock PO patterns contain the following events:

1. Load values into the scan chains.
2. Force values on all primary inputs, (potentially) including clocks (with constrained pins at their constrained values).
3. Measure all primary outputs that are connected to scan clocks.

The tool generates clock PO patterns whenever it learns that a clock connects to a primary output and if it determines that it can only detect faults associated with the circuitry by using a clock PO pattern. If you do not want the tool to generate clock PO patterns, you can turn off the capability as follows:

SETUP> set_pattern_type -clock_po off

Clock Sequential Patterns

The ATPG tool's clock sequential pattern type handles limited sequential circuitry, and can also help in testing designs with RAM. This kind of pattern contains the following events:

1. Load the scan chains.
2. Apply the clock sequential cycle.
 - a. Force values on all primary inputs, except clocks (with constrained pins at their constrained values).
 - b. Pulse the write lines, read lines, capture clock, and/or apply selected clock procedure.
 - c. Repeat steps a and b for a total of "N" times, where N is the clock sequential depth - 1.
3. Apply the capture cycle.
 - a. Force pi.
 - b. Measure po.
 - c. Pulse capture clock.
4. Unload the scan chains as you load the next pattern.

To instruct the tool to generate clock sequential patterns, you must set the sequential depth to some number greater than one, using the [set_pattern_type](#) command as follows:

SETUP> set_pattern_type -sequential 2

A depth of zero indicates combinational circuitry. A depth greater than one indicates limited sequential circuitry. You should, however, be careful of the depth you specify. You should start off using the lowest sequential depth and analyzing the run results. You can perform several runs, if necessary, increasing the sequential depth each time. Although the maximum allowable depth limit is 255, you should typically limit the value you specify to five or less, for performance reasons.

Multiple Load Patterns

The tool can optionally include multiple scan chain loads in a clock sequential pattern. By creating patterns that use multiple loads, the tool is capable of the following:

- Take advantage of a design's non-scan sequential cells that are capable of retaining their state through a scan load operation.
- Test through a RAM/ROM.

You enable the multiple load capability by using “-multiple_load on” with the [set_pattern_type](#) command and setting the sequential depth to some number greater than one. When you activate this capability, you allow the tool to include a scan load before any pattern cycle.

Note

 An exception is at-speed sequences in named capture procedures. A load may not occur between the at-speed launch and capture cycles. For more information, see the description of the “load” cycle type in “[Internal and External Modes Definition](#)” on page 360.

Generally, multiple load patterns require a sequential depth for every functional mode clock pulsed. A minimum sequential depth of 4 is required to enable the tool to create the multiple cycle patterns necessary for RAM testing. The patterns are very similar to RAM sequential patterns, but for many designs will give better coverage than RAM sequential patterns. This method also supports certain tool features (MacroTest, dynamic compression, split-capture cycle, clock-off simulation) not supported by RAM sequential patterns.

RAM Sequential Patterns

To propagate fault effects through RAM, and to thoroughly test the circuitry associated with a RAM, the tool generates a special type of pattern called RAM sequential. RAM sequential patterns are single patterns with multiple loads, which model some sequential events necessary to test RAM operations. The multiple load events include two address writes and possibly a read (if the RAM has data hold). This type of pattern contains the following events:

1. Load scan cells.
2. Force primary inputs.
3. Pulse write line(s).
4. Repeat steps 1 through 3 for a different address.
5. Load scan cells.
6. Force primary inputs.
7. Pulse read lines (optional, depending on the RAM’s data hold attribute).
8. Load scan cells.
9. Force primary inputs
10. Measure primary outputs.
11. Pulse capture clock.
12. Unload values from scan cells.

The following example explains the operations depicted in this type of pattern. Assume you want to test a stuck-at-1 fault on the highest order bit of the address lines. You could do this by

writing some data, D, to location 1000. You could then write different data, D', to location 0000. If a stuck-at-1 fault was present on the highest address bit, the faulty machine would overwrite location 1000 with the value D'. Next, you would attempt to read from address location 1000. With the stuck-at-1 fault on the address line, you would read D'.

Conversely, if the fault on the highest order bit of the address line is a stuck-at-0 fault, you would want to write the initial data, D, to location 0000. You would then write different data, D', to location 1000. If a stuck-at-0 fault was present on the highest address bit, the faulty machine would overwrite location 0000 with the value D'. Next, you would attempt to read from address location 0000. With the stuck-at-0 fault on the address line, you would read D'.

You can instruct the tool to generate RAM sequential patterns by issuing the `set_pattern_type` command as follows:

SETUP> `set_pattern_type -ram_sequential on`

Sequential Transparent Patterns

Designs containing some non-scan latches can use basic scan patterns if the latches behave transparently between the time of the primary input force and the primary output measure. A latch behaves transparently if it passes rule D6.

For latches that do not behave transparently, a user-defined procedure can force some of them to behave transparently between the primary input force and primary output measure. A test procedure, which is called `seq_transparent`, defines the appropriate conditions necessary to force transparent behavior of some latches. The events in sequential transparent patterns include:

1. Load scan chains.
2. Force primary inputs.
3. Apply `seq_transparent` procedure(s).
4. Measure primary outputs.
5. Unload scan chains.

For more information on sequential transparent procedures, refer to “[The Procedures](#)” section in the *Tessent Shell User’s Manual*.

ATPG Procedures

This section describes some operations you may need to perform with the ATPG tool.

Tool Invocation	285
Circuit Behavior Setup	286
Tool Behavior Setup	292
Scan Data Definition	296

Tool Invocation

You access ATPG functionality by invoking Tesson Shell and then setting the context to “patterns -scan.”

```
% tessent -shell  
SETUP> set_context patterns -scan
```

When Tesson Shell invokes, the tool assumes the first thing you want to do is set up circuit behavior, so it automatically puts you in setup mode. To change the system mode to analysis, use the [set_system_mode](#) command.

Circuit Behavior Setup

The ATPG tool provides a number of commands that let you set up circuit behavior. You must execute these commands while in setup mode.

A convenient way to execute the circuit setup commands is to place these commands in a dofile, as explained previously in “[Tessent Shell Batch Jobs](#)”. The following subsections describe typical circuit behavior set up tasks.

Equivalent or Inverted Primary Input Definition	286
Primary Inputs and Outputs Addition	286
Bidirectional Pins as Primary Inputs or Outputs.....	287
How to Tie Undriven Signals	290
Primary Input Constraints	290
How to Mask Primary Outputs	290
Slow Pad Addition	291

Equivalent or Inverted Primary Input Definition

Within the circuit application environment, often multiple primary inputs of the circuit being tested must always have the same (equivalent) or opposite values. Specifying pin equivalences constrains selected primary input pins to equivalent or inverted values relative to the last entered primary input pin.

The following commands are useful when working with pin equivalences:

- [add_input_constraints](#) — Adds pin equivalences.
- [delete_input_constraints](#) — Deletes the specified pin equivalences.
- [report_input_constraints](#) — Displays the specified pin equivalences.

Primary Inputs and Outputs Addition

In some cases, you may need to change the test pattern application points (primary inputs) or the output value measurement points (primary outputs). When you add previously undefined primary inputs, they are called user class primary inputs, while the original primary inputs are called system class primary inputs.

To add primary inputs to a circuit, at the setup mode prompt, use the [add_primary_inputs](#) command. When you add previously undefined primary outputs, they are called user class primary outputs, while the original primary outputs are called system class primary outputs.

To add primary outputs to a circuit, at the setup mode prompt, use the [add_primary_outputs](#) command.

You use the following command to report and delete primary inputs and outputs:

- `delete_primary_inputs` — Deletes the specified types of primary inputs.
- `report_primary_inputs` — Reports the specified types of primary inputs.
- `delete_primary_outputs` — Deletes the specified types of primary outputs.
- `report_primary_outputs` — Reports the specified types of primary outputs.

Bidirectional Pins as Primary Inputs or Outputs

During pattern generation, the ATPG tool automatically determines the mode of bidirectional pins (bidis) and avoids creating patterns that drive values on these pins when they are not in input mode. In some situations, however, you might prefer to have the tool treat a bidirectional pin as a PI or PO. For example, some testers require more memory to store bidirectional pin data than PI or PO data. Treating each bidi as a PI or PO when generating and saving patterns will reduce the amount of memory required to store the pin data on these testers.

From the tool's perspective, a bidi consists of several gates and includes an input port and an output port. You can use the commands, `report_primary_inputs` and `report_primary_outputs`, to view PIs and POs. Pins that are listed by both commands are bidirectional pins.

Certain other PI-specific and PO-specific commands accept a bidi pinnname argument, and enable you to act on just the applicable port functionality (input or output) of the bidi. For example, you can use the `delete_primary_inputs` command with a bidirectional pin argument to remove the input port of the bidi from the design interface. From then on, the tool will treat that pin as a PO. You can use the `delete_primary_outputs` command similarly to delete the output port of a bidi from the design interface, so the tool treats that bidi as a PI.

Note

 Altering the design's interface will result in generated patterns that are different than those the tool would generate for the original interface. It also prevents verification of the saved patterns using the original netlist interface. If you want to be able to verify saved patterns by performing simulation using the original netlist interface, you must use the commands described in the following subsections instead of the `delete_primary_inputs/outputs` commands.

Bidirectional Pin as a Primary Output for ATPG Only

With the `add_input_constraints` command you can get the tool to treat a bidi as a PO during ATPG only, without altering the design interface within the tool. You do this by constraining the input part of the bidi to a constant high impedance (CZ) state. The generated patterns will then contain PO data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist.

Bidirectional Pin as a Primary Input for ATPG Only

With the [add_output_masks](#) command, you can get the tool to treat a bidi as a PI during ATPG only, without altering the design interface. This command blocks observability of the output part of the bidi. The generated patterns will then contain PI data for the bidi, and you will be able to verify saved patterns by performing simulation using the original design netlist.

If the Bidirectional Pin Control Logic is Unknown

Sometimes the control logic for a bidi is unknown. In this situation, you can model the control logic as a black box. If you want the tool to treat the bidi as a PI, model the output of the black box to be 0. If you want the bidi treated as a PO, model the output of the black box to be 1.

If the Bidirectional Pin has a Pull-up or Pull-down Resistor

Using default settings, the ATPG tool generates a known value for a bidirectional pad having pull-up or pull-down resistors. In reality, however, the pull-up or pull-down time is typically very slow and will result in simulation mismatches when a test is carried out at high speed.

To prevent such mismatches, you should use the [add_input_constraints](#) command. This command changes the tool's simulation of the I/O pad so that instead of a known value, an X is captured for all observation points that depend on the pad. The X masks the observation point, preventing simulation mismatches.

Examples of Setup for Bidirectional Pins as PIs or POs

The following examples demonstrate the use of the commands described in the preceding sections about bidirectional pins (bidis). Assume the following pins exist in an example design:

- Bidirectional pins: /my_inout[0].../my_inout[2]
- Primary inputs (PIs): /clk, /rst, /scan_in, /scan_en, /my_en
- Primary outputs (POs): /my_out[0].../my_out[4]

You can view the bidis by issuing the following two commands:

SETUP> report_primary_inputs

```
SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en
SYSTEM: /my_inout [2]
SYSTEM: /my_inout [1]
SYSTEM: /my_inout [0]
```

SETUP> report_primary_outputs

```
SYSTEM: /x_out [4]
SYSTEM: /x_out [3]
SYSTEM: /x_out [2]
SYSTEM: /x_out [1]
SYSTEM: /x_out [0]
SYSTEM: /my_inout [2]
SYSTEM: /my_inout [1]
SYSTEM: /my_inout [0]
```

Pins listed in the output of both commands (shown in bold font) are pins the tool will treat as bidis during test generation. To force the tool to treat a bidi as a PI or PO, you can remove the definition of the unwanted input or output port. The following example removes the input port definition, then reports the PIs and POs. You can see the tool now only reports the bidis as POs, which reflects how those pins will be treated during ATPG:

```
SETUP> delete_primary_inputs /my_inout[0] /my_inout[1] /my_inout[2]
SETUP> report_primary_inputs
```

```
SYSTEM: /clk
SYSTEM: /rst
SYSTEM: /scan_in
SYSTEM: /scan_en
SYSTEM: /my_en
```

```
SETUP> report_primary_outputs
```

```
SYSTEM: /x_out [4]
SYSTEM: /x_out [3]
SYSTEM: /x_out [2]
SYSTEM: /x_out [1]
SYSTEM: /x_out [0]
SYSTEM: /my_inout [2]
SYSTEM: /my_inout [1]
SYSTEM: /my_inout [0]
```

Because deleting the unwanted primary inputs and outputs alters the design's interface within the tool, it may not be acceptable in all cases. Another approach, explained earlier, is to have the tool treat a bidi as a PI or PO during ATPG only, without altering the design interface. To obtain PO treatment for a bidi, constrain the input part of the bidi to the high impedance state. The following command does this for the /my_inout[0] bidi:

```
SETUP> add_input_constraints /my_inout[0] -cz
```

To have the tool treat a bidi as a PI during ATPG only, direct the tool to mask (ignore) the output part of the bidi. The following example does this for the /my_inout[0] and /my_inout[1] pins:

```
SETUP> add_output_masks /my_inout[0] /my_inout[2]
SETUP> report_output_masks
```

```
TIEX /my_inout [0]
TIEX /my_inout [2]
```

The “TIEX” in the output of “report_output_masks” indicates the two pins are now tied to X, which blocks their observability and prevents the tool from using them during ATPG.

How to Tie Undriven Signals

Within your design, there could be several undriven nets, which are input signals not tied to fixed values. When you read a netlist, the application issues a warning message for each undriven net or floating pin in the module. The ATPG tool must “virtually” tie these pins to a fixed logic value during ATPG.

If you do not specify a value, the application uses the default value X, which you can change with the [set_tied_signals](#) command. The [set_tied_signals](#) command assigns a fixed value to every named floating net or pin, that you do not specify with the [add_tied_signals](#) command, in every module of the circuit under test.

To add tied signals, use the [add_tied_signals](#) command at the setup mode prompt.

The [delete_tied_signals](#), [report_tied_signals](#), and [set_tied_signals](#) commands are useful when working with undriven signals.

- [delete_tied_signals](#) — Deletes the current list of specified tied signals.
- [report_tied_signals](#) — Displays current list of specified tied nets and pins.
- [set_tied_signals](#) — Sets default for tying unspecified undriven signals.

Primary Input Constraints

The tool can constrain primary inputs during the ATPG process.

To add a pin constraint to a specific pin, use the [add_input_constraints](#) command. You can specify one or more primary input pin pathnames to be constrained to one of the following formats: constant 0 (C0), constant 1 (C1), high impedance (CZ), or unknown (CX).

For detailed information on the tool-specific usages of this command, refer to [add_input_constraints](#) in the *Tessent Shell Reference Manual*.

How to Mask Primary Outputs

Your design may contain certain primary output pins that have no strobe capability. Or in a similar situation, you may want to mask certain outputs from observation for design trade-off experimentation.

In these cases, you could mask these primary outputs using the [add_output_masks](#) command.

Note

 The tool places faults that can be detected only through masked outputs in the AU category—not the UO category.

Slow Pad Addition

While running tests at high speed, as might be used for path delay test patterns, it is not always safe to assume that the loopback path from internal registers, via the I/O pad back to internal registers, can stabilize within a single clock cycle. Assuming that the loopback path stabilizes within a single clock cycle may cause problems verifying ATPG patterns or may lead to yield loss during testing.

To prevent a problem caused by this loopback, use the [add_input_constraints](#) command to modify the simulated behavior of the bidirectional I/O pin, on a pin by pin basis.

For a slow pad, the simulation of the I/O pad changes so that the value propagated into the internal logic is X whenever the primary input is not driven. This causes an X to be captured for all observation points dependent on the loopback value.

When modifying the behavior of I/O pins, these commands are useful:

- [delete_input_constraints](#) — Resets the specified I/O pin back to the default simulation mode.
- [report_input_constraints](#) — Displays all I/O pins marked as slow.

Tool Behavior Setup

In addition to specifying information about the design to the ATPG tool, you can also set up how you want the ATPG tool to handle certain situations and how much effort to put into various processes.

The following commands are useful for setting up the ATPG tool:

- [set_learn_report](#) — Enables access to certain data learned during analysis.
- [set_loop_handling](#) — Specifies the method in which to break loops.
- [set_pattern_buffer](#) — Enables the use of temporary buffer files for pattern data.
- [set_possible_credit](#) — Sets credit for possibly-detected faults.

Bus Contention Checks	292
Multi-Driven Net Behavior Setup	293
Z-State Handling Setup	293
The ATPG Learning Process	294
Capture Handling Setup	294
Transient Detection Setup	295

Bus Contention Checks

If you use contention checking on tri-state driver busses and multiple-port flip-flops and latches, the tool rejects (from the internal test pattern set) patterns generated by the ATPG process that can cause bus contention.

To set contention checking, you use the [set_contention_check](#) command.

By default, contention checking is on, as are the switches -Warning and -Bus, causing the tool to check tri-state driver buses and issue a warning if bus contention occurs during simulation. For more information on the different contention checking options, refer to the [set_contention_check](#) description in the *Tessent Shell Reference Manual*.

To display the current status of contention checking, use the [report_environment](#) command.

You may use the following commands when you are checking bus contention:

- [analyze_bus](#) — Analyzes the selected buses for mutual exclusion.
- [set_bus_handling](#) — Specifies how to handle contention on buses.
- [set_driver_restriction](#) — Specifies whether only a single driver or multiple drivers can be on for buses or ports.
- [report_bus_data](#) — Reports data for either a single bus or a category of buses.

- [report_gates](#) — Reports netlist information for the specified gates.

Multi-Driven Net Behavior Setup

When you specify the fault effect of bus contention on tri-state nets with the [set_net_dominance](#) command, you are giving the tool the ability to detect some faults on the enable lines of tri-state drivers that connect to a tri-state bus.

At the setup mode prompt, you use the [set_net_dominance](#) command.

The three choices for bus contention fault effect are And, Or, and Wire (unknown behavior), Wire being the default. The Wire option means that any different binary value results in an X state. The truth tables for each type of bus contention fault effect are shown on the references pages for the [set_net_dominance](#) description in the *Tessent Shell Reference Manual*.

If you have a net with multiple non-tri-state drivers, you may want to specify this type of net's output value when its drivers have different values. Using the [set_net_resolution](#) command, you can set the net's behavior to And, Or, or Wire (unknown behavior). The default Wire option requires all inputs to be at the same state to create a known output value. Some loss of test coverage can result unless the behavior is set to And (wired-and) or Or (wired-or). To set the multi-driver net behavior, at the setup mode prompt, you use the [set_net_resolution](#) command.

Z-State Handling Setup

If your tester has the ability to distinguish the high impedance (Z) state, you should use the Z state for fault detection to improve your test coverage. If the tester can distinguish a high impedance value from a binary value, certain faults may become detectable which otherwise would at best be possibly detected (pos_det). This capability is particularly important for fault detection in the enable line circuitry of tri-state drivers.

The default for the ATPG tool is to treat a Z state as an X state. If you want to account for Z state values during simulation, you can issue the [set_z_handling](#) command.

Internal Z handling specifies how to treat the high impedance state when the tri-state network feeds internal logic gates. External handling specifies how to treat the high impedance state at the circuit primary outputs. The ability of the tester normally determines this behavior.

To set the internal or external Z handling, use the [set_z_handling](#) command at the setup mode prompt.

For internal tri-state driver nets, you can specify the treatment of high impedance as a 0 state, a 1 state, and an unknown state.

Note

-  This command is not necessary if the circuit model already reflects the existence of a pull gate on the tri-state net.
-

For example, to specify that the tester does not measure high impedance, enter the following:

SETUP> set_z_handling external X

For external tri-state nets, you can also specify that the tool measures high impedance as a 0 state and distinguished from a 1 state (0), measures high impedance as a 1 state and distinguished from a 0 state (1), measures high impedance as unique and distinguishable from both a 1 and 0 state (Z).

The ATPG Learning Process

The ATPG tool performs extensive learning on the circuit during the transition from setup to some other system mode. This learning reduces the amount of effort necessary during ATPG. The tool allows you to control this learning process.

For example, the tool lets you turn the learning process off or change the amount of effort put into the analysis. You can accomplish this for combinational logic using the [set_static_learning](#) command.

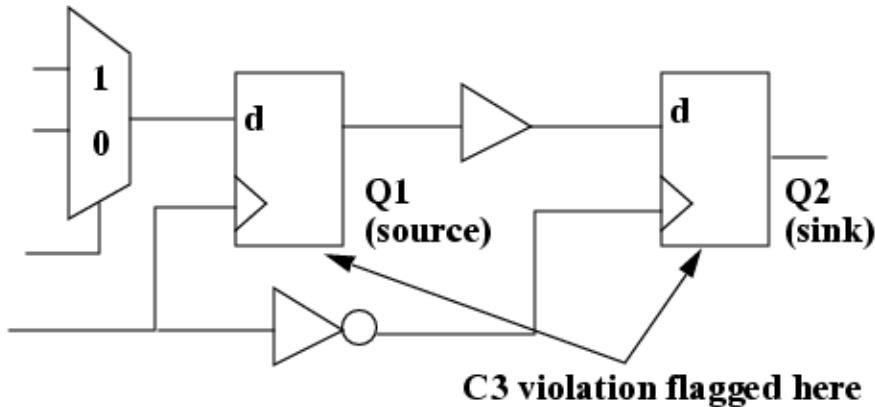
By default, static learning is on and the simulation activity limit is 1000. This number ensures a good trade-off between analysis effort and process time. If you want the ATPG tool to perform maximum circuit learning, you should set the activity limit to the number of gates in the design.

By default, state transition graph extraction is on. For more information on the learning process, refer to “[Learning Analysis](#)” on page 104.

Capture Handling Setup

The ATPG tool evaluates gates only once during simulation, simulating all combinational gates before sequential gates. This default simulation behavior correlates well with the normal behavior of a synchronous design, if the design model passes design rules checks—particularly rules C3 and C4. However, if your design fails these checks, you should examine the situation to see if your design would benefit from a different type of data capture simulation.

For example, examine the design of [Figure 8-4](#). It shows a design fragment which fails the C3 rules check.

Figure 8-4. Data Capture Handling Example

The rules checker flags the C3 rule because Q2 captures data on the trailing edge of the same clock that Q1 uses. The ATPG tool considers sequential gate Q1 as the data *source* and Q2 as the data *sink*. By default, the tool simulates Q2 capturing old data from Q1. However, this behavior most likely does not correspond to the way the circuit really operates. In this case, the C3 violation should alert you that simulation could differ from real circuit operation.

To allow greater flexibility of capture handling for these types of situations, the tool provides some commands that alter the default simulation behavior. The `set_split_capture_cycle` command, for example, effects whether or not the tool updates simulation data between clock edges. When set to “on,” the tool is able to determine correct capture values for trailing edge and level-sensitive state elements despite C3 and C4 violations. If you get these violations, issue the `set_split_capture_cycle ON` command.

Transient Detection Setup

You can set how the tool handles zero-width events on the clock lines of state elements. The tool lets you turn transient detection on or off with the `set_transient_detection` command.

With transient detection off, DRC simulation treats all events on state elements as valid. Because the simulator is a zero delay simulator, it is possible for DRC to simulate zero-width monostable circuits with ideal behavior, which is rarely matched in silicon. The tool treats the resulting zero-width output pulse from the monostable circuit as a valid clocking event for other state elements. Thus, state elements change state although their clock lines show no clocking event.

With transient detection on, the tool sets state elements to a value of X if the zero-width event causes a change of state in the state elements. This is the default behavior upon invocation of the tool.

Scan Data Definition

You must define the scan clocks and scan chains before the application performs rules checking (which occurs upon exiting setup mode). The following subsections describe how to define the various types of scan data.

Scan Clocks Definition	296
Scan Groups Definition	296
Scan Chains Definition	297
Clock Restriction Setup	297
How to Add Constraint to Scan Cells	297
Nofault Settings	298

Scan Clocks Definition

The tool considers any signals that capture data into sequential elements (such as system clocks, sets, and resets) to be scan clocks. Therefore, to take advantage of the scan circuitry, you need to define these “clock signals” by adding them to the clock list.

You must specify the *off-state* for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off-state is the clock value prior to the clock’s capturing transition. You add clock pins to the list by using the [add_clocks](#) command. The [delete_clocks](#) command deletes the specified pins from the clock list. The [report_clocks](#) command all defined clock pins.

You can constrain a clock pin to its off-state to suppress its usage as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state, otherwise an error occurs. If you add an equivalence pin to the clock list, all of its defined equivalent pins are also automatically added to the clock list.

Scan Groups Definition

A scan group contains a set of scan chains controlled by a single test procedure file. You must create this test procedure file prior to defining the scan chain group that references it.

To define scan groups, you use the [add_scan_groups](#) command. These commands are also useful:

- [delete_scan_groups](#) — Deletes specified scan groups and associated chains.
- [report_scan_groups](#) — Displays current list of scan chain groups.

Scan Chains Definition

After defining scan groups, you can define the scan chains associated with the groups. For each scan chain, you must specify the name assigned to the chain, the name of the chain's group, the scan chain input pin, and the scan chain output pin.

Note

 Scan chains of a scan group can share a common scan input pin, but this condition requires that both scan chains contain the same data after loading.

To define scan chains and their associated scan groups, you use the [add_scan_chains](#) command. These commands are also useful:

- [delete_scan_chains](#) — Deletes the specified scan chains.
- [report_scan_chains](#) — Displays current list of scan chains.

Clock Restriction Setup

You can specify whether or not to allow the test generator to create patterns that have more than one non-equivalent capture clock active at the same time.

To set the clock restriction, you use the [set_clock_restriction](#) command.

Note

 If you choose to turn off the clock restriction, you should verify the generated pattern set using a timing simulator—to ensure there are no timing errors.

How to Add Constraint to Scan Cells

The tool can constrain scan cells to a constant value (C0 or C1) during the ATPG process to enhance controllability or observability. Additionally, the tools can constrain scan cells to be either uncontrollable (CX), unobservable (OX), or both (XX).

You identify a scan cell by either a pin pathname or a scan chain name plus the cell's position in the scan chain.

To add constraints to scan cells, you use the [add_cell_constraints](#) command. To delete constraints from specific scan cells, use the [delete_cell_constraints](#) command. You use the [report_cell_constraints](#) to report all defined scan cell constraints.

If you specify the pin pathname, it must be the name of an output pin directly connected (through only buffers and inverters) to a scan memory element. In this case, the tool sets the scan memory element to a value such that the pin is at the constrained value. An error condition occurs if the pin pathname does not resolve to a scan memory element.

If you identify the scan cell by chain and position, the scan chain must be a currently-defined scan chain and the position is a valid scan cell position number. The scan cell closest to the scan-out pin is in position 0. The tool constrains the scan cell's MASTER memory element to the selected value. If there are inverters between the MASTER element and the scan cell output, they may invert the output's value.

Nofault Settings

Within your design, you may have instances that should not have internal faults included in the fault list. You can label these parts with a nofault setting.

To add a nofault setting, you use the [add_nofaults](#) command. To delete a specified nofault setting, use the [delete_nofaults](#) command. Use the [report_nofaults](#) command to report all specified nofault settings.

You can specify that the listed pin pathnames, or all the pins on the boundary and inside the named instances, are not allowed to have faults included in the fault list.

Rules Checks and Violations

If an error occurs during the rules checking process, the application remains in setup mode so you can correct the error. You can easily resolve the cause of many such errors; for instance, those that occur during parsing of the test procedure file. Other errors may be more complex and difficult to resolve, such as those associated with proper clock definitions or with shifting data through the scan chain.

The ATPG tool performs model flattening, learning analysis, and rules checking when you try to exit setup mode. Each of these processes is explained in detail in “[Common Tool Terminology and Concepts](#)” on page 87. To change from setup to one of the other system modes, you enter the [set_system_mode](#) command.

Note

 The ATPG tool does not require the DRC mode because it uses the same internal design model for all of its processes.

“[How to Troubleshoot Rules Violations](#)” in the *Tessent Shell Reference Manual* discusses some procedures for debugging rules violations. The Debug window of DFTVisualizer is especially useful for analyzing and debugging certain rules violations. The “[Attributes and DFTVisualizer](#)” section in the *Tessent User’s Manual* discusses DFTVisualizer in detail.

Good/Fault Simulation with Existing Patterns

The purpose of fault simulation is to determine the fault coverage of the current pattern source for the faults in the active fault list. The purpose of “good” simulation is to verify the simulation model. Typically, you use the good and fault simulation capabilities of the ATPG tool to grade existing hand- or ATPG-generated pattern sets.

Fault Simulation.....	299
Good-Machine Simulation	301

Fault Simulation

The following subsections discuss the procedures for setting up and running fault simulation using the ATPG tool.

Fault simulation runs in analysis mode without additional setup. You enter analysis mode using the following command:

SETUP> set_system_mode analysis

Fault Type Designation

By default, the fault type is stuck-at. If you want to simulate patterns to detect stuck-at faults, you do not need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDQ), transition, path delay, or bridge, you can issue the [set_fault_type](#) command.

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Faults List Creation

Before you can run fault simulation, you need an active fault list from which to run. You create the faults list using the [add_faults](#) command. Typically, you would create this list using all faults as follows:

> add_faults -all

“[Fault Information Setup for ATPG](#)” on page 303 provides more information on creating the fault list and specifying other fault information.

Pattern Source Designation

You can have the tools perform simulation and test generation on a selected pattern source, which you can change at any time.

To use an external pattern source, you use the [read_patterns](#) command.

Note

 You may notice a slight drop in test coverage when using an external pattern set as compared to using generated patterns. This is an artificial drop.

The ATPG tool can perform simulation with a select number of random patterns. Refer to the *Tessent Shell Reference Manual* for additional information about these application-specific [simulate_patterns](#) command options.

These commands are used with pattern simulation:

- [set_capture_clock](#) — Specifies the capture clock for random pattern simulation.
- [set_random_clocks](#) — Specifies the selection of `clock_sequential` patterns for random pattern simulation.
- [set_random_patterns](#) — Specifies the number of random patterns to be simulated.

Fault Simulation Execution

You execute the fault simulation process by using the [simulate_patterns](#) command. You can repeat this command as many times as you want for different pattern sources.

These commands are used with fault simulation:

- [simulate_patterns](#)— Executes the fault simulation process.
- [report_faults](#) — Displays faults for selected fault classes.
- [report_statistics](#) — Displays a statistics report.

Undetected Faults List Writing

Typically, after performing fault simulation on an external pattern set, you will want to save the faults list. You can then use this list as a starting point for ATPG.

To save the faults, you use the [write_faults](#) command. For more information, refer to “[How to Write Faults to an External File](#)” on page 305.

To read the faults back in for ATPG, go to analysis mode (using [set_system_mode](#)) and enter the [read_faults](#) command.

Fault Simulation Debugging

To debug your fault simulation, you can write a list of pin values that differ between the faulty and good machine using the [add_lists](#) and [set_list_file](#) commands.

The `add_lists` command specifies which pins you want reported. The `set_list_file` command specifies the name of the file in which to place simulation values for the selected pins. The default behavior is to write pin values to standard output.

Circuit and Fault Status Reset

You can reset the circuit status and status of all testable faults in the fault list to undetected. Doing so lets you re-run the fault simulation using the current fault list, which does not cause deletion of the current internal pattern set.

To reset the testable faults in the current fault list, enter the `reset_state` command.

Good-Machine Simulation

Given a test vector, you use good machine simulation to predict the logic values in the good (fault-free) circuit at all the circuit outputs. The following subsections discuss the procedures for running good simulation on existing hand- or ATPG-generated pattern sets using the ATPG tool.

Good-Machine Simulation Preparation

Good-machine simulation runs in analysis mode without additional setup. You enter analysis mode using the following command:

SETUP> set_system_mode analysis

External Pattern Source Specification

By default, simulation runs using an internal ATPG-generated pattern source. To run simulation using an external set of patterns, enter the following command:

ANALYSIS> read_patterns filename

Good Machine Simulation Debug

You can debug your good machine simulation in several ways. If you want to run the simulation and save the values of certain pins in batch mode, you can use the `add_lists` and `set_list_file` commands. The `add_lists` command specifies which pins to report. The `set_list_file` command specifies the name of the file in which you want to place simulation values for the selected pins.

If you prefer to perform interactive debugging, you can use the `simulate_patterns` and `report_gates` commands to examine internal pin values.

How to Reset Circuit Status

In analysis mode, you can reset the circuit status by using the `reset_state` command.

Random Pattern Simulation

The following subsections show the typical procedure for running random pattern simulation.

Tool Setup for Random Pattern Simulation	302
Faults List Addition.....	302
Running Random Pattern Simulation	302

Tool Setup for Random Pattern Simulation

You run random pattern simulation in the analysis system mode.

If you are not already in the analysis system mode, use the [set_system_mode](#) command as in the following example:

```
SETUP> set_system_mode analysis
```

Faults List Addition

You can generate the faults list and eliminate all untestable faults.

To generate the faults list and eliminate all untestable faults, use the [add_faults](#) and [delete_faults](#) commands together as in the following example:

```
> add_faults -all  
> delete_faults -untestable
```

In this example, the [delete_faults](#) command with the [-untestable](#) switch removes faults from the fault list that are untestable using random patterns.

Running Random Pattern Simulation

This describes how you run random pattern simulation.

Procedure

1. Enter the “[simulate_patterns -source random](#)” command.
2. After the simulation run, you display the undetected faults with the [report_faults](#) command.
3. Some of the undetected faults may be redundant. You run ATPG on the undetected faults to identify those that are redundant.

Fault Information Setup for ATPG

Prior to performing test generation, you must set up a list of all faults the application has to evaluate. The tool can either read the list in from an external source, or generate the list itself. The type of faults in the fault list vary depending on the fault model and your targeted test type.

For more information on fault modeling and the supported models, refer to “[Fault Modeling Overview](#)” on page 47.

After the application identifies all the faults, it implements a process of structural equivalence fault collapsing from the original uncollapsed fault list. From this point on, the application works on the collapsed fault list. The results, however, are reported for both the uncollapsed and collapsed fault lists. Executing any command that changes the fault list causes the tool to discard all patterns in the current internal test pattern set due to the probable introduction of inconsistencies. Also, whenever you re-enter setup mode, it deletes all faults from the current fault list. The following subsections describe how to create a fault list and define fault related information.

Tool Setup for ATPG.....	303
Faults List Creation.....	304
Faults Addition to an Existing List	304
How to Load Faults from an External List.....	304
How to Write Faults to an External File	305
Fault Sampling Percentage Setup	305
Fault Mode Setup.....	305
Possible-Detect Credit Setup.....	305

Tool Setup for ATPG

Switch from setup to the analysis mode using the `set_system_mode` command.

Assuming your circuit passes rules checking with no violations, you can exit the setup system mode and enter the analysis system mode as follows:

SETUP> `set_system_mode analysis`

By default, the fault type is stuck-at. If you want to generate patterns to detect stuck-at faults, you *do* not need to issue the `set_fault_type` command. If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay, you can issue the `set_fault_type` command.

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Faults List Creation

The application creates the internal fault list the first time you add_faults or load in external faults. Typically, you would create a fault list with all possible faults of the selected type, although you can place some restrictions on the types of faults in the list.

To create a list with all faults of the given type, issue the following command:

ANALYSIS> add_faults -all

If you do not want all possible faults in the list, you can use other options of the add_faults command to restrict the added faults. You can also specify no-faulted instances to limit placing faults in the list. You flag instances as “Nofault” while in setup mode. For more information, refer to “[Nofault Settings](#)” on page 298.

When the tool first generates the fault list, it classifies all faults as uncontrolled (UC).

- [delete_faults](#) — Deletes the specified faults from the current fault list.
- [report_faults](#) — Displays the specified types of faults.

Faults Addition to an Existing List

You can add new faults to the current fault list using the add_faults command.

To add new faults to the current fault list, enter the [add_faults](#) command. You must enter either a list of object names (pin pathnames or instance names) or use the -All switch to indicate the pins whose faults you want added to the fault list. You can use the -Stuck-at switch to indicate which stuck faults on the selected pins you want added to the list. If you do not use the -Stuck-at switch, the tool adds both stuck-at-0 and stuck-at-1 faults. The tool initially places faults added to a fault list in the undetected-uncontrolled (UC) fault class.

How to Load Faults from an External List

You can place faults from a previous run (from an external file) into the internal fault list.

To read faults from an external file into the current fault list, enter the [read_faults](#) command.

The -Retain option causes the application to retain the fault class (second column of information) from the external fault list. The -Delete option deletes all faults in the specified file from the internal faults list. The -DELETE_Equivalent option, in the ATPG tool, deletes from the internal fault list all faults listed in the file, as well as all their equivalent faults.

Note

- In the ATPG tool, the **filename** specified cannot have fault information lines with comments appended to the end of the lines or fault information lines greater than five columns. The tool will not recognize the line properly and will not add the fault on that line to the fault list.
-

How to Write Faults to an External File

You can write all or only selected faults from a current fault list into an external file. You can then edit or load this file to create a new fault list.

To write _faults to a file, enter the [write_faults](#) command. You must specify the name of the file you want to write.

Fault Sampling Percentage Setup

By reducing the fault sampling percentage (which by default is 100 percent), you can decrease the process time to evaluate a large circuit by telling the application to process only a fraction of the total collapsed faults.

To set the fault sampling percentage, use the [set_fault_sampling](#) command. You must specify a percentage (between 1 and 100) of the total faults you want processed.

Fault Mode Setup

You can specify use of either the collapsed or uncollapsed fault list for fault counts, test coverages, and fault reports. The default is to use uncollapsed faults.

To set the fault mode, you use the [set_fault_mode](#) command.

Possible-Detect Credit Setup

Before reporting test coverage, fault coverage, and ATPG effectiveness, you should specify the credit you want given to possible-detected faults.

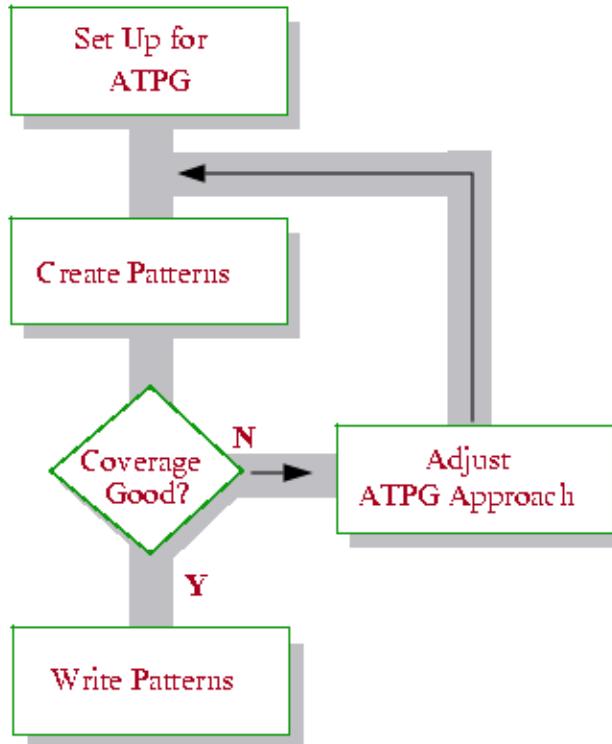
To set the credit to be given to possible-detected faults, use the [set_possible_credit](#) command. The selected credit may be any positive integer less than or equal to 100, the default being 50 percent.

ATPG Operations

Obtaining the optimal test set in the least amount of time is a desirable goal.

Figure 8-5 outlines how to most effectively meet this goal.

Figure 8-5. Efficient ATPG Flow



The first step in the process is to perform any special setup you may want for ATPG. This includes such things as setting limits on the pattern creation process itself. The second step is to create patterns with default settings (see “[Pattern Creation with Default Settings](#)” on page 314). This is a very fast way to determine how close you are to your testability goals. You may even obtain the test coverage you desire from your very first run. However, if your test coverage is not at the required level, you may have to troubleshoot the reasons for the inadequate coverage and create additional patterns using other approaches (see “[Approaches for Improving ATPG Efficiency](#)” on page 316).

ATPG Setup	307
Pattern Creation with Default Settings	314
Approaches for Improving ATPG Efficiency	316
How to Save the Test Patterns	318

ATPG Setup

Prior to ATPG, you may need to set certain criteria that aid the test generators in the test generation process.

If you just want to generate patterns quickly in the ATPG tool using default settings, the recommended method for pattern creation is using the [create_patterns](#) command.

If the initial patterns are unsatisfactory, then run the [create_patterns](#) command a second time. If, however, you are still unable to create a satisfactory pattern set, then use the [set_pattern_type](#) command in conjunction with the [create_patterns](#) command using the following sequence:

ANALYSIS> set_pattern_type -sequential 2

ANALYSIS> create_patterns

A reasonable practice is creating patterns using these two commands with the sequential depth set to 2. This is described in more detail in “[Pattern Creation with Default Settings](#)” on page 314.

ATPG Constraints Definition	307
Power and Ground Ports Exclusion from the Pattern Set.....	309
ATPG Limits Setup	310
Event Simulation for DFFs and Latches	310
Flattened Model Saves Time and Memory	312
Pattern Buffer Area Creation.....	312
Fault Sampling to Save Processing Time.....	313
Checkpointing Setup	313

ATPG Constraints Definition

ATPG constraints are similar to pin constraints and scan cell constraints. Pin constraints and scan cell constraints restrict the values of pins and scan cells, respectively. ATPG constraints place restrictions on the acceptable kinds of values at any location in the circuit. For example, you can use ATPG constraints to prevent bus contention or other undesirable events within a design. Additionally, your design may have certain conditions that can never occur under normal system operation. If you want to place these same constraints on the circuit during ATPG, use ATPG constraints.

During deterministic pattern generation, only the restricted values on the constrained circuitry are allowed. Unlike pin and scan cell constraints, which are only available in setup mode, you can define ATPG constraints in any system mode after design flattening. If you want to set ATPG constraints prior to performing design rules checking, you must first create a flattened model of the design using the [create_flat_model](#) command.

ATPG constraints are useful when you know something about the way the circuit behaves that you want the ATPG process to examine. For example, the design may have a portion of circuitry that behaves like a bus system; that is, only one of various inputs may be on, or selected, at a time. Using ATPG constraints, combined with a defined ATPG function, you can specify this information to the ATPG tool. ATPG functions place artificial Boolean relationships on circuitry within your design. After defining the functionality of a portion of circuitry with an ATPG function, you can then constrain the value of the function as desired with an ATPG constraint. This is more useful than just constraining a point in a design to a specific value.

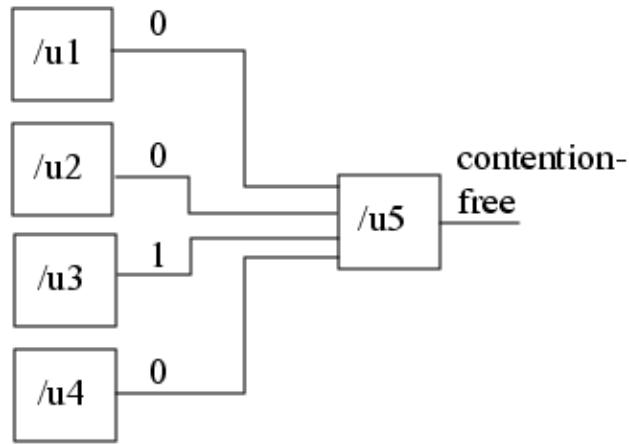
To define ATPG functions, use the [add_atpg_functions](#) command. When using this command, you specify a name, a function type, and an object to which the function applies.

You can specify ATPG constraints with the [add_atpg_constraints](#) command. When using this command, you specify a value, an object, a location, and a type.

Test generation considers all current constraints. However, design rules checking considers only static constraints. You can only add or delete static constraints in setup mode. Design rules checking generally does not consider dynamic constraints, but there are some exceptions detailed in the [set_drc_handling](#) command reference description (see the Atpg_analysis and ATPGC options). You can add or delete dynamic constraints at any time during the session. By default, ATPG constraints are dynamic.

[Figure 8-6](#) and the following commands give an example of how you use ATPG constraints and functions together.

Figure 8-6. Circuitry with Natural “Select” Functionality



The circuitry of [Figure 8-6](#) includes four gates whose outputs are the inputs of a fifth gate. Assume you know that only one of the four inputs to gate /u5 can be on at a time, such as would be true of four tri-state enables to a bus gate whose output must be contention-free. You can specify this using the following commands:

```
ANALYSIS> add_atpg_functions sel_func1 select1 /u1/o /u2/o /u3/o /u4/o
```

ANALYSIS> add_atpg_constraints 1 sel_func1

These commands specify that the “select1” function applies to gates /u1, /u2, /u3, and /u4 and the output of the select1 function should always be a 1. Deterministic pattern generation must ensure these conditions are met. The conditions causing this constraint to be true are shown in [Table 8-3](#). When this constraint is true, gate /u5 will be contention-free.

Table 8-3. ATPG Constraint Conditions

/u1	/u2	/u3	/u4	sel_func1	/u5
0	0	0	1	1	contention-free
0	0	1	0	1	contention-free
0	1	0	0	1	contention-free
1	0	0	0	1	contention-free

Given the defined function and ATPG constraint you placed on the circuitry, the ATPG tool only generates patterns using the values shown in [Table 8-3](#).

Typically, if you have defined ATPG constraints, the tools do not perform random pattern generation during ATPG. However, using the ATPG tool you can perform random pattern simulation using the [simulate_patterns](#) command. In this situation, the tool rejects patterns during fault simulation that do not meet the currently-defined ATPG constraints.

These commands are used when you are dealing with ATPG constraints:

- [analyze_atpg_constraints](#) — Analyzes a given constraint for either its ability to be satisfied or for mutual exclusivity.
- [analyze_restrictions](#) — Performs an analysis to automatically determine the source of the problems from a failed ATPG run.
- [delete_atpg_constraints](#) — Removes the specified constraint from the list.
- [delete_atpg_functions](#) — Removes the specified function definition from the list.
- [report_atpg_constraints](#) — Reports all ATPG constraints in the list.
- [report_atpg_functions](#) — Reports all defined ATPG functions.

Power and Ground Ports Exclusion from the Pattern Set

You can constrain power and ground ports during circuit setup and DRC, and also exclude those ports from the pattern set.

The first step is to specify the power and ground ports by setting the value of the “function” attribute on a top-level input or inout port. For example, the following command designates the “vcc” port as a power input:

SETUP> set_attribute_value vcc -name function -value power

Note that the port can only be an input or inout port on a top-level module, and the only allowable values are “power” and “ground.” Also, the port cannot be an IJTAG port in the ICL file, if present. Also, you cannot change the “function” attribute after reading in the flat model.

Setting the value to “power” has the effect of adding an input constraint of CT1 on that port; and setting the value to “ground” has the effect of adding an input constraint of CT0 on that port. The only way to remove these inferred constraints is by using the `reset_attribute_value` command. That is, the `delete_input_constraints` and `delete_pin_constraints` commands do not work in this situation.

After specifying the power and ground ports, you can write patterns that exclude those ports using the “`write_patterns -parameter_list`” command. Also, the parameter file has a keyword `ALL_EXCLUDE_POWER_GROUND` that allows you to control whether power and ground ports are excluded from tester pattern file formats, such as STIL and WGL. For more information, refer to the `ALL_EXCLUDE_POWER_GROUND` keyword description in the [“Parameter File Format and Keywords”](#) section of the *Tessent Shell Reference Manual*.

ATPG Limits Setup

Normally, there is no need to limit the ATPG process when creating patterns. There may be an occasional special case, however, when you want the tool to terminate the ATPG process if CPU time, test coverage, or pattern (cycle) count limits are met.

To set these limits, use the `set_atpg_limits` command.

Event Simulation for DFFs and Latches

The following explains how the tool’s simulation kernel models DFFs and latches. The kernel simulates clock pulses as “010” or “101.” The three-digit notation refers to the state of the three frames that comprise a clock cycle: `clock_off` frame, post-LE frame, and post-TE frame. There are always at least three events per cycle.

The kernel distinguishes only between “edge-triggered” and “level-sensitive” sequential elements:

- Edge-triggered elements update during 0→1 transitions of their clock input.
- Level-sensitive elements update when their clock is high.

The kernel updates all elements immediately in the same frame. For more information, refer to [Figure 8-7](#).

Figure 8-7. Simulation Frames

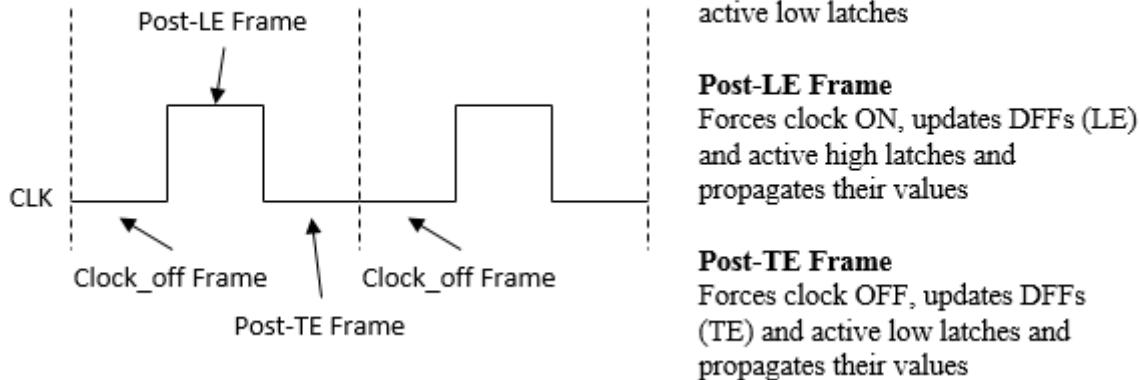
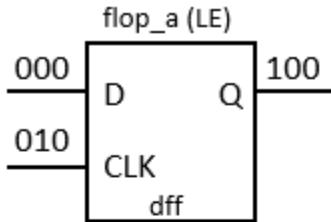
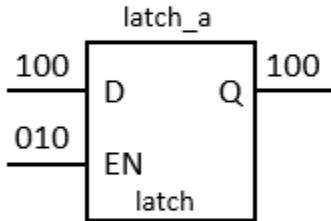


Figure 8-8 explains how the simulation kernel models waveforms with DFFs and latches.

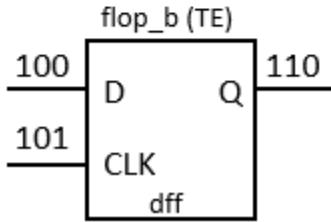
Figure 8-8. Waveform Modeling for DFFs and Latches



- 1) In the first frame the CLK input forces the DFF to output its initial value of 1 on the Q pin.
- 2) In the second frame the 0 to 1 edge of CLK causes the DFF to store its D input value and output Q=0.
- 3) In the third frame the CLK goes to OFF and the DFF holds its state and continues to output Q=0.



- 1) In the first frame the EN input forces the latch to output its initial value of 1 on the Q pin.
- 2) In the second frame the EN input is 1 which causes the level-sensitive latch to store its D input value and output Q=0.
- 3) In the third frame the EN goes to 0 and the level-sensitive latch holds its state and continues to output Q=0.



- 1) In the first frame the CLK input forces the DFF to output its initial value of 1 on the Q pin.
- 2) In the second frame the CLK goes to OFF and the DFF hold its state and continues to output Q=1.
- 3) In the third frame the 0 to 1 edge of CLK causes the DFF to store its D input value and output Q=0.

Event Simulation Data Display for a Gate

You can display event simulation data for a gate by using the [set_gate_report](#) and [report_gates](#) commands. For examples of how to do this refer to “[Example 5](#)” in the [set_gate_report](#) command description in the *Tessent Shell Reference Manual*.

Flattened Model Saves Time and Memory

You can flatten your design and subsequently use the flattened model instead of the design netlist with the tool. An advantage of using a flattened netlist rather a regular netlist, is that you save memory and have room for more patterns.

Note

 Before you save a flattened version of your design, ensure you have specified all necessary settings accurately. Some design information, such as that related to hierarchy, is lost when you flatten the design. Consequently, commands that require this information do not operate with the flattened netlist. Some settings, once incorporated in the flattened netlist, cannot be changed (for example, a tied constraint you apply to a primary input pin).

When you reinvoke the tool, you use this flattened netlist by specifying the “-flat” switch. The tool reinvokes in the same mode (setup or analysis) the tool was in when you saved the flattened model.

You flatten your design with the tool using one of the following methods depending on the tool’s current *mode*:

- **analysis mode** — The tool automatically creates the flat model when you change from setup to analysis mode using the [set_system_mode](#) command. After model flattening, you save the flattened design using the [write_flat_model](#) command.
- **setup mode** — You can manually create a flattened model in setup mode using the [create_flat_model](#) command. After model flattening, you save the flattened design using the [write_flat_model](#) command.

You can read a flat model into the tool in setup mode using the [read_flat_model](#) command.

Pattern Buffer Area Creation

To reduce demands on virtual memory when you are running the tool with large designs, use the [set_pattern_buffer](#) command with the ATPG tool. The tool then stores runtime pattern data in temporary files rather than in virtual memory.

Fault Sampling to Save Processing Time

Another command, `set_fault_sampling`, enables you to perform quick evaluation runs of large designs prior to final pattern generation. Intended for trial runs only, you can use this command to reduce the processing time when you want a quick estimate of the coverage to expect with your design.

Checkpointing Setup

The term “checkpointing” refers to when the tool automatically saves test patterns at regular periods, called checkpoints, throughout the pattern creation process. This is useful when ATPG takes a long time and there is a possibility it could be interrupted accidentally. For example, if a system failure occurs during ATPG, checkpointing enables you to recover and continue the run from close to the interruption point. You do not have to redo the entire pattern creation process from the beginning. The continuation run uses the data saved at the checkpoint, just prior to the interruption, saving you the time required to recreate the patterns that would otherwise have been lost.

The `set_checkpointing_options` command turns the checkpoint functionality on or off and specifies the time period between each write of the test patterns, as well as the name of the pattern file to which the tool writes the patterns.

Example Checkpointing

Suppose a large design takes several days for the ATPG tool to process. You do not want to restart pattern creation from the beginning if a system failure ends ATPG one day after it begins. The following dofile segment defines a checkpoint interval of 90 minutes and enables checkpointing.

```
set_checkpointing_options on -pattern_file my_checkpoint_file -period 90 \
-replace -pattern_format ascii -faultlist_file my_checkpoint_fault_file
```

If you need to perform a continuation run, invoking on a flattened model can be much faster than reflattening the netlist (see “[Flattened Model Saves Time and Memory](#)” on page 312 for more information). After the tool loads the design, but before you continue the interrupted run,

be sure to set all the same constraints you used in the interrupted run. The next dofile segment uses checkpoint data to resume the interrupted run:

```
# Load the fault population stored by the checkpoint.  
#  
# The ATPG process can spend a great deal of time proving  
# faults to be redundant (RE) or ATPG untestable (AU). By  
# loading the fault population using the -retain option, the  
# status of these fault sites will be restored. This will  
# save the time required to reevaluate these fault sites.  
read_faults my_checkpoint_fault_file -retain  
#  
# The report_statistics command shows if the fault coverage  
# is at the same level as at the last checkpoint the tool  
# encountered.  
report_statistics  
#  
# Set the pattern source to the pattern set that was stored  
# by the checkpoint. Then fault simulate these patterns.  
# During the fault simulation, the external patterns will be  
# copied into the tool's internal pattern set. Then, by  
# setting the pattern source back to the internal pattern  
# set, additional patterns can be added during a subsequent  
# ATPG run. This sequence is accomplished with the following  
# segment of the dofile.  
#  
# Fault grade the checkpoint pattern set.  
read_patterns my_checkpoint_file  
#  
# Reset the fault status to assure that the patterns  
# simulated do detect faults. When the pattern set is fault  
# simulated, if no faults are detected, the tool will not  
# retain the patterns in the internal pattern set.  
reset_state  
simulate_patterns  
report_statistics  
#  
# Create additional ATPG patterns  
create_patterns
```

After it executes the above commands, the tool should be at the same fault grade and number of patterns as when it last saved checkpoint data during the interrupted run. To complete the pattern creation process, you can now use the [create_patterns](#) command as described in “[Pattern Creation with Default Settings](#)”.

Pattern Creation with Default Settings

By default, the `create_patterns` command initiates an optimal ATPG process that includes highly efficient pattern compression.

This is the basic command for creating patterns:

ANALYSIS> create_patterns

Review the transcript for any command or setting changes the tool suggests and implement those that will help you achieve your test goals. Refer to the [create_patterns](#) command description in the *Tessent Shell Reference Manual* for more information.

If the first pattern creation run gives inadequate coverage, refer to “[Approaches for Improving ATPG Efficiency](#)” on page 316. To analyze the results if pattern creation fails, use the [analyze_atpg_constraints](#) command and the [analyze_restrictions](#) command.

Approaches for Improving ATPG Efficiency

If you are not satisfied with the test coverage after initially creating patterns, or if the resulting pattern set is unacceptably large, you can make adjustments to several system defaults to improve results in another ATPG run. The following subsections provide helpful information and strategies for obtaining better results during pattern creation.

Reasons for Low Test Coverage	316
Analysis of a Specific Fault	317
Aborted Faults Report.	317
Abort Limit Setup	317
Random Pattern Usage Setup.....	318
Decision Reordering for Pattern Creation	318

Reasons for Low Test Coverage

There are two basic reasons for low test coverage: constraints on the tool and abort conditions.

A high number of faults in the **ATPG_untestable (AU)** or PU fault categories indicates the problem lies with tool constraints. PU faults are a type of possible-detected, or **Posdet (PD)**, fault. A high number of UC and UO faults, which are both **Undetected (UD)** faults, indicates the problem lies with abort conditions. If you are unfamiliar with these fault categories, refer to “[Fault Classes](#)” on page 72.

When trying to establish the cause of low test coverage, you should examine the messages the tool prints during the deterministic test generation phase. These messages can alert you to what might be wrong with respect to **Redundant (RE)** faults, **ATPG_untestable (AU)** faults, and aborts. If you do not like the progress of the run, you can terminate the process with CTRL-C.

If a high number of aborted faults (UC or UO) appears to cause the problem, you can set the abort limit to a higher number, or modify some command defaults to change the way the application makes decisions. The number of aborted faults is high if reclassifying them as **Detected (DT)** or **Posdet (PD)** would result in a meaningful improvement in test coverage. In the tool’s coverage calculation (see “[Testability Calculations](#)” on page 84), these reclassified faults would increase the numerator of the formula. You can quickly estimate how much improvement would be possible using the formula and the fault statistics from your ATPG run. The following subsections discuss several ways to handle aborted faults.

Note

 Changing the abort limit is not always a viable solution for a low coverage problem. The tool cannot detect **ATPG_untestable (AU)** faults, the most common cause of low test coverage, even with an increased abort limit. Sometimes you may need to analyze why a fault, or set of faults, remains undetected to understand what you can do.

Also, if you have defined several ATPG constraints or have specified set_contention_check On -Atpg, the tool may not abort because of the fault, but because it cannot satisfy the required conditions. In either of these cases, you should analyze the buses or ATPG constraints to ensure the tool *can* satisfy the specified requirements.

Analysis of a Specific Fault

You can report on all faults in a specific fault category with the [report_faults](#) command.

You can analyze each fault individually, using the pin pathnames and types listed by [report_faults](#), with the [analyze_fault](#) command.

The [analyze_fault](#) command runs ATPG on the specified fault, displaying information about the processing and the end results. The application displays different data depending on the circumstances. You can optionally display relevant circuitry in DFTVisualizer using the -Display option. See the [analyze_fault](#) description in the *Tessent Shell Reference Manual* for more information.

Aborted Faults Report

During the ATPG process, the tool may terminate attempts to detect certain faults given the ATPG effort required. The tools place these types of faults, called *aborted faults*, in the AU fault class, which includes the UC and UO sub-classes.

You can determine why these faults are undetected by using the [report_aborted_faults](#) command.

Abort Limit Setup

If the fault list contains a number of aborted faults, the tools may be able to detect these faults if you change the abort limit. You can increase the abort limit for the number of backtracks, test cycles, or CPU time and recreate patterns.

To set the abort limit using the ATPG tool, use the [set_abort_limit](#) command.

The default for combinational ATPG is 30. The clock sequential abort limit defaults to the limit set for combinational. Both the [report_environment](#) command and a message at the start of deterministic test generation indicate the combinational and sequential abort limits. If they differ, the sequential limit follows the combinational abort limit.

The application classifies any faults that remain undetected after reaching the limits as *aborted faults*—which it considers undetected faults.

The [report_aborted_faults](#) command displays and identifies the cause of aborted faults.

Random Pattern Usage Setup

The ATPG tool lets you specify whether to use random test generation processes when creating uncompressed patterns. In general, if you use random patterns, the test generation process runs faster and the number of test patterns in the set is larger. If not specified, the default is to use random patterns in addition to deterministic patterns. If you use random patterns exclusively, test coverage is typically very low.

To set random pattern usage for ATPG, use the [set_random_atpg](#) command.

Note

 The [create_patterns](#) command does not use random patterns when generating compressed patterns.

Decision Reordering for Pattern Creation

Prior to ATPG, the tool learns which inputs of multiple input gates it can most easily control. It then orders these inputs from easiest to most difficult to control. Likewise, the tool learns which outputs can most easily observe a fault and orders these in a similar manner. Then during ATPG, the tool uses this information to generate patterns in the simplest way possible.

Decision reordering for pattern creation facilitates the ATPG process; however, it also minimizes random pattern detection. This is not always desirable, as you typically want generated patterns to randomly detect as many faults as possible. To maximize random pattern detection, the tool provides the [set_decision_order](#) command to allow flexible selection of control inputs and observe outputs during pattern generation.

How to Save the Test Patterns

To save generated test patterns, enter the `write_patterns` command.

For more information about the test data formats, refer to “[Saving Timing Patterns](#)” on page 544.

Low-Power ATPG

Low-power ATPG allows you to create test patterns that minimize the amount of switching activity during test to reduce power consumption. Excessive power consumption can overwhelm the circuit under test, causing it to malfunction.

Low-power ATPG controls switching activity during scan load/unload and capture:

- [Low-Power Capture](#)
- [Low-Power Shift](#)

Note

 Low-power constraints are directly related to the number of test patterns generated in an application. For example, using stricter low-power constraints results in more test patterns.

Low-Power Capture	319
Low-Power Shift	319
Setting up Low-Power ATPG.....	320

Low-Power Capture

Low-power capture employs the clock gaters in a design to achieve the power target. Clock gaters controlling untargeted portions of the design are turned off, while clock gaters controlling targeted portions are turned on.

Power is controlled most effectively in designs that employ clock gaters, and especially multiple levels of clock gaters (hierarchy), to control a majority of the state elements.

This low-power feature is available using the *Capture* option of the [set_power_control](#) command.

Low-Power Shift

Low-power shift minimizes the switching activity during shift with a constant fill algorithm where random values in scan chains are replaced with constant values as they are shifted through the core; a repeat fill heuristic is used to generate the constant values.

This low-power feature is available in the ATPG tool using the [set_power_control](#) command during test pattern generation.

Setting up Low-Power ATPG

You can enable low-power for capture and shift before test patterns are generated. Setting up low-power ATPG is an iterative process that includes the setup, generation, and analysis of test patterns.

Prerequisites

- Gate-level netlist with scan chains inserted.
- DFT strategy for your design. A test strategy helps define the most effective testing process for your design.

Procedure

1. Invoke Tessent Shell:

```
% tessent -shell my_gate_scan.v -library my_lib.atpg \
    -logfile log/atpg_cg.log -replace
```

2. Set the context to “patterns -scan”:

```
SETUP> set_context patterns -scan
```

3. Set up test patterns.

4. Exit setup mode and run DRC:

```
SETUP> set_system_mode analysis
```

5. Correct any DRC violations.

6. Turn on low-power capture. For example:

```
ANALYSIS> set_power_control capture -switching_threshold_percentage 30 \
    -rejection_threshold_percentage 35
```

Switching during the capture cycle is minimized to 30 percent and any test patterns that exceed a 35 percent rejection threshold are discarded.

7. Turn on low-power shift. For example

```
ANALYSIS> set_power_control shift on -switching_threshold_percentage 20 \
    -rejection_threshold_percentage 25
```

Switching during scan chain loading is minimized to 20 percent and any test patterns that exceed a 25 percent rejection threshold are discarded.

8. Create test patterns:

```
ANALYSIS> create_patterns
```

Test patterns are generated and the test pattern statistics and power metrics display.

9. Analyze reports, adjust power and test pattern settings until power and test coverage goals are met. You can use the [report_power_metrics](#) command to report the capture power usage associated with specific instances or modules in the design.

10. Save test patterns. For example:

```
ANALYSIS> write_patterns ..../generated/patterns_edt_p.stil -stil -replace
```

Related Topics

[Low-Power Capture](#)

[Low-Power Shift](#)

IDQ Test Set Creation

The ATPG tool supports the pseudo stuck-at fault model for IDQ testing.

This fault model allows detection of most of the common defects in CMOS circuits (such as resistive shorts) without costly transistor level modeling. “[IDQ Test](#)” on page 44 introduces IDQ testing.

The tool creates an IDQ patterns based on the pseudo stuck-at fault model. You must first set the fault type to IDQ with the [set_fault_type](#) command.

During IDQ test generation, the tool classifies faults at the inputs of sequential devices such as scan cells or non-scan cells as **Blocked (BL)** faults. This is because the diversity of flip-flop and latch implementations means the pseudo stuck-at fault model cannot reliably guide ATPG to create a good IDQ test. In contrast, a simple combinational logic gate has one common, fully complementary implementation (a NAND gate, for example, has two parallel pFETs between its output and Vdd and two series nFETs between its output and Vss), so the tool can more reliably declare pseudo stuck-at faults as detected. The switch level implementation of a flip-flop varies so greatly that assuming a particular implementation is highly suspect. The tool therefore takes a pessimistic view and reports coverage lower than it actually is, because it is unlikely such defects will go undetected for all IDQ patterns.

Using the ATPG tool, you can generate IDQ patterns using several user-specified checks. These checks can help ensure that the IDQ test vectors do not increase IDQ in the good circuit. These sections provide more details for IDQ test generation and user-specified checks:

IDQ Test Set Generation	323
Leakage Current Checks.....	324

IDDQ Test Set Generation

These topics discuss the basic IDDQ pattern generation process and provide an example of a typical IDDQ pattern generation run.

When to Perform the Measures	323
Pattern Generation	323
Running ATPG for IDDQ.....	323

When to Perform the Measures

IDDQ test patterns must contain statements that tell the tester to make an IDDQ measure.

In the Text format, this IDDQ measure statement, or label, appears as follows:

```
measure IDDQ ALL <time>;
```

By default, the ATPG tool places these statements at the end of patterns (cycles) that can contain IDDQ measurements. You can manually add these statements to patterns (cycles) within the external pattern set.

Pattern Generation

Prior to pattern generation, you may want to set up restrictions that the tool must abide by when creating the best IDDQ patterns.

For more information on IDDQ restrictions, see “[Leakage Current Checks](#)” on page 324. As with any other fault type, you issue the [create_patterns](#) command within analysis mode. This generates an internal pattern set targeting the IDDQ faults in the current list.

Running ATPG for IDDQ

This procedure shows how to run ATPG for the IDDQ fault type.

Procedure

1. Invoke Tesson Shell, set the context to “patterns -scan,” read in the netlist, set up the appropriate parameters for the ATPG run, pass rules checking, and then enter analysis mode.

```
SETUP> set_system_mode analysis
```

2. Set the fault type to IDDQ.

```
ANALYSIS> set_fault_type iddq
```

3. Specify the number of desirable IDDQ patterns.

```
ANALYSIS> set_atpg_limits -pattern_count 128
```

4. Run ATPG, generating patterns that target the IDDQ faults in the current fault list.

Note that you could use the [set_iddq_checks](#) command prior to the ATPG run to place restrictions on the generated patterns. You can also optionally issue the [read_faults](#) command to add the target fault list at this point.

ANALYSIS> create_patterns -patterns_per_pass 1

The tool then adds all faults automatically.

Note that the generated internal pattern source already contains the appropriate IDDQ measure statements.

5. Save these IDDQ patterns into a file.

ANALYSIS> write_patterns iddq.pats

Leakage Current Checks

For CMOS circuits with pull-up or pull-down resistors or tri-state buffers, the good circuit should have a nearly zero IDDQ current. The ATPG tool allows you to specify various IDDQ measurement checks to ensure that the good circuit does not raise IDDQ current during the measurement.

Use the [set_iddq_checks](#) command to specify these options.

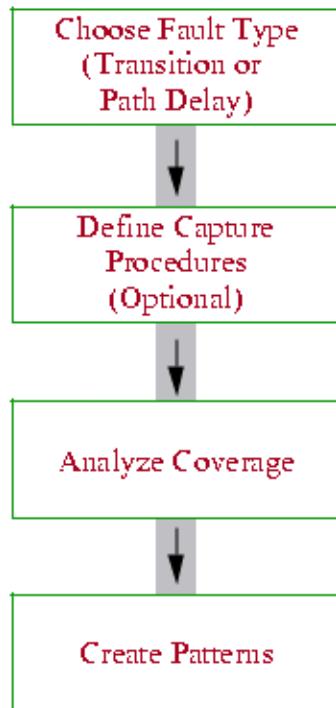
By default, the tool does not perform IDDQ checks. Both ATPG and fault simulation processes consider the checks you specify.

Delay Test Set Creation

Delay, or “at-speed” tests in the ATPG tool are of two types: transition delay and path delay.

Figure 8-9 shows a general flow for creating a delay pattern set.

Figure 8-9. Flow for Creating a Delay Test Set



Your process may be different and it may involve multiple iterations through some of the steps, based on your design and coverage goals. This section describes these two test types in more detail and how you create them using the ATPG tool.

Transition Delay Test Set Creation	326
Transition Fault Detection	327
Generating a Transition Test Set	330
Timing for Transition Delay Tests	331
Transition Fault Detection and Multiple Clocks	334
Pattern Failures Due to Timing Exception Paths	337
SDC Timing Exception Effects	343
Debugging Test Coverage with Timing Exceptions	347
Path Delay Test Set Creation	349
At-Speed Test With Named Capture Procedures	359
Mux-DFF Example	367

Support for Internal Clock Control.....	373
Generating Test Patterns for Different Fault Models and Fault Grading.....	380
Timing-Aware ATPG	383

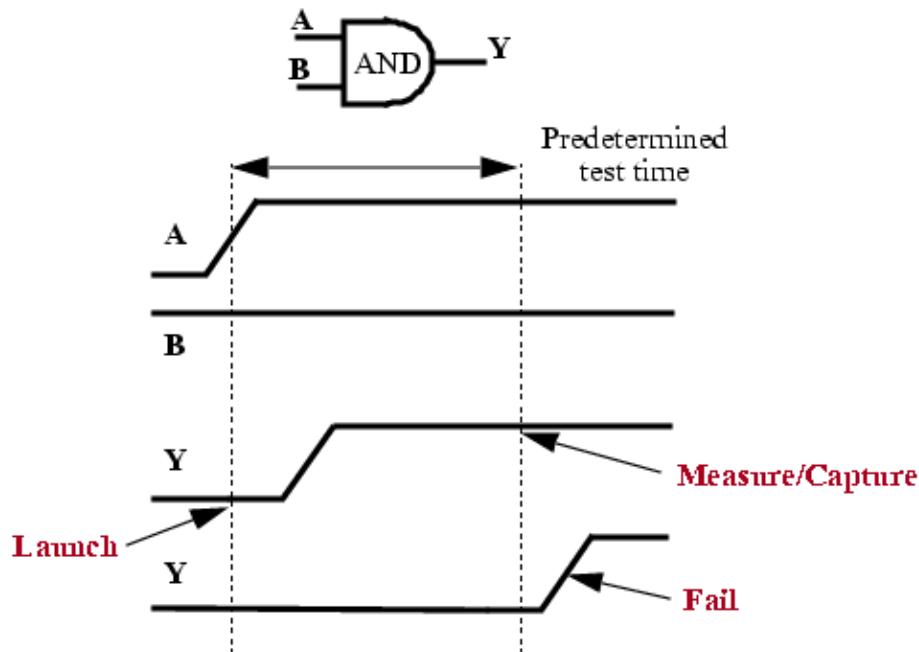
Transition Delay Test Set Creation

The tool can generate patterns to detect transition faults.

“At-Speed Testing and the Transition Fault Model” on page 52 introduced the transition fault model. Transition faults model gross delays on gate terminals (or nodes), allowing each terminal to be tested for slow-to-rise or slow-to-fall behavior. The defects these represent may include things like partially conducting transistors or interconnections.

Figure 8-10 illustrates a simple AND gate which has six potential transition faults. These faults are comprised of slow-to-rise and slow-to-fall transitions for each of the three terminals. Because a transition delay test checks the speed at which a device can operate, it requires a two cycle test. First, all the conditions for the test are set. In the figure, A and B are 0 and 1 respectively. Then a change is launched on A, which should cause a change on Y within a pre-determined time. At the end of the test time, a circuit response is captured and the value on Y is measured. Y might not be stuck at 0, but if the value of Y is still 0 when the measurement is taken at the capture point, the device is considered faulty. The ATPG tool automatically chooses the launch and capture scan cells.

Figure 8-10. Transition Delay

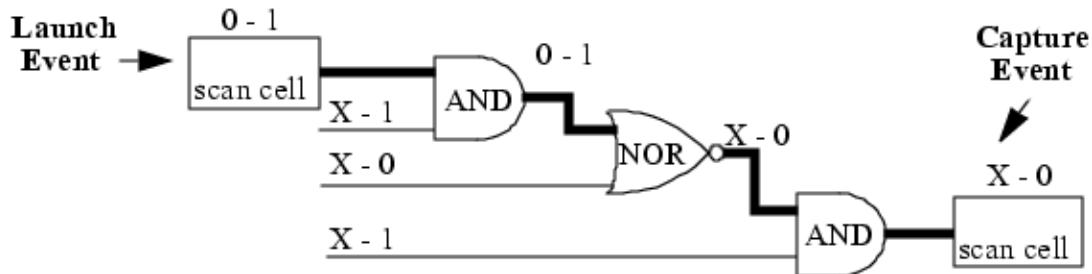


Transition Fault Detection

To detect transition faults, the tool must verify that certain conditions are met. The corresponding stuck-at fault must be detected. Also, within a single previous cycle, the node value must be at the opposite value than the value detected in the current cycle.

[Figure 8-11](#) depicts the launch and capture events of a small circuit during transition testing. Transition faults can be detected on any pin.

Figure 8-11. Transition Launch and Capture Events

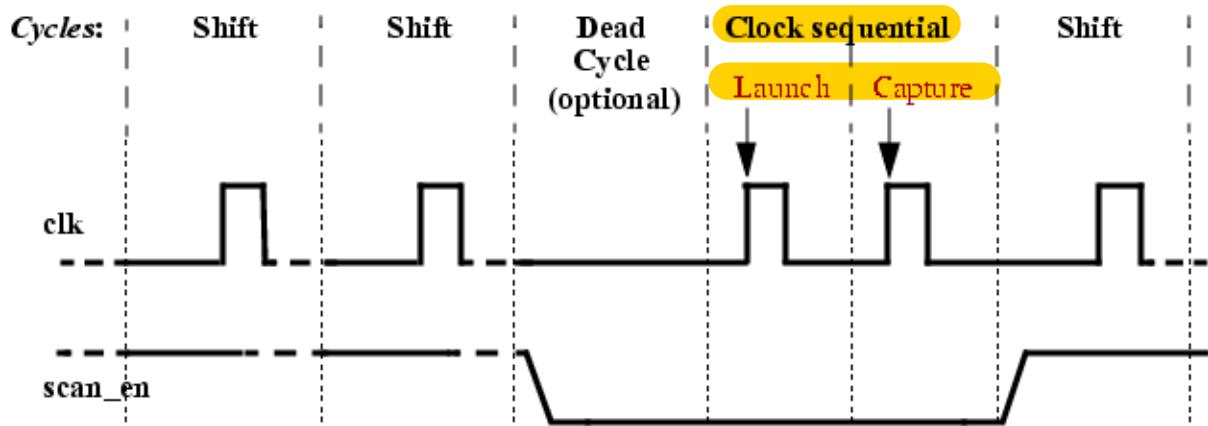


Broadside (Launch-Off-Capture) Patterns

This is a clock sequential pattern, commonly referred to as a broadside pattern. It has basic timing similar to that shown in [Figure 8-12](#) and is the kind of pattern the ATPG tool attempts to create by default when the clock-sequential depth (the depth of non-scan sequential elements in the design) is two or larger. You can specify this depth with the “`set_pattern_type -sequential`” command, although the `create_patterns` command will automatically select and set the optimal sequential depth for you.

Typically, this type of pattern eases restrictions on scan enable timing because of the relatively large amount of time between the last shift and the launch. After the last shift, the clock is pulsed at speed for the launch and capture cycles.

Figure 8-12. Basic Broadside Timing



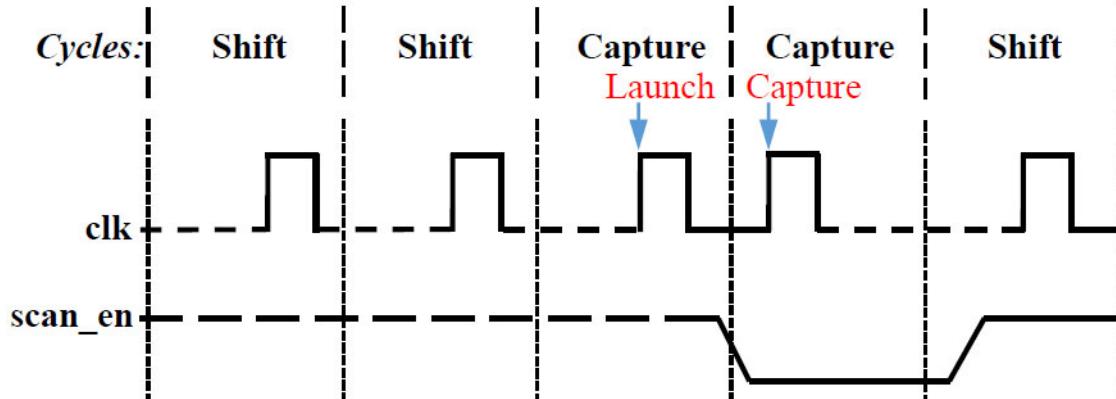
The following are example commands you could use at the command line or in a dofile to generate broadside transition patterns:

```
SETUP> add_input_constraints scan_en -c0 // force for launch & capture.
ANALYSIS> set_fault_type transition
ANALYSIS> create_patterns
```

Pseudo Launch-Off-Shift Patterns

This method of pattern generation is modeled within the capture cycles of ATPG. The patterns typically include two cycles. During the first capture cycle, the design is kept in shift mode. During the second cycle, the scan enable is de-asserted and the capture is performed. This method is more commonly used because it allows the tool to perform shift and capture at-speed using PLL clocks.

Figure 8-13. Pseudo Launch Off Shift Timing



You use a named capture procedure to force scan_en to change from 1 to 0 at the second capture cycle. The scan_en is typically a pipelined signal, not a PI, so it can have at-speed timing to switch at the capture cycle, which is usually much faster than the shift cycles.

Native Launch-Off-Shift Patterns

If the -allow_shift_launch is used with the “set_fault_type transition” command, the tool may attempt to generate a pattern that includes the events shown in Table 8-4.

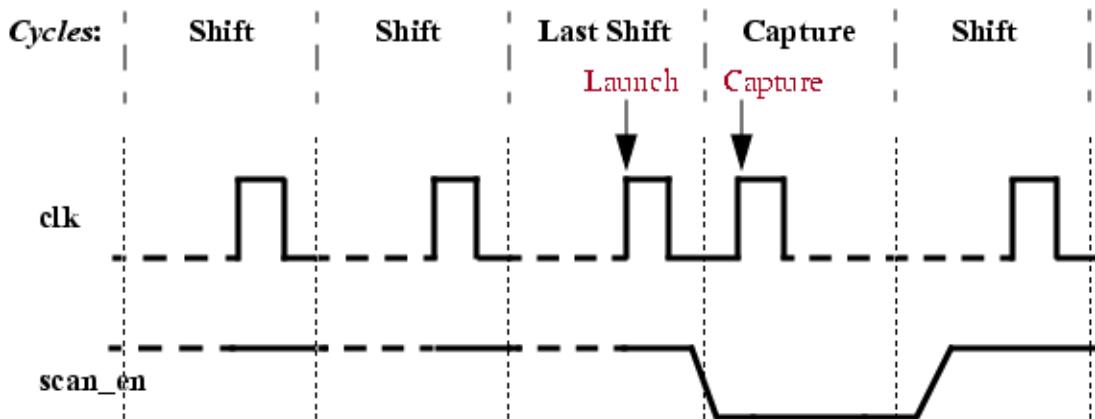
With native launch-off-shift patterns, the transition tested is launched by the last shift applied by the shift procedure. The capture event occurs when ATPG generates a single cycle test. When generating retargetable patterns for the transition fault type, launch-off-shift is automatically disabled even if you invoke the “set_fault_type transition” command with the -allow_shift_launch switch.

Table 8-4. Typical Events in a Launch-Off-Shift Pattern

1. Init_force primary inputs
2. Load scan chains
3. Force primary inputs
4. Pulse clock
5. Unload scan chains

In this type of pattern, commonly referred to as a launch off last shift or just launch off shift pattern, the transition occurs because of the last shift in the load scan chains procedure (event #2) or the forcing of the primary inputs (event #3). Figure 8-14 shows the basic timing for a launch that is triggered by the last shift.

Figure 8-14. Native Launch-Off-Shift Timing



If your design supports this requirement, you can direct the tool to create launch off shift patterns by including the -allow_shift_launch switch when specifying transition faults with the set_fault_type command.

Note

 Launch-off-shift patterns require the scan enable signal for mux-scan designs to transition from shift to capture mode at speed. Therefore, the scan enable must be globally routed and timed similar to a clock. Also, because launch off shift patterns detect a certain number of faults in non-functional paths as well as in the scan path, the test coverage reported is usually higher than for broadside patterns which do not capture faults in non-functional paths.

To create transition patterns that launch off the last shift, use a sequence of commands similar to this:

```
ANALYSIS> set_fault_type transition -allow_shift_launch
ANALYSIS> set_pattern_type -sequential 0 //prevent broadside patterns.
ANALYSIS> create_patterns
```

Related Topics

[set_abort_limit](#) [Tessent Shell Reference Manual]
[set_fault_type](#) [Tessent Shell Reference Manual]
[set_pattern_type](#) [Tessent Shell Reference Manual]

Generating a Transition Test Set

Transition faults model large delay defects at gate terminals in the circuit under test. This basic procedure generates the transition test set to test gate terminals or nodes for slow-to-rise or slow-to-fall behavior.

Procedure

1. Perform circuit setup tasks, as discussed in “[ATPG Basic Tool Flow](#)” on page 275 and “[ATPG Setup](#)” on page 307 in this manual.
2. Constrain the scan enable pin to its inactive state. For example:

```
SETUP> add_input_constraints scan_en -c0
```

3. Set the sequential depth to two or greater (optional):

```
SETUP> set_pattern_type -sequential 2
```

4. Enter analysis system mode. This triggers the tool’s automatic design flattening and rules checking processes.

```
SETUP> set_system_mode analysis
```

5. Set the fault type to transition:

```
ANALYSIS> set_fault_type transition
```

6. Add faults to the fault list:

ANALYSIS> add_faults -all

7. Run test generation:

ANALYSIS> create_patterns

Related Topics

[Delay Test Set Creation](#)

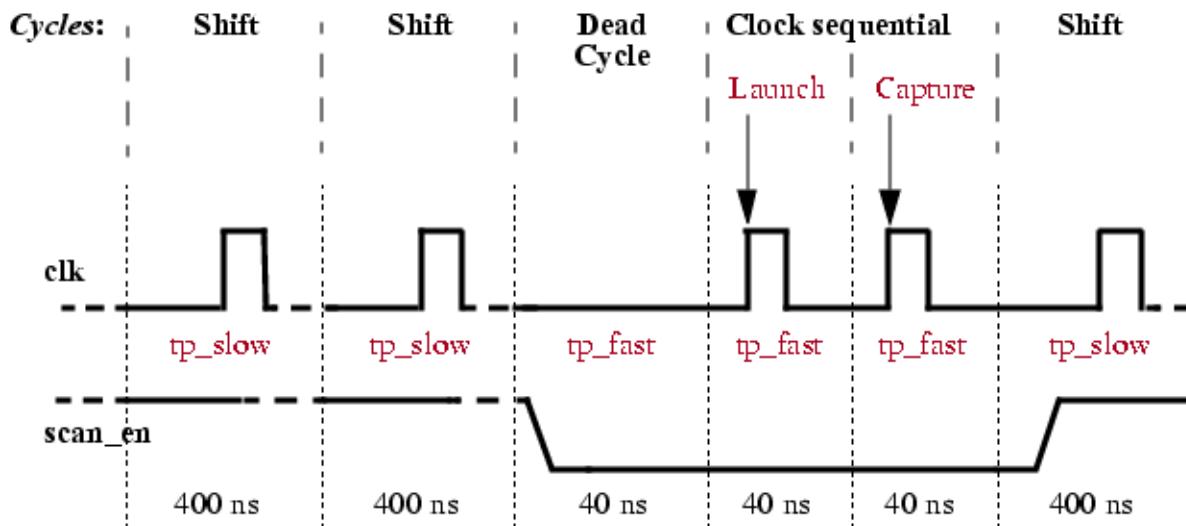
Timing for Transition Delay Tests

For transition delay tests, the tool obtains the timing information from the test procedure file. This file describes the scan circuitry operation to the tool. You can create scan circuitry manually, or use Tessent Scan to create the scan circuitry for you after it inserts scan circuitry into the design.

The test procedure file contains cycle-based procedures and timing definitions that tell the ATPG tool how to operate the scan structures within a design. For more information, refer to “[Test Procedure File](#)” in the *Tessent Shell User’s Manual*.

Within the test procedure file, timeplates are the mechanism used to define tester cycles and specify where all event edges are placed in each cycle. As shown conceptually in [Figure 8-12](#) for broadside testing, slow cycles are used for shifting (load and unload cycles) and fast cycles for the launch and capture. [Figure 8-15](#) shows the same diagram with example timing added.

Figure 8-15. Broadside Timing Example



This diagram now shows 400 nanosecond periods for the slow shift cycles defined in a timeplate called *tp_slow* and 40 nanosecond periods for the fast launch and capture cycles defined in a timeplate called *tp_fast*.

The following are example timeplates and procedures that would provide the timing shown in [Figure 8-15](#). For brevity, these excerpts do not comprise a complete test procedure. Normally, there would be other procedures as well, like setup procedures.

```
timeplate tp_slow =
    force_pi 0;
    measure_po 100;
    pulse clk 200 100;
    period 400;
end;

procedure load_unload =
    scan_group grp1;
    timeplate tp_slow;
    cycle =
        force clk 0;
        force scan_en 1;
    end;
    apply shift 127;
end;

procedure shift =
    timeplate tp_slow;
    cycle =
        force_sci;
        measure_sco;
        pulse clk;
    end;
end;

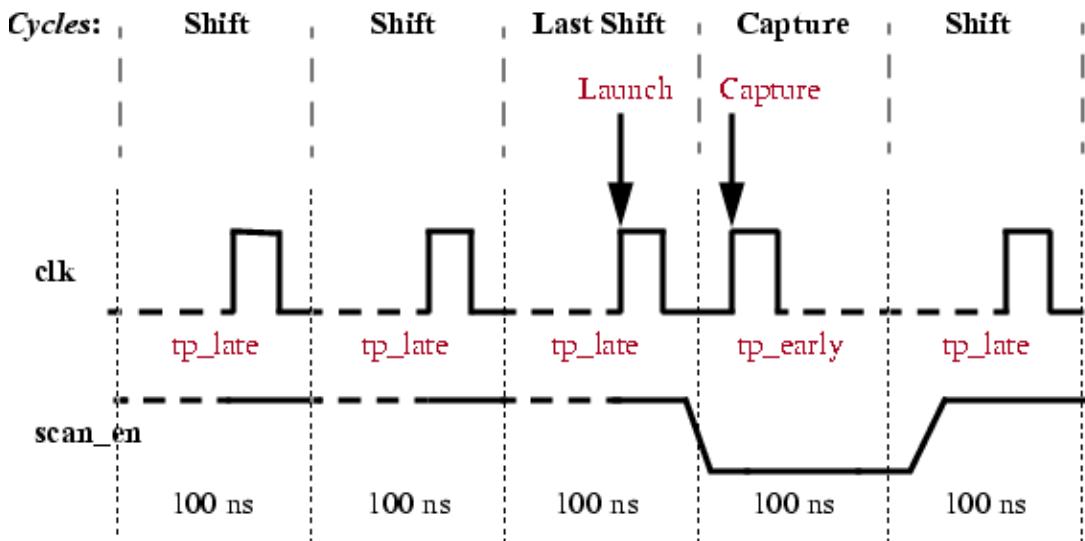
timeplate tp_fast =
    force_pi 0;
    measure_po 10;
    pulse clk 20 10;
    period 40;
end;

procedure capture =
    timeplate tp_fast;
    cycle =
        force_pi;
        measure_po;
        pulse_capture_clock;
    end;
end;

procedure clock_sequential =
    timeplate tp_fast;
    cycle =
        force_pi;
        pulse_capture_clock;
        pulse_read_clock;
        pulse_write_clock;
    end;
end;
```

In this example, there are 40 nanoseconds between the launch and capture clocks. If you want to create this same timing between launch and capture events, but all your clock cycles have the same period, you can skew the clock pulses within their cycle periods—if your tester can provide this capability. [Figure 8-16](#) shows how this skewed timing might look.

Figure 8-16. Launch Off Shift (Skewed) Timing Example



The following timeplate and procedure excerpts show how skewed launch off shift pattern events might be managed by timeplate definitions called *tp_late* and *tp_early*, in a test procedure file:

Note

For brevity, these excerpts do not comprise a complete test procedure. The shift procedure is not shown and normally there would be other procedures as well, like setup procedures.

```

timeplate tp_late =
  force_pi 0;
  measure_po 10;
  pulse clk 80 10;
  period 100;
end;

procedure load_unload =
  scan_group grp1;
  timeplate tp_late;
  cycle =
    force clock 0;
    force scan_en 1;
  end;
  apply shift 7;
end;

timeplate tp_early =
  force_pi 0;
  measure_po 10;
  pulse clk 20 10;
  period 100;
end;

procedure capture =
  timeplate tp_early;
  cycle =
    force_pi;
    measure_po;
    pulse_capture_clock;
  end;
end;

```

By moving the clock pulse later in the period for the load_unload and shift cycles and earlier in the period for the capture cycle, the 40 nanosecond time period between the launch and capture clocks is achieved.

Transition Fault Detection and Multiple Clocks

When you are creating transition fault patterns, it is important to understand the available commands to handle multiple clocks.

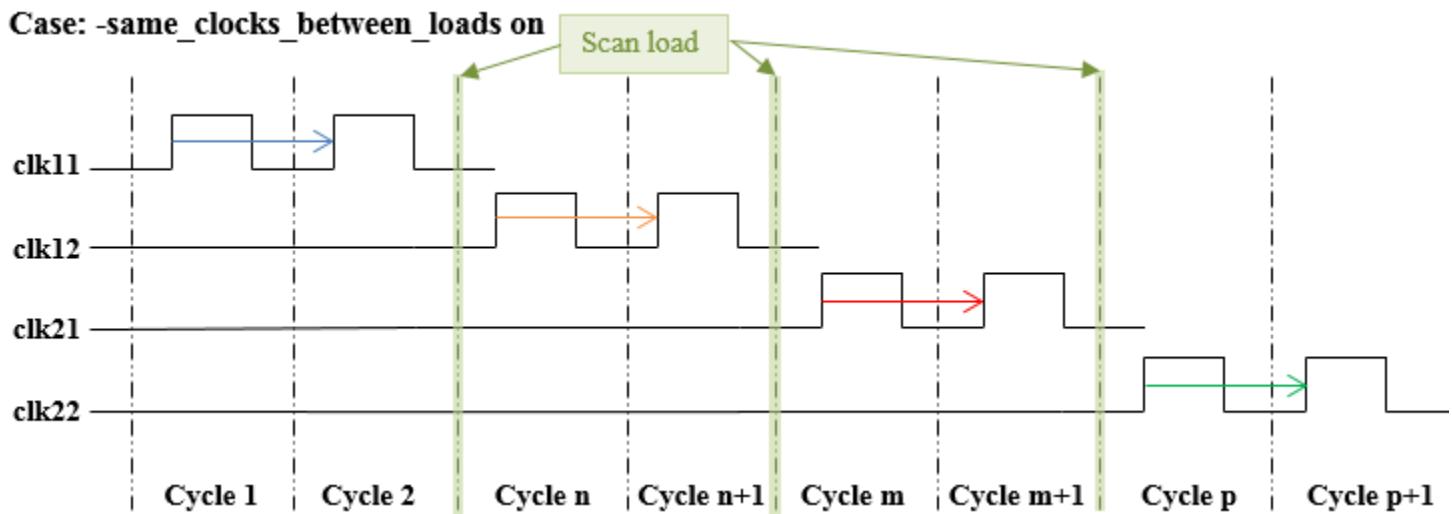
The `set_clock_restriction` command specifies whether ATPG can create patterns with more than one active clock. The `domain_clock` literal for that command enables the tool to generate patterns that pulse compatible scan chain capture or system clocks at the same time.

There are three separate ways to control the behavior of the tool regarding the creation of test patterns with more than one active clock.

- `set_clock_restriction domain_clock -same_clocks_between_loads on`

When you use “`-same_clocks_between_loads on`” switch for the “`set_clock_restriction domain_clock` command”, the tool uses the same clocks in every cycle within a load/unload interval during pattern creation. See [Figure 8-17](#) for example waveforms. The green highlights between Cycle 2/Cycle n, Cycle n+1/Cycle m, and Cycle m+1/Cycle p represent load cycles that occur between them.

Figure 8-17. Same Clocks Between Loads On

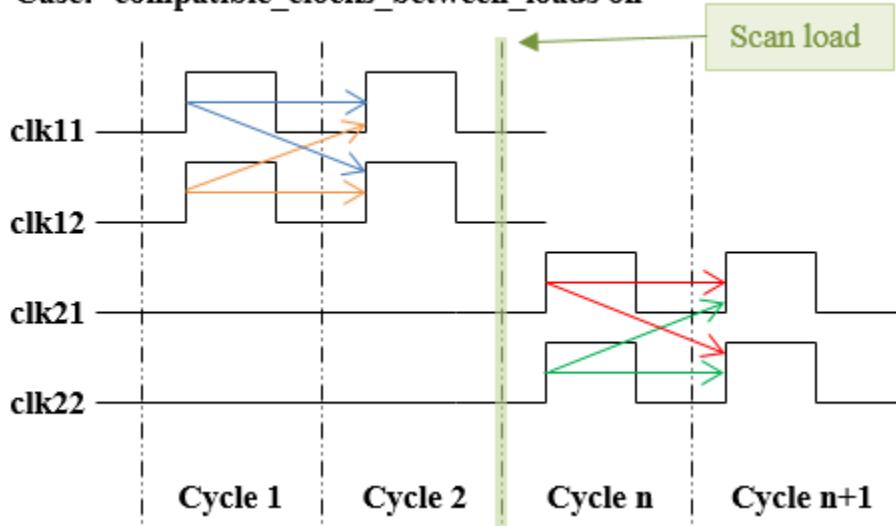


- `set_clock_restriction domain_clock -compatible_clocks_between_loads on`

When you use “`-compatible_clocks_between_loads on`” switch for the “`set_clock_restriction domain_clock` command”, the tool will only allow compatible clocks to pulse within capture cycles. Incompatible clocks are not allowed to pulse between load/unload operations. See [Figure 8-18](#) for example waveforms. In this case, `clk11` and `clk12` are compatible clocks. The green highlight between Cycle 2/Cycle n, represents a load cycle that occurs between them

Figure 8-18. Compatible Clocks Between Loads On

Case: -compatible_clocks_between_loads on



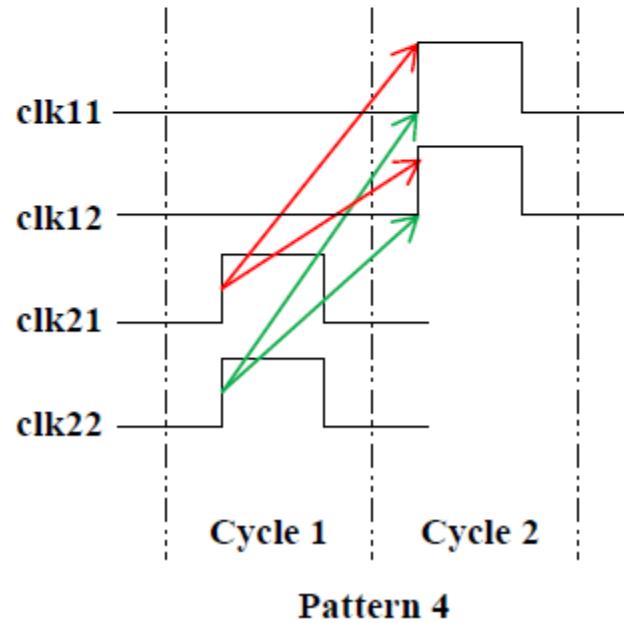
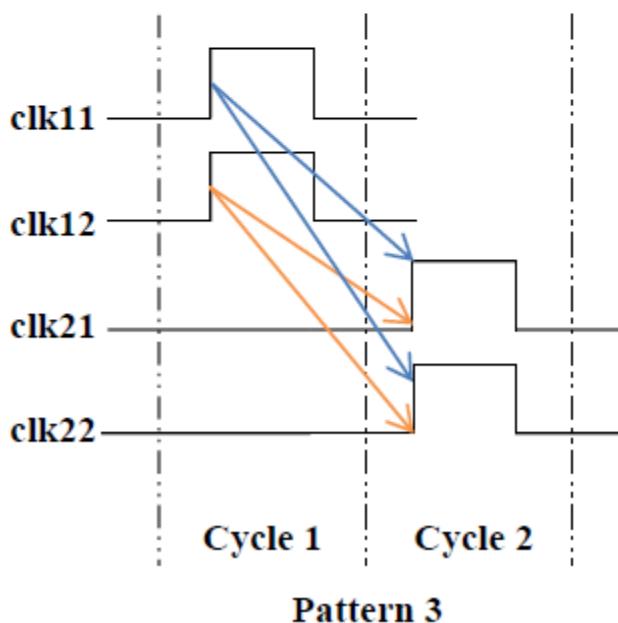
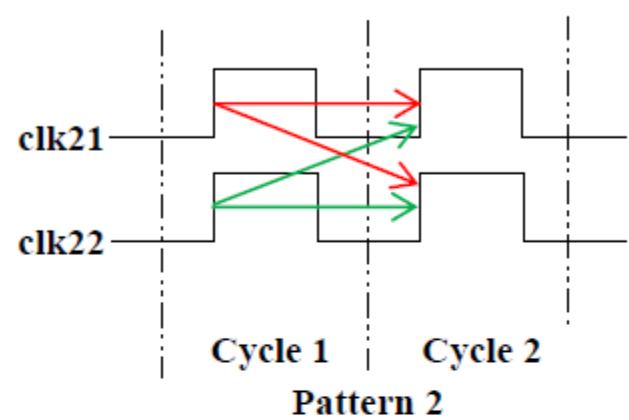
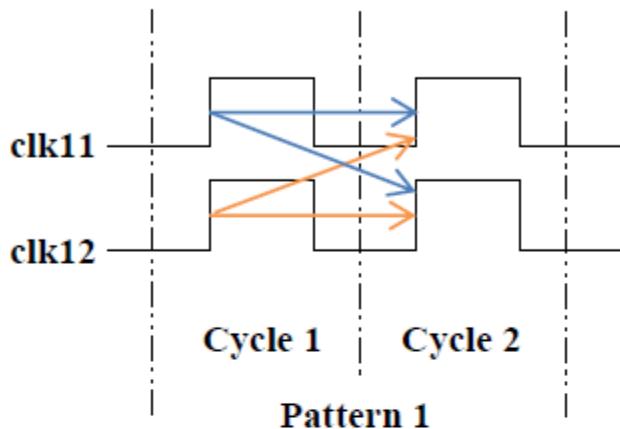
For at-speed fault models, when you are not using named capture procedures, and there are no user defined settings or you are not calling `create_patterns` with the `-override` switch, the tool will automatically turn on `hold_pi` and mask all primary outputs. It will also automatically set clock restriction to “domain_clock” with `-any_interaction` and `-compatible_clocks_between_loads on`.

- `set_clock_restriction domain_clock`

When you use the “`set_clock_restriction domain_clock`” without the `-same_clocks_between_loads` or `-compatible_clocks_between_loads` switches, the tool is not required to use the same clocks in every cycle within a load/unload interval during pattern creation. The tool enforces the compatible clock requirement so that incompatible clocks are not allowed to pulse in the same cycle but allowed to pulse at different capture cycles. See [Figure 8-19](#) for example waveforms. In this case, clk11 and clk12 are compatible clocks and clk21 and clk22 are compatible clocks. However, clk11 and clk12 are not compatible with clk21 and clk22.

Figure 8-19. domain_clock

Case: domain_clock without either switch
 (-compatible_clocks_between_loads off
 -same_clocks_between_loads off)



When using `create_patterns` to create transition fault patterns, by default the tool will set the clock restrictions to `domain_clock` (edge interaction) and the `-compatible_clocks_between_loads` switch to on. It will also set the `set_transition_holdpi` and `set_output_masks` to on.

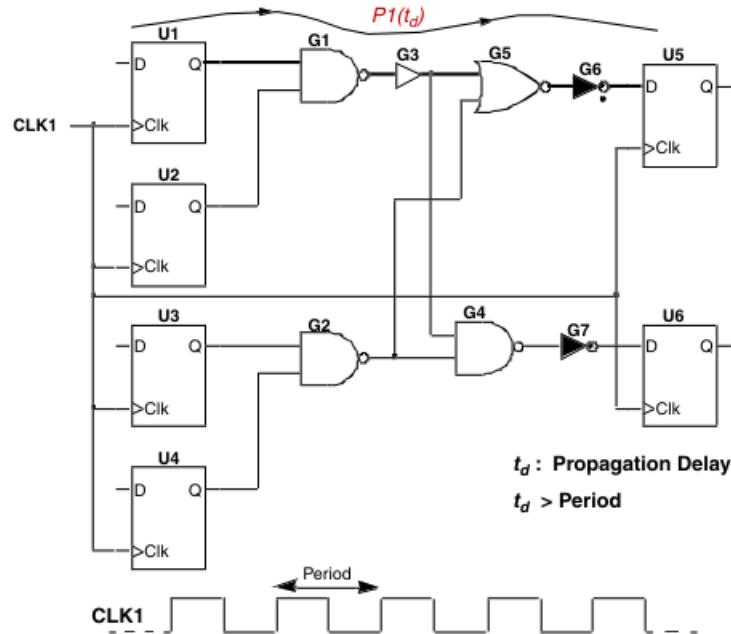
Pattern Failures Due to Timing Exception Paths

Prior to ATPG, you can perform a timing optimization on a design using a static timing analysis (STA) tool.

This process also defines timing exception paths consisting of one of the following:

- **False Path** — A path that cannot be sensitized in the functional mode of operation (the STA tool ignores these paths when determining the timing performance of a circuit).
- **Multicycle Path** — A path with a signal propagation delay of more than one clock cycle. [Figure 8-20](#) shows path $P1$ beginning at flip-flop U1, going through gates G1, G3, G5, and G6, and ending at flip-flop U5. This path has a total propagation delay longer than the clock period.

Figure 8-20. Multicycle Path Example



You should evaluate the effect of timing exception paths for any sequential pattern containing multiple at-speed capture clock pulses, either from the same clock or from different clocks. This includes the following pattern types:

- Clock sequential (broadside transition patterns and stuck-at patterns)
- RAM sequential
- Path delay

The ATPG tool automatically evaluates false and multicycle paths before creating the test patterns. When simulating patterns during ATPG, the tool identifies transitions that propagate through false or multicycle paths and masks them (modifies them to X) at the capturing flops in the resultant patterns when the transitions may not be stable due to the timing exceptions. For multicycle paths, expected values are accurately simulated and stored in the pattern set.

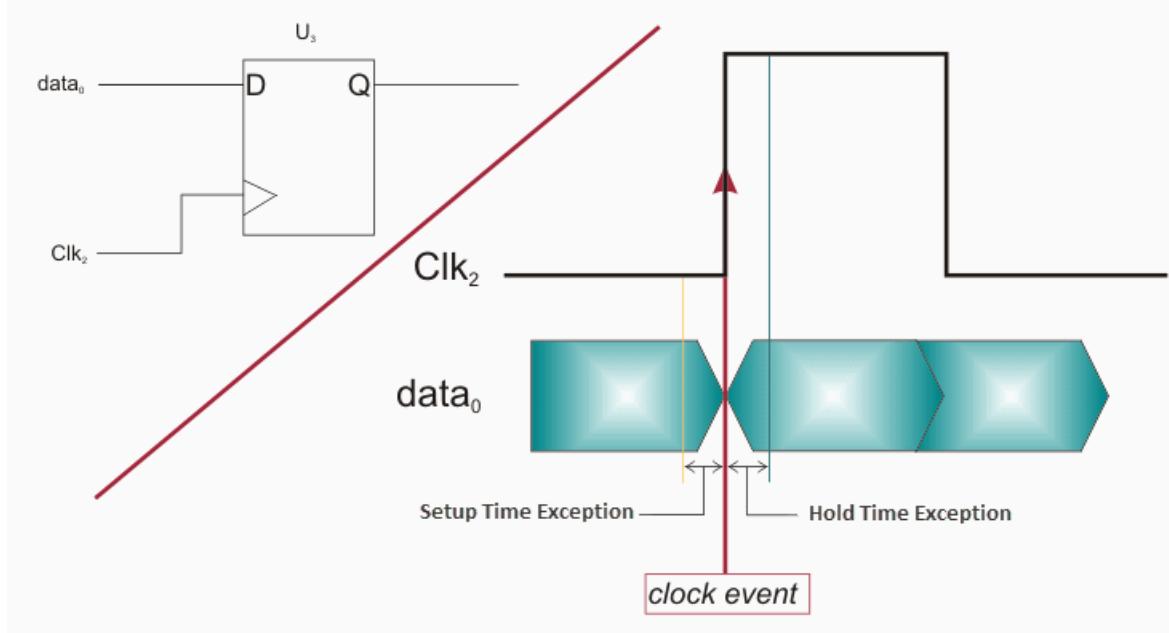
Types of Timing Exception Paths.....	338
Timing Exception Paths from an SDC File.....	340
Does the SDC File Contain Valid SDC Information?	341
Manual Definition of False Paths	342

Types of Timing Exception Paths

Most designs hold data and control inputs constant for specified time periods before and after any clock events. In this context, the time period before the clock event is the setup time, and the time period after the clock event is hold time.

Figure 8-21 illustrates how setup and hold time exceptions can produce the following timing exception paths: “Setup Time Exceptions” on page 338 and “Hold Time Exceptions” on page 339.

Figure 8-21. Setup Time and Hold Time Exceptions



Setup Time Exceptions

A timing exception path with a setup time exception does not meet the setup time requirements. This type of exception can affect test response for at-speed test patterns.

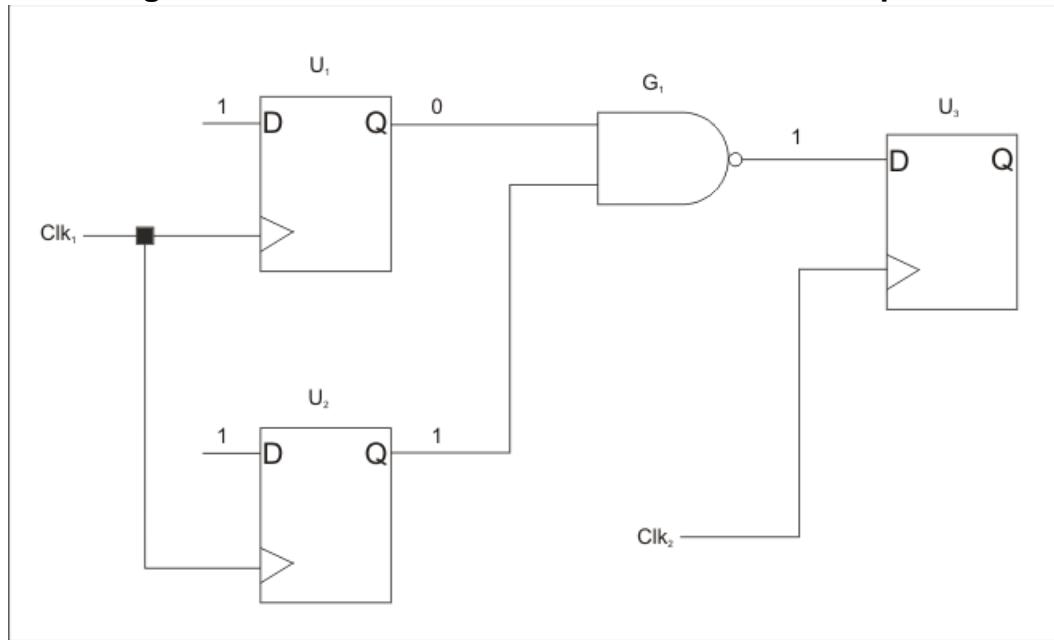
Hold Time Exceptions

A timing exception path with a hold time exception does not meet the hold time requirements. This type of exception can affect test response of *any* test pattern and usually occurs across different clock domains. The ATPG tool simulates hold time false paths for the following timing exception paths:

- False paths you manually specify with the [add_false_paths](#) command using the -hold switch—see “[Manual Specification of False Paths for Hold Time Exception Checks](#)”.
- False paths between two clock domains.
- Multicycle paths with a path multiplier of 0 (zero).

[Figure 8-22](#) illustrates a hold time exception occurring across different clock domains.

Figure 8-22. Across Clock Domain Hold Time Exception



[Figure 8-22](#) shows the false paths from the clock domain Clk1 to the clock domain Clk2. In this figure, when a test pattern has a clock sequence of simultaneously pulsing both Clk1 and Clk2, there can be a hold time exception from the flip-flop U1 or U2 to the flip-flop U3.

In [Figure 8-22](#), pulsing clocks Clk1 and Clk2 simultaneously places the new value 1 at the D input of flip-flop U1, creating a rising transition on the flip-flop U1’s Q output. This transition sensitizes the path from the flip-flop U1 to the flip-flop U3. If the clock Clk2 arrives late at the flip-flop U3 the new value 0 is captured at the flip-flop U3 instead of the old value 1.

Timing Exception Paths from an SDC File

STA tools typically provide a command to write out the false and multicycle path information identified during the STA process into a file or script in Synopsys Design Constraint (SDC) format. For example, the Synopsys PrimeTime tool has the `write_sdc` command. You can also read in both the `-setup` and `-hold` SDC information.

Tip

i Before using the `write_sdc` command, use the PrimeTime `transform_exceptions` command to eliminate redundant, overridden or invalid exceptions. It is recommended to use this command only with the default options. In particular, it is recommended that you *not* use the `-flatten` option, because it can result in excessively large SDC files.

If you can get the information into an SDC file, you can use the `read_sdc` command to read in the false path definitions from the file. Note that the tool does not infer false and multicycle paths from the delays in an SDF file.

The following is an example of the use of this command in a typical command sequence for creating broadside transition patterns:

```
<Define clocks, scan chains, constraints, and so on>
ANALYSIS> set_fault_type transition
ANALYSIS> set_pattern_type -sequential 2
ANALYSIS> read_sdc my_sdc_file
...
ANALYSIS> create_patterns
```

If you already have a pattern set for your design and want to see the effect of adding the false and multicycle path information, the command sequence is slightly different:

```
<Define clocks, scan chains, constraints, and so on>
ANALYSIS> read_sdc my_sdc_file
ANALYSIS> add_faults -all
ANALYSIS> read_patterns my_patterns.ascii
ANALYSIS> simulate_patterns
ANALYSIS> report_statistics
```

As a result of simulating the patterns using the false and multicycle path information, the patterns read in from the external pattern file will now be stored in the tool's internal pattern set, with some capture values in the internal patterns changed to "X." These changed values represent masking the tool applied to adjust for false and multicycle path effects. The Xs will increase the number of undetected faults slightly and lower the test coverage; however, the patterns will be more correct and will eliminate mismatches related to those capture values.

Note

 You can save the patterns that include the false and/or multicycle path information as usual.

For example:

```
ANALYSIS> write_patterns my_patterns_falsepaths.v -verilog
ANALYSIS> write_patterns my_patterns_falsepaths.ascii -ascii
```

Does the SDC File Contain Valid SDC Information?

The read_sdc command reads an SDC file and parses it, looking for supported commands that are relevant for ATPG.

ATPG tools support the following SDC commands and arguments:

- all_clocks
- create_clock [-name clock_name]
- create_generated_clock [-name clock name]
- get_clocks
- get_generated_clocks
- get_pins
- get_ports
- set_case_analysis value port_or_pin_list
- set_clock_groups
- set_disable_timing [-from from_pin_name] [-to to_pin_name] cell_pin_list
- **set_false_path** [-setup] [-hold] [-from from_list] [-to to_list] [-through through_list]
[-rise_from from_list] [-rise_to to_list] [-fall_from from_list] [-fall_to to_list]
- set_hierarchy_separator
- **set_multicycle_path** [-setup] [-hold] [-from from_list] [-to to_list]
[-through through_list] [-rise_from from_list] [-rise_to to_list] [-fall_from from_list]
[-fall_to to_list]

Note

 For complete information on these commands and arguments, refer to your SDC documentation.

To avoid problems extracting the timing exception paths from the SDC specifications, the best results are obtained when the SDC file is written out by the PrimeTime static timing analysis (STA) tool. Mentor Graphics highly recommends that within PrimeTime you use the `transform_exceptions` command with only the default options before saving the SDC file. This command removes any redundant, invalid and overridden path exceptions. Then use the `write_sdc` command to save the updated information to a file you can use with `read_sdc`.

The following summarizes the recommended steps:

1. Read the original SDC file(s) in PrimeTime.
2. Execute the `transform_exceptions` command in PrimeTime.
3. Execute the `write_sdc` command in PrimeTime, to write out a valid SDC file.
4. In the ATPG tool, use the `read_sdc` command to read the SDC file written out in step 3.
5. Generate at-speed patterns.

Manual Definition of False Paths

Alternatively using the `add_false_paths` command, you can manually specify false path definitions for both specifying false paths for setup time exception checks and specifying false paths for hold time exception checks.

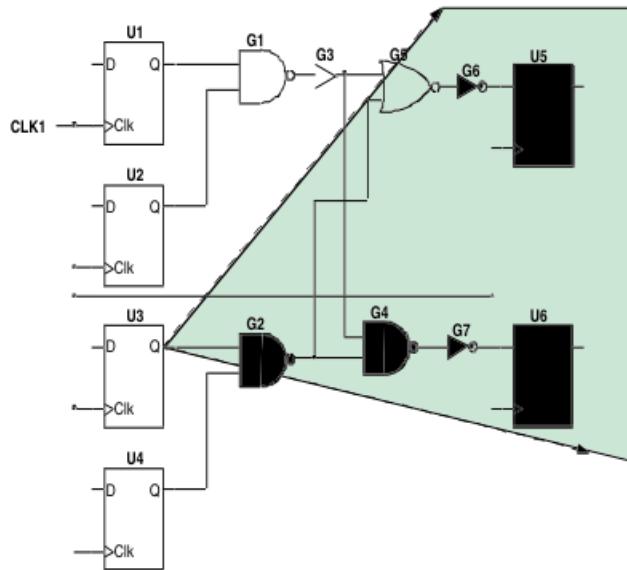
When performing this operation, you must be specific and accurate when specifying false (and multicycle) paths during the STA process, and to maintain the same accuracy when using the `add_false_paths` command.

For example, defining non-specify false path definition with the following command:

`add_false_paths -from U3`

would result in the propagation of the false path out through the design in an effect cone encompassing all possible paths from that node. [Figure 8-23](#) shows an illustration of this.

Figure 8-23. Effect Cone of a Non-specific False Path Definition



Manual Specification of False Paths for Setup Time Exception Checks

By default, the tool evaluates setup time exceptions for the false paths you manually specify with the `add_false_paths` command.

Manual Specification of False Paths for Hold Time Exception Checks

For hold time exceptions, you identify the path with the `add_false_paths` command and also specify the `-hold` switch. The following are useful commands for working with false paths:

- `add_false_paths` — Specifies one or more false paths.
- `delete_false_paths` — Deletes the specified false path definitions.
- `report_false_paths` — Displays the specified false path definitions.
- `delete_multicycle_paths` — Deletes the specified multicycle path definitions.
- `report_multicycle_paths` — Displays the specified multicycle path definitions.

SDC Timing Exception Effects

The false path and multicycle path timing exception information provided by the SDC file have an impact on the tool's internal pattern set.

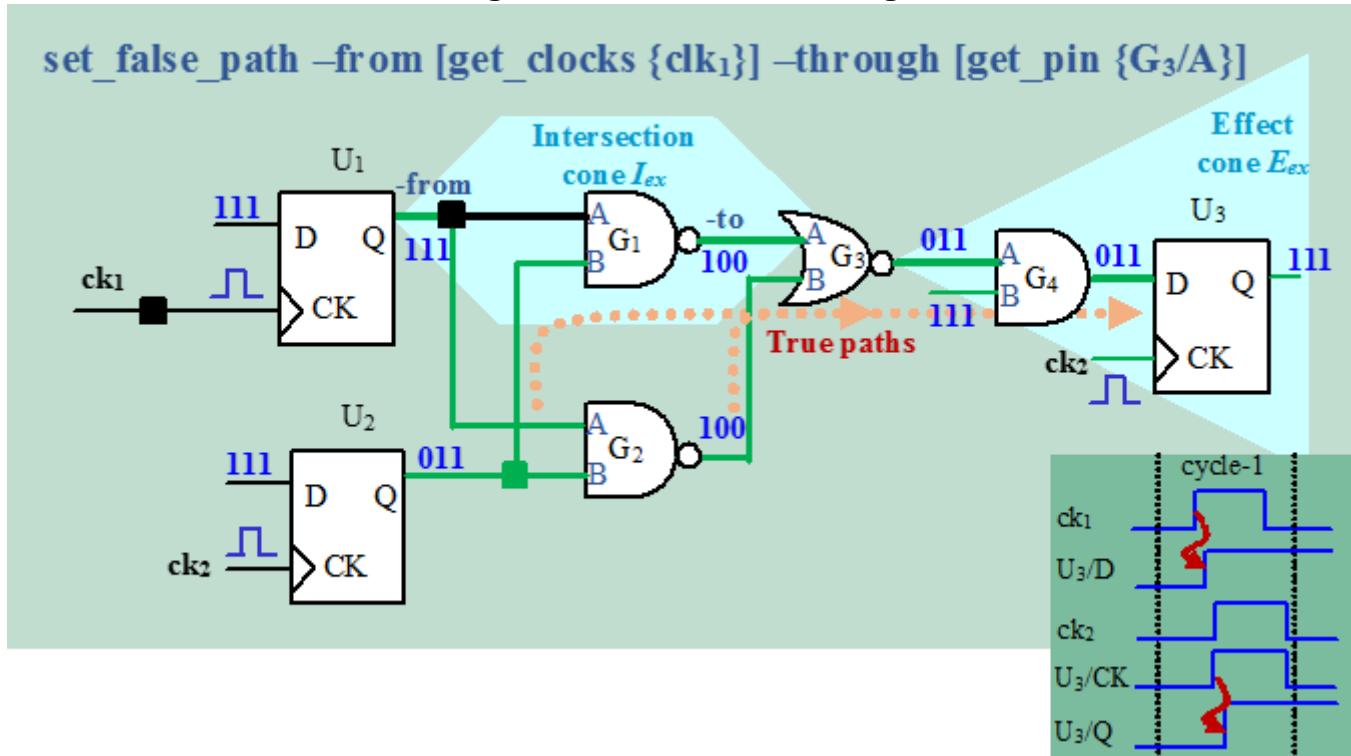
Consider these effects when using SDC provided timing exceptions:

- Scan loading values are stable and are not impacted by SDC constraints.
- The tool considers both setup and hold time exceptions for every capture frame.
- Hold time exceptions impact the current clock edge.
- Hold time exceptions impact both the stuck-at and transition fault test coverage.
- Setup time exceptions impact the following clock edges.
- Setup time exceptions mainly impact the transition fault test coverage, except training-edge (TE) edge flops.
- ATPG considers multicycle paths to determine the proper sequential depth to test the faults on multicycle paths.

No Masking

Figure 8-24 shows that when there is no transition on the false path, no masking occurs.

Figure 8-24. SDC No Masking

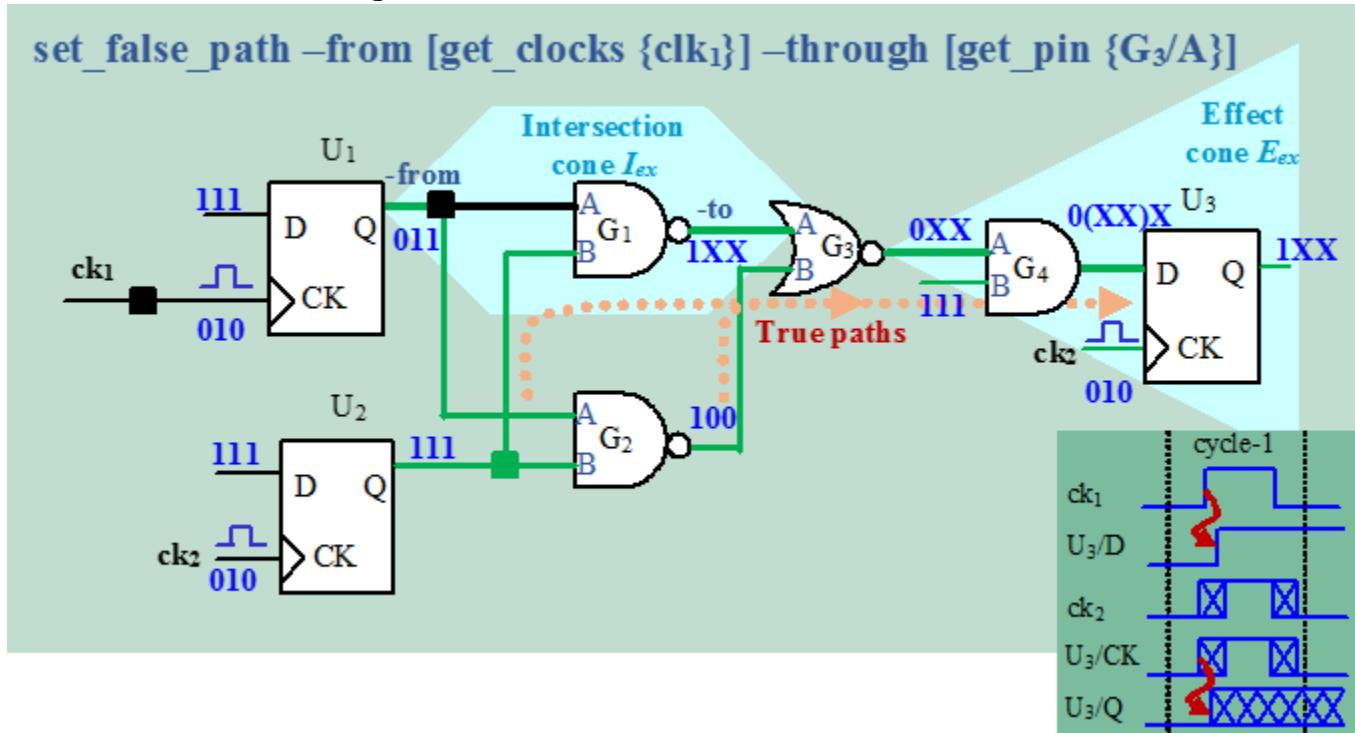


SDC False Path

Hold Time Exceptions

Figure 8-25 shows the effect of the hold time exception. The hold time exception impacts the current clock edge. The result is Xs on G_1/Q which propagate through U_3/Q .

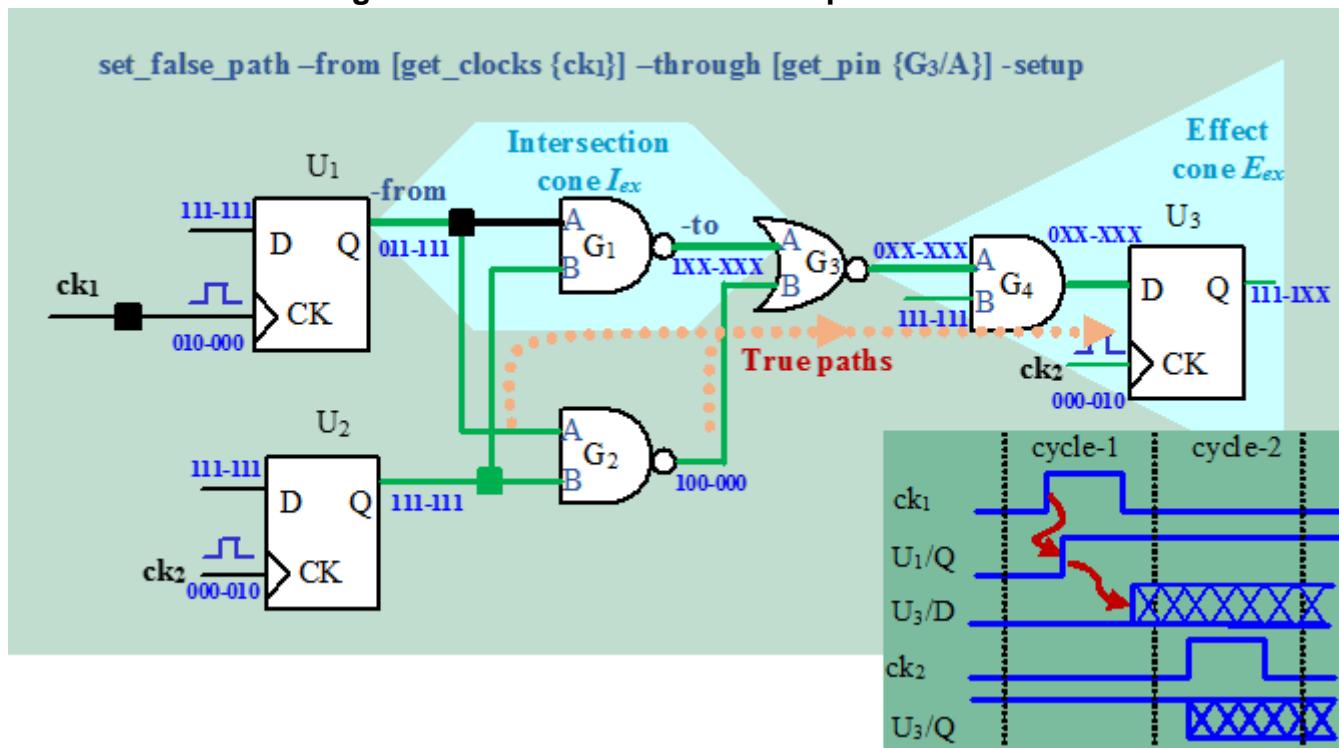
Figure 8-25. SDC False Path Hold Time Effect



Setup Time Exceptions

Figure 8-26 shows a circuit and the logic cones formed by the SDC set_false_path command. There is an intersection cone from U_1 through G_3/A and an effect cone from the output of G_3 . It also shows the effect of the setup time exception for this path on U_3/D and U_3/Q which are inside the effect cone of the false path. The setup time exception impacts the following clock edge. The result is Xs on U_3/D and U_3/Q into the second cycle.

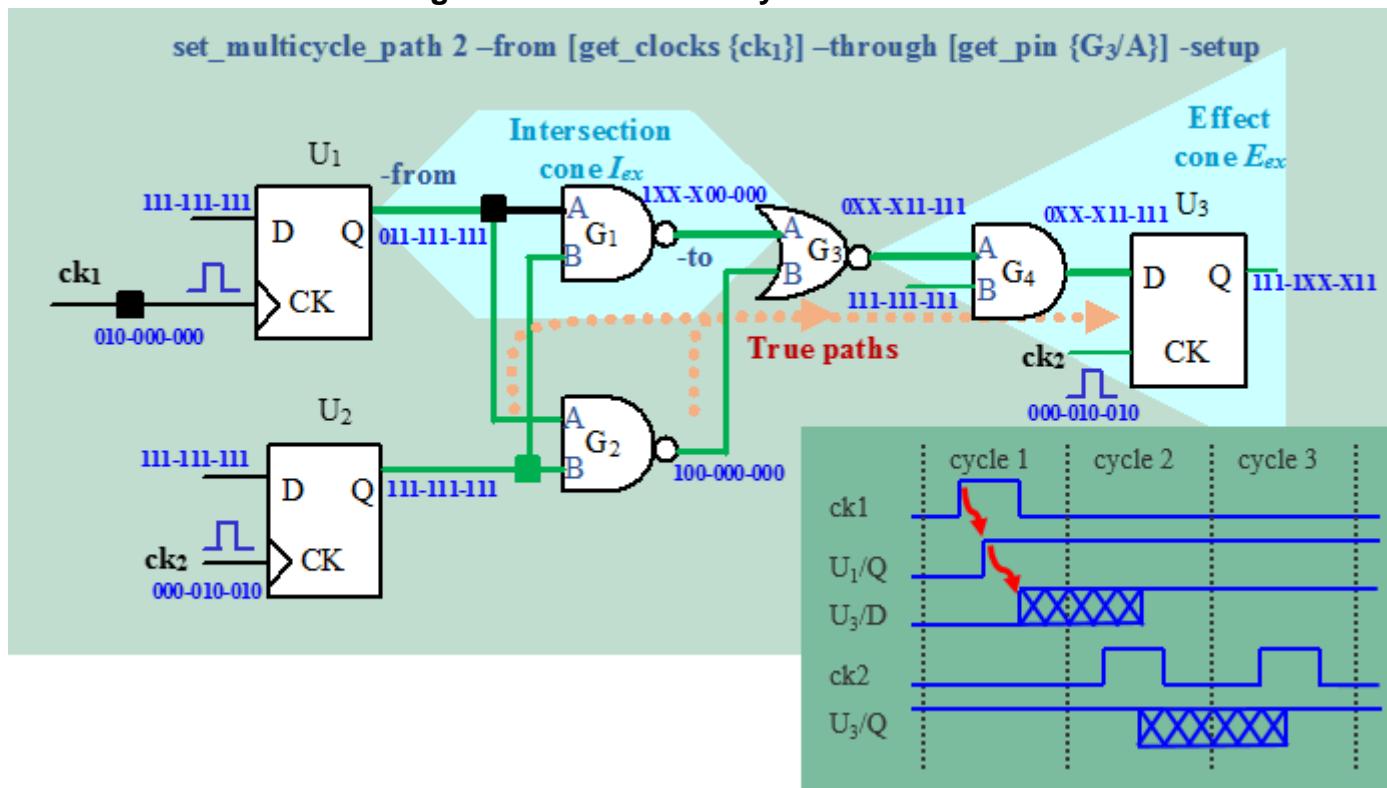
Figure 8-26. SDC False Path Setup Time Effect



Multicycle Path

Figure 8-27 illustrates that masking will occur during the second cycle, and becomes stable after the second cycle.

Figure 8-27. SDC Multicycle Path Effect



Debugging Test Coverage with Timing Exceptions

Due to the masking of unreliable transitions through timing exception paths, timing exceptions, typically from SDC files, can impact the test coverage for static and at-speed test.

This procedure can be used to improve test coverage if your coverage goals are not achieved because of timing exceptions.

Procedure

1. Enable the timing exception X-Statistics functionality to identify the timing exceptions that have the highest masking impact.

`set_timing_exceptions_handling -x_statistics on`
2. When you have identified the timing exceptions with the highest masking impact, remove the offending SDC timing exceptions from the SDC file. Create patterns using the updated SDC file and evaluate the new test coverage results.
3. Run static timing analysis using the modified SDC from step 2 and check that timing verification passes with the updated timing exceptions.
4. If the static timing analysis does not pass with the updated SDC file, you must make the necessary modifications to the design to meet the relaxed SDC timing.

5. For static tests when the clock frequency is sufficiently slow, use the “`set_timing_exceptions_handling -setup off`” command to disable setup timing exception masking. This helps improve static test coverage for sequential patterns. To verify the accuracy of this change, remove setup timing exceptions from the SDC file and check using static timing analysis.

Examples

This example demonstrates how to enable SDC X-Statistics using 1 percent fault sampling and generate 1,000 patterns with a sequential depth of one. After pattern creation, the example reports the top three exceptions with the highest masking in the created patterns. This example focuses on stuck-at fault test coverage impact, which is mainly due to hold time exceptions and can be remedied more easily without design changes.

```
read_sdc original.sdc
set_timing_exceptions_handling -x_statistics on
set_fault_sampling 1
set_atpg_limits -pattern_count 1024
set_pattern_type -sequential 1
create_patterns
...
report_false_paths -x_statistics -count 3

// Total 255 paths out of 350 paths produced Xs.
// -----
// Path Type           Number of Xs   File and line number
// -----
// 1    false path     4424790    original.sdc (line 32761)
// 2    false path     3460790    original.sdc (line 32761)
// 3    false path     463897     original.sdc (line 30390)
// Only the first 3 paths producing the most Xs were reported.
```

This report identifies the top three offending false paths. Edit the SDC file and manually remove the three false paths on lines 32761 and 30390. Save the file to *modify.sdc*. Then issue these commands:

```
delete_sdc -all
read_sdc modify.sdc
create_patterns
```

Compare the test coverage difference from the run with the *original.sdc*.

Repeat the same process if the test coverage goal is not reached and there are still false paths with large masking.

Re-run static timing analysis to determine that timing verification passes with the most currently revised SDC file, *modify.sdc*.

Path Delay Test Set Creation

The ATPG tool can generate patterns to detect path delay faults. These patterns determine if specific user-defined paths operate correctly at-speed.

[“At-Speed Testing and the Path Delay Fault Model”](#) on page 53 introduced the path delay fault model. You determine the paths you want tested (most people use a static timing analysis tool to determine these paths), and list them in an ASCII path definition file you create. You then load the list of paths into the tool. [The Path Definition File](#) describes how to create and use this file.

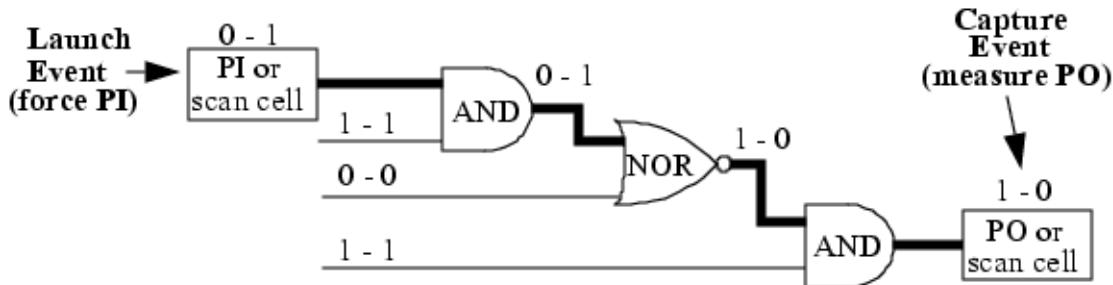
Path Delay Fault Detection	349
The Path Definition File	352
Path Definition Checks	355
Generating a Path Delay Test Set	357
Path Delay Testing Limitations	358

Path Delay Fault Detection

Path delay testing requires a logic value transition, which implies two events need to occur to detect a fault. These events include a launch event and a capture event. Typically, both the launch and capture occur at scan cells, but they can occur at RAMs or, depending on the timing and precision of the ATE to test around a chip’s I/O, at PIs and POs.

Figure 8-28 depicts the launch and capture events of a small circuit during a path delay test.

Figure 8-28. Path Delay Launch and Capture Events



Path delay patterns are a variant of clock-sequential patterns. A typical pattern to detect a path delay fault includes the following events:

1. Load scan chains.
2. Force primary inputs.
3. Pulse clock (to create a launch event for a launch point that is a state element).
4. Force primary inputs (to create a launch event for a launch point that is a primary input).

5. Measure primary outputs (to create a capture event for a capture point that is a primary output).
6. Pulse clock (to create a capture event for a capture point that is a state element).
7. Unload scan chains.

The additional force_pi/pulse_clock cycles may occur before or after the launch or capture events. The cycles depend on the sequential depth required to set the launch conditions or sensitize the captured value to an observe point.

Note

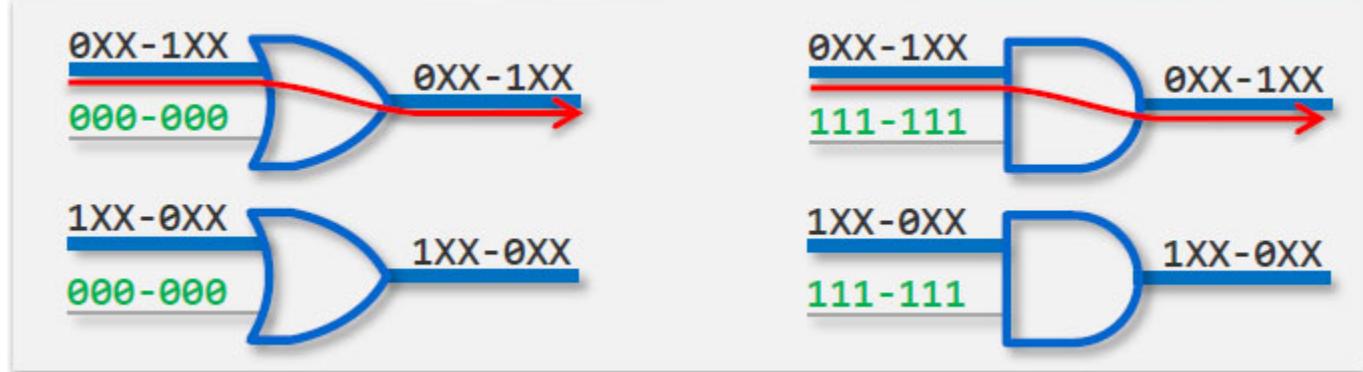
 Path delay testing often requires greater depth than for stuck-at fault testing. The sequential depths that the tool calculates and reports are the minimums for stuck-at testing.

To get maximum benefit from path delay testing, the launch and capture events must have accurate timing. The timing for all other events is not critical.

The ATPG tool detects a path delay fault with either a hazard-free test, a robust test, a non-robust test, or a functional test. If you save a path delay pattern in ASCII format, the tool includes comments in the file that indicate which of these types of detection the pattern uses. The hazard free test combined with robust detection is the most rigid detection test.

Hazard-free detection occurs when the gating input used to sensitize the path value is a non-controlling value that is held constant in every frame of every cycle as shown in Figure 8-29. Hazard-free detection, when used with robust detection, provides a delay test with additional restrictions than when using robust detection alone. Also, when used with robust detection, it guarantees that any robust detection patterns created by the tool are hazard free.

Figure 8-29. Hazard-free Detection Example

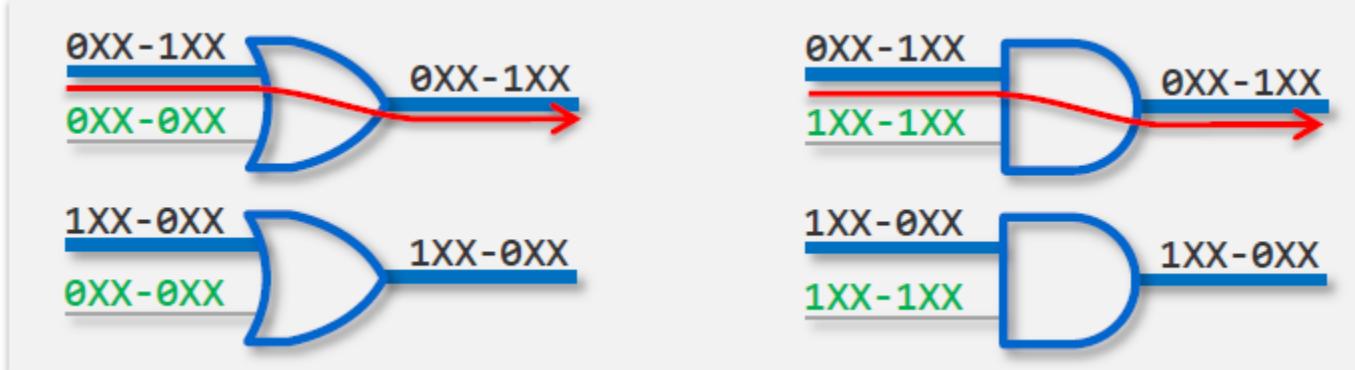


Unlike hazard-free detection, which requires that the gating input is held at a constant non-controlling value through every frame of every cycle, robust detection requires that the gating input used to sensitize the path is held at a non-controlling value only on the first frame of every cycle. Similar to hazard-free, robust detection keeps the gating of the path constant during fault detection and thus, does not affect the path timing. Because it avoids any possible

reconvergent timing effects, it is a desirable type of detection and for that reason is the approach the ATPG tool tries first. The tool, however, cannot use robust detection on many paths because of its restrictive nature and if it is unable to create a robust test, it will automatically try to create a non-robust test. The application places faults detected by robust detection in the DR (det_robust) fault class.

Figure 8-30 gives an example of robust detection within a simple path. Notice that the gating value on the OR and AND gates are stable on the first frame of each cycle and able to retain the proper value for detection during launch and capture events.

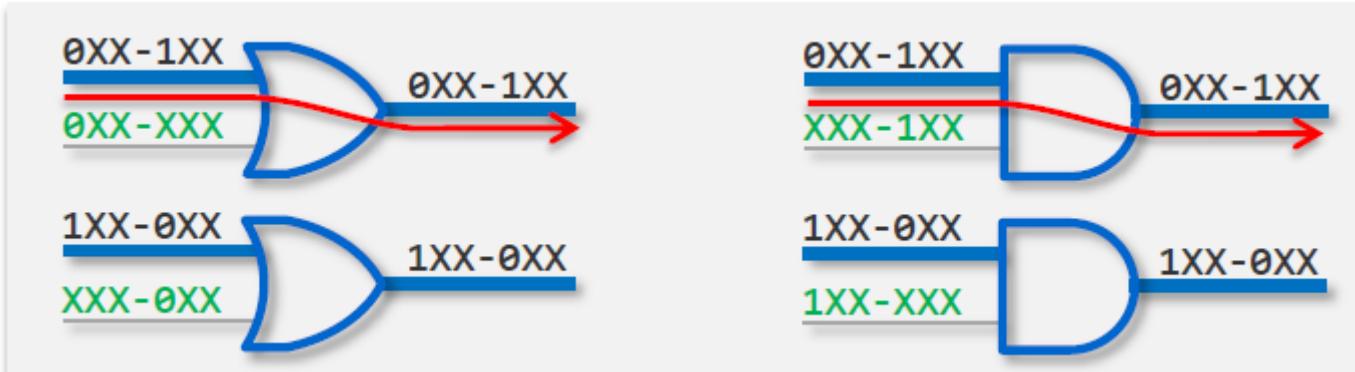
Figure 8-30. Robust Detection Example



Non-robust detection does not require a constant value on the gating input used to sensitize the path. It only requires the proper gating value, meaning it must be non-controlling, at the time of the capture event. The ATPG tool places faults detected by non-robust detection in the DS (det_simulation) fault class.

Figure 8-31 gives an example of non-robust detection within a simple path. Notice that the gating value (highlighted in green) is not constant in every frame or cycle. While the value may transition, it must be non-controlling at the start of the capture cycle.

Figure 8-31. Non-robust Detection Example

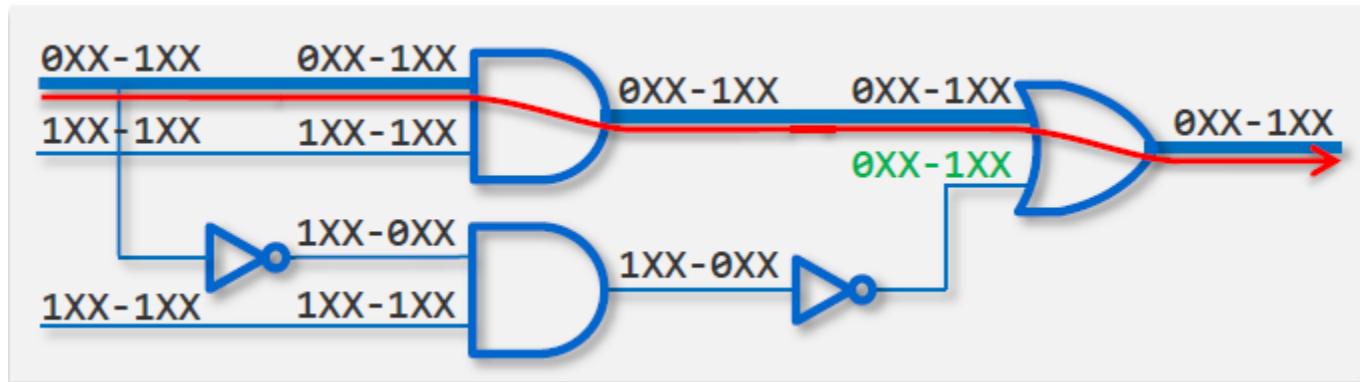


Functional detection further relaxes the requirements on the gating input used to sensitize the path. The gating of the path does not have to be stable as in robust detection, nor does it have to be sensitizing at the capture event, as required by non-robust detection. Functional detection

requires only that the gating inputs do not block propagation of a transition along the path. The tool places faults detected by functional detection in the `det_functional` (DF) fault class.

Figure 8-32 gives an example of functional detection for a rising-edge transition within a simple path. Notice that the gating (off-path) value on the gates is neither stable, nor sensitizing at the time of the capture event. However, the path input transition still propagates to the path output.

Figure 8-32. Functional Detection Example



You can use these commands as you create path delay detection test sets:

- [analyze_fault](#) — Analyzes a fault, including path delay faults, to determine why it was not detected.
- [delete_fault_sites](#) — Deletes paths from the internal path list.
- [read_fault_sites](#) — Loads in a file of path definitions from an external file.
- [report_fault_sites](#) — Reports information on paths in the path list.
- [report_statistics](#) — Displays simulation statistics, including the number of detected faults in each fault class.
- [set_fault_type](#) — Specifies the number of paths the tool should expand to when encountering an ambiguous path, by using the `path_delay` literal and `-expand_ambiguous_paths` switch.
- [set_pathdelay_holdpi](#) — Sets whether non-clock primary inputs can change after the first pattern force, during ATPG.
- [write_fault_sites](#) — Writes information on paths in the path list to an external file.

The Path Definition File

In an external ASCII file, you must use the path definition file to define all the paths that you want tested in the test set.

For each path, you must specify the following:

- **Path_name** — A unique name you define to identify the path.
- **Path_definition** — The topology of the path from launch to capture point as defined by an ordered list of pin pathnames. Each path must be unique.

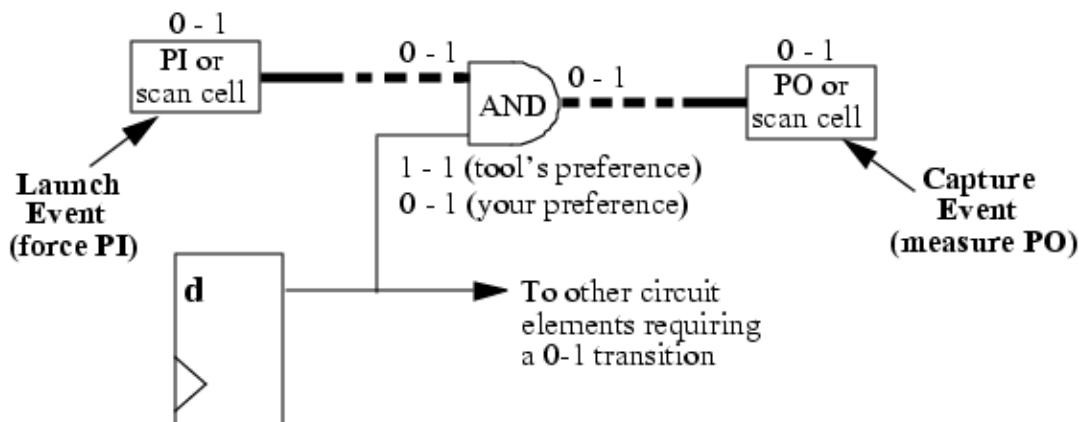
The ASCII path definition file has several syntax requirements. The tools ignore as a comment any line that begins with a double slash (//) or pound sign (#). Each statement must be on its own line. The four types of statements include:

- **Path** — A required statement that specifies the unique pathname of a path.
- **Condition** — An optional statement that specifies any conditions necessary for the launch and capture events. Each condition statement contains two arguments: a full pin pathname for either an internal or external pin, and a value for that pin. Valid pin values for condition statements are 0, 1, or Z. Condition statements must occur between the path statement and the first pin statement for the path.
- **Transition_condition** — An optional statement that specifies additional transitions required in the test pattern. Each transition_condition statement contains two arguments: a full pin pathname for either an internal or external pin and a direction. Transition_condition statements must occur between the path statement and the first pin statement for the path.

The direction can be one of the following: rising, falling, same, or opposite. Rising and falling specify that a rising edge and falling edge, respectively, are required on the specified pin at the same time as launching a transition into the first pin of the path. Same specifies for the tool to create a transition in the same direction as the one on the first pin in the path definition. Opposite creates a transition in the opposite direction.

[Figure 8-33](#) shows an example where a transition_condition statement could be advantageous.

Figure 8-33. Example Use of Transition_condition Statement



A defined path includes a 2-input AND gate with one input on the path, the other connected to the output of a scan cell. For a robust test, the AND gate's off-path or gating input needs a constant 1. The tool, in exercising its preference for a robust test, would try to create a pattern that achieved this. Suppose however that you wanted the circuit elements fed by the scan cell to receive a 0-1 transition. You could add a `transition_condition` statement to the path definition, specifying a rising transition for the scan cell. The path capture point maintains a 0-1 transition, so remains testable with a non-robust test, and you also get the desired transition for the other circuit elements.

- **Pin** — A required statement that identifies a pin in the path by its full pin pathname. Pin statements in a path must be ordered from launch point to capture point. A “+” or “-” after the pin pathname indicates the inversion of the pin with respect to the launch point. A “+” indicates no inversion (you want a transition identical to the launch transition on that pin), while a “-” indicates inversion (you want a transition opposite the launch transition).

Note

 If you use “+” or “-” in any pin statement, you must include a “+” for the launch point. The polarity of the launch transition must always be “+”.

You must specify a minimum of two pin statements, the first being a valid launch point (primary input or data output of a state element or RAM) and the last being a valid capture point (primary output, data or clk input of a state element, or data input of a RAM). The current pin must have a combinational connectivity path to the previous pin and the edge parity must be consistent with the path circuitry. If a statement violates either of these conditions, the tool issues an error. If the path has edge or path ambiguity, it issues a warning.

Paths can include state elements (through data or clock inputs), but you must explicitly name the data or clock pins in the path. If you do not, the tool does not recognize the path and issues a corresponding message.

- **End** — A required statement that signals the completion of data for the current path. Optionally, following the end statement, you can specify the name of the path. However, if the name does not match the pathname specified with the path statement, the tool issues an error.

The following shows the path definition syntax:

```
PATH <pathname> =
  Condition <pin.pathname> <0|1|Z>;
  TRANSition_condition <pin.pathname> <Rising|Falling|Same|Opposite>;
  PIN <pin.pathname> [+|-];
  PIN <pin.pathname> [+|-];
  ...
  PIN <pin.pathname> [+|-];
END [pathname];
```

The following is an example of a path definition file:

```

PATH "path0" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$951/I$1/IN - ;
    PIN /I$1/I$951/I$1/OUT + ;
    PIN /A_EQ_B + ;
END ;
PATH "path1" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$684/I1 - ;
    PIN /I$1/I$684/OUT - ;
    PIN /I$5/D - ;
END ;
PATH "path2" =
    PIN /I$5/Q + ;
    PIN /I$35/B1 + ;
    PIN /I$35/C1 + ;
    PIN /I$1/I$649/IN + ;
    PIN /I$1/I$649/OUT - ;
    PIN /I$1/I$622/I2 - ;
    PIN /I$1/I$622/OUT - ;
    PIN /A_EQ_B + ;
END ;
PATH "path3" =
    PIN /I$5/QB + ;
    PIN /I$6/TI + ;
END ;

```

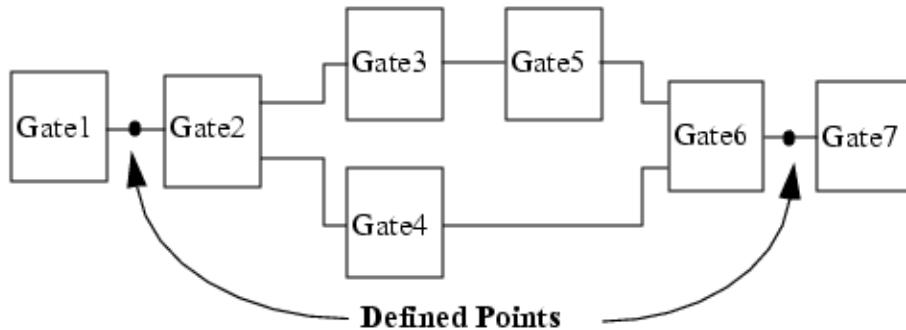
You use the [read_fault_sites](#) command to read in the path definition file. The tool loads the paths from this file into an internal path list. You can add to this list by adding paths to a new file and re-issuing the `read_fault_sites` command with the new filename.

Path Definition Checks

The ATPG tool checks the points along the defined path for proper connectivity and to determine if the path is ambiguous. Path ambiguity indicates there are several different paths from one defined point to the next.

[Figure 8-34](#) indicates a path definition that creates ambiguity.

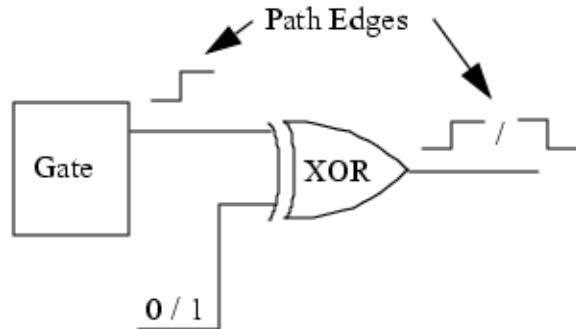
Figure 8-34. Example of Ambiguous Path Definition



In this example, the defined points are an input of Gate2 and an input of Gate7. Two paths exist between these points, thus creating path ambiguity. When the ATPG tool encounters this situation, it expands the ambiguous path to up to 10 unambiguous paths by default. If you want the tool to consider a different number of paths, you can specify this with the “[set_fault_type path_delay -expand_ambiguous_paths](#)” command.

During path checking, the tool can also encounter *edge ambiguity*. Edge ambiguity occurs when a gate along the path has the ability to either keep or invert the path edge, depending on the value of another input of the gate. [Figure 8-35](#) shows a path with edge ambiguity due to the XOR gate in the path.

Figure 8-35. Example of Ambiguous Path Edges



The XOR gate in this path can act as an inverter or buffer of the input path edge, depending on the value at its other input. Thus, the edge at the output of the XOR is ambiguous. The path definition file lets you indicate edge relationships of the defined points in the path. You do this by specifying a “+” or “-” for each defined point, as was previously described in “[The Path Definition File](#)” on page 352.

The “[set_fault_type path_delay -expand_ambiguous_paths](#)” command can also expand paths with edge ambiguity.

Generating a Path Delay Test Set

When you have a path definition file with the paths you want to test, you use this procedure to generate a path delay test set.

Procedure

1. Perform circuit setup tasks as described in the [ATPG Basic Tool Flow](#) and [ATPG Setup](#) sections in this manual.

2. Constrain the scan enable pin to its inactive state. For example:

```
SETUP> add_input_constraints scan_en -c0
```

3. (Optional) Turn on output masking.

```
SETUP> set_output_masks on
```

4. Addnofaults <x, y, z>

5. Set the sequential depth to two or greater:

```
SETUP> set_pattern_type -sequential 2
```

6. Enter analysis system mode. This triggers the tool's automatic design flattening and rules checking processes.

7. Set the fault type to path delay:

```
ANALYSIS> set_fault_type path_delay
```

8. Write a path definition file with all the paths you want to test. “[The Path Definition File](#)” on page 352 describes this file in detail. If you want, you can do this prior to the session. You can only add faults based on the paths defined in this file.

9. Load the path definition file (assumed for the purpose of illustration to be named *path_file_1*):

```
ANALYSIS> read_fault_sites path_file_1
```

10. Specify any changes you want in how the tool expands ambiguous paths. By default, the tool expands each ambiguous path internally with up to 10 unambiguous paths. The following example limits the number of expanded unambiguous paths to a maximum of 4.

```
ANALYSIS> set_fault_type path_delay -expand_ambiguous_paths 4
```

11. Define faults for the paths in the tool's internal path list:

```
ANALYSIS> add_faults -all
```

This adds a rising edge and falling edge fault to the tool's path delay fault list for each defined path.

12. Perform an analysis on the specified paths and delete those the analysis proves are unsensitizable:

ANALYSIS> delete_fault_sites -unsensitizable_paths

13. Run test generation:

ANALYSIS> create_patterns

Path Delay Testing Limitations

Path delay testing does not support several types of circuit configurations.

- **RAMs Within a Specified Path** — A RAM as a launch point is supported only if the launch point is at the RAM’s output. A RAM as a capture point is supported only if the capture point is at the RAM’s input.
- **Paths Starting at a Combinationally Transparent Latch** — A combinational transparent latch as a capture point is supported only if the capture point is at the latch’s input.
- **Path Starting and/or Ending at ROM** — You should model ROM as a read-only CRAM primitive (that is, without any _write operation) to enable the tool to support path delay testing starting and/or ending at ROM.

At-Speed Test With Named Capture Procedures

To create at-speed test patterns for designs with complicated clocking schemes, you may need to specify the actual launch and capture clocking sequences. For example, in an LSSD type design with master and slave clocks, the number and order of clock pulses might need to be organized in a specific way. You can do this using a named capture procedure in the test procedure file.

A named capture procedure is an optional procedure, with a unique name, used to define explicit clock cycles. Named capture procedures can be used for generating stuck-at, path delay, and broadside transition patterns, but not launch off shift transition patterns. You can create named capture procedures using the [create_capture_procedures](#) command and then write out the procedures using the [write_procfile](#) command. You can also manually create or edit named capture procedures using an external editor if needed. For information on manually creating and editing named capture procedures, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User’s Manual*.

When the test procedure file contains named capture procedures, the ATPG tool generates patterns that conform to the waveforms described by those procedures. Alternatively, you can use the [set_capture_procedures](#) command to disable a subset of the named capture procedures, and only the enabled subset is used. For example, you might want to exclude named capture procedures that are unable to detect certain types of faults during test pattern generation.

You can have multiple named capture procedures within one test procedure file in addition to the default capture procedure the file typically contains. Each named capture procedure must reflect clock behavior that the clocking circuitry is actually capable of producing. When you use a named capture procedure to define a waveform, it is assumed you have expert design knowledge; the ATPG tool does not verify that the clocking circuitry is capable of delivering the waveform to the defined internal pins.

The ATPG tool uses either all named capture procedures (the default) or only those named capture procedures you enable with the [set_capture_procedures](#) command. When the test procedure file does not contain named capture procedures, or you use the “[set_capture_procedures off -all](#)” command, the tool uses the default capture procedure. However, usually you would not use the default procedure to generate at-speed tests. The tool does not currently support use of both named capture procedures and clock procedures in a single ATPG session.

Note

 If a DRC error prevents the use of a capture procedure, the run will abort.

For more information on named capture procedures, see the “[Capture Procedures Optional](#)” section in the *Tessent Shell User’s Manual*.

Support for On-Chip Clocks (PLLs) **360**

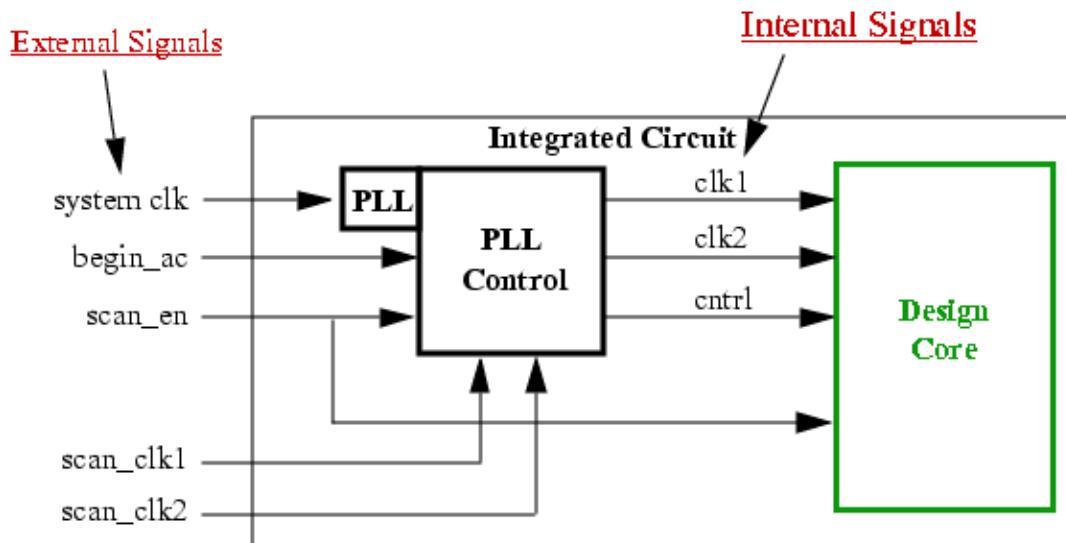
Internal and External Modes Definition	360
Named Capture Procedures Display	363
Achieving the Test Coverage Goal Using Named Capture Procedures	364
Debugging Low Test Coverage Using Pre-Defined Named Capture Procedures	365
At-speed Fault Simulation Clocking Considerations	365
Internal Signals and Clocks	365
How to Save Internal and External Patterns	366

Support for On-Chip Clocks (PLLs)

You can use named capture procedures for the support of on-chip or internal clocks.

These are clocks generated on-chip by a phased-locked loop (PLL) or other clock generating circuitry as shown in [Figure 8-36](#). In addition, an example timing diagram for this circuit is shown in [Figure 8-37](#). A PLL can support only certain clock waveforms and named capture procedures let you specify the allowed set of clock waveforms. In this case, if there are multiple named capture procedures, the ATPG engine will use these named capture procedures instead of assuming the default capture behavior.

Figure 8-36. On-chip Clock Generation

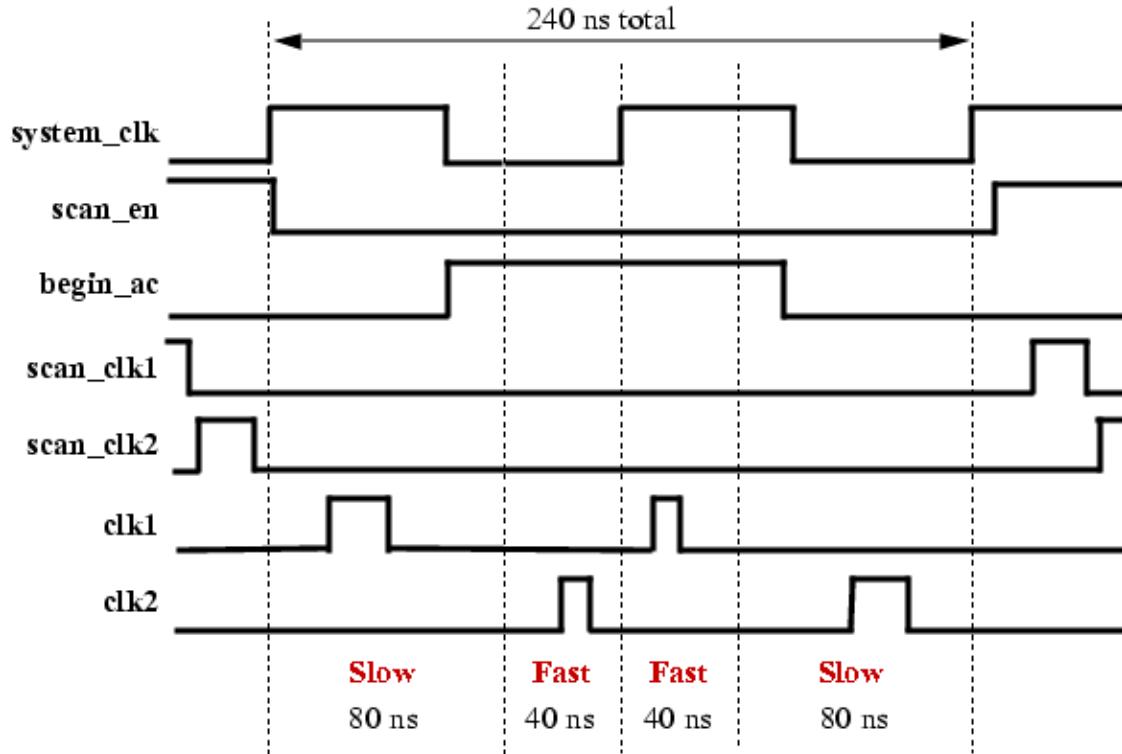


Internal and External Modes Definition

When manually creating or editing named capture procedures, you can use the optional keyword “mode” with two mode blocks, “internal” and “external” to describe what happens on the internal and external sides of an on-chip phase-locked loop (PLL) or other on-chip clock-generating circuitry. You use “mode internal =” and “mode external =” to define mode blocks in

which you put procedures to exercise internal and external signals. You must use the internal and external modes together, and ensure no cycles are defined outside the mode definitions.

Figure 8-37. PLL-Generated Clock and Control Signals



The internal mode is used to describe what happens on the internal side of the on-chip PLL control logic, while the external mode is used to describe what happens on the external side of the on-chip PLL. Figure 8-36 shows how this might look. The internal mode uses the internal clocks (/pll/clk1 and /pll/clk2) and signals while the external mode uses the external clocks (system_clk) and signals (begin_ac and scan_en). If any external clocks or signals go to both the

PLL and to other internal chip circuitry (scan_en), you need to specify their behavior in both modes and they need to match, as shown in the following example (timing is from [Figure 8-37](#)):

```
timeplate tp_cap_clk_slow =
    force_pi 0;
    pulse /pll/clk1 20 20;
    pulse /pll/clk2 40 20;
    period 80;
end;
timeplate tp_cap_clk_fast =
    force_pi 0;
    pulse /pll/clk1 10 10;
    pulse /pll/clk2 20 10;
    period 40;
end;

timeplate tp_ext =
    force_pi 0;
    measure_po 10;
    force begin_ac 60;
    pulse system_clk 0 60;
    period 120;
end;

procedure capture clk1 =
    observe_method master;

    mode internal =
        cycle slow =
            timeplate tp_cap_clk_slow;
                force system_clk 0;
                force scan_clk1 0;
                force scan_clk2 0;
                force scan_en 0;
                force_pi;
                force /pll/clk1 0;
                force /pll/clk2 0;
                pulse /pll/clk1;
            end;
        // launch cycle
        cycle =
            timeplate tp_cap_clk_fast;
                pulse /pll/clk2;
            end;
        // capture cycle
        cycle =
            timeplate tp_cap_clk_fast;
                pulse /pll/clk1;
            end;
        cycle slow =
            timeplate tp_cap_clk_slow;
                pulse /pll/clk2;
            end;
    end;

    mode external =
        timeplate tp_ext;
```

```
cycle =
    force system_clk 0;
    force scan_clk1 0;
    force scan_clk2 0;
    force scan_en 0;
    force_pi;
    force begin_ac 1;
    pulse system_clk;
end;
cycle =
    force begin_ac 0;
    pulse system_clk;
end;
end;
end;
```

For more information about internal and external modes, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User’s Manual*.

Named Capture Procedures Display

When the ATPG tool uses a named capture procedure, it uses a “cyclized” translation of the internal mode. The tool may merge certain internal mode cycles in order to optimize them, and it may expand others to ensure correct simulation results. These modifications are internal only; the tool does not alter the named capture procedure in the test procedure file.

You can use the [report_capture_procedures](#) command to display the cyclized procedure information with annotations that indicate the timing of the cycles and where the at-speed sequences begin and end. If you want to view the procedures in their unaltered form in the test procedure file, use the [report_procedures](#) command.

After cyclizing the internal mode information, the tool automatically adjusts the sequential depth to match the number of cycles that resulted from the cyclizing process. Patterns will automatically reflect any sequential depth adjustment the tool performs.

[Figure 8-38](#) illustrates cycle merging.

Figure 8-38. Cycles Merged for ATPG

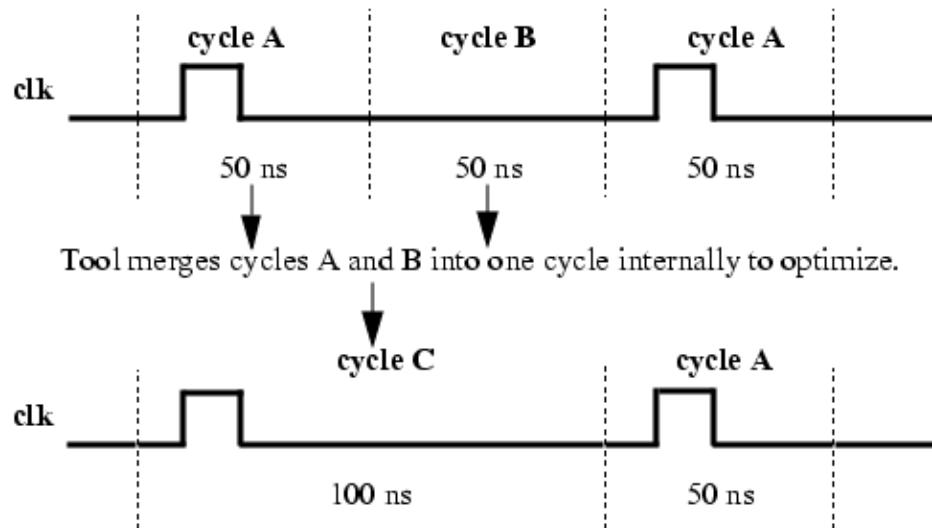
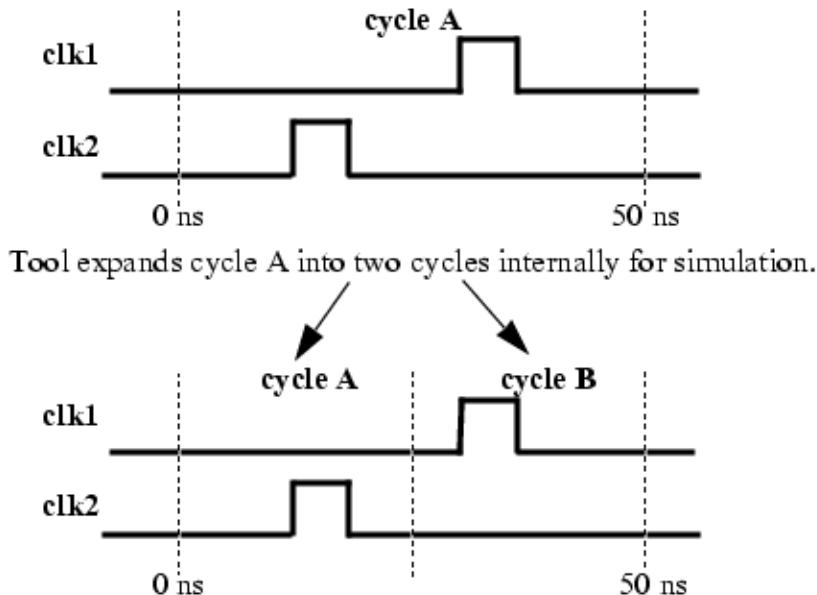


Figure 8-39 illustrates cycle expansion.

Figure 8-39. Cycles Expanded for ATPG



Achieving the Test Coverage Goal Using Named Capture Procedures

Use this procedure when you know the repeated clock sequences to which the test clocks will be applied and want to investigate the best sequential depth of each clock domain to achieve the test coverage goal.

Procedure

1. Use the [create_capture_procedures](#) command to define the minimum clock sequences in a named capture procedure.
2. Gradually add more capture procedures with higher sequential depth until the test coverage goal is achieved or the pattern count limit is reached.

Debugging Low Test Coverage Using Pre-Defined Named Capture Procedures

Use this procedure when you want to know which clock sequences will provide the best test coverage and then create a named capture procedure from it.

Procedure

1. Start ATPG from the default capture procedure.
2. Use the [create_capture_procedures](#) command to create named capture procedures by extracting the most often applied clock sequences from the pattern set.
3. Use the [write_procedure](#) command to write the named capture procedure into the test procedure file.

Note

 You may need to manually edit the named capture procedure in the test procedure file to achieve the functionality you want. For example, you may need to add condition statements or add external mode definitions. For information on rules to follow when editing named capture procedures, see the “[Rules for Creating and Editing Named Capture Procedures](#)” section in the *Tessent Shell User’s Manual*.

At-speed Fault Simulation Clocking Considerations

Not all clocks specified in the capture procedures are applied at-speed. During at-speed fault simulation, the tool does not activate at-speed related faults when slow clock sequences are fault simulated, even if a transition occurs in two consecutive cycles. Generally, the clock sequence defined in a capture procedure can consist of zero or more slow clock sequences, followed by zero or more at-speed clock sequences, followed by zero or more slow clock sequences.

Related Topics

[Delay Test Set Creation](#)

Internal Signals and Clocks

The tool provides a way for you to specify internal signals and clocks.

For clocks and signals that come out of the PLL or clock generating circuitry that are not available at the real I/O interface of the design, you can use the [add_clocks](#) or [add_primary_inputs](#) commands to define the internal signals and clocks for use in ATPG. If the pin you specify with the command is an internal pin, the tool automatically creates an internal PI for it.

For example, when setting up for pattern generation for the example circuit shown in [Figure 8-36](#), you would issue this command to define the internal clocks:

```
SETUP> add_clocks 0 /pll/clk1 /pll/clk2
```

The two PLL clocks would then be available to the tool's ATPG engine for pattern generation.

For those PIs created from internal pins by the [add_clocks](#) and [add_primary_inputs](#) commands, fault sites in the driving logic of the internal pins will be classified as AU (for example, AU.SEQ or AU.PC) or DI (for example, DI.CLK), as appropriate.

How to Save Internal and External Patterns

By default, the ATPG tool uses only the primary input clocks when creating test patterns.

However, if you use named capture procedures with internal mode clocks and control signals you define with the [add_clocks](#) or [add_primary_inputs](#) commands, the tool uses those internal clocks and signals for pattern generation and simulation. To save the patterns using the same internal clocks and signals, you must use the [-Mode_internal](#) switch with the [write_patterns](#) command. The [-Mode_internal](#) switch is the default when saving patterns in ASCII or binary format.

Note

 The [-Mode_internal](#) switch is also necessary if you want patterns to include internal pin events specified in scan procedures (test_setup, shift, load_unload).

To obtain pattern sets that can run on a tester, you need to write patterns that contain only the true primary inputs to the chip. These are the clocks and signals used in the external mode of any named capture procedures, not the internal mode. To accomplish this, you must use the [-Mode_external](#) switch with the [write_patterns](#) command. This switch directs the tool to map the information contained in the internal mode blocks back to the external signals and clocks that comprise the I/O of the chip. The [-Mode_external](#) switch is the default when saving patterns in a tester format (for example, WGL) and Verilog format.

Note

 The [-Mode_external](#) switch ignores internal pin events in scan procedures (test_setup, shift, load_unload).

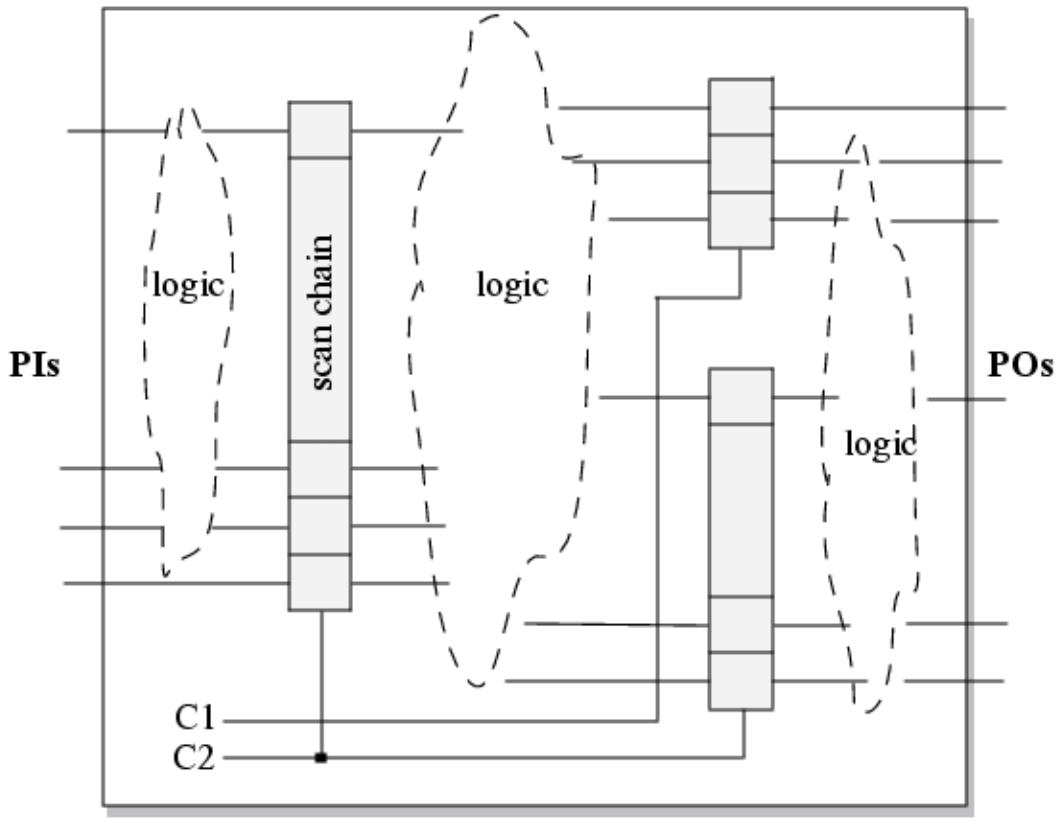
Mux-DFF Example

In a full scan design, the vast majority of transition faults are between scan cells (or cell to cell) in the design. There are also some faults between the PI to cells and cells to the PO. Targeting these latter faults can be more complicated, mostly because running these test patterns on the tester can be challenging. For example, the tester performance or timing resolution at regular I/O pins may not be as good as that for clock pins. This section shows a mux-DFF type scan design example and covers some of the issues regarding creating transition patterns for the faults in these three areas.

[Figure 8-40](#) shows a conceptual model of an example chip design. There are two clocks in this mux-DFF design, which increases the possible number of launch and capture combinations in creating transition patterns. For example, depending on how the design is actually put together, there might be faults that require these launch and capture combinations: C1-C1, C2-C2, C1-C2, and C2-C1. The clocks may be either external or are created by some on-chip clock generator circuitry or PLL.

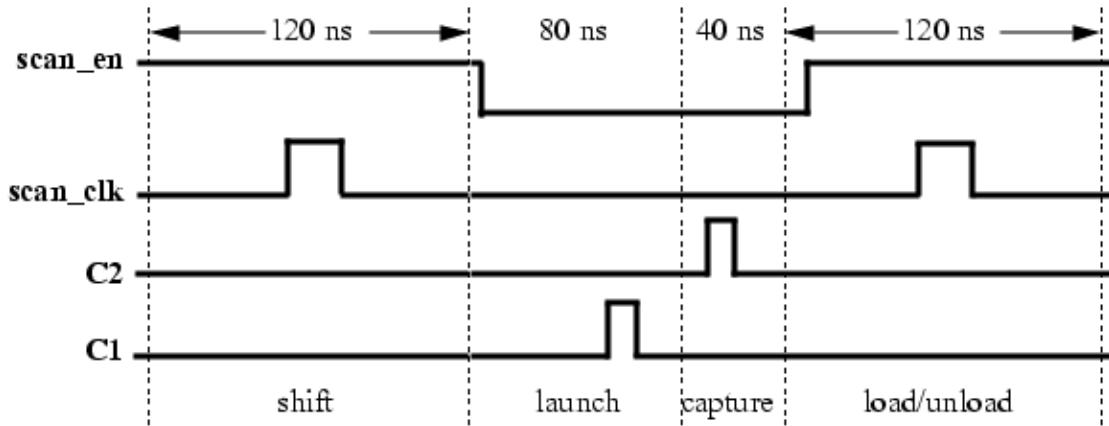
[“Timing for Transition Delay Tests” on page 331](#) shows the basic waveforms and partial test procedure files for creating broadside and launch off shift transition patterns. For this example, named capture procedures are used to specify the timing and sequence of events. The example focuses on broadside patterns and shows only some of the possible named capture procedures that might be used in this kind of design.

Figure 8-40. Mux-DFF Example Design



A timing diagram for cell to cell broadside transition faults that are launched by clock C1 and captured by clock C2 is shown in [Figure 8-41](#).

Figure 8-41. Mux-DFF Broadside Timing, Cell to Cell



Following is the capture procedure for a matching test procedure file that uses a named capture procedure to accomplish the clocking sequence. Other clocking combinations would be handled with additional named capture procedures that pulse the clocks in the correct sequences.

```
set time scale 1.000000 ns ;
timeplate tp1 =
    force_pi 0;
    measure_po 10;
    pulse scan_clk 50 20;
    period 120;
end;
timeplate tp2 =
    force_pi 0;
    pulse c1 10 10;
    pulse c2 10 10;
    measure_po 30;
    period 40;
end;
timeplate tp3 =
    force_pi 0;
    pulse c1 50 10;
    pulse c2 10 10;
    period 80;
end;
procedure load_unload =
    timeplate tp1;
    cycle =
        force c1 0;
        force c2 0;
        force scan_en 1;
    end;
    apply shift 255;
end;
procedure shift =
    timeplate tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse scan_clk;
    end;
end;
procedure capture launch_c1_cap_c2 =
    cycle =
        timeplate tp3;
        force c1 0;
        force c2 0;
        force scan_clk 0;
        force_pi; //force scan_en to 0
        pulse c1; //launch clock
    end;
    cycle =
        timeplate tp2;
        pulse c2; //capture clock
    end;
end;
```

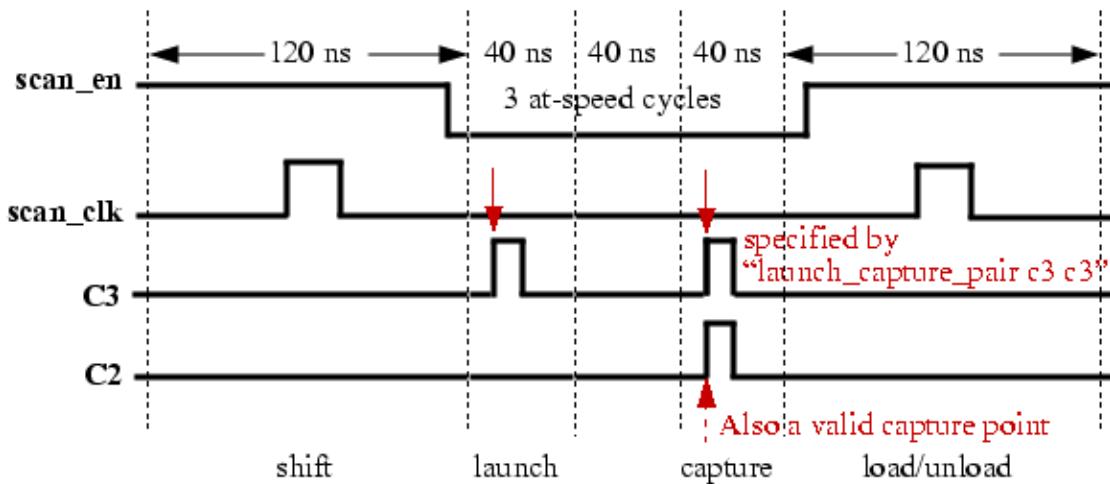
Be aware that this is just one example and your implementation may vary depending on your design and tester. For example, if your design can turn off scan_en quickly and have it settle before the launch clock is pulsed, you may be able to shorten the launch cycle to use a shorter

period; that is, the first cycle in the launch_c1_cap_c2 capture procedure could be switched from using timeplate tp3 to using timeplate tp2.

Another way to make sure scan enable is turned off well before the launch clock is to add a cycle to the load_unload procedure right after the “apply shift” line. This cycle would only need to include the statement, “force scan_en 0;”.

Notice that the launch and capture clocks shown in [Figure 8-41](#) pulse in adjacent cycles. The tool can also use clocks that pulse in non-adjacent cycles, as shown in [Figure 8-42](#), if the intervening cycles are at-speed cycles.

Figure 8-42. Broadside Timing, Clock Pulses in Non-adjacent Cycles



To define a pair of nonadjacent clocks for the tool to use as the launch clock and capture clock, include a “`launch_capture_pair`” statement at the beginning of the named capture procedure. Multiple “`launch_capture_pair`” statements are permitted, but the tool will use just one of the statements for a given fault. Without this statement, the tool defaults to using adjacent clocks.

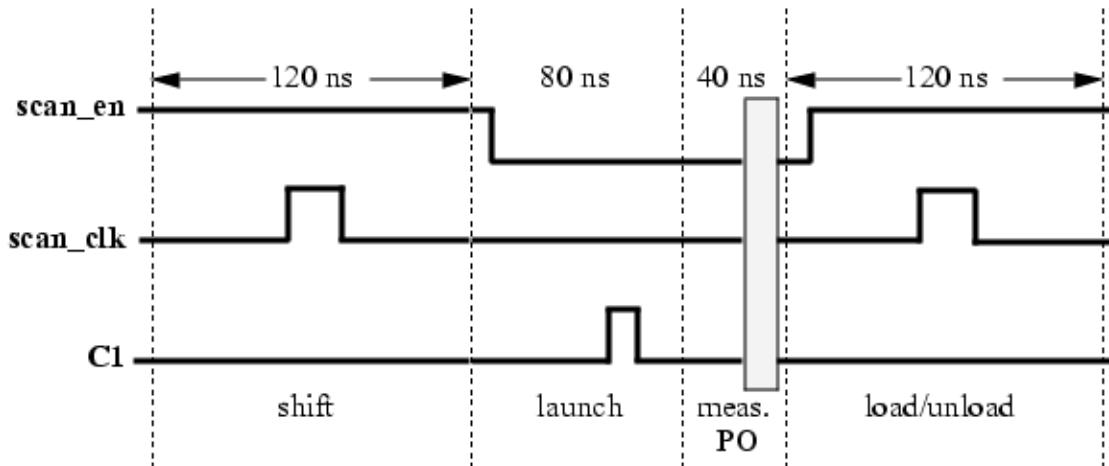
When its choice of a launch and capture clock is guided by a `launch_capture_pair` statement, the tool may use for launch, the clock specified as the launch clock in the statement or another clock that is pulsed between the launch and capture clocks specified in the statement. The capture clock, however, will be the one specified in the statement or another clock that has the *same period* as the specified capture clock.

If a named capture procedure for example pulses clocks `clk1`, `clk2` and `clk3` in that order in each of three successive at-speed cycles and the `launch_capture_pair {clk1, clk3}` is defined, the tool could use *either* `clk1` or `clk2` to launch and `clk3` to capture. The idea of the launch and capture pair is that it allows you to specify the capture clock and the farthest launch clock from the capture clock. In this example, the `{clk1, clk3}` pair directs the tool to use `clk3` to capture and the farthest launch clock to be `clk1`. The tool considers it all right for `clk2` to launch since if `{clk1, clk3}` is at speed, `{clk2, clk3}` should be at speed as well.

For more information on using the “`launch_capture_pair`” statement, see the “[“`launch_capture_pair` Statement](#)” section in the *Tessent Shell User’s Manual*.

If you want to try to create transition patterns for faults between the scan cells and the primary outputs, make sure your tester can accurately measure the PO pins with adequate resolution. In this scenario, the timing looks similar to that shown in [Figure 8-41](#) except that there is no capture clock. [Figure 8-43](#) shows the timing diagram for these cell to PO patterns.

Figure 8-43. Mux-DFF Cell to PO Timing



Following is the additional capture procedure that is required:

```
procedure capture launch_c1_meas_PO=
    cycle =
    timeplate tp3;
        force_pi; //force scan_en to 0
        pulse c1; //launch clock
    end;
    cycle =
    timeplate tp2;
        measure_po; //measure PO values
    end;
end;
```

Note

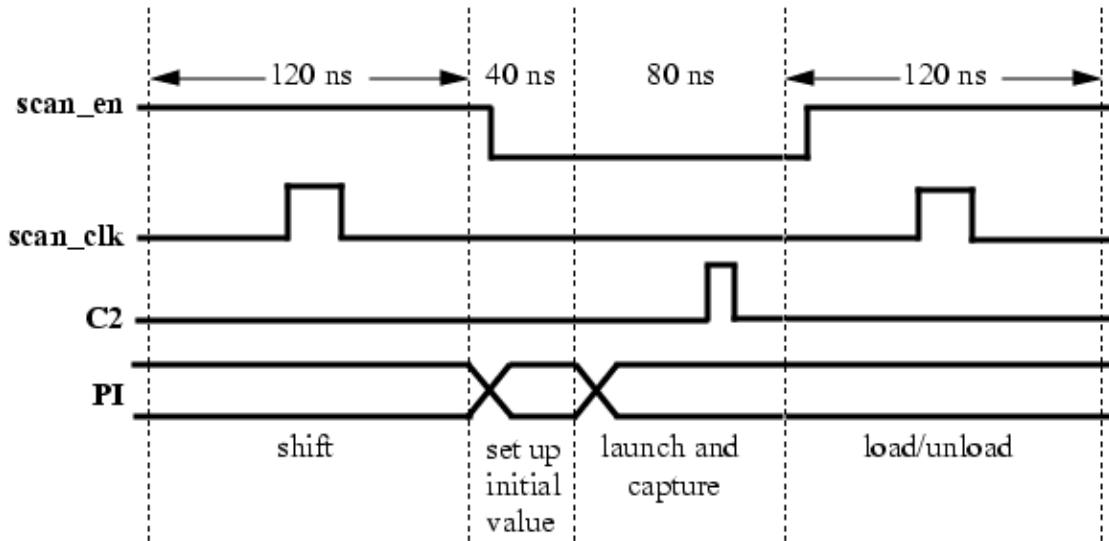
 You will need a separate named capture procedure for each clock in the design that can cause a launch event.

What you specify in named capture procedures is what you get. As you can see in the two preceding named capture procedures (`launch_c1_cap_c2` and `launch_c1_meas_PO`), both procedures used two cycles, with timeplate tp3 followed by timeplate tp2. The difference is that in the first case (cell to cell), the second cycle only performed a pulse of C2 while in the second case (cell to PO), the second cycle performed a `measure_po`. The key point to remember is that

even though both cycles used the same timeplate, they only used a subset of what was specified in the timeplate.

To create effective transition patterns for faults between the PI and scan cells, you also may have restrictions due to tester performance and tolerance. One way to create these patterns can be found in the example timing diagram in [Figure 8-44](#). The corresponding named capture procedure is shown after the figure.

Figure 8-44. Mux-DFF PI to Cell Timing



```
procedure capture launch_PI_cap_C2 =
    cycle =
        timeplate tp2;
            force_pi; //force initial values
        end;
    cycle =
        timeplate tp3;
            force_pi; //force updated values
            pulse c2; //capture clock
        end;
end;
```

As before, you would need other named capture procedures for capturing with other clocks in the design. This example shows the very basic PI to cell situation where you first set up the initial PI values with a force, then in the next cycle force changed values on the PI and quickly capture them into the scan cells with a capture clock.

Note

 You do not need to perform at-speed testing for all possible faults in the design. You can eliminate testing things like the boundary scan logic, the memory BIST, and the scan shift path by using the [add_nofaults](#) command.

Support for Internal Clock Control

You can use clock control definitions in your test procedure file to specify the operation of on-chip/internal clocks during capture cycles. A clock control definition is one or more blocks in the test procedure file that define internal clock operation by specifying source clocks and conditions at scan cell outputs when a clock can be pulsed. ATPG interprets these definitions and determines which clock to pulse during the capture cycle to detect the most faults.

When a clock control is defined, the clock control bits are included in chain test pattern to turn off capture clocks when the clock under control has no source clock or when the source clock defined in the clock under control is a always-pulse clock.

By default in the ATPG tool, one capture cycle is included in chain test patterns if there is no pulse-always clock, or if the pulse-always clock drives neither observation points nor buses.

You can manually create clock control definitions or use the stil2mgc tool to generate them automatically from a STIL Procedure File (SPF) that contains a ClockStructures block.

To specify explicit launch/capture sequences or external/internal clock relationship definitions, you must use Named Capture Procedures (NCPs).

Note that you can turn off clock control using the “set_clock_controls off” command. You should turn off clock controls only for debug purposes, typically to determine if a fault is untestable because of the clock control constraints and if the fault is testable when those constraints are not present. For more information, refer to the [set_clock_controls](#) command description in the *Tessent Shell Reference Manual*.

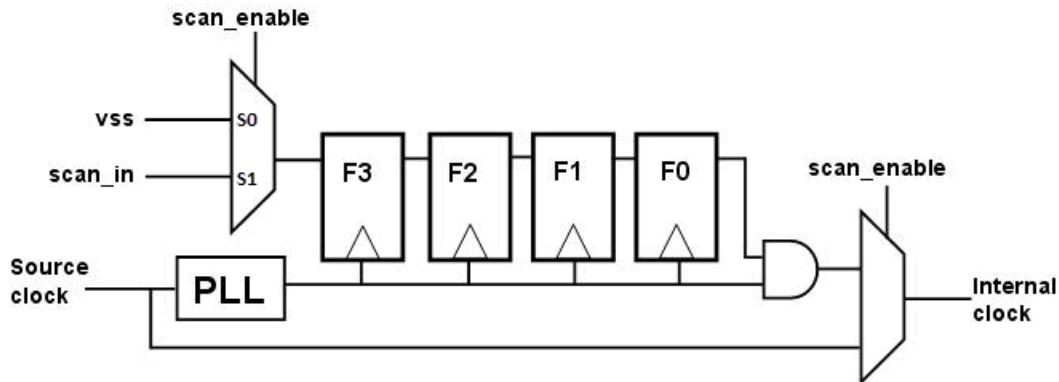
Per-Cycle Clock Control	373
Sequence Clock Control	376
Capture Cycle Determination.....	377
Applying Internal Clock Control.....	378

Per-Cycle Clock Control

Per-cycle clock control allows you to define the internal clock output based on a single capture cycle using scan cell values. Per-cycle clock control is commonly used for pipeline-based clock generation where one bit controls the clock in each cycle.

[Figure 8-45](#) shows a simplified per-cycle clock control model.

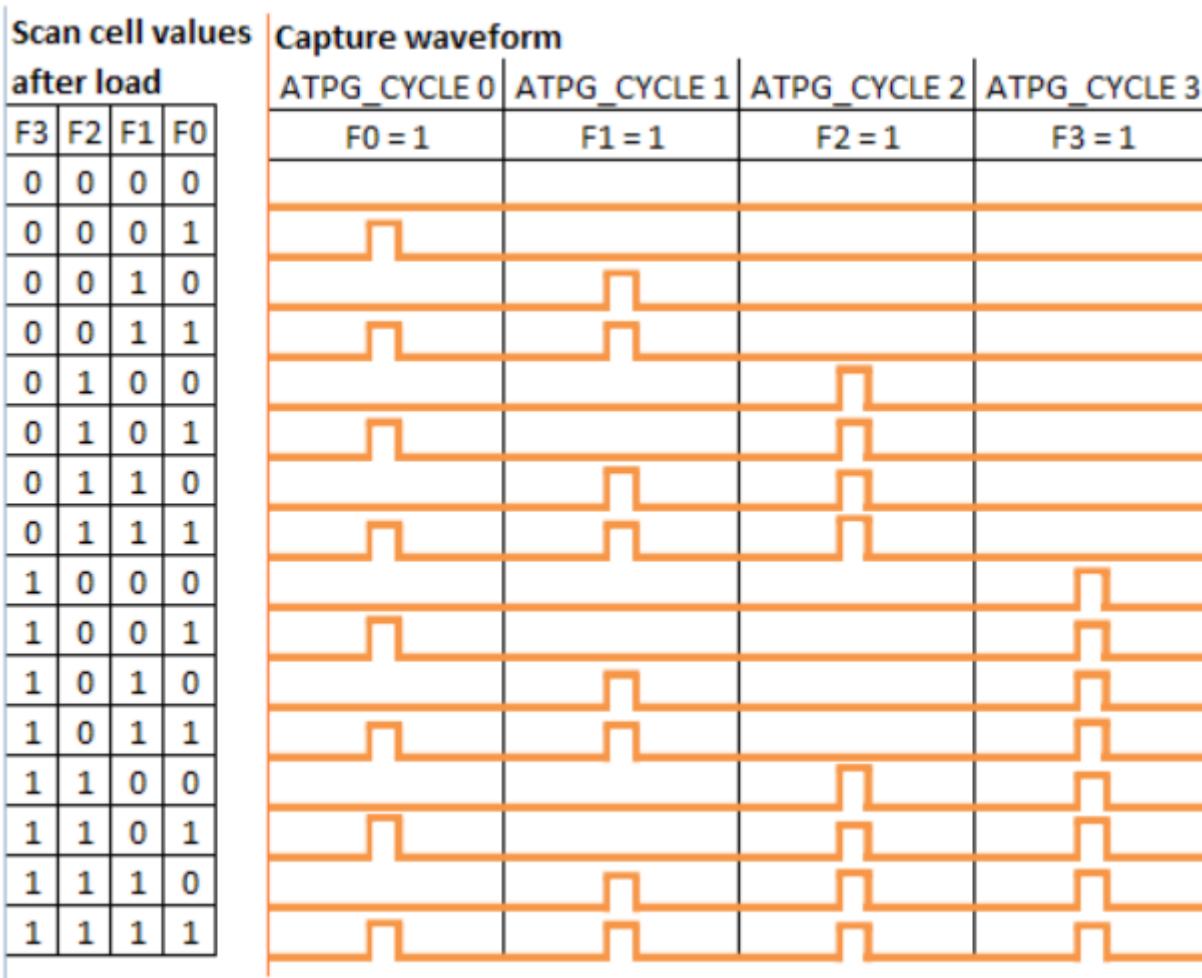
Figure 8-45. Simplified Per-Cycle Clock Control Model



The values on the shift register (F3-F2-F1-F0) determine if the AND gate passes the clock through on each cycle (4 cycles in this circuit). All of the flops in this circuit are scan flops. At the end of scan chain shifting, the scan cells are initialized to a value. The value determines the number of clocks that the circuitry can produce.

The hardware, in this circuit, is defined so that based on the values of the four scan cells the circuitry can provide up to four capture cycles. The combinations are shown in [Figure 8-46](#)

Figure 8-46. Per-Cycle Clock Control Waveform



Using “ATPG_CYCLE” you can represent the above circuit and waveforms as follows:

```
// Define clock control for internal clock "/clk_ctrl/int_clk1"
CLOCK_CONTROL /clk_ctrl/int_clk1 =
    // Define a source clock if one exists
    SOURCE_CLOCK ref_clk;

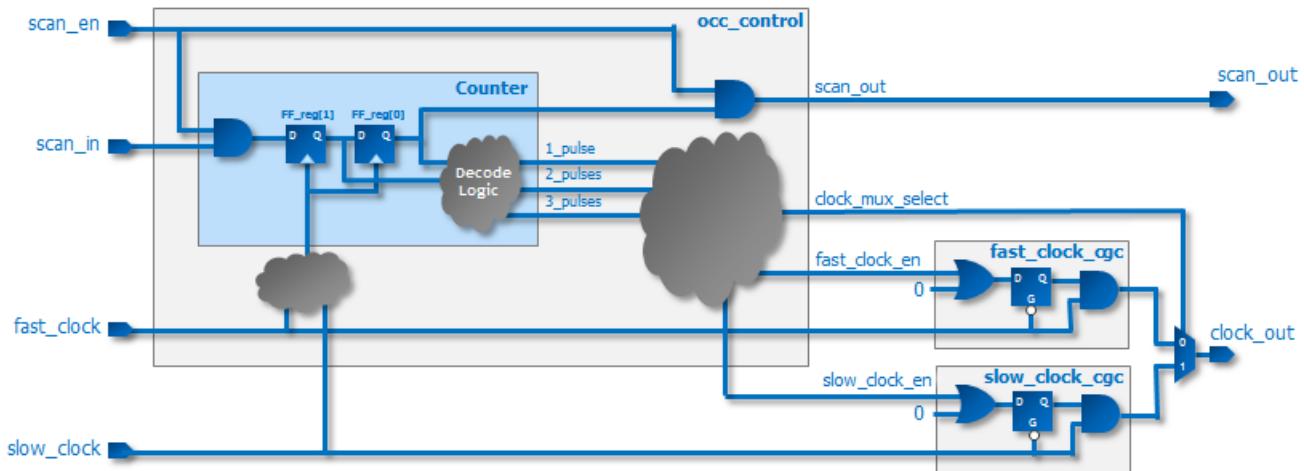
    // Define specific local conditions
    ATPG_CYCLE 0 =
        CONDITION /clk_ctrl/F0/q 1;
    END;
    ATPG_CYCLE 1 =
        CONDITION /clk_ctrl/F1/q 1;
    END;
    ATPG_CYCLE 2 =
        CONDITION /clk_ctrl/F2/q 1;
    END;
    ATPG_CYCLE 3 =
        CONDITION /clk_ctrl/F3/q 1;
    END
END;
```

Sequence Clock Control

Sequence clock control allows you to define the internal clock output based on a sequence of capture cycles using scan cell values. Sequence clock control is commonly used for counter-based clock generation circuitry.

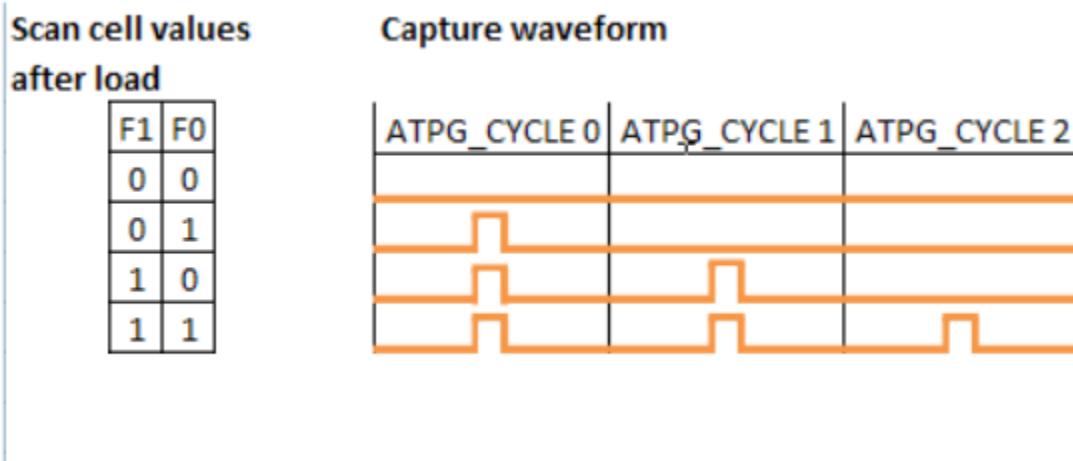
Figure 8-47 shows a simplified counter-based clock control example.

Figure 8-47. Simplified Counter-based Clock Control Example



The combinations are shown in Figure 8-48

Figure 8-48. Counter-based Clock Control Capture Waveform



In this case, the values loaded into the OCC flops determine the number of consecutive clock pulses. Three different sequences can be generated: one clock pulse, two clock pulses, and three clock pulses.

This can be described with this ATPG_SEQUENCE statement:

```
// Define clock control for internal clock "/clk_ctrl/int_clk1"
CLOCK_CONTROL /clk_ctrl/int_clk1 =
    // Define a source clock if one exists
    SOURCE_CLOCK ref_clk;
    // Define specific local conditions
    ATPG_SEQUENCE 0 0 = // '0 0' is a 1 clock pulse sequence
        CONDITION /clk_ctrl/FF_reg[1]/q 0;
        CONDITION /clk_ctrl/FF_reg[0]/q 1;
    END;
    ATPG_SEQUENCE 0 1 = // '0 1' is a 2 clock pulse sequence
        CONDITION /clk_ctrl/FF_reg[1]/q 1;
        CONDITION /clk_ctrl/FF_reg[0]/q 0;
    END;
    ATPG_SEQUENCE 0 2 = // '0 2' is a 3 clock pulse sequence
        CONDITION /clk_ctrl/FF_reg[1]/q 1;
        CONDITION /clk_ctrl/FF_reg[0]/q 1;
    END
END;
```

Capture Cycle Determination

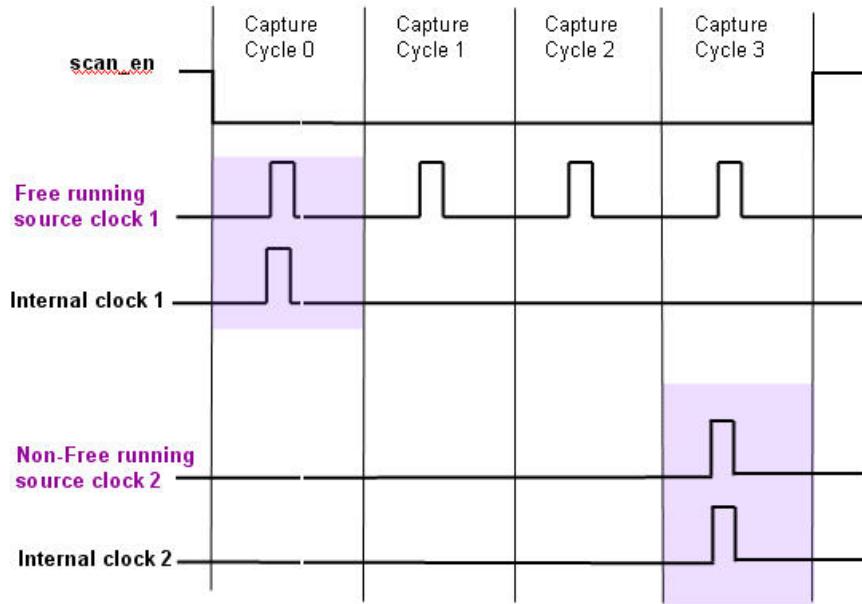
The actual capture cycle is relative to how many times the source clock pulses between scan loads as follows.

- **If the Source Clock is Pulse-Always** — The specified capture cycle and the actual capture cycle are always the same because the source clock pulses in every ATPG cycle as shown in [Figure 8-49](#). No source clock or a pulse-in-capture clock are considered equivalent to a pulse-always source clock.

- **If the Source Clock is Defined as Anything but Pulse-Always or Pulse-in-Capture** — The specified capture cycle is determined by the pulse of the source clock. For example, an internal clock defined to pulse for cycle 0 may not pulse in cycle 0 but in the cycle that corresponds to the source clock pulse as shown in [Figure 8-49](#).

Figure 8-49. Clock Control Capture Cycles

ATPG_CYCLE = 0 in the clock control definition



Applying Internal Clock Control

Use this procedure to apply internal clock control for ATPG via the test procedure file.

Restrictions and Limitations

Clock control definitions and Named Capture Procedures (NCPs) cannot both be enabled during test pattern generation. By default, clock control definitions are disabled when NCPs are enabled.

Note

 Test patterns created with NCPs and test patterns created with the clock control definitions can be fault simulated simultaneously.

Procedure

1. Depending on your application, either:
 - Create clock control definitions in your test procedure file.
OR

- Run the stil2mgc tool on an SPF to generate a test procedure file with clock control definitions. Refer to the [stil2mgc](#) description in the *Tessent Shell Reference Manual*.

For more information, see “[Clock Control Definition](#)” in the Tessent Shell User’s Manual.

2. Invoke Tessent Shell, set the context to “patterns -scan,” read in the netlist, and set up the appropriate parameters for the ATPG run.
3. Load the test procedure file created in step 1 and set up for ATPG. For example:

```
SETUP> add_scan_groups group1 scan_g1.procfile
SETUP> add_scan_chains chain1 group1 indata2 testout2
SETUP> add_scan_chains chain2 group1 indata4 testout4
SETUP> add_clocks0 clk1
```

The [add_scan_groups](#) command loads the specified test procedure file from setup mode.

You can also use the [read_procfile](#) command to load the test procedure file from analysis mode. For more information, see “[Test Pattern Formatting and Timing](#)” on page 531.

4. Exit setup mode and run DRC. For example:

```
SETUP> set_system_mode analysis
```

5. Correct any DRC violations. For information on clock control definition DRCs, see “[Procedure Rules \(P Rules\)](#)” in the *Tessent Shell Reference Manual*.

Clock control definitions are enabled by default unless there are NCPs enabled in the test procedure file. If NCPs exist and are enabled, they override clock control definitions.

6. Report the clock control configurations. For example:

```
ANALYSIS> report_clock_controls
CLOCK_CONTROL "/top/core/clk1 (3457) " =
    SOURCE_CLOCK "/pll_clk (4)" ;
    ATPG_CYCLE 0 =
        CONDITION "/ctl_dff2/ (56)" 1;
    END;
    ATPG_CYCLE 1 =
        CONDITION "/ctl_dff1/ (55)" 1;
    END;
END;
```

Values in parenthesis are tool-assigned gate ID numbers.

7. Generate test patterns. For example:

```
ANALYSIS> create_patterns
```

8. Save test patterns. For example:

```
ANALYSIS> write_patterns ..../generated/patterns_edt_p.stil -stil -replace
```

Generating Test Patterns for Different Fault Models and Fault Grading

Use this procedure when you want to create test patterns for path delay, transition, stuck-at, and bridge fault models and fault grade the test patterns to improve fault coverage.

Note

 You can use N-detect for stuck-at and transition patterns. If you use N-detect, replace the stuck-at and/or transition patterns described in the procedure with the N-detect patterns.

Prerequisites

- A test procedure file must be available. See “[Test Procedure File](#)” in the *Tessent Shell User’s Manual*.
- A path definition file must be available. See “[The Path Definition File](#)” on page 352.
- For bridge faults, refer to the requirements outlined in “[Bridge and Open, and Cell Neighborhood Defects UDFM Creation](#)” on page 396.

Procedure

1. Create path delay test patterns for your critical path(s) and save them to a file. Then fault grade the path delay test patterns for transition fault coverage. Following are example commands in a dofile.

```

//-----Create path delay patterns-----
// Enable two functional pulses (launch and capture).
set_fault_type path_delay
read_fault_sites my_critical_paths
report_fault_sites path0
// Uncomment next 2 lines to display path in DFTVisualizer.
// set_gate_level primitive
// report fault sites path0 -display debug
create_patterns
// Save path delay patterns.
write_patterns pathdelay_pat.bin.gz -binary -replace
//-----
//-----Grade for broadside transition fault coverage-----
// Change the fault model (when you change the fault model, the
// the internal pattern set database is emptied).
set_fault_type transition
add_faults -all
// Read the previously saved path delay patterns into the external
// pattern set database; include the -All_patterns switch so the
// patterns are copied to the now empty internal pattern set
// database when they are simulated.
read_patterns pathdelay_pat.bin.gz
// Simulate all the path delay patterns for transition fault
// coverage, copying them into the internal pattern set as they
// are simulated.
simulate_patterns -store_patterns all
report_statistics
//-----

```

2. Create additional transition test patterns for any remaining transition faults, and add these test patterns to the original test pattern set. Then fault grade the enlarged test pattern set for stuck-at fault coverage. Following are example commands in a dofile.

```

//-----Create add'l transition fault patterns-----
// Create transition patterns that detect the remaining transition
// faults the path delay patterns did not detect during
// simulation.
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns and add'l transition patterns.
write_patterns pathdelay_trans_pat.bin.gz -binary -replace
//-----
//-----Grade for stuck-at fault coverage-----
set_fault_type stuck
add_faults -all
// Read in previously saved path delay and transition patterns and
// add them to the internal pattern set when they are simulated.
read_patterns pathdelay_trans_pat.bin.gz -all_patterns
simulate_patterns -store_patterns all
report_statistics
//-----

```

3. Create additional stuck-at test patterns for any remaining stuck-at faults and add them to the test pattern set. Then fault grade the enlarged test pattern set for bridge fault coverage. Following are example commands in a dofile.

```
//-----Create add'l (top-up) stuck-at patterns-----
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns and transition patterns, plus
// the add'l stuck-at patterns.
write_patterns pathdelay_trans_stuck_pat.bin.gz -binary -replace
//-----
//-----Grade for bridge fault coverage-----
set_fault_type udfm
read_fault_sites my_fault_definitions
add_faults -all
// Read in previously saved path delay, transition, and stuck-at
// patterns and add them to the internal pattern set
// when they are simulated.
read_patterns pathdelay_trans_stuck_pat.bin.gz
simulate_patterns -store_patterns all
report_statistics
//-----
```

4. Create additional bridge test patterns for any remaining bridge faults and add these test patterns to the test pattern set. Following are example commands in a dofile.

```
//-----Create add'l bridge patterns-----
create_patterns
order_patterns 3 // optimize the pattern set
// Save original path delay patterns, transition patterns, stuck-at
// patterns, plus the add'l bridge patterns.
write_patterns pathdelay_trans_stuck_bridge_pat.bin.gz -binary \
-replace
// Close the session and exit.
exit
```

Timing-Aware ATPG

Timing-aware ATPG reads timing information from a Standard Delay Format (SDF) file and tries to generate patterns that detect transition faults using the longest detection path.

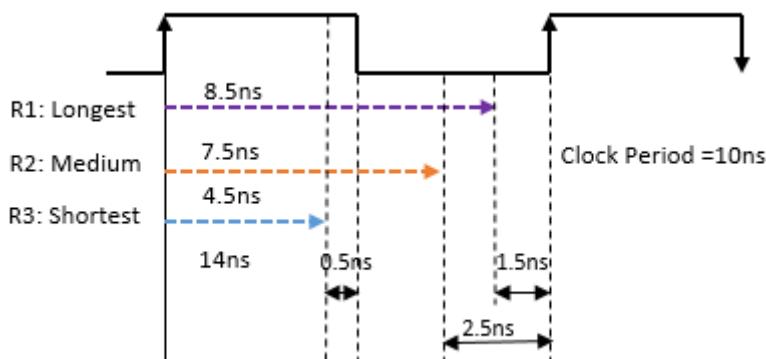
Slack Calculation	383
Synopsys Design Constraint Derived Clock Timing.....	384
Delay Test Coverage Calculation	384
Timing-Aware ATPG vs. Transition ATPG	385
Timing-Aware ATPG Limitations	385
Inaccuracies in Timing Calculations	386
Running Timing-Aware ATPG	387
Troubleshooting Topics.....	390

Slack Calculation

Slack is equal to the margin between the path delay and the clock period. Slack within Small Delay Fault Model represents the smallest delay defect that can be detected.

Figure 8-50 illustrates slack calculations. Assume there are three paths that can detect a fault. The paths have a 8.5 ns, 7.5 ns, and 4.5 ns delay, respectively. The clock period is 10 ns. The slacks for the paths are calculated as 1.5 ns, 2.5 ns, and 0.5 ns, respectively. The longest path launches from the leading edge and captures at the leading edge of the next cycle, giving it a 1.5 ns slack. Therefore the smallest delay defect that can be detected through this path is 1.5 ns. The shortest path launches from the leading edge and captures at the falling edge of the same cycle, giving it a 0.5 ns slack. To detect the small delay at the fault site, the test pattern should be generated to detect the fault through this path since it has the smallest slack.

Figure 8-50. Timing Slack



Synopsys Design Constraint Derived Clock Timing

Timing-aware ATPG uses the divided-by ratio information from the Synopsys Design Constraint (SDC) derived clocks and adjusts them according to the definitions in the SDC.

The clock period shown in [Figure 8-50](#) is 20 ns if it is a divided-by-two clock.

Delay Test Coverage Calculation

ATPG calculates a metric called Delay Test Coverage to determine the quality of the test patterns. The delay test coverage is automatically included in the ATPG statistics report when timing-aware ATPG is enabled.

The weight given to a detected fault is modified based upon the slack. It gives a greater weight to a path whose slack is closest to the minimum static slack. This is the formula for delay test coverage (DTC):

$$DTC = (\max_static_interval - dynamic_slack) / (\max_static_interval - static_slack) * 100\%$$

- **max_static_interval** — This is defined as the maximum time interval between the launch edge and capture edge among all the physical paths passing through the fault site. For example, in [Figure 8-50](#), the time interval for R1 and R2 is the full clock period, while it is one-half of the clock period for R3. The maximum static time interval through the fault site, in this case, is one full clock period.
- **dynamic_slack** — This is defined as the difference between the delay of the path used to detect the fault by the pattern and the interval between the launch edge and the capture edge associated with the path.
- **static_slack** — This is minimum slack among all the physical paths through the fault site.

Using the example paths from [Figure 8-50](#) the minimum slack is 0.5 ns. These are the delay test coverages for detecting the fault at each path:

- R1: $((10.0 \text{ ns} - 1.5 \text{ ns}) / (10.0 \text{ ns} - 0.5 \text{ ns})) = 89.47\%$
- R2: $((10.0 \text{ ns} - 2.5 \text{ ns}) / (10.0 \text{ ns} - 0.5 \text{ ns})) = 78.95\%$
- R3: $((10.0 \text{ ns} - 0.5 \text{ ns}) / (10.0 \text{ ns} - 0.5 \text{ ns})) = 100\%$

Undetected faults have a delay test coverage of 0 percent. DI faults (Detected by Implication) have a delay test coverage of 100 percent. Chip-level delay test coverage is calculated by averaging the delay test coverage values for all faults.

Timing-Aware ATPG vs. Transition ATPG

The following data was gathered from the STARC03 testcase, which STARC and Mentor Graphics used to evaluate timing-aware ATPG.

[Table 8-5](#) compares transition fault ATPG and timing-aware ATPG. The testcase has the following characteristics:

- Design Size: 2.4M sim_gate
- Number of FFs: 69,153
- CPU: 2.2Ghz
- 315 sec to read SDF file that has 10,674,239 lines

Table 8-5. Testcase 2 Data

Run Parameters	Pattern Count	Transition TC	Delay TC	CPU Time	Memory
Before ATPG	0	0.00%	0.00%	0	2.1G
Transition ATPG (1 detection)	3,668	91.23%	66.13%	4,180 sec	1.0G
Transition ATPG (7 detections)	7,979	91.27%	68.07%	11,987 sec	1.0G
Timing-Aware ATPG (SMFD ¹ = 100%)	3,508	91.19%	67.39%	17,373 sec	2.2G
Timing-Aware ATPG (SMFD = 50%)	8,642	91.26%	76.26%	129,735 sec	2.2G
Timing-Aware ATPG (SMFD = 0%)	24,493	91.28%	77.71%	178,673 sec	2.2G

1. Slack Margin for Fault Dropping = $(Ta-Tms)*100/Ta$.

For more information about setting run parameters, see the [set_atpg_timing](#) command.

Timing-Aware ATPG Limitations

There are many limitations involved in performing timing-aware ATPG.

The following is a list of timing-aware ATPG limitations:

- For limitations regarding the SDF file, see the [read_sdf](#) command.
- Launch-off shift is not supported.

- The ATPG run time for timing-aware is about eight times slower than the normal transition fault ATPG. For more information, see “[Timing-Aware ATPG vs. Transition ATPG](#)”. Targeting critical faults may help.
- A large combinational loop may slow down the analysis to calculate the static slack. It also makes the actual delay analysis less accurate.
- The transition test coverage on timing-aware ATPG may be lower than the normal transition fault ATPG. Because timing-aware ATPG tries to detect a fault with a longer path, it is more likely to hit the abort limit. You may use “-coverage_effort high” switch with the [create_patterns](#) command to improve transition test coverage. Be aware that using this option increases the run time.
- When calculating slack, the tool does not explicitly consider false or multicycle path effect, so it adds all path delays into the slack calculation. When the result slack is negative, meaning the maximum delay of the fault is larger than the clock period, the tool rounds the slack up to zero.
- Timing information in the Named Capture Procedure is not included in static slack calculations.
- When saving check point, the SDF database is not stored. You must reload the SDF data, using the [read_sdf](#) command, when using a flattened model.
- Static Compression ([compress_patterns](#) command) and pattern ordering ([order_patterns](#) command) cannot be used when [-Slack_margin_for_fault_dropping](#) is specified.
- Clock skew is ignored. Clock skew is caused by a delay on clock paths, and timing-aware ATPG does not use the delay on clock path. Consequently, clock skew (the delay on clock path) is not accounted for in the timing-calculation.
- The SDF file does not impact the good machine simulation value. The SDF file is used mainly to guide timing-aware ATPG to detect a fault along a long path as well as for calculating the delay test coverage. That is, the tool does not extract (or infer) false and multicycle paths from the delays in the SDF file. You have to provide that information with an SDC file or [add_false_paths](#) command.

Inaccuracies in Timing Calculations

When performing timing-aware ATPG based on SDF, several factors can result in inaccurate timing calculations.

Those factors are as follows:

- Device delay is not supported.
- Conditional delay is not supported. For a given IOPATH or INTERCONNECT pin pair, the maximal number among all conditional delays defined for this pair will be used by the tool when calculating static and actual delays.

- Negative delay is supported. However, if path delay is a negative number, the tool forces the delay value to 0 when calculating delay coverage, path delay, and slack, and so on.
- Static timing is calculated in a pessimistic way that does not consider false or multicycle paths.

Running Timing-Aware ATPG

Use this procedure to create Timing-Aware test patterns with the ATPG tool.

Prerequisites

- Because timing-aware ATPG is built on transition ATPG technology, you must set up for Transition ATPG first before starting this procedure. See “[Transition Delay Test Set Creation](#).”
- SDF file from static timing analysis.

Procedure

1. Load the timing information from an SDF file using the `read_sdf` command. For example:

```
ANALYSIS> read_sdf top_worst.sdf
```

If you encounter problems reading the SDF file, see “[Errors and Warnings While Reading SDF Files](#)” on page 390.

2. Define clock information using the `set_atpg_timing` command. You must define the clock information for all clocks in the design, even for those not used for ATPG (not used in a named capture procedure). For example:

```
ANALYSIS> set_atpg_timing -clock clk_in 36000 18000 18000
```

```
ANALYSIS> set_atpg_timing -clock default 36000 18000 18000
```

3. Enable timing-aware ATPG using the `set_atpg_timing` command. For example:

```
ANALYSIS> set_atpg_timing on -slack_margin_for_fault_dropping 50%
```

If you specify a slack margin for fault dropping, the fault simulation keeps faults for pattern generation until the threshold is met. During normal transition fault simulation, faults are dropped as soon as they are detected.

4. Select timing-critical faults using the `set_atpg_timing` command. For example:

```
ANALYSIS> set_atpg_timing -timing_critical 90%
```

5. Run ATPG. For example:

```
ANALYSIS> create_patterns -coverage_effort high
```

6. Report delay_fault test coverage using the `report_statistics` command. For example:

```
ANALYSIS> report_statistics
```

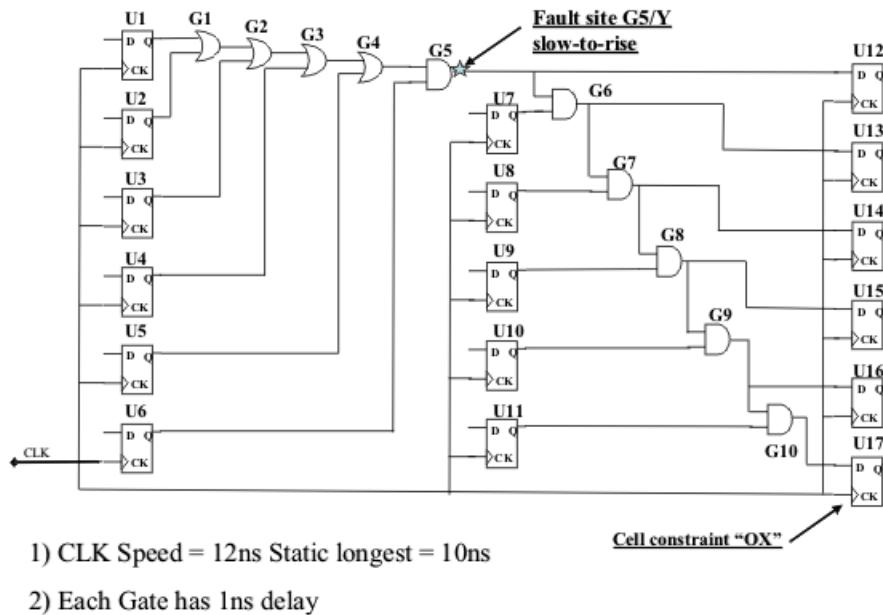
Examples

[Figure 8-51](#) shows a testcase where there are 17 scan flip-flops and 10 combinational gates. Each gate has a 1 ns delay and there is no delay on the scan flip-flops. A slow-to-rise fault is injected in G5/Y. The test period is 12 ns. The last scan flip-flop (U17) has an OX cell constraint so that it cannot be used as an observation point.

The longest path starts at U1, moving through G1 through G10 and ending at U17. The total path delay is 10 ns. Because U17 cannot be used as an observation point, timing-aware ATPG uses the path starting at U1, moving through G1 through G9 and ending at U16. The total path delay is 9 ns.

Therefore static minimum slack is $12\text{ ns} - 10\text{ ns} = 2\text{ ns}$, and the best actual slack is $12\text{ ns} - 9\text{ ns} = 3\text{ ns}$.

Figure 8-51. Testcase 1 Logic



[Figure 8-52](#) shows the dofile used for this example. As you can see in the comments, the dofile goes through the procedure outlined in “[Running Timing-Aware ATPG](#).”

Figure 8-52. Dofile

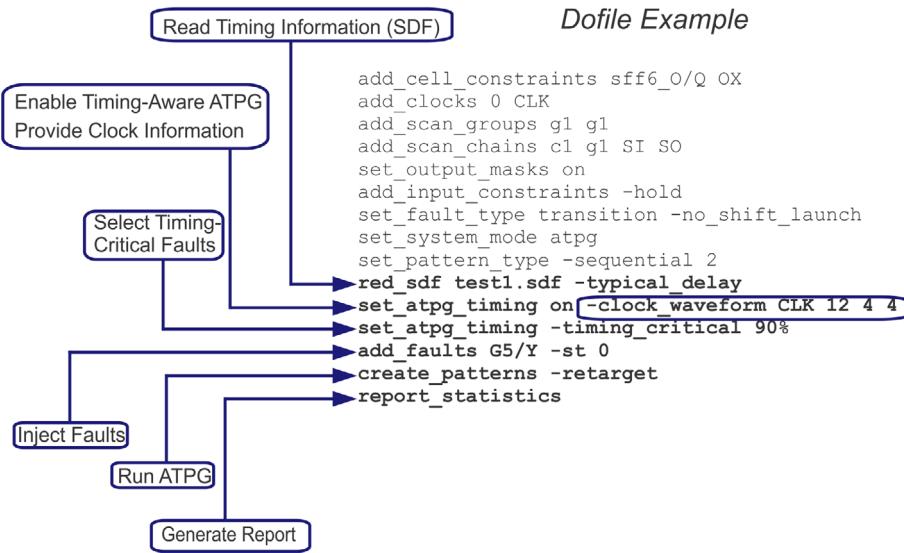


Figure 8-53 shows the fault statistics report. Timing-aware ATPG used the longest possible path, which is 9 ns. Static longest path is 10 ns. The delay test coverage is $9 \text{ ns} / 10 \text{ ns} = 90\%$.

Figure 8-53. Reports

Statistics report		
fault class	#faults (coll.)	#faults (total)
FU (full)	1	1
DS (det_simulation)	1	1
test_coverage	100.00%	100.00%
fault_coverage	100.00%	100.00%
atpg_effectiveness	100.00%	100.00%
Delay_test_coverage	90%	90%
#test_patterns	1	
#clock_sequential_patterns		1
#simulated_patterns		32
CPU_time (secs)		0.1

Troubleshooting Topics

The following topics describe common issues related to timing-aware ATPG.

Run Time Reduction for Timing-Aware ATPG	390
Errors and Warnings While Reading SDF Files.....	390
Warnings During ATPG	393
Actual Slack Smaller Than Tms.....	393

Run Time Reduction for Timing-Aware ATPG

Timing-aware ATPG takes much longer than the regular transition ATPG. You can make it faster by targeting for timing-critical faults only.

For more information about why timing-aware ATPG is slower, refer to [Table 8-5](#) on page 385.

You can use the `set_atpg_timing` command to make a fault list that includes faults with less slack time than you specified. The fault is put in the fault list if its (Longest delay)/(Test time) is more than your specified threshold.

For example, assume there are three faults and their longest paths are 9.5 ns, 7.5 ns and 6.0 ns respectively and the test time is 10 ns. The (Longest delay)/(Test time) is calculated 95 percent, 75 percent, and 60 percent respectively. If you set the threshold to 80 percent, only the first fault is included. If you set it to 70 percent, the first and second faults are included.

The following series of commands inject only timing-critical faults with 70 percent or more.

```
ANALYSIS> add_faults -all
ANALYSIS> set_atpg_timing -timing_critical 70%
ANALYSIS> write_faults fault_list1 -timing_critical -replace
ANALYSIS> delete_faults -all
ANALYSIS> read_faults fault_list1
```

Errors and Warnings While Reading SDF Files

There are many errors and warnings that can occur when reading SDF files.

The most common are as follows:

- Error: Near line N — The destination pin A/B/C is mapped to gate output
For an interconnect delay, the destination pin has to be a gate input pin. If a gate output pin is used, the tool issues the error message and ignores the delay.
- Error: Near line N — The pin A/B/C for conditional delay is mapped to gate output.

The signals used for a condition on a conditional delay must all be gate inputs. If a gate output is used for the condition, the tool issues the error message and ignores the delay.

For example, there is a component that has two outputs “O1” and “O2”. If a conditional delay on “O1” is defined by using “O2”, it will produce an error.

- Error: Near line N — Unable to map destination SDF pin.

Ignore INTERCONNECT delay from pin A/B/C to pin D/E/F.

The destination pin “D/E/F” does not have a receiver. It is likely to be a floating net, where no delay can be added.

- Error: Near line N — Unable to map source SDF pin.

Ignore INTERCONNECT delay from pin A/B/C to pin D/E/F.

The source pin “A/B/C” does not have a driver. It is likely to be a floating net, where no delay can be added.

- Error: Near line N — Unable to map flatten hierarchical source pin A/B/C derived from D/E/F.

“D/E/F” is a source pin defined in SDF (either interconnect delay or IO Path delay), and “A/B/C” is a flattened (cell library based) pin corresponding to “D/E/F,” and there is no driver found.

- Error: Near line N — Unable to map flatten hierarchical destination pin A/B/C derived from D/E/F.

“D/E/F” is a destination pin defined in SDF (either internet delay or IO Path delay), and “A/B/C” is a flattened (cell library based) pin corresponding to “D/E/F,” and there is no receiver found.

- Error: Near line N — There is no net path from pin A/B/C to pin D/E/F (fanin=1).
Ignore current net delay.

There is no connection between “A/B/C” (source) and “D/E/F” (destination). The “fanin=1” means the first input of the gate.

- Warning: Near line N — Negative delay value is declared.

Negative delay is stored in the timing database and the negative value can be seen as the minimum delay, but the negative number is not used for delay calculation. It will be treated as a zero.

- Error: Near line N — Flatten hierarchical destination pin "A" is not included in instance "B".

In this case, “B” is an instance name where both the source and destination pins are located (for example, an SDF file was loaded with “-instance B” switch). But a flattened

hierarchical destination pin (cell based library) “A” traced from the destination pin is outside of the instance “B.”

- Error: Near line N — Flatten hierarchical source pin “A” is not included in instance “B”.

In this case, “B” is an instance name where both the source and destination pins are located (for example, an SDF file was loaded with “-instance B” switch). But a flattened hierarchical source pin (cell based library) “A” traced from the source pin is outside of the instance “B.”

- Error: Near line N — Unable to add IOPATH delay from pin A/B/C to pin D/E/F even when the pins are treated as hierarchical pin. Ignore IOPATH delay.

There was an error when mapping the hierarchical source “A/B/C” and/or destination “D/E/F” to their flattened hierarchical pins. The delay is ignored. Most likely, one or both hierarchical pins are floating.

- Error: Near line N — Unable to add INTERCONNECT delay from pin "A/B/C" to pin "D/E/F" even when the pins are treated as hierarchical pin. Ignore INTERCONNECT delay.

There was an error when mapping the hierarchical source “A/B/C” and/or destination “D/E/F” to their flattened hierarchical pins. The delay is ignored. Most likely, one or both hierarchical pins are floating.

- Error: Near line N — The conditional delay expression is not supported. Ignore the delay.

The delay was ignored because the condition was too complex.

- Error: Near line N — Delay from pin A/B/C (fanin=1) to pin D/E/F has been defined before. Ignore current IOPATH delay.

Two or more IOPATH delays defined from SDF pin names map to the same flattened gate pin. The tool will keep the value for the first definition and ignore all the subsequent duplicates. This error with IOPATH delay is likely an SDF problem. For interconnect delay, the tool cannot handle multiple definitions. For example, the Verilog path is: /u1/Z -> net1 -> net2 -> /u2/A. If you define the first interconnect delay /u1/Z -> net1, the second net1 -> net2, and the third net2 -> /u2/A, then all the three interconnect delays are mapped to /u1/Z -> /u2/A in the gate level. This will cause an error.

- Error: Near line N — There is no combinational path from pin A/B/C (fanin=3) to pin D/E/F. Ignore current IOPATH delay.

The tool can only handle the IOPATH delay through combinational gates. Delays passing through state elements cannot be used.

Warnings During ATPG

It is possible to generate a warning during ATPG that is due to the conflicting setting for holding PI and masking PO between ATPG and static timing analysis.

In this case, you may see the following message when you start ATPG:

```
Warning: Inconsistent holding PI attribute is set between test generation
and static timing analysis.
Test generation with capture procedures holds PI, but static timing
analysis allows PI change.
The inconsistent settings will impact the timing metrics reported by the
tool.
```

For ATPG, you can hold PI and mask PO as follows:

- Including the “add_input_constraints -hold” and set_output_masks commands in your dofile.
- Including the “force_pi” and “measure_po” statements in a named capture procedure.

For static timing analysis, holding PI and masking PO can be set using the “-hold_pi” and “-mask_po” switches for the [set_atpg_timing](#) command.

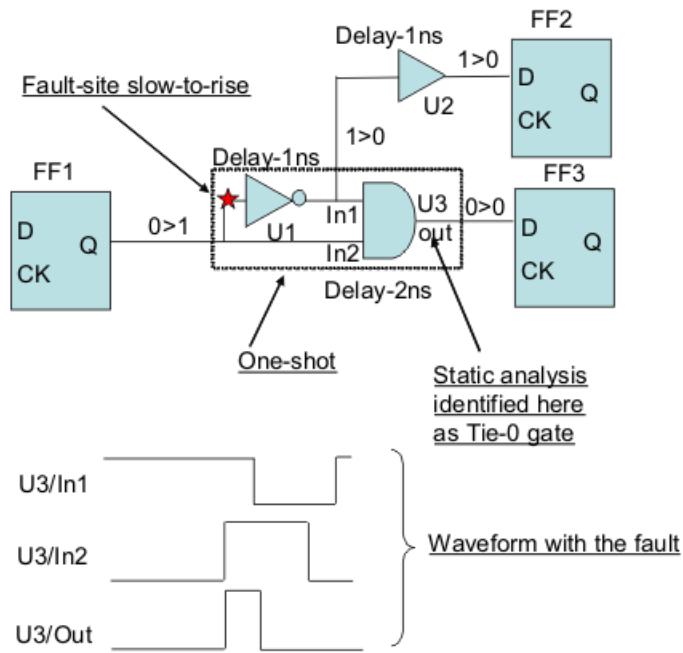
The static timing database is created once, before running ATPG, with the information of holding PI and/or masking PO, whereas the setting can change for ATPG especially when using NCP (for example, one NCP has force_pi and the other does not).

Actual Slack Smaller Than Tms

This condition occurs when a one-shot circuit is used in the data path. In static analysis, it is identified as a “static-inactive” (STAT) path but it could be used as a fault detection path.

For more information, refer to the circuit in [Figure 8-54](#).

Figure 8-54. Glitch Detection Case



A slow-to-rise fault is injected at the input of the inverter (U1). There are two paths to detect it. One is FF2 through U1 and U2 and the other is FF3 through U1 and U3. In the second path, the fault is detected as a glitch detection as shown in the waveform.

But in the static analysis (when calculating T_{ms}), this path is blocked because U3/out is identified as a Tie-0 gate by DRC. Therefore the maximum static delay will be 2 ns (first path). If the test cycle is 10 ns, the T_{ms} will be 8 ns. And if ATPG uses the second path to detect the fault, the actual slack will be 7 ns (10-1-2).

You will see the following message during DRC if DRC identifies a Tie-AND or Tie-OR.

```
// Learned tied gates: #TIED_ANDs=1 #TIED_ORs=1
```

Following are reports for the example shown in [Figure 8-54](#).

```
// command: report_faults -delay
// type    code   pin.pathname
//                                     static_slack actual_min_slack
// ----- -----
//      0     DS    /U1/A           8.0000      7.0000
// command: set_gate_report delay actual_path 0
// command: report_gates /U1/A
// /U1 inv01
//      A     I  (0-1) (3.0000)  /sff1/Q
//      Y     O  (1-0) (3.0000)  /U2/A  /U3/A1
// command: report_gates /U3/A1
// /U3 and02
//      A0    I  (0-1) (3.0000)  /sff1/Q
//      A1    I  (1-0) (3.0000)  /U1/Y
//      Y     O  (0-0) (-)     /sff3/D

// command: set_gate_report delay static_path
// command: report_gates /U3
// /U3 and02
//      A0    I  (2.0000, 2.0000)/(0.0000, 0.0000)  /sff1/Q
//      A1    I  (2.0000, 2.0000)/(2.0000, 2.0000)  /U1/Y
//      Y     O  (STAT)  /sff3/D
```

Bridge and Open, and Cell Neighborhood Defects UDFM Creation

You use Tesson Shell in conjunction with Tesson CellModelGen to extract interconnect bridges and opens as well as cell neighborhood defects, and create UDFMs. These UDFMs are subsequently used in ATPG for critical area based bridge and open, and cell neighborhood pattern generation.

This process uses a Layout Database (LDB), which is a prerequisite to this flow. For complete information, see “[Layout Database](#)” in the *Tesson Diagnosis User’s Manual* for complete information and procedures for creating a LDB.

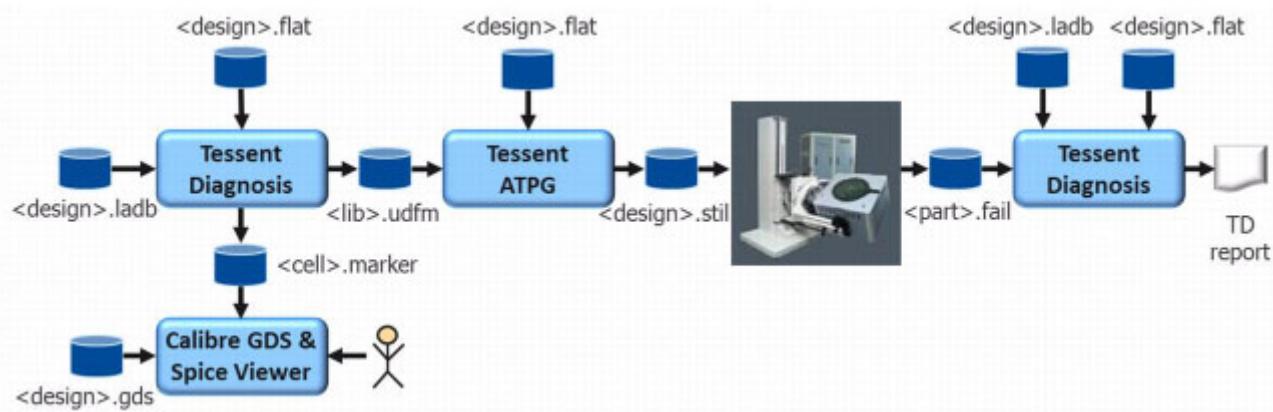
Interconnect Bridge and Open Extraction and UDFM Creation	397
Cell Neighborhood Defects UDFM Creation.....	419

Interconnect Bridge and Open Extraction and UDFM Creation

Extracting defects from a chip layout (GDS) file is a complex task and requires deep knowledge in layout extraction tools such as Calibre®.

The Tessent layout extraction feature does not require specific layout extraction knowledge and enables the extraction of critical area-based bridges and opens. Input to the Tessent fault site extraction tool is not a chip layout (GDS) file but the [Tessent Layout Database \(LDB\)](#), which is created for the layout-based diagnosis. An overview is shown in [Figure 8-55](#).

Figure 8-55. Tessent Bridge and Open Extraction Flow



From the LDB, interconnect bridges and opens can be extracted, and for each bridge and open defect its corresponding critical area is calculated. [Figure 8-58](#) on page 399 shows the critical area calculation for bridges for a certain particle size and its location between two adjacent nets.

The tool writes the considered bridges and opens into UDFM files that you use as input to ATPG for the critical area based bridge and open pattern generation. See “[About User-Defined Fault Modeling](#)” on page 56.

In addition to the UDFM files for ATPG purpose, also you can instruct the tool to create a marker file to back-annotate (highlight) defects of interest within the chip layout using the Calibre layout viewer—see “[Viewing Extracted Bridge and Open Defects](#)” on page 405.

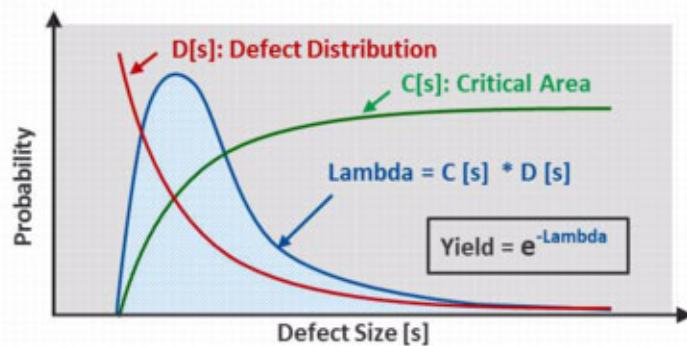
Critical Area Calculation	398
Viewing Extracted Bridge and Open Defects	405
Bridge and Open Extraction Output Files	409
Critical Area Based Bridge Pattern Generation.....	413
Critical Area Based Open Pattern Generation.....	417

Critical Area Calculation

The extraction of the fault sites can be influenced by various settings that refer to the defect probability and the critical area.

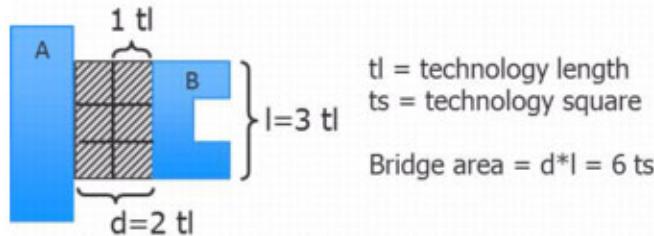
The tool calculates critical area based on the relation between the size and density of particles that may cause a bridge or open fault. The bridge probability depends, among others, on the distance between two adjacent objects. The open probability depends, among others, on the width of a net object and its length. [Figure 8-56](#) shows the generic relation between the probability, defect size, the critical area, and lambda, which is the multiplication product of defect size and critical area.

Figure 8-56. Probability and Critical Area



Based on the calculation shown in [Figure 8-56](#), the particle sizes that most probably will cause a bridge defect will be taken into account for the bridge area calculation as shown in [Figure 8-57](#), and also for the critical area calculation as explained in [Figure 8-58](#) on page 399. The assumed defect sizes are normalized to the technology length [tl]. That means, a value of 1 tl is equal to the length of the technology node.

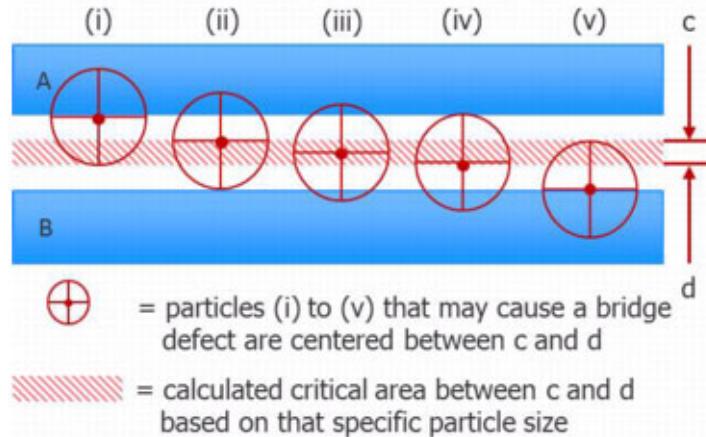
Figure 8-57. Bridge Area Calculation



In the example shown in [Figure 8-57](#), there are two adjacent objects in the cell layout on the same layer (for example metal2) and the distance between the adjacent objects is two technology lengths. The length of the bridging area is three technology lengths. As a result, the bridging area is six technology squares.

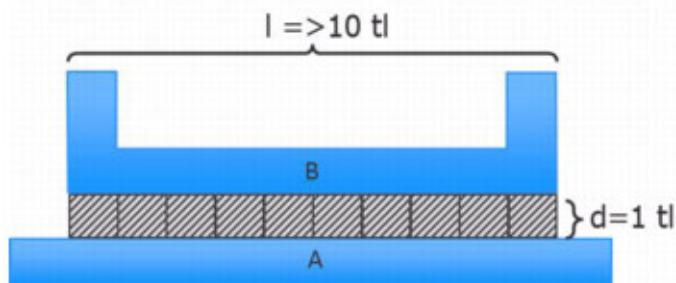
With the information about the maximum defect size that shall be considered, as shown in Figure 8-56 on page 398, the critical area of each bridge defect is calculated; this is illustrated in Figure 8-58.

Figure 8-58. Critical Area Calculation



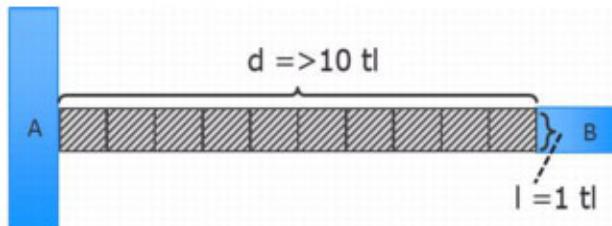
For the probability of each bridge defect, the distance between the adjacent objects and the length of the bridging area are taken into account. A high bridge probability is reached, when the distance between the adjacent objects is for example just 1 technology length and the length of the bridging area is for example 10 or more technology lengths, see the example shown in Figure 8-59.

Figure 8-59. High Bridge Probability

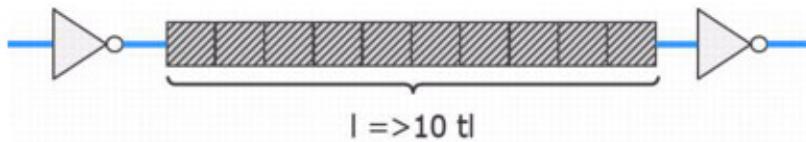


A low probability is present, when the bridging length is for example just 1 technology length and the distance between the 2 adjacent objects is for example 10 times or more the technology length. This is illustrated in Figure 8-60.

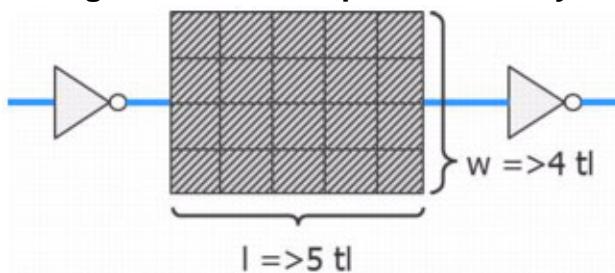
Figure 8-60. Low Bridge Probability



For the probability of each open defect, the width and length of the net object are taken into account. A high open probability is reached when the width of the net object (for example metal1, metal2 and so on) is small and the length is large, see an example shown in [Figure 8-61](#).

Figure 8-61. High Open Probability

A low open probability is given when the width is large and the length is small, see an example shown in [Figure 8-62](#).

Figure 8-62. Low Open Probability

Critical Area Calculation Formula for Bridges	400
Critical Area Calculation Formula for Opens & Vias	402
Alternative Critical Area Calculation Formula for Bridges	404

Critical Area Calculation Formula for Bridges

The formula for calculating the critical area for bridges is shown in the following figure.

Figure 8-63. Critical Area Calculation Formula for Bridges

$$\int_{s_{\min}}^{s_{\max}} \text{Probability}(s) * \text{Area}(s) ds \rightarrow \int_{s_{\min}}^{s_{\max}} \frac{3 * \text{dist}_{\min}^2 * (s + \text{length}) * (s - \text{dist})}{s^3} ds$$

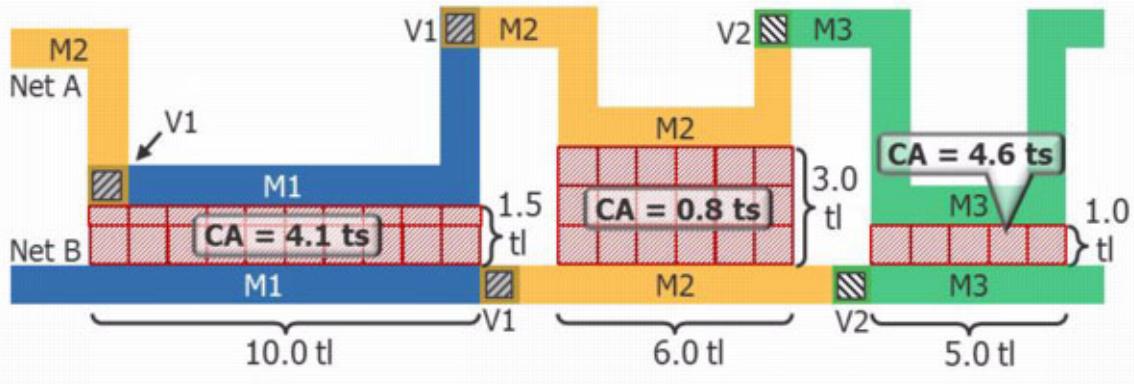
The definitions for the variables are as follows:

- s = spot size in technology length [tl]
- s_{\min} = minimum spot size in [tl], that can create a bridge
- s_{\max} = maximum spot size in [tl] to be considered

- distmin = technology dependent minimum distance between objects in [tl]
- length = length of the bridging area in [tl]
- dist = distance between the bridging objects in [tl]

[Figure 8-64](#) shows the result of such a calculation in three examples of side-to-side (S2S) bridges, with critical areas in layer M1, M2, and M3, each with a different distance and a different length.

Figure 8-64. Critical Area Example - Side-to-Side Bridges



The total critical area (TCA) in technology squares [ts] in the example in [Figure 8-64](#) is:

$$4.1 \text{ ts} + 0.8 \text{ ts} + 4.6 \text{ ts} = 9.5 \text{ ts}$$

The minimum spot size smin is different for the three bridge areas, that means for the bridge area in layer M1 in above example it is $s_{min} = 1.5$; for the M2 bridge area it is $s_{min} = 3.0$; and for the M3 bridge area it is $s_{min} = 1.0$.

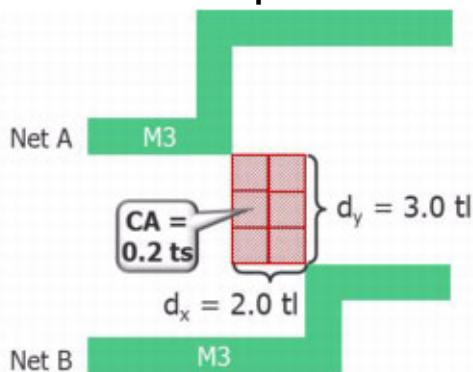
In all cases, the maximum spot size that is considered is $s_{max} = s_{min} + 2$

Note

 The calculation of the critical area for bridges to power and ground is the same as for side-to-side (S2S) bridges. The only difference is that they are written out to the UDFM file as B2P (bridge-to-power) and B2G (bridge-to-ground).

[Figure 8-65](#) shows the result of the critical area calculation for a corner-to-corner (C2C) bridge in layer M3.

Figure 8-65. Critical Area Example - Corner-to-Corner Bridge



The definitions for the variables for calculating the CA for C2C bridges slightly differ from the definitions for S2S bridges:

- length = - (higher value of dx or dy) in [tl]
- dist = lower value of dx or dy in [tl]

Note

Note that in this definition, the “length” is negative since the sides do not overlap. Also, in the example in Figure 8-65, the dy value is higher than the dx value. So, “length” is defined to be $-3tl$, and “dist” is defined to be $2tl$.

The critical area for this C2C bridge is 0.2 ts.

Critical Area Calculation Formula for Opens & Vias

The formula for calculating the critical area for opens and vias is shown in the following figure.

Figure 8-66. Critical Area Calculation Formula for Opens and Vias

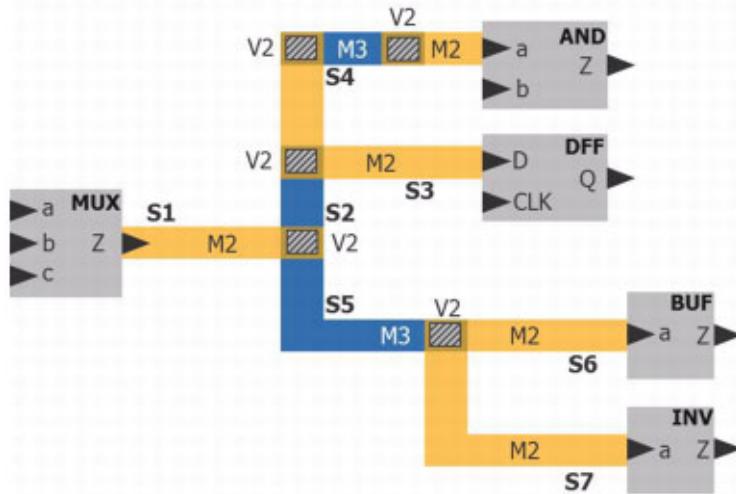
$$\int_{s_{\min}}^{s_{\max}} \frac{3 * \text{width}_{\min}^2 * (s + \text{length}) * (s - \text{width})}{s^3} ds$$

The definitions for the variables are as follows:

- s = spot size in technology length [tl]
- smin = minimum spot size in [tl], that can create an open
- smax = maximum spot size in [tl] to be considered
- widthmin = technology dependent minimum width of objects in [tl]

- length = length of the net or via object in [tl]
- width = width of the net or via object in [tl]

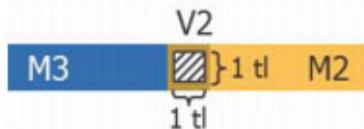
Figure 8-67. Open Example - Net with Seven Layout Segments



In [Figure 8-67](#) shows the layout of one net with seven segments, driven by the MUX Z output port. A layout segment is the part of the net that is in front of a fan-out or in front of a receiver port. Each layout segment defines an open defect. Vias will be considered together with the corresponding segment and will not result in a separate open defect.

[Figure 8-68](#) shows one via, which has a length and width of each 1 tl.

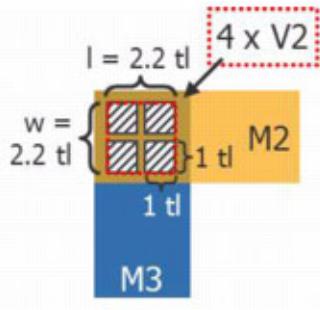
Figure 8-68. One Via



Using the critical area calculation formula shown in [Figure 8-66](#) on page 402, this results in a TCA of 1.96 ts.

[Figure 8-69](#) shows four single vias to connect an M3 object with an M2 object. Having four vias instead of just one as in [Figure 8-68](#), will result in a smaller TCA for the set of four vias. The calculation of the TCA for the set of four vias is based on a length and width of each 2.2 tl, so the surrounding rectangle (shown in red in [Figure 8-69](#)) is used for the critical area calculation.

Figure 8-69. Four Vias



Using the formula shown in [Figure 8-66](#) on page 402, this results in a TCA of 0.85 ts. This is in addition divided by the number of vias. So, the actual TCA for this example is 0.2125 ts.

Alternative Critical Area Calculation Formula for Bridges

An alternative method for calculating the critical area for bridges is an analytical solution based on a wider range of defects but taking critical area saturation into account as follows:

Figure 8-70. Alternative Critical Area Calculation Formula for Bridges

$$CA = \ln\left(\frac{2d + w}{d}\right) + \left(\frac{l - d}{2}\right) \times \left(\frac{1}{d} - \frac{1}{2d + w}\right)$$

The definitions for the variables are as follows:

- d = distance between the bridging objects
- l = length of the bridging area
- w = width of the net

Note that this alternative critical area calculation formula omits the units.

Based on this calculation method, the TCA of the S2S bridges shown in [Figure 8-64](#) is:

$$2.7 + 1.1 + 2.4 = 6.2$$

Accordingly, the TCA for the C2C example in [Figure 8-65](#) is 0.3.

Note

 Method 2 is only supported for bridges. When using method 2 with “-defect_types opens”, an error message will be issued .

Viewing Extracted Bridge and Open Defects

You can use the GDS viewer of Calibre to get a detailed view of the extracted interconnect defects within the layout of the design.

Using the “`extract_fault_sites -marker_file`” command and switch, you can use this marker file with the GDS viewer in Calibre to get a detailed view of the extracted interconnect defects within the layout of the design. For this, you need, in addition to the generated marker file, the `<design>.gds` file and optionally the layer properties file.

To open the design layout with the Calibre GDS viewer, issue the following on the command line:

```
calibredrv -m <design>.gds -rve <filename>.marker [ -l <layer.info> ]
```

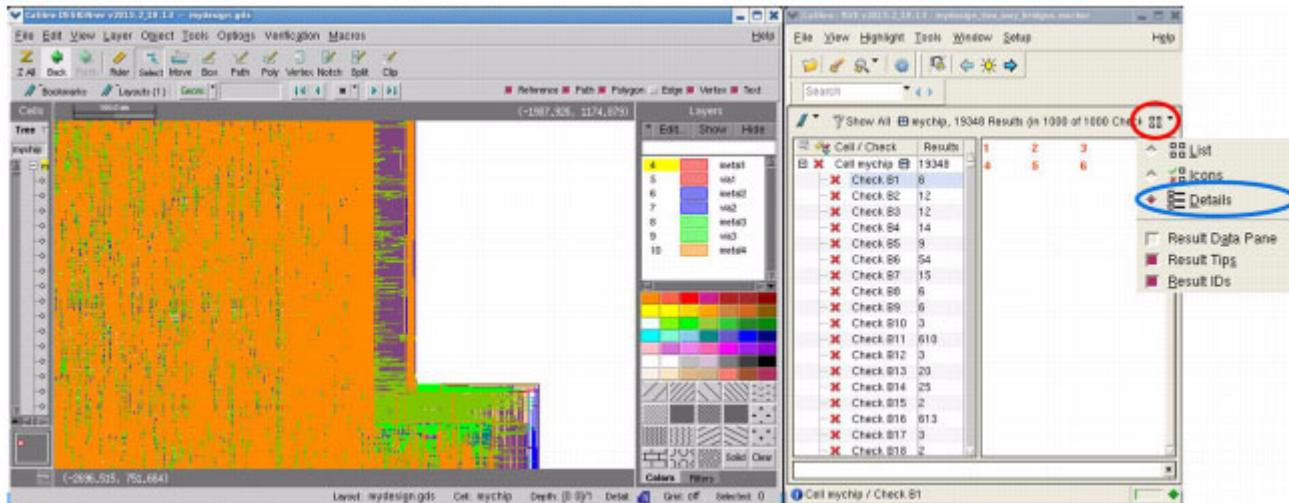
This opens Calibre showing the layout of your design, and the Calibre RVE window containing the list of all extracted interconnect defects. If you have no layer properties file for your design, you can omit the `-l` option and start Calibre by issuing the following on the command line:

```
calibredrv -m <design>.gds -rve <filename>.marker
```

The design layout opens containing all layers showing the Calibre defined properties.

A zoomed-in view of an example design with the RVE window is shown in [Figure 8-71](#).

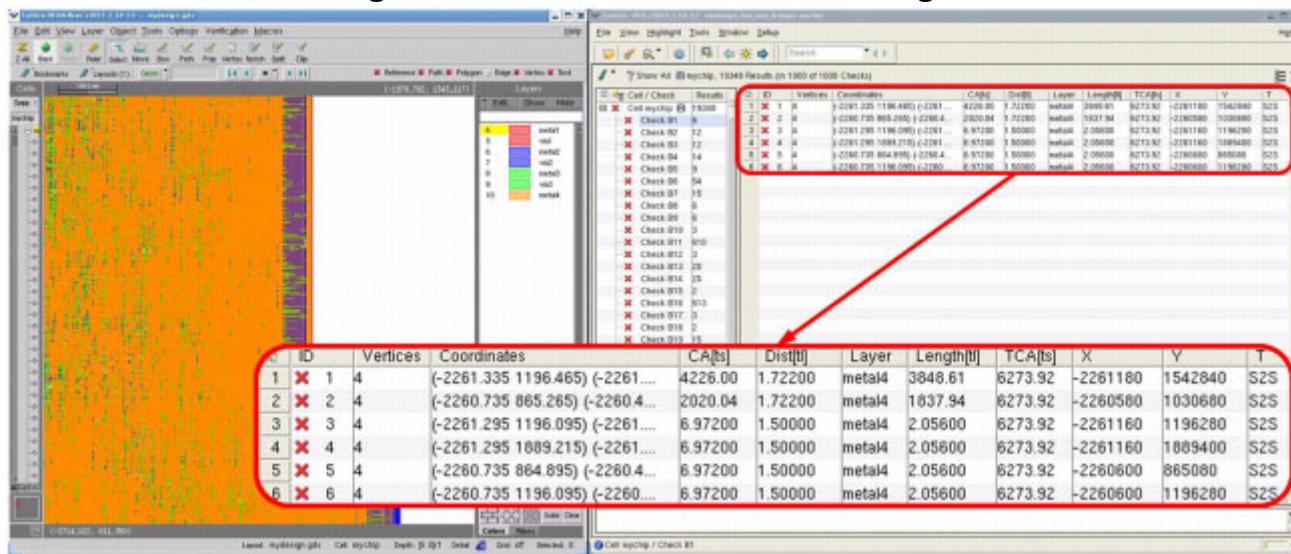
Figure 8-71. GDS View with Calibre RVE



In the RVE window on the right side of [Figure 8-71](#), you see the list of all extracted interconnect defects. You can change the view of the result window to not only see the Calibre internal ID(s) of the selected defect objects (in this example the bridge areas of bridge B1, numbered from 1 to 6), but also detailed information on the selected defects. For this, click on the red circled button shown in [Figure 8-71](#), then select “Details”, see the blue shape.

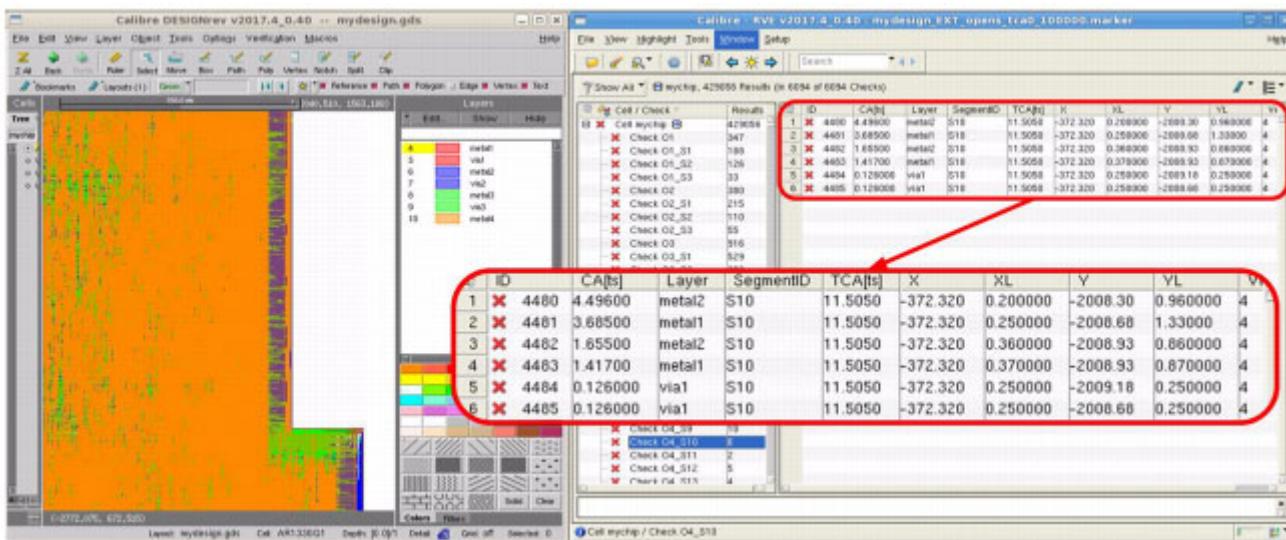
When having selected the details view, the RVE window displays detailed information on the selected defect as shown in [Figure 8-72](#).

Figure 8-72. Calibre RVE View for Bridges



Looking for example at bridge fault B1, you see that the bridge consists of 6 bridge areas, numbered in the “ID” column from 1 to 6. The column “Coordinates” contains the X/Y coordinates of each of the four vertices of each bridge area. These are not important because there is also an “X” and a “Y” column. The column “CA[ts]” displays the calculated critical area per bridge area in technology squares (ts). The column “Dist[tl]” contains the distance between the bridging objects per bridge area in technology length (tl). The “Layer” column displays the layer on which the bridge is detected. The “Length[tl]” column contains the length of each bridge area in technology length [tl]. The column “TCA[ts]” displays the calculated total critical area in technology squares (ts), that is the sum of all calculated critical areas for that bridge fault. The “X” and “Y” columns contain the X and Y coordinates pointing to the center of each bridge area. The “T” column displays the bridge type, which can be either S2S (side-to-side) or C2C (corner-to corner).

Figure 8-73. Calibre RVE View for Opens



The content of the RVE details deviates a bit for open defects, as shown in [Figure 8-73](#). Looking for example at open group O1, you see, that the open defect O1 consists of 3 layout segments named O1_S1, O1_S2, and O1_S3.

When you select the group “O1”, you will see the list of object details for all 3 segments being displayed in the RVE details window. When you select the group “O1_S1”, you will see only the objects details for segment S1 being shown in the RVE details window.

The example in [Figure 8-73](#) shows the details for the segment “O4_S10” which consists of 6 net objects. The column “CA[ts]” contains the calculated critical area per net object. The “Layer” column contains the layer per net object. The column “SegmentID” shows the layout segment for the corresponding net object. The “TCA[ts]” column contains the calculated total critical area of the corresponding layout segment, which is the sum of the “CA[ts]” column for that segment. Also, the “X” and “Y” coordinates per net object are given, pointing to the center of each net object. The “XL” and “YL” columns contain the x-length and the y-length of each layout object.

Highlighting Defects in Design Layout 407

Highlighting Defects in Design Layout

From the Calibre RVE window you can select any defect to be highlighted in the design layout. Select one item from the list, click the right mouse button and select “Highlight” or press the “H” key.

You can also highlight dedicated single bridge areas or open segments from the details window. An example for highlighting bridge fault B1 and two of the bridge areas is shown in [Figure 8-74](#).

Figure 8-74. Highlighted Bridge Fault B1

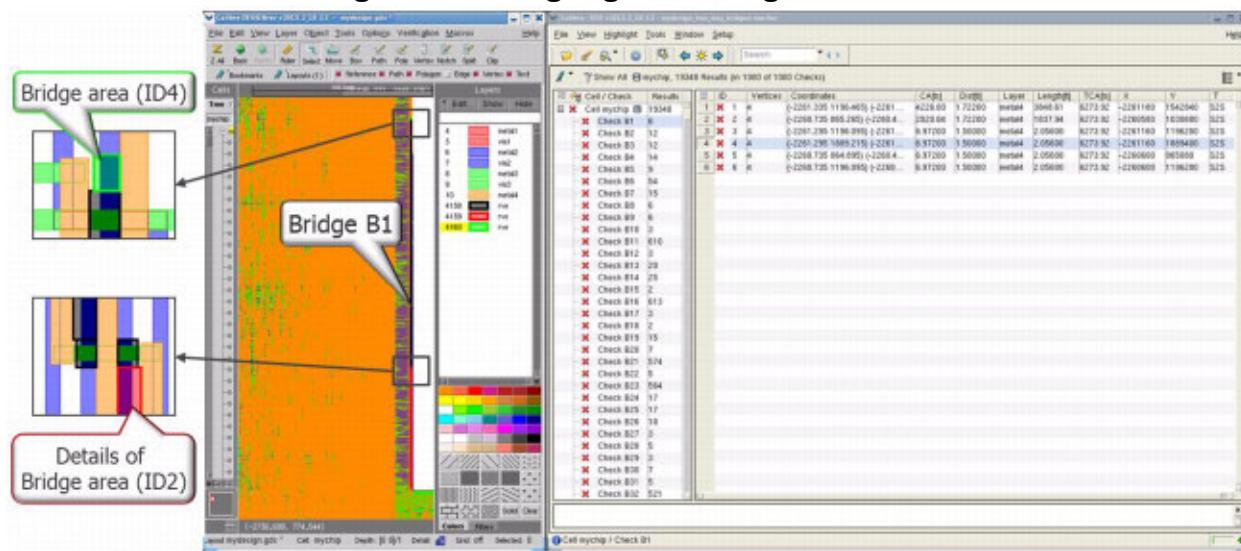


Figure 8-74 shows the bridge fault B1 as a black line on the right side of the layout window, beginning on top with a green dot for bridge area 4 (see “ID” column) and ending at the bottom with the red line for bridge area 2. On the left side of Figure 8-74 you see the enlarged picture details of bridge area 4 and bridge area 2.

To clear the highlights, click “Highlight” in the menu bar of the Calibre RVE window, then select “Clear Highlights”, or simply press the “F4” button on your keyboard.

Bridge and Open Extraction Output Files

This section describes the UDFM output files from bridge and open extraction.

Log File	409
UDFM File	409

Log File

If you are saving a log file of the tool session, the extract_fault_sites command fills it with a distribution graph, which displays the number of defects in relation to the total critical area. Otherwise this distribution graph is displayed to the standard output.

For more information and an example, refer to the [extract_fault_sites](#) command in the *Tessent Shell Reference Manual*.

UDFM File

When running the extract_fault_sites command, a partly encrypted UDFM file will be generated.

The UDFM file begins with a summary of the applied fault site extraction settings followed by a distribution graph.

This part is followed by the UDFM section containing the encrypted net names, this section is called “EncryptedLocationAliases”:

```
UDFM {  
    Version : 3;  
    EncryptedLocationAliases {  
        "zIYEAAAAAAABqEAAAAGQOMQHm2zPeHPtI8RYLEBpudnJEyNtSP...3jQ4P71IS";  
        "zIYEAAAAAAABqEAAAAGQOMQHm2zPedfjrdjfssfhbpsulakjtSP...3jQ4P71IS";  
        "zIYEAAAAAAABqEAAAAGQOMQHm2zPshfgajhfRYLEBdnJEyNtSP...3jQ4P71IS";  
        .  
        .  
        .  
    }  
}
```

See also the [extract_fault_sites](#) command in the *Tessent Shell Reference Manual*.

The UDFM Section for Extracted Bridges

The encrypted section is then followed by a readable part containing the data for the extracted bridges.

```
UdfmType("interconnect_bridges") {
    Instance("/")
        Bridge("B1")
    {Type:S2S;TCA:6285.370;Layer:"metal4";Net1:$N22071;Net2:$N22074;
     CA:4225.996;Distance:1.722;Length:3848.611;X:-2261.180;Y:1542.840;
     XL:0.310;YL:692.750;}
        Bridge("B2")
    {Type:S2S;TCA:5740.685;Layer:"metal4";Net1:$N28132;Net2:$N28134;
     CA:2455.966;Distance:1.722;Length:2235.278;X:-2307.980;Y:1570.920;
     XL:0.310;YL:402.350;}
    .
    .
    .
    Bridge("B13") {Type:B2G;TCA:3835.294;Layer:"metal4";Net1:$N93624;
     CA:3060.823;Distance:2.806;Length:6749.944;X:-2210.683;Y:-144.840;
     XL:0.505;YL:1214.990;}
    .
    .
    .
    Bridge("B2762347")
{Type:C2C;TCA:0.043;Layer:"metal2";Net1:$N288288;Net2:$N892868;
 CA:0.043;Distance:9.742;X:-2684.720;Y:1891.080;XL:1.240;YL:1.240;}
    Bridge("B2762348")
{Type:C2C;TCA:0.043;Layer:"metal2";Net1:$N877331;Net2:$N787491;
 CA:0.043;Distance:9.742;X:-1682.000;Y:2735.400;XL:1.240;YL:1.240;
 }
}
```

The top lines show the bridge faults with the highest defect probability reflected by the “TCA” value; the bottom lines show the bridges with the lowest probability.

Each fault is summarized in one line containing the following information:

- **Bridge ID** — This is the defect id, for example “B1”.
- **Type** — This is the bridge type of the largest bridge area, which can be a side-to-side (S2S), a corner-to-corner (C2C) bridge, a bridge-to-ground (B2G), or a bridge-to-power (B2P).
- **TCA** — This is the calculated total critical area, which is the sum of all critical areas for that bridge fault.
- **Layer** — This is the layer with the largest bridge area, where the bridge fault is detected, for example “metal5”.
- **Net1, Net2** — These are the corresponding location aliases for the net names at which the bridge fault is located. In case of a power bridge (B2P), or a bridge-to-ground (B2G) Net2 is not specified.
- **CA** — This is the calculated critical area for the largest bridge area of that bridge fault.
- **Distance** — This is the distance between the bridging objects of the largest bridge area, defined in technology length [tl].

- **Length** — This is the length of the largest bridge area, defined in technology length [tl]. In case of a corner-to-corner (C2C) bridge, the length is excluded.
- **X, Y** — These are the X/Y coordinates in microns, pointing to the center of the largest bridge area.
- **XL, YL** — These are the width and length values of the largest bridge area, defined in microns.

The UDFM Section for Extracted Opens

When using the “[extract_fault_sites -defect_types opens](#)” command and switch, a section for opens will be written to the UDFM file. In the following, the UDFM section for opens is shown:

```

UdfmType("interconnectOpens") {
    Instance("/")
        Fault("O6_S1_R") {DefectInfo("O6"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6654.603;}Test{StaticFault{$N3:0;}}Test{DelayFault{$N3:0;}}}
        Fault("O6_S1_F") {DefectInfo("O6"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6654.603;}Test{StaticFault{$N3:1;}}Test{DelayFault{$N3:1;}}}
        Fault("O8_S1_R") {DefectInfo("O8"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6544.646;}Test{StaticFault{$N6:0;}}Test{DelayFault{$N6:0;}}}
        Fault("O8_S1_F") {DefectInfo("O8"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6544.646;}Test{StaticFault{$N6:1;}}Test{DelayFault{$N6:1;}}}
        Fault("O11_S1_R") {DefectInfo("O11"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6435.698;}Test{StaticFault{$N9:0;}}Test{DelayFault{$N9:0;}}}
        Fault("O11_S1_F") {DefectInfo("O11"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6435.698;}Test{StaticFault{$N9:1;}}Test{DelayFault{$N9:1;}}}
        Fault("O13_S1_R") {DefectInfo("O13"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6391.743;}Test{StaticFault{$N11:0;}}Test{DelayFault{$N11:0;}}}
        Fault("O13_S1_F") {DefectInfo("O13"){Type:Open;Layer:"metal1-metal2-metal3-metal4-via1-
            via2-via3";TCA:6391.743;}Test{StaticFault{$N11:1;}}Test{DelayFault{$N11:1;}}}.
        .
        .
        .
        Fault("O16688_S21_R") {DefectInfo("O16688"){Type:Open;Layer:"metal3";TCA:0.539;}}
            Test{StaticFault{$N515045:0;}}Test{DelayFault{$N515045:0;}}
            Test{StaticFault{$N515047:0;}}Test{DelayFault{$N515047:0;}}
            Test{StaticFault{$N515046:0;}}Test{DelayFault{$N515046:0;}}
            Test{StaticFault{$N515049:0;}}Test{DelayFault{$N515049:0;}}
            Test{StaticFault{$N515050:0;}}Test{DelayFault{$N515050:0;}}
            Test{StaticFault{$N515051:0;}}Test{DelayFault{$N515051:0;}}
            Test{StaticFault{$N515052:0;}}Test{DelayFault{$N515052:0;}}
        Fault("O16688_S21_F") {DefectInfo("O16688"){Type:Open;Layer:"metal3";TCA:0.539;}}
            Test{StaticFault{$N515045:1;}}Test{DelayFault{$N515045:1;}}
            Test{StaticFault{$N515047:1;}}Test{DelayFault{$N515047:1;}}
            Test{StaticFault{$N515046:1;}}Test{DelayFault{$N515046:1;}}
            Test{StaticFault{$N515049:1;}}Test{DelayFault{$N515049:1;}}
            Test{StaticFault{$N515050:1;}}Test{DelayFault{$N515050:1;}}
            Test{StaticFault{$N515051:1;}}Test{DelayFault{$N515051:1;}}
            Test{StaticFault{$N515052:1;}}Test{DelayFault{$N515052:1;}}}
    }
}

```

Each fault definition represents an open defect segment containing the following information:

- **Fault ID** — This is the unique fault id, constructed by the open group id, followed by the corresponding segment id. For each open, the results for both a rising and a falling edge will be written to the UDFM file, which is reflected by the appendix to the fault ID, for example “O6_S1_R” and “O6_S1_F”.
- **DefectInfo** — This is the overall open group id, under which the open segments are summarized, for example “O6”.
- **Type** — This is the defect type, which is in this section always “Open”.
- **Layer** — This is the list of layers of the open segment, for example “metal3” or “metal1-metal2-via1”.
- **TCA** — This is the calculated total critical area for that open segment.
- **StaticFault** — These are the corresponding location aliases for the port names at which the static open defect is observable.
- **DelayFault** — These are the corresponding location aliases for the port names at which the delay open defect is observable.

By default, the open defects are sorted in descending order by the total critical area (TCA) of the corresponding net segments. When the option “[extract_fault_sites -open_sorting net](#)” is used, the list of fault definitions is sorted in two levels. In the first level it is sorted by the summarized total critical area (TCA) of all segments for each open group in descending order. In the second level, all fault definitions for the segments of a particular open group are also sorted in descending order by the specific TCA value.

Critical Area Based Bridge Pattern Generation

The following sections describe bridge pattern generation.

Bridge Fault Model	413
Four-Way Bridge Model for Static ATPG	413
Two-Way Bridge Model for Delay ATPG.....	414
How to Generate Bridge Patterns	415

Bridge Fault Model

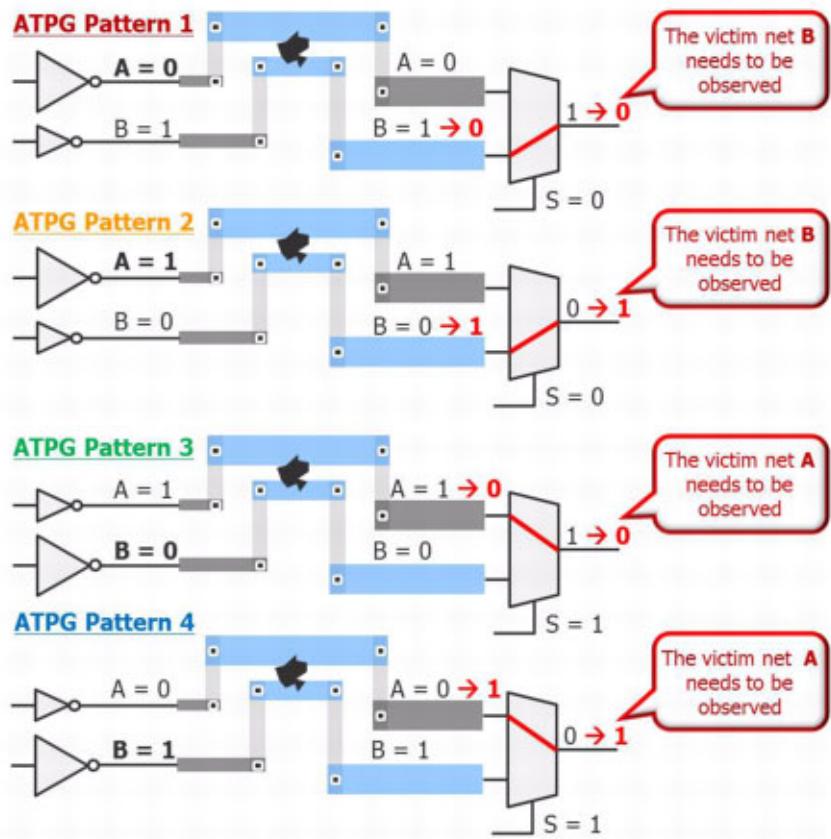
Bridge Faults can be extracted as four-way bridges or two-way bridges. By default, the bridge static faults will be written out as four-way bridges, and the bridge delay faults will be written out as two-way bridges. The following sections describe these two bridge models.

Four-Way Bridge Model for Static ATPG

For the bridge static model, the ATPG is forced to generate in total four static patterns, two patterns assuming net A is the aggressor, and net B is the victim, and two additional patterns assuming net B is the aggressor, and net A is the victim.

This is illustrated in [Figure 8-75](#). In case that both nets can be observed concurrently, it is possible that only two patterns are generated.

Figure 8-75. Four-Way Bridge Static Model

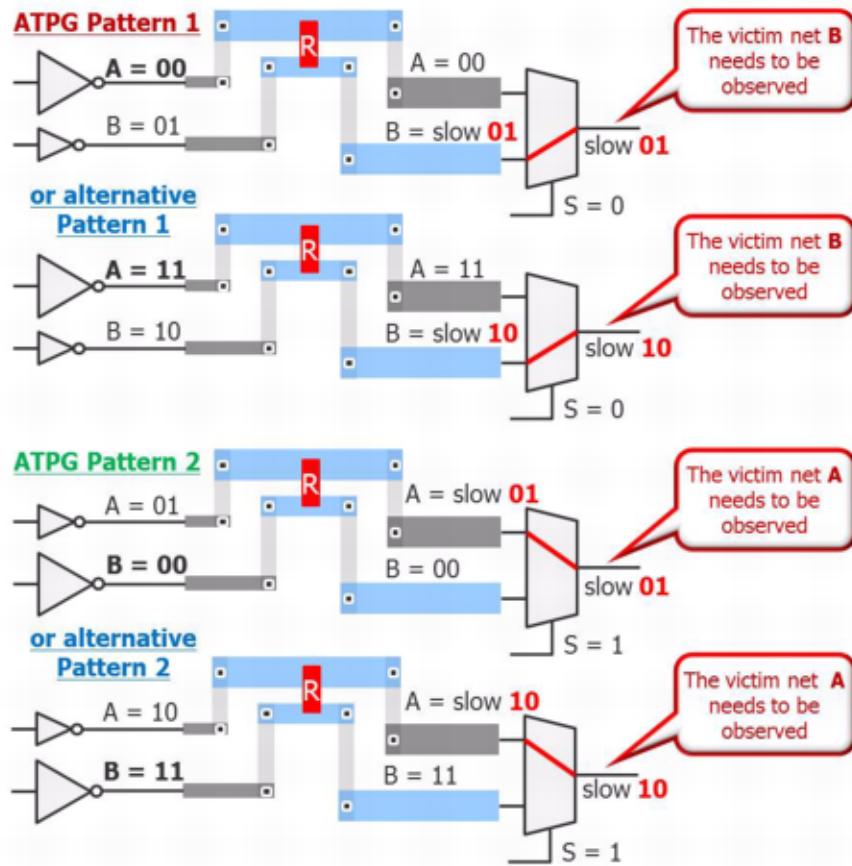


Two-Way Bridge Model for Delay ATPG

For the bridge delay model, the ATPG is forced to generate in total two delay patterns. One pattern is generated assuming net A is the aggressor with a constant state, and net B is the victim with a rising or a falling edge. Depending on the edge on net B, the constant value on net A is either 00 or 11.

The second pattern is generated assuming net B is the aggressor with a constant state, and net A is the victim with a rising or a falling edge. Depending on the edge on net A, the constant value on net B is either 00 or 11. This is illustrated in [Figure 8-76](#).

Figure 8-76. Two-Way Bridge Delay Model



How to Generate Bridge Patterns

After extracting the bridges using the `extract_fault_sites` command.

As explained in the `extract_fault_sites` command description, the created UDFM file can be used within Tessent Shell to generate critical area based bridge patterns. An example for the corresponding dofile for static pattern generation from scratch is shown below:

```
set_context patterns -scan
read_flat_model my_design.flat_model.gz
set_fault_type udfm -static_fault
read_fault_sites my_design_bridges.udfm
add_faults -all
create_patterns
write_patterns my_design_static_bridges.stil.gz -stil -replace
```

For the delay pattern generation, replace the fault type as follows:

```
set_fault_type udfm -delay_fault
```

and rename the pattern file defined with the `write_patterns` command accordingly.

The “[add_faults](#) -all” will add the bridge faults as defined in the UDFM. You may specify the -verbose option to the add_faults command to get a reporting of netnames that could not be matched to netnames of the design.

The following is an example of a dofile for the critical area based bridge static top off ATPG that might need to be adjusted for your design.

```
set_context patterns -scan
read_flat_model my_design.flat_model.gz
set_fault_type udfm -static_fault
read_fault_sites my_design_bridges.udfm
add_faults -all

read_patterns apt1_CA1_atpg.pat.gz
simulate_patterns -source external
report_statistics
report_udfm_statistics -set_baseline

set_pattern_source internal
create_patterns
report_udfm_statistics

write_patterns my_design_static_bridges.stil.gz -stil -replace
```

For the delay top off ATPG, again replace the fault type as follows:

```
set_fault_type udfm -delay_fault
```

and rename the pattern files specified with the commands [read_patterns](#) and [write_patterns](#) accordingly.

Critical Area Based Open Pattern Generation

This section describes open pattern generation.

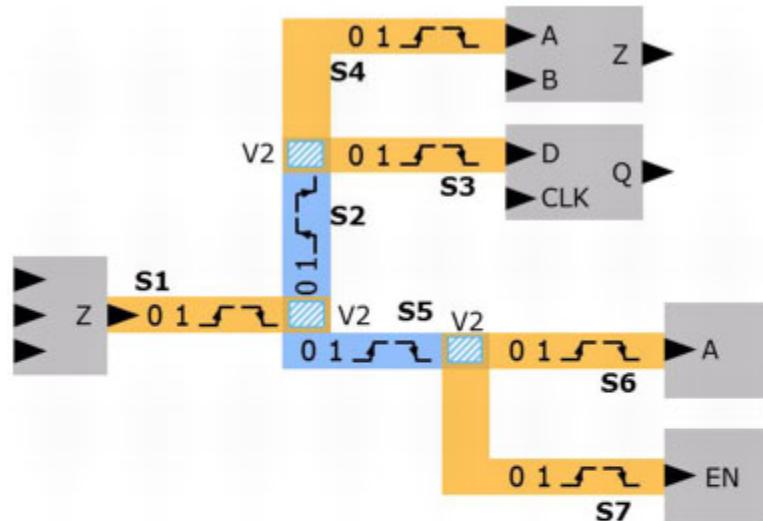
Open Fault Model	417
How to Generate Open Patterns.....	418

Open Fault Model

The open fault model is based on the actual layout of the chip. For this, each net in the layout from the driven cell to all its receivers is analyzed. As such, each net consists of one or more net segments, and for each net segment its own total critical area (TCA) is calculated. Each net segment will have two static and two delay faults.

In the open fault model, the static ATPG is forced to generate both 0 and 1 states for each net segment, and the delay ATPG is forced to generate both a rising & a falling edge, which is illustrated in [Figure 8-77](#).

Figure 8-77. Open Fault Model - Static and Delay



How to Generate Open Patterns

The following is an example of a dofile for the critical area based open static ATPG run from scratch that might need to be adjusted for your design.

```
set_context patterns -scan
read_flat_model my_design.flat_model.gz
set_fault_type udfm -static_fault
read_fault_sites my_design_open.udfm.gz
add_faults -all
create_patterns
report_udfm_statistics
write_patterns my_design_staticOpens.pat.gz -ascii -replace
```

Please notice that the fault type is set to “udfm -static_fault”, and that only the dedicated open UDFM file is read.

For the open delay ATPG, replace the fault type as follows:

```
set_fault_type udfm -delay_fault
```

and rename the pattern file defined with the [write_patterns](#) command accordingly.

The following is an example of a dofile for the critical area based open static top off ATPG that might need to be adjusted for your design.

```
set_context patterns -scan
read_flat_model my_design.flat_model.gz
set_fault_type udfm -static_fault
read_fault_sites my_design_open.udfm.gz
add_faults -all
read_patterns my_design_Ca1_atpg.pat.gz
simulate_patterns -source external
report_udfm_statistics -set_baseline
report_statistics
create_patterns
report_udfm_statistics
write_patterns my_design_staticOpens.pat.gz -ascii -replace
```

Please notice that the fault type is set to “udfm -static_fault”, and that only the dedicated open UDFM file is read. Then, first the existing Cell-Aware Test (CAT-static) patterns are read and then these are fault simulated using the UDFM open fault model. Performing this fault simulation of the existing CAT-static pattern will typically result in more or less no additional patterns to be generated with the [create_patterns](#) command.

For the open delay top off ATPG, again replace the fault type as follows:

```
set_fault_type udfm -delay_fault
```

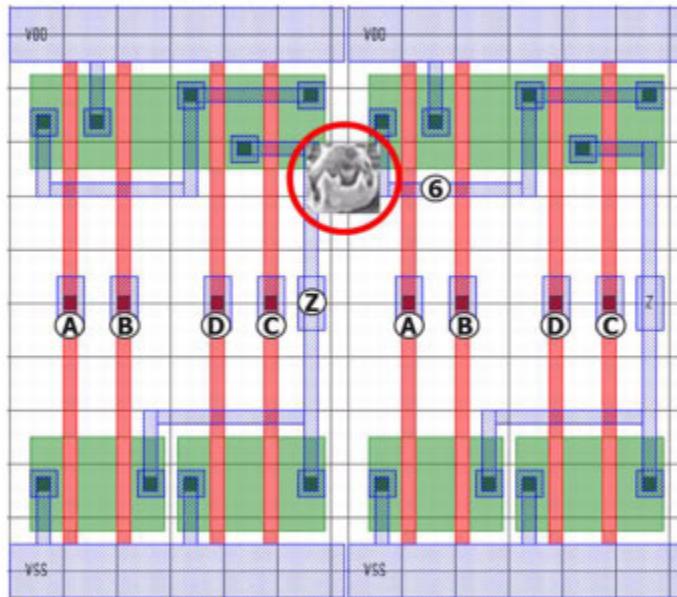
and rename the pattern files specified with the commands [read_patterns](#) and [write_patterns](#) accordingly.

Cell Neighborhood Defects UDFM Creation

Cell neighborhood defects can occur in any process technology. These are chip-dependent bridge defects on the interface of one instantiated standard cell to another standard cell that is placed right, left, top or at the bottom of a certain cell.

The difficulty with such defects is that these bridge defects are not purely on the interconnect between standard cells, but that bridges are possible to cell-internal nets (layers) that are not accessible directly via the nets on the cell interconnects. Such a case is illustrated in [Figure 8-78](#), which shows an AndOr cell on the left and the same AndOr cell on the right.

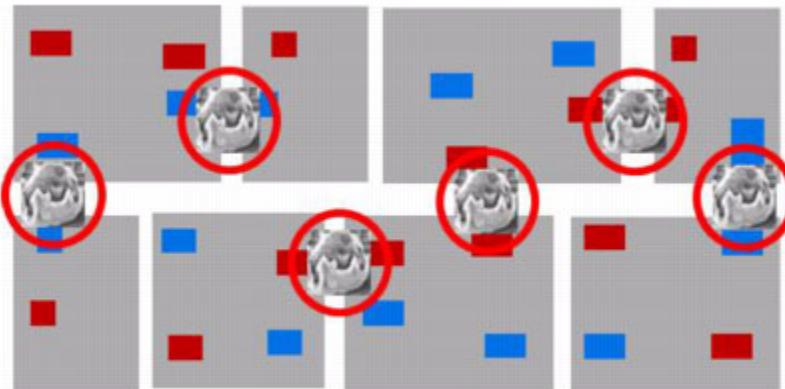
Figure 8-78. Neighborhood Cell Layout



The bridge defect that is not targeted explicitly by the Cell-Aware ATPG and also not targeted by doing an interconnect bridge extraction, is the bridge defect between the “Z” output of the left cell and the cell-internal net “6” of the cell on the right. This bridge defect can only be targeted explicitly when the cell content of the two neighbor cells are given together to [Tessent CellModelGen](#). Tessent CellModelGen is then able to analyze the bridge to the cell-internal net “6” and to create a CAT-view (UDFM) file that is passed on to CAT ATPG to generate the chip-dependent test pattern to detect such neighborhood defects. See also “[About User-Defined Fault Modeling](#)” on page 56.

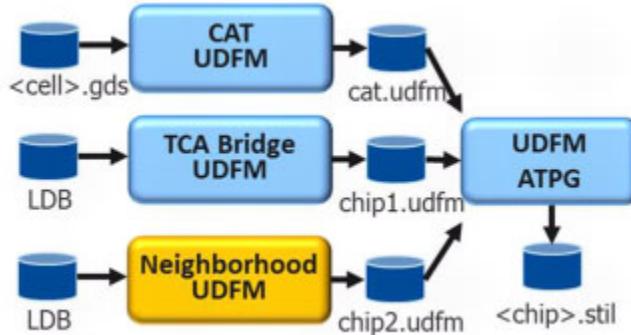
[Figure 8-79](#) shows a simplified chip layout with potential bridge defects between various adjacent cells. This illustrates that bridge defects may occur between horizontally adjacent cells, but also to cells that are on top or bottom to each other.

Figure 8-79. Cell Neighborhood Defects



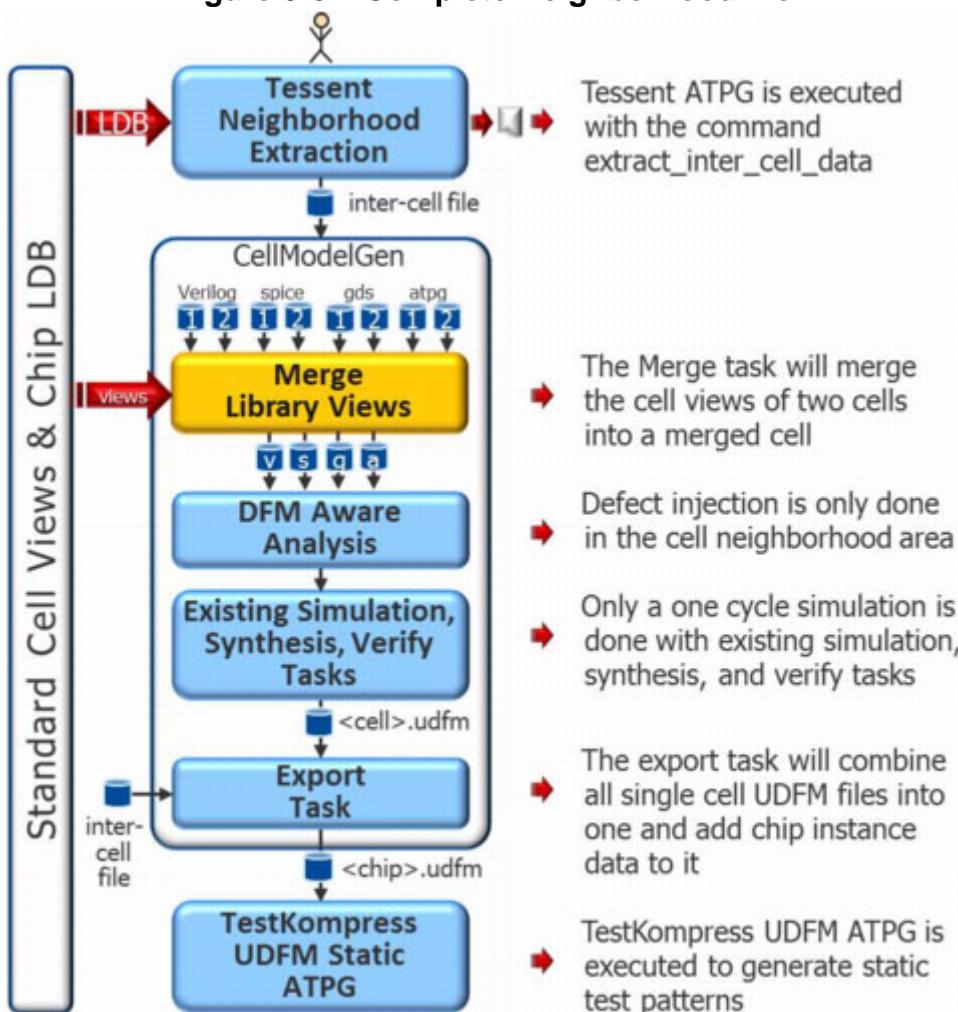
The general Neighborhood UDFM generation is shown in [Figure 8-80](#). This is to point out that that the traditional technology-dependent Cell-Aware Test (CAT) UDFM still needs to be generated as well to target all cell-internal defects, and that the Total Critical Area (TCA) Bridge UDFM file to target the chip-dependent interconnect bridge defects is also required. The Neighborhood UDFM generation is to ensure that the highest product quality is achieved. The UDFM ATPG runs can all be combined into just one static UDFM ATPG run and a second delay UDFM ATPG run. For experimental purposes, also single UDFM ATPG runs can be done; that is one each with the CAT UDFM, the TCA Bridge UDFM, and the Neighborhood UDFM file.

Figure 8-80. Neighborhood UDFM Generation Flow



Further details for the generation of the Neighborhood UDFM file are illustrated in [Figure 8-81](#).

Figure 8-81. Complete Neighborhood Flow



The first step in the flow of [Figure 8-81](#) is to extract the cell neighborhood data file using Tessent TestKompress, Tessent FastScan, or Tessent Diagnosis. The second step is the Tessent CellModelGen run, which starts with the [merge task](#), merging the GDS, Verilog, ATPG and Spice view of the two neighboring cells into just one view each, such that the Tessent CellModelGen view generation can be executed on the combination of two cells. The defect injection will be specific in case of such merge runs; that is only defects in the adjacent neighborhood area will be injected. The purpose of this Tessent CellModelGen run is to create a UDFM file that only targets bridge defects in the neighborhood area. When all wanted cell combinations are executed by Tessent CellModelGen, the export task can be started, which will create a chip level UDFM file. Finally, Tessent TestKompress can be executed to generate the test patterns, that will detect the cell-neighborhood defects explicitly.

Note

Executing the -merge task requires a Tessent TestKompress Automotive-grade ATPG license. For the CellModelGen view generation with the merge task, the *run_flow.merge* script can be used. For the export task on the merged cells, a *run_export.merge* script is available. Both are example scripts that are tailored to Mentor Graphics 24nm test library and need to be adapted to your design.

Inter-Cell Data File	422
The Marker File.....	424

Inter-Cell Data File

One of the output files from cell pair extraction is the inter-cell data file. This file starts with a summary of the applied command line options followed by a distribution graph, displaying the number of required Tessent CellModelGen runs in relation to the total bridging length (TBL) of the cell pairs.

Both are also shown in the Tessent log file. See the explanation in the [extract_inter_cell_data](#) command description in the *Tessent Shell Reference Manual*.

This is then followed by the UDFM section for encrypted location aliases:

```
UDFM {
    Version : 3;
    EncryptedLocationAliases {

        "zIYEAAAAAAAfkAAAAADnvDUR+Jcd6rRtM7lQRHQsXJDc2kkvY4ro3CSnVikIkofzRxsBi
        +dTlpgwP",
        "e/maKK6Xwn7P667RcMJKX/
        nYCrdr3+q7FRYygVRVYYs14jv19Ae1HvnaOu8yt0gvoWP4GBbAJNxYTef",
        "w46GDWpPjbyLVDIiXwb1SV4ydn94YjMB06Iyk0l7bpHTiGNPItgihnm8bUyccbz+Zu/
        5MP1Vod/PIyam",
        "iibsQtGjme3loCvVpnRW+dYNy+yStfVRL+jBrhzxqosnr8cMIY6W6I79/s0xXm0+/
        1zRH29J8i6qMjbi",
        "c1ynIVpmQEtoT7wEpj+/
        Kb3m3J+Hdsrw1JplWkk1ZwVisLBNoaHqsWo274zpGibYbJqTTniWLQuxsbyI",
        "PUn0o3AQfHMKBEDGJB1sRbVnR8oRp0b8ofbKp+dnahPv+KAqwNNH61mBYb1A2EwFT5ZfJKh
        A1z+pjcN",
        "ZaKh5LHY6w6iRzeEReXwWQsqFLxdFzp+sbyGTqpptz4KJvuWUpwP3kRTZe2fVBiA37WMZsT
        9X8v4div",
        "OyGTwqdIaa/
        8FhqSPYwhhteQrt0Gron28yhhEAshUXu20ddBGCMIdt6cayplcgdpv2P7py48e8T7rcJi",
        "AB1183gbwXtzxTiuVngQvOGrpK5VomsimPr79XQJpGARGFhE2MiZgpJ/+56ydDx";
    }
}
```

The UDFM Section for Inter-Cell Bridges

The encrypted section is followed by a readable part containing the data for the extracted cell pairs.

```

UdfmType("inter_cell_bridges") {
    VirtualModule("sffq_01_std_thn_dnd_N_sffq_01_std_thn_dnd_FS_top_0_4320") { // #loc: 9
        tbl: 528.000000 tl
        CellInfo("C1") {Cell:"sffq_01_std_thn_dnd"; Ori:"N"; X:0.000000e+00; Y:0.000000e+00;
                      XL:1.056000e-05; YL:4.320000e-06; }
        CellInfo("C2") {Cell:"sffq_01_std_thn_dnd"; Ori:"FS"; X:0.000000e+00; Y:4.320000e-06;
                      XL:1.056000e-05; YL:4.320000e-06; }
        DesignLocations {
            Location {C1:$N406; C2:$N407; Ori:"N"; X:-1.842320e-03; Y:-2.174760e-03; }
            Location {C1:$N174; C2:$N249; Ori:"FN"; X:-2.534960e-03; Y:1.505880e-03; }
            Location {C1:$N408; C2:$N409; Ori:"FN"; X:-1.100000e-04; Y:2.378520e-03; }
            Location {C1:$N120; C2:$N410; Ori:"FN"; X:-2.615120e-03; Y:1.324440e-03; }
            Location {C1:$N411; C2:$N204; Ori:"FN"; X:-2.125040e-03; Y:-2.278440e-03; }
            Location {C1:$N135; C2:$N121; Ori:"FN"; X:-2.047760e-03; Y:-1.449000e-03; }
            Location {C1:$N412; C2:$N62; Ori:"FN"; X:-2.577680e-03; Y:-2.131560e-03; }
            Location {C1:$N258; C2:$N413; Ori:"FN"; X:-2.330000e-03; Y:-1.742760e-03; }
            Location {C1:$N67; C2:$N257; Ori:"FN"; X:-2.333840e-03; Y:-1.751400e-03; }
        }
    }
    VirtualModule("sffq_01_std_thn_dnd_FS_sffq_01_std_thn_dnd_N_top_0_4320") { // #loc: 6
        tbl: 352.000000 tl
        CellInfo("C1") {Cell:"sffq_01_std_thn_dnd"; Ori:"FS"; X:0.000000e+00; Y:0.000000e+00;
                      XL:1.056000e-05; YL:4.320000e-06; }
        CellInfo("C2") {Cell:"sffq_01_std_thn_dnd"; Ori:"N"; X:0.000000e+00; Y:4.320000e-06;
                      XL:1.056000e-05; YL:4.320000e-06; }
        DesignLocations {
            Location {C1:$N216; C2:$N217; Ori:"N"; X:-1.950320e-03; Y:2.486520e-03; }
            Location {C1:$N218; C2:$N219; Ori:"N"; X:-2.469200e-03; Y:-2.351880e-03; }
            Location {C1:$N220; C2:$N221; Ori:"N"; X:-1.210400e-04; Y:-2.248200e-03; }
            Location {C1:$N222; C2:$N223; Ori:"FN"; X:-2.681840e-03; Y:1.276920e-03; }
            Location {C1:$N224; C2:$N225; Ori:"N"; X:7.141600e-04; Y:2.598840e-03; }
            Location {C1:$N226; C2:$N227; Ori:"N"; X:-2.626160e-03; Y:1.389240e-03; }
        }
    }
}

.

.

VirtualModule("sffq_01_std_thn_dnd_N_sffq_01_std_thn_dnd_FS_top_n10080_4320") { // #loc: 1
    tbl: 2.666667 tl
    CellInfo("C1") {Cell:"sffq_01_std_thn_dnd"; Ori:"N"; X:1.008000e-05; Y:0.000000e+00;
                  XL:1.056000e-05; YL:4.320000e-06; }
    CellInfo("C2") {Cell:"sffq_01_std_thn_dnd"; Ori:"FS"; X:0.000000e+00; Y:4.320000e-06;
                  XL:1.056000e-05; YL:4.320000e-06; }
    DesignLocations {
        Location {C1:$N148; C2:$N401; Ori:"N"; X:-2.162000e-03; Y:-2.157480e-03; }
    }
}
}
}

```

The top lines show the cell pairs with the largest total bridging length (TBL); the bottom lines show the cell pairs with the smallest TBL.

Each cell pair information is headed by the merged cell name of the cell pair, followed by the number of occurrences of that cell pair (#loc) and the TBL in technology lengths (tl).

The “CellInfo” lines for cell1 (C1) and cell2 (C2) contain the following information:

- **Cell** — This is the cell name of the related cell.
- **Ori** — This is the orientation of that cell within the bounding box, for example “FS”— for more information including the orientation types, see the [-orientation switch](#) in the -cell1 switch description in the *Tessent CellModelGen Tool Reference* manual.
- **X, Y** — These are the X- and Y-offsets of the cell within the bounding box, defined in meters—for more information including the types of offsets, see the [-y_offset switch](#) in the -cell1 switch description in the *Tessent CellModelGen Tool Reference* manual.
- **XL, YL** — These are the width and length values of the cell defined in meters.

This section is followed by the “DesignLocations” showing as many lines as there are locations, where each line contains the following information:

- **C1, C2** — These are the location aliases giving the instance name of the corresponding cell.
- **Ori** — This is the orientation of the bounding box within the design, for example “FN”.
- **X, Y** — These are the X/Y coordinates in meters, pointing to the lower-left corner of the bounding box, but taking the orientation of the bounding box into account. That is, when the orientation of the bounding box is “FN”, then the originally lower-left corner is also flipped to be at the lower-right corner.

The Marker File

When running the extract_inter_cell_data command by using the -marker_file switch, also a marker file will be generated, containing layout information for all extracted cell pairs.

[Figure 8-82](#) shows a part of such a marker file with added explanation for each entry in callout boxes.

Figure 8-82. Marker File

```

MyDesign 1000
1 sffq_01_std_thn_dnd_N_sffq_01_std_thn_dnd_FS_top_0_4320
9 9 1 Tessent
Locations 9 TBL 528
# Check text lines (1), tool
p 1 4
1_x1 -1842320
1_y1 -2174760
1_o1 N
2_x1 -1842320}
2_y1 -2170440
2_o1 FS
-1831760 -2174760
-1842320 -2174760
-1842320 -2166120
-1831760 -2166120
p 2 4
1_x1 -2545520
1_y1 1505880
1_o1 FN
2_x1 -2545520
2_y1 1510200
2_o1 S
-2534960 1505880
-2545520 1505880
-2545520 1514520
-2534960 1514520
...
p 9 4
1_x1 -23444400
1_y1 -1751400
1_o1 FN
2_x1 -23444400
2_y1 -1747080
2_o1 S
-2333840 -1751400
-23444400 -1751400
-23444400 -1742760
-2333840 -1742760

```

The diagram illustrates the structure of a marker file. It shows various sections of the file with corresponding annotations:

- Top-cell name:** MyDesign 1000
- # Current DRC results:** 1 sffq_01_std_thn_dnd_N_sffq_01_std_thn_dnd_FS_top_0_4320
- # Original DRC results:** 9 9 1 Tessent
- Polygon, ordinal, # vertices:** Locations 9 TBL 528
- Cell 1 coordinates lower-left corner:** 1_x1 -1842320, 1_y1 -2174760, 1_o1 N
- Cell 1 orientation:** 2_x1 -1842320}, 2_y1 -2170440, 2_o1 FS
- Cell pair number & name:** Cell pair number 1
- # Check text lines (1), tool:** # Check text lines (1), tool
- Total bridging length:** Total bridging length
- # Locations:** # Locations
- Cell 2 coordinates lower-left corner:** Cell 2 coordinates lower-left corner
- Cell 2 orientation:** Cell 2 orientation
- Bounding box coordinate data:** Bounding box coordinate data

In this example, cell pair number 1 occurs 9 times within the design (# Locations). For each occurrence the layout data is listed. The extracted cell pairs are sorted by the total bridging length from largest to smallest. See also [extract_inter_cell_data](#) in the *Tessent Shell Reference Manual*.

Pattern Generation for a Boundary Scan Circuit

This example shows how to create a test set for an IEEE 1149.1 (boundary scan)-based circuit.

The following subsections list and explain the dofile and test procedure file.

About Dofile	426
TAP Controller State Machine.	427
About the Test Procedure File	428

About Dofile

The following dofile shows the commands you could use to specify the scan data in the ATPG tool.

```
add_clocks 0 tck
add_scan_groups grp1 proc_fscan
add_scan_chains chain1 grp1 tdi tdo
add_input_constraints tms -c0
add_input_constraints trstz -c1
set_capture_clock TCK -atpg
```

You must define the tck signal as a clock because it captures data. There is one scan group, grp1, which uses the proc_fscan test procedure file (see page 428). There is one scan chain, chain1, that belongs to the scan group. The input and output of the scan chain are tdi and tdo, respectively.

The listed pin constraints only constrain the signals to the specified values during ATPG—not during the test procedures. Thus, the tool constrains tms to a 0 during ATPG (for proper pattern generation), but not within the test procedures, where the signal transitions the TAP controller state machine for testing. This outlines the basic scan testing process:

1. Initialize scan chain.
2. Apply PI values.
3. Measure PO values.
4. Pulse capture clock.
5. Unload scan chain.

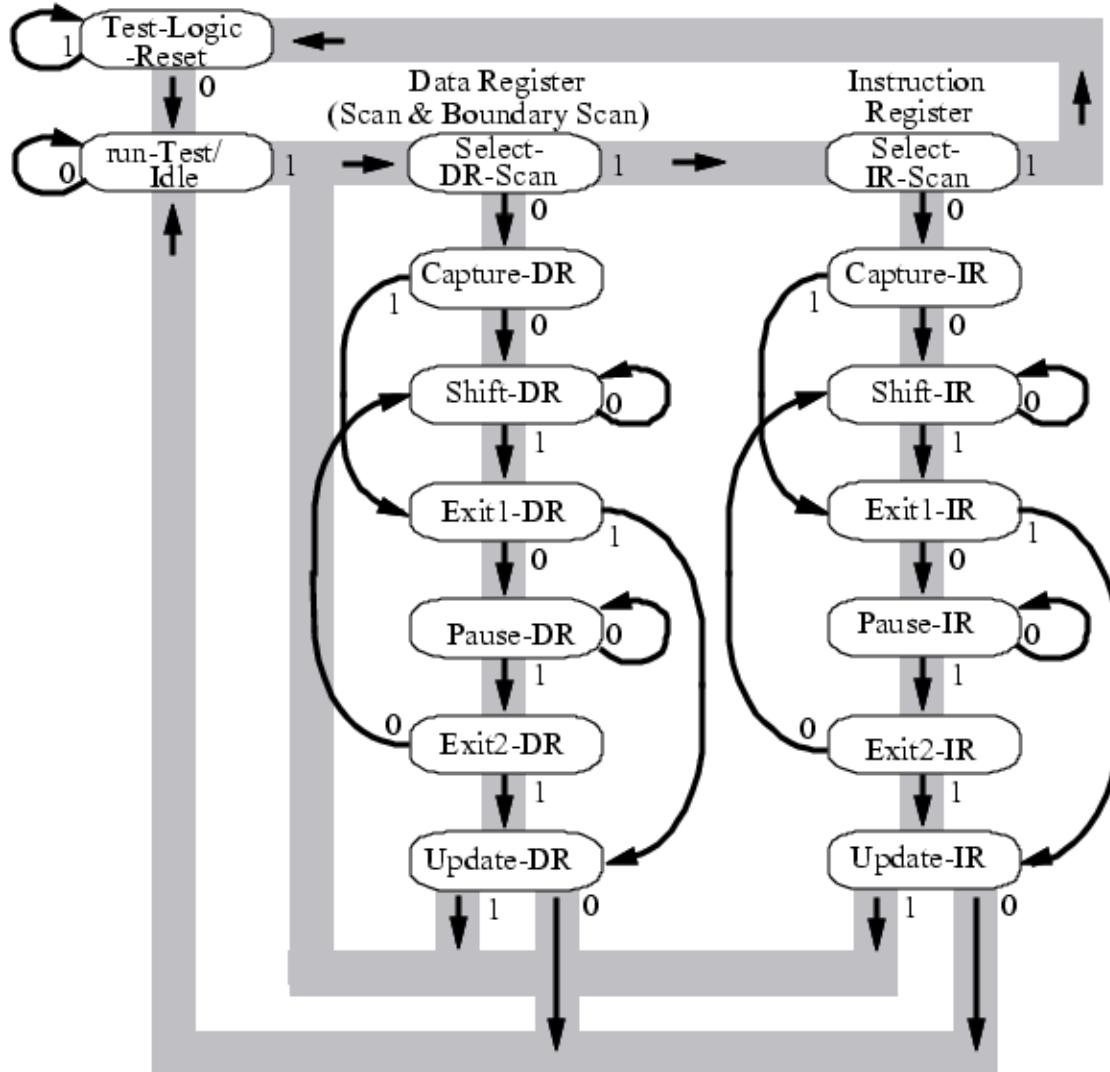
During Step 2, you must constrain tms to 0 so that the Tap controller’s finite state machine (Figure 8-83) can go to the Shift-DR state when you pulse the capture clock (tck). You constrain the trstz signal to its off-state for the same reason. If you do not do this, the Tap controller goes to the Test-Logic-reset_state at the end of the Capture-DR sequence.

The `set_capture_clock` TCK -ATPG command defines tck as the capture clock and that the capture clock must be used for each pattern (as the ATPG tool is able to create patterns where the capture clock never gets pulsed). This ensures that the Capture-DR state properly transitions to the Shift-DR state.

TAP Controller State Machine

The following shows the finite state machine for the TAP controller of a IEEE 1149.1 circuit.

Figure 8-83. State Diagram of TAP Controller Circuitry



The TMS signal controls the state transitions. The rising edge of the TCK clock captures the TAP controller inputs. You may find this diagram useful when writing your own test procedure file or trying to understand the example test procedure file shown in [About the Test Procedure File](#).

About the Test Procedure File

This is the test procedure file *proc_fscan*.

```
set time scale 1 ns;
set strobe_window time 1;
timeplate tp0 =
    force_pi 100;
    measure_po 200;
    pulse TCK 300 100;
    period 500;
end;

procedure test_setup =
    timeplate tp0;

    // Apply reset procedure
    // Test cycle one

    cycle =
        force TMS 1;
        force TDI 0;
        force TRST 0;
        pulse TCK;
    end;

    // "TMS"=0 change to run-test-idle
    // Test cycle two

    cycle =
        force TMS 0;
        force TRST 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-DR
    // Test cycle three

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=1 change to select-IR
    // Test cycle four

    cycle =
        force TMS 1;
        pulse TCK;
    end;

    // "TMS"=0 change to capture-IR
    // Test cycle five

    cycle =
        force TMS 0;
        pulse TCK;
    end;

    // "TMS"=0 change to shift-IR
    // Test cycle six
```

```
cycle =
    force TMS 0;
    pulse TCK;
end;

// load MULT_SCAN instruction "1000" in IR
// Test cycle seven

cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle eight

cycle =
    force TMS 0;
    pulse TCK;
end;

// Test cycle nine

cycle =
    force TMS 0;
    pulse TCK;
end;

// Last shift in exit-IR Stage
// Test cycle ten

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// Change to shift-dr stage for shifting in data
// "TMS" = 11100
// "TMS"=1 change to update-IR state
// Test cycle eleven

cycle =
    force TMS 1;
    force TDI 1;
    pulse TCK;
end;

// "TMS"=1 change to select-DR state
// Test cycle twelve

cycle =
    force TMS 1;
    pulse TCK;
end;

// "TMS"=0 change to capture-DR state
// Test cycle thirteen
```

```
cycle =
    force TMS 0;
    pulse TCK;
end;

// "TMS"=0 change to shift-DR state
// Test cycle fourteen

cycle =
    force TMS 0;
    force TEST_MODE 1;
    force TRST 1;
    pulse TCK;
end;
end;

procedure shift =
    scan_group grp1;
    timeplate tp0;
    cycle =
        force_sci;
        measure_sco;
        pulse TCK;
    end;
end;

procedure load_unload =
    scan_group grp1;
    timeplate tp0;
    cycle =
        force TMS 0;
        force TCK 0;
    end;
    apply shift 77;

// "TMS"=1 change to exit-1-DR state

cycle =
    force TMS 1;
end;
apply shift 1;

// "TMS"=1 change to update-DR state

cycle =
    force TMS 1;
    pulse TCK;
end;

// "TMS"=1 change to select-DR-scan state

cycle =
    force TMS 1;
    pulse TCK;
end;

// "TMS"=0 change to capture-DR state
```

```
cycle =  
    force TMS 0;  
    pulse TCK;  
end;  
end;
```

Upon completion of the **test_setup** procedure, the tap controller is in the shift-DR state in preparation for loading the scan chain(s). It is then placed back into the shift-DR state for the next scan cycle. This is achieved by the following:

- The items that result in the correct behavior are the pin constraint on tms of C1 and the fact that the capture clock has been specified as TCK.
- At the end of the **load_unload** procedure, the tool asserts the pin constraint on TMS, which forces tms to 0.
- The capture clock (TCK) occurs for the cycle and this results in the tap controller cycling from the run-test-idle to the Select-DR-Scan state.
- The **load_unload** procedure is again applied. This will start the next load/unloading the scan chain.

The first procedure in the test procedure file is **test_setup**. This procedure begins by resetting the test circuitry by forcing trstz to 0. The next set of actions moves the state machine to the Shift-IR state to load the instruction register with the internal scan instruction code (1000) for the MULT_SCAN instruction. This is accomplished by shifting in 3 bits of data (tdi=0 for three cycles) with tms=0, and the 4th bit (tdi=1 for one cycle) when tms=1 (at the transition to the Exit1-IR state). The next move is to sequence the TAP to the Shift-DR state to prepare for internal scan testing.

The second procedure in the test procedure file is **shift**. This procedure forces the scan inputs, measures the scan outputs, and pulses the clock. Because the output data transitions on the falling edge of tck, the measure_sco command at time 0 occurs as tck is falling. The result is a rules violation unless you increase the period of the **shift** procedure so tck has adequate time to transition to 0 before repeating the shift. The **load_unload** procedure, which is next in the file, calls the **shift** procedure.

This is the basic flow of the **load_unload** procedure:

1. Force circuit stability (all clocks off, etc.).
2. Apply the **shift** procedure n-1 times with tms=0
3. Apply the **shift** procedure one more time with tms=1
4. Set the TAP controller to the Capture-DR state.

The **load_unload** procedure inactivates the reset mechanisms, because you cannot assume they hold their values from the **test_setup** procedure. It then applies the **shift** procedure 77 times with tms=0 and once more with tms=1 (one shift for each of the 77 scan registers within the

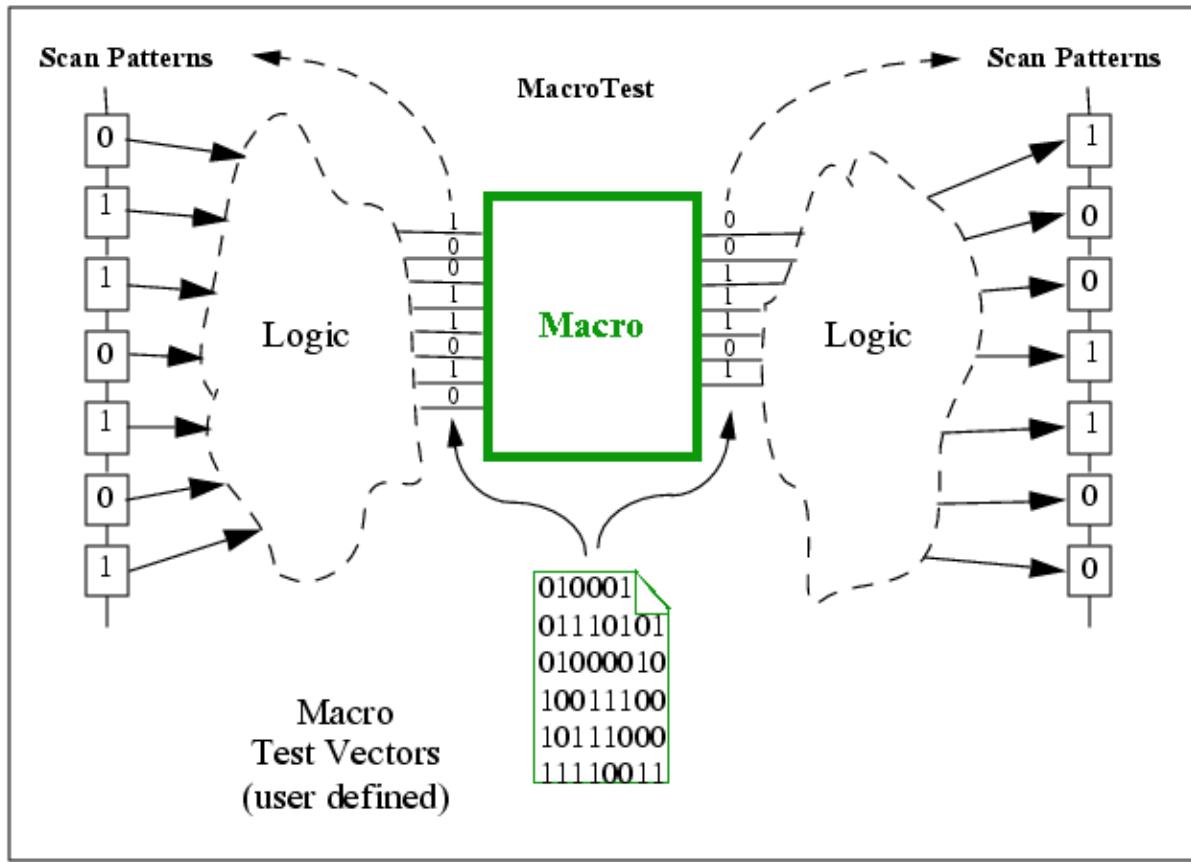
design). The procedure then sequences through the states to return to the Capture-DR state. You must also set tck to 0 to meet the requirement that all clocks be off at the end of the procedure.

MacroTest Overview

MacroTest is a utility that helps automate the testing of embedded logic and memories (macros) by automatically translating user-defined patterns for the macros into scan patterns. Because it enables you to apply your macro test vectors in the embedded environment, MacroTest improves overall IC test quality. It is particularly useful for testing small RAMs and embedded memories but can also be used for a disjoint set of internal sites or a single block of hardware represented by an instance in HDL.

This is illustrated conceptually in [Figure 8-84](#).

Figure 8-84. Conceptual View of MacroTest



MacroTest provides the following capabilities and features:

- Supports user-selected scan observation points.
- Supports synchronous memories; for example, supports positive (or negative) edge-triggered memories embedded between positive (or negative) edge-triggered scan chains.
- Enables you to test multiple macros in parallel.

- Allows you to define macro output values that do not require observation.
- Has no impact on area or performance.

The MacroTest Process Flow	435
Macro Qualification for MacroTest	437
When to Use MacroTest	438
Macro Boundary Definition	441
Test Values Definition	445
Recommendations for MacroTest Use	447
Macro Test Examples	449

The MacroTest Process Flow

The MacroTest flow requires a set of patterns and MacroTest. The patterns are a sequence of tests (inputs and expected outputs) that you develop to test the macro. To use MacroTest effectively, you need to be familiar with two commands.

- **set_mactest_options** — Modifies two rules of the DRCs to allow otherwise illegal circuits to be processed by MacroTest. Black box (unmodeled) macros may require this command.
- **macrotest** — Runs the MacroTest utility to read functional patterns you provide and convert them into scan-based manufacturing test patterns.

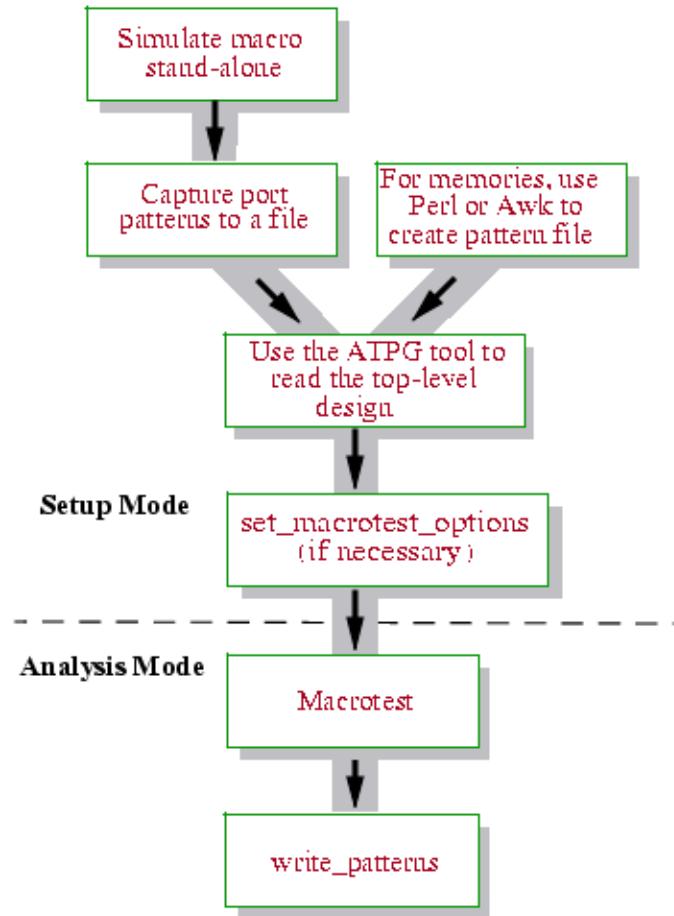
For a memory, this is a sequence of writes and reads. You may need to take embedding restrictions into account as you develop your patterns. Next, you set up and run MacroTest to convert these cycle-based patterns into scan-based test patterns. The converted patterns, when applied to the chip, reproduce your input sequence at the macro's inputs through the intervening logic. The converted patterns also ensure that the macro's output sequence is as you specified in your set of patterns.

Note

 You can generate a wide range of pattern sets: From simple patterns that verify basic functionality, to complex, modified March algorithms that exercise every address location multiple times. Some embeddings (the logic surrounding the macro) do not allow arbitrary sequences, however.

[Figure 8-85](#) shows the basic flow for creating scan-based test patterns with MacroTest.

Figure 8-85. Basic Scan Pattern Creation Flow with MacroTest



Note

The patterns produced by MacroTest cannot be read back into the ATPG tool. This is because the simulation and assumptions about the original macro patterns are no longer valid and the original macro patterns are not preserved in the MacroTest patterns.

Tessent Diagnosis, a Mentor Graphics test failure diagnosis tool, also cannot read a pattern set that includes MacroTest patterns. If Tessent Diagnosis is used in your manufacturing test flow, save MacroTest patterns separately from your other patterns. This will enable a test engineer to remove them from the set of patterns applied on ATE before attempting to read that set into Tessent Diagnosis for diagnosis.

When you run the macrotest command, MacroTest reads your pattern file and begins analyzing the patterns. For each pattern, the tool searches back from each of the macro's inputs to find a scan flip-flop or primary input. Likewise, the tool analyzes observation points for the macro's output ports. When it has justified and recorded all macro input values and output values, MacroTest moves on to the next pattern and repeats the process until it has converted all the patterns. The default MacroTest effort exhaustively tries to convert all patterns. If successful,

then the set of scan test patterns MacroTest creates will detect any defect inside the macro that changes any macro output from the expected value.

Note

 If you add faults prior to running MacroTest, the ATPG tool automatically fault simulates the patterns as they are created. This is time consuming, but is retained for backward compatibility. It is advised that you generate and save macrotest patterns in a separate run from normal ATPG and faultsim and that you not issue the add_faults command in the MacroTest run.

The patterns you supply to MacroTest must be consistent with the macro surroundings (embedding) to assure success. In addition, the macro must meet certain design requirements. The following sections detail these requirements, describe how and when to use MacroTest, and conclude with some examples.

Macro Qualification for MacroTest

If a design meets three basic criteria, then you can use MacroTest to convert a sequence of functional cycles (that describe I/O behavior at the macro boundary) into a sequence of scan patterns.

- The design has at least one combinational observation path for each macro output pin that requires observation (usually all outputs).
- All I/O of the RAM/macro block to be controlled or observed are unidirectional.
- The macro/block can hold its state while the scan chain shifts, if the test patterns require that the state be held across patterns. This is the case for a March algorithm, for example.

If you write data to a RAM macro (RAM), for example, then later read the data from the RAM, typically you will need to use one scan pattern to do the write, and a different scan pattern to do the read. Each scan pattern has a load/unload that shifts the scan chain, and you must ensure that the DFT was inserted, if necessary, to allow the scan chain to be shifted without writing into the RAM. If the shift clock can also cause the RAM to write and there is no way to protect the RAM, then it is very likely that the RAM contents will be destroyed during shift; the data written in the early pattern will not be preserved for reading during the latter pattern. Only if it is truly possible to do a write followed by a read, all in one scan pattern, then you may be able to use MacroTest even with an unprotected RAM.

Because converting such a multicycle pattern is a sequential ATPG search problem, success is not guaranteed even if success is possible. Therefore, you should try to convert a few patterns before you depend on MacroTest to be able to successfully convert a given embedded macro. This is a good idea even for combinational conversions.

If you intend to convert a sequence of functional cycles to a sequence of scan patterns, you can insert the DFT to protect the RAM during shift: The RAM should have a write enable that is PI-controllable throughout test mode to prevent destroying the state of the RAM. This ensures the tool can create a state inside the macro and retain the state during the scan loading of the next functional cycle (the next scan pattern after conversion by MacroTest).

The easiest case to identify is where the ATPG tool issues a message saying it can use the RAM test mode, RAM_SEQUENTIAL. This message occurs because the tool can independently operate the scan chains and the RAM. The tool can operate the scan chain without changing the state of the macro as well as operate the macro without changing the state loaded into the scan chain. This allows the most flexibility for ATPG, but the most DFT also.

However, there are cases where the tool can operate the scan chain without disturbing the macro, while the opposite is not true. If the scan cells are affected or updated when the macro is operated (usually because a single clock captures values into the scan chain and is also an input into the macro), the tool cannot use RAM_SEQUENTIAL mode. Instead, the tool can use a sequential MacroTest pattern (multiple cycles per scan load), or it can use multiple single cycle patterns if the user's patterns keep the write enable or write clock turned off during shift.

For example, suppose a RAM has a write enable that comes from a PI in test mode. This makes it possible to retain written values in the RAM during shift. However, it also has a single edge-triggered read control signal (no separate read enable) so the RAM's outputs change any time the address lines change followed by a pulse of the read clock/strobe. The read clock is a shared clock and is also used as the scan clock to shift the scan chains (composed of MUX scan cells). In this case, it is not possible to load the scan chains without changing the read values on the output of the macro. For this example, you will need to describe a sequential read operation to MacroTest. This can be a two-cycle operation. In the first cycle, MacroTest pulses the read clock. In the second cycle, MacroTest observes and captures the macro outputs into the downstream scan cells. This works because there is no intervening scan shift to change the values on the macro's output pins. If a PI-controllable read enable existed, or if you used a non-shift clock (clocked scan and LSSD have separate shift and capture clocks), an intervening scan load could occur between the pulse of the read clock and the capture of the output data. This is possible because the macro read port does not have to be clocked while shifting the scan chain.

When to Use MacroTest

MacroTest is primarily used to test small memories (register file, cache, FIFO, and so on). Although the ATPG tool can test the faults at the boundaries of such devices, and can propagate the fault effects through them (using the _ram or _cram primitives), it does not attempt to create a set of patterns to test them internally. This is consistent with how it treats all primitives. Because memory primitives are far more complex than a typical primitive (such as a NAND gate), you may prefer to augment ATPG tool patterns with patterns that you create to test the internals of the more complex memory primitives. Such complex primitives are usually packaged as models in the ATPG library, or as HDL modules that are given the generic name "macro."

Note

Although the ATPG library has specific higher level collections of models called macros, MacroTest is not intended for testing such macros; they are tested by normal ATPG. Only small embedded memories, such as register files, are tested using macrotest. MBIST is recommended as the testing solution for those and all memories.

Here, the term “macro” simply means some block of logic, or even a distributed set of lines that you want to control and observe. You must provide the input values and expected output values for the macro. Typically you are given, or must create, a set of tests. You can then simulate these tests in some time-based simulator, and use the results predicted by that simulator as the expected outputs of the macro. For memories, you can almost always create both the inputs and expected outputs without any time-based simulation. For example, you might create a test that writes a value, V, to each address. It is trivial to predict that when subsequent memory reads occur, the expected output value will be V.

MacroTest converts these functional patterns to scan patterns that can test the device after it is embedded in systems (where its inputs and outputs are not directly accessible, and so the tests cannot be directly applied and observed). For example, a single macro input enable might be the output of two enables which are ANDed outside the macro. The tests must be converted so that the inputs of the AND are values which cause the AND’s output to have the correct value at the single macro enable input (the value specified by the user as the macro input value). MacroTest converts the tests (provided in a file) and provides the inputs to the macro as specified in the file, and then observes the outputs of the macro specified in the file. If a particular macro output is specified as having an expected 0 (or 1) output, and this output is a data input to a MUX between the macro output and the scan chain, the select input of that MUX must have the appropriate value to propagate the macro’s output value to the scan chain for observation. MacroTest automatically selects the path(s) from the macro output(s) to the scan chain(s), and delivers the values necessary for observation, such as the MUX select input value in this case.

Often, each row of a MacroTest file converts to a single 1-system cycle scan test (sometimes called a basic scan pattern in the ATPG tool). A scan chain load, PI assertion, output measure, clock pulse, and scan chain unload result for each row of the file if you specify such patterns. To specify a write with no expected known outputs, specify the values to apply at the inputs to the device and give X output values (don’t care or don’t measure). To specify a read with expected known outputs, specify both the inputs to apply, and the outputs that are expected (as a result of those and all prior inputs applied in the file so far). For example, an address and read enable would have specified inputs, whereas the data inputs could be X (don’t care) for a memory read.

Mentor Graphics highly recommends that you not over-specify patterns. It may be impossible, due to the surrounding logic, to justify all inputs otherwise. For example, if the memory has a write clock and write enable, and is embedded in a way that the write enable is independent but the clock is shared with other memories, it is best to turn off the write using the write enable, and leave the clock X so it can be asserted or de-asserted as needed. If the clock is turned off instead of the write enable, and the clock is shared with the scan chain, it is not possible to pulse the shared clock to capture and observe the outputs during a memory read. If instead, the write

enable is shared and the memory has its own clock (not likely, but used for illustration), then it is best to turn off the write with the clock and leave the shared write enable X.

Realize that although the scan tests produced appear to be independent tests, the tool assumes that the sequence being converted has dependencies from one cycle to the next. Thus, the scan patterns have dependencies from one scan test to the next. Because this is atypical, the tool marks MacroTest patterns as such, and you must save such MacroTest patterns using the write_patterns command. The MacroTest patterns cannot be reordered or reduced using compress_patterns; reading back MacroTest patterns is not allowed for that reason. You must preserve the sequence of MacroTest patterns as a complete, ordered set, all the way to the tester, if the assumption of cycle-to-cycle dependencies in the original functional sequence is correct.

To illustrate, if you write a value to an address, and then read the value in a subsequent scan pattern, this will work as long as you preserve the original pattern sequence. If the patterns are reordered, and the read occurs before the write, the patterns will then mismatch during simulation or fail on the tester. The reason is that the reordered scan patterns try to read the data before it has been written. This is untrue of all other ATPG tool patterns. They are independent and can be reordered (for example, to allow pattern compaction to reduce test set size). MacroTest patterns are never reordered or reduced, and the number of input patterns directly determines the number of output patterns.

Macro Boundary Definition

The macro boundary is typically defined by its instance name with the macrotest command. If no instance name is given, then the macro boundary is defined by a list of hierarchical pin names (one per macro pin) given in the header of the MacroTest patterns file.

Macro Boundary Definition by Instance Name	441
Macro Boundary Definition Without Using an Instance Name	442
Observation Site Specification and Reporting	443
Macro Boundary Definition With Trailing Edge Inputs	443

Macro Boundary Definition by Instance Name

The macro is a particular instance, almost always represented by a top-level model in the ATPG library. More than one instance may occur in the netlist, but each instance has a unique name that identifies it. Therefore, the instance name is all that is needed to define the macro boundary.

The definition of the instance/macro is accessed to determine the pin order as defined in the port list of the definition. MacroTest expects that pin order to be used in the file specifying the I/O (input and expected output) values for the macro (the tests). For example, the command:

```
macrotest regfile_8 file_with_tests
```

would specify for MacroTest to find the instance “regfile_8”, look up its model definition, and record the name and position of each pin in the port list. Given that the netlist is written in Verilog, with the command:

```
regfile_definition_name regfile_8 (net1, net2, ... );
```

the portlist of regfile_definition_name (not the instance port list “net1, net2, …”) is used to get the pin names, directions, and the ordering expected in the test file, file_with_tests. If the library definition is:

```
model "regfile_definition_name"
  ("Dout_0", "Dout_1", Addr_0, "Addr_1", "Write_enable", ...)
  ( input ("Addr_0") () ... output ("Dout_0") () ... )
```

then MacroTest knows to expect the output value Dout_0 as the first value (character) mentioned in each row (test) of the file, file_with_tests. The output Dout_1 should be the 2nd pin, input pin Addr_0 should be the 3rd pin value encountered, etc. If it is inconvenient to use this ordering, the ordering can be changed at the top of the test file, file_with_tests. This can be done using the following syntax:

```
macro_inputs Addr_0  Addr_1
macro_output Dout_1
macro_inputs Write_enable
...
end
```

which would cause MacroTest to expect the value for input Addr_0 to be the first value in each test, followed by the value for input Addr_1, the expected output value for Dout_1, the input value for Write_enable, and so on.

Note

 Only the pin names need be specified, because the instance name “regfile_8” was given for the macrotest command.

Macro Boundary Definition Without Using an Instance Name

If an instance name is not given with the macrotest command, then you must provide an entire hierarchical path/pin name for each pin of the macro. This is given in the header of the MacroTest patterns file.

There must be one name per data bit in the data (test values) section which follows the header. For example:

```
macro_inputs regfile_8/Addr_0regfile_8/Addr_1
macro_outputregfile_8/Dout_1
macro_inputsregfile_8/write_enable
...
end
```

The code sample defines the same macro boundary as was previously defined for regfile_8 using only pin names to illustrate the format. Because the macro is a single instance, this would not normally be done, because the instance name is repeated for each pin. However, you can use this entire pathname form to define a distributed macro that covers pieces of different instances. This more general form of boundary definition allows a macro to be any set of pins at any level(s) of hierarchy down to the top library model. If you use names which are inside a model in the library, the pin pathname must exist in the flattened data structures. (In other words, it must be inside a model where all instances have names, and it must be a fault site, because these are the requirements for a name inside a model to be preserved in the tool).

This full path/pin name form of “macro boundary” definition is a way to treat any set of pins/wires in the design as points to be controlled, and any set of pins/wires in the design as points to be observed. For example, some pin might be defined as a macro_input which is then given {0,1} values for some patterns, but X for others. In some sense, this “macro input” can be thought of as a programmable ATPG constraint (see [add_atpg_constraints](#)), whose value can be changed on a pattern by pattern basis. There is no requirement that inputs be connected to outputs. It would even be possible to define a distributed macro such that the “output” is really the input to an inverter, and the “input” is really the output of the same inverter. If you specified that the input = 0, and the expected output = 1, MacroTest would ensure that the macro “input” was 0 (so the inverter output is 0, and its input is 1), and would sensitize the input of the inverter to some scan cell in a scan chain. Although this is indeed strange, it is included to emphasize the

point that full path/pin forms of macro boundary definition are completely flexible and unrelated to netlist boundaries or connectivity. Any set of connected or disjoint points can be inputs and/or outputs.

Observation Site Specification and Reporting

You can report the set of possible observation sites using the macrotest command switch, -Report_observation_candidates. This switch reports, for each macro output, the reachable scan cells and whether the scan cell is already known to be unable to capture/observe. Usually, all reachable scan cells can capture, so all are reported as possible observation sites. The report gives the full instance name of the scan cell's memory element, and its gate id (which follows the name and is surrounded by parentheses).

Although rarely done, you can specify for one macro output at a time exactly which of those reported scan cells is to be used to observe that particular macro output pin. Any subset can be so specified. For example, if you want to force macro output pin Dout_1 to be observed at one of its reported observation sites, such as “/top/middle/bottom/ (13125)”, then you can specify this as follows:

```
macro_output regfile_8/Dout_1
observe_at13125
```

Note

 There can be only one macro_output statement on the line above the observe_at directive. Also, you must specify only one observe_at site, which is always associated with the single macro_output line that precedes it. If a macro_input line immediately precedes the observe_at line, MacroTest will issue an error message and exit.

The preceding example uses the gate id (number in parentheses in the -Report output) to specify the scan cell DFF to observe at, but you can also use the instance pathname. Instances inside models may not have unique names, so the gate id is always an unambiguous way to specify exactly where to observe. If you use the full name and the name does not exactly match, the tool selects the closest match from the reported candidate observation sites. The tool also warns you that an exact match did not occur and specifies the observation site that it selected.

Macro Boundary Definition With Trailing Edge Inputs

MacroTest treats macros as black boxes, even if modeled, so do not assume that this information will be gathered using connectivity. Assuming nothing is known about the macro's internals, MacroTest forces the user-specified expected outputs onto the macro outputs for each pattern. This allows black-boxed macros to be used, or you to create models for normal ATPG using the _cram primitive, but treat the macro as a black box for internal testing. A _cram primitive may be adequate for passing data through a RAM, for example, but not for modeling it

for internal faults. MacroTest trusts the output values you provide regardless of what would normally be calculated in the tool, allowing you to specify outputs for these and other situations.

Due to its black box treatment of even modeled RAMs/macros, MacroTest must sometimes get additional information from you. MacroTest assumes that all macro inputs capture on the leading edge of any clock that reaches them. So, for a negative pulse, MacroTest assumes that the leading (falling) edge causes the write into the macro, whereas for a positive pulse, MacroTest assumes that the leading (rising) edge causes the write. If these assumptions are not true, you must specify which data or address inputs (if such pins occur) are latched into the macro on a trailing edge.

Occasionally, a circuit uses leading DFF updates followed by trailing edge writes to the memory driven by those DFFs. For trailing edge macro inputs, you must indicate that the leading edge assumption does not hold for any input pin value that must be presented to the macro for processing on the trailing edge. For a macro which models a RAM with a trailing edge write, you must specify this fact for the write address and data inputs to the macro which are associated with the falling edge write. To specify the trailing edge input, you must use a boundary description which lists the macro's pins (you cannot use the instance name only form).

Regardless of whether you use just pin names or full path/pin names, you can replace “macro_inputs” with “te_macro_inputs” to indicate that the inputs that follow must have their values available for the trailing edge of the shared clock. This allows MacroTest to ensure that the values arrive at the macro input in time for the trailing edge, and also that the values are not overwritten by any leading edge DFF or latch updates. If a leading edge DFF drives the trailing edge macro input pin, the value needed at the macro input will be obtained from the D input side of the DFF rather than its Q output. The leading edge will make Q=D at the DFF, and then that new value will propagate to the macro input and be waiting for the trailing edge to use. Without the user specification as a trailing edge input, MacroTest would obtain the needed input value from the Q output of the DFF. This is because MacroTest would assume that the leading edge of the clock would write to the macro before the leading edge DFF could update and propagate the new value to the macro input.

It is not necessary to specify leading edge macro inputs because this is the default behavior. It is also unnecessary to indicate leading or trailing edges for macro outputs. You can control the cycle in which macro outputs are captured. This ensures that the tool correctly handles any combination of macro outputs and capturing scan cells as long as all scan cells are of the same polarity (all leading edge capture/observe or all trailing edge capture/observe).

In the rare case that a particular macro output could be captured into either a leading or a trailing edge scan cell, you must specify which you prefer by using the -Le_observation_only switch or -Te_observation_only switch with the macrotest command for that macro. For more information on these switches, see “[Example 3 — Using Leading Edge & Trailing Edge Observation Only](#)” and the **macrotest** description in the *Tessent Shell Reference Manual*.

An example of the TE macro input declaration follows:

```
macro_input clock
te_macro_inputs Addr_0  Addr_1 // TE write address inputs
macro_output Dout_1
...
end
```

Test Values Definition

It is important to be familiar with the elements contained in the test file.

There are four different elements than can be in the test file:

- Comments (a line starting with “//” or #)
- Blank lines
- An optional pin reordering section (which must come before any values) that begins with “MACRO_INPutS” or “MACRO_OUTPutS” and ends with “END”
- The tests (one cycle per row of the file)

Normal (nonpulseable) input pin values include {0,1,X,Z}. Some macro inputs may be driven by PIs declared as pulseable pins (add_clocks, add_read_controls, and add_write_controls specify these pins in the tool). These pins can have values from {P,N} where P designates a positive pulse and N designates a negative pulse. Although you can specify a P or N on any pin, the tool issues a warning if it cannot verify that the pin connects to a pulseable primary input (PI). If the tool can pulse the control and cause a pulse at that macro pin, then the pulse will occur. If it cannot, the pulse will not occur. Users are warned if they specify the wrong polarity of pulse (an N, for example, when there is a direct, non-inverting connection to a clock PI that has been specified with an off value of 0, which means that it can only be pulsed positively). A P would need to be specified in such a case, and some macro inputs would probably have to be specified as te_macro_inputs since the N was probably used due to a negative edge macro. P and N denote the actual pulse, not the triggering edge of the macro. It is the embedding that determines whether a P or N can be produced.

Note

 It is the declaration of the PI pin driving the macro input, not any declaration of the macro input itself, which determines whether a pin can be pulsed in the tool.

Normal observable output values include {L,H}, which are analogous to {0,1}. L represents output 0, and H represents output 1. You can give X as an output value to indicate Don't Compare, and F for a Floating output (output Z). Neither a Z nor an X output value will be observed. Occasionally an output cannot be observed, but must be known in order to prevent bus contention or to allow observation of some other macro output.

If you provide a file with these characters, a check is done to ensure that an input pin gets an input value, and an output pin gets an output value. If an “L” is specified in an input pin position, for example, an error message is issued. This helps detect ordering mismatches between the port list and the test file. If you prefer to use 0 and 1 for both inputs and outputs, then use the -No_L_h switch with the macrotest command:

macrotest regfile_8 file_with_tests -no_L_h

Assuming that the -L_h default is used, the following might be the testfile contents for our example register file, if the default port list pin order is used.

```
// Tests for regfile_definition_name.  
//  
//      W  
//      r  
//      i  
//      t  
//      e  
//  
// DD AA e  
// oo dd n  
// uu dd a  
// tt rr b  
//      l  
// 01 01 e  
  
XX 00 0  
XX 00 1  
HH 00 0
```

The example file above has only comments and data; spaces are used to separate the data into fields for convenience. Each row must have exactly as many value characters as pins mentioned in the original port list of the definition, or the exact number of pins in the header, if pins were specified there. Pins can be left off of an instance if macro_inputs and macro_outputs are specified in the header, so the header names are counted and that count is used unless the instance name only form of macro boundary definition is used (no header names exist).

To specify less than all pins of an instance, omit the pins from the header when reordering the pins. The omitted pins are ignored for purposes of MacroTest. If the correct number of values do not exist on every row, an error occurs and a message is issued.

The following is an example where the address lines are exchanged, and only Dout_0 is to be tested:

```
// Tests for regfile_definition_name testing only Dout_0
macro_output Dout_0
macro_inputs Addr_1 Addr_0 write_enable ...
...
end
//      W
//      r
//      i
//      t
//      e
//
// D AA e
// o dd n
// u dd a
// t rr b
//   l
// 0 10 e

X 00 0
X 00 1
H 00 0
```

It is not necessary to have all macro_inputs together. You can repeat the direction designators as necessary:

```
macro_input write_enable
macro_output Dout_0
macro_inputs Addr_1 Addr_0
macro_outputs Dout_1 ...
...
end
```

Recommendations for MacroTest Use

When using MacroTest, you should begin early in the process. This is because the environment surrounding a regfile or memory may prevent the successful delivery of the original user-specified tests, and Design-for-Test hardware may have to be added to allow the tests to be delivered, or the tests may have to be changed to match the surroundings so that the conversion can occur successfully.

For example, if the write enable line outside the macro is the complement of the read enable line (perhaps due to a line which drives the read enable directly and also fans out to an inverter which drives the write enable), and you specify that both the read enable and write enable pins should be 0 for some test, then MacroTest will be unable to deliver both values. It stops and reports the line of the test file, as well as the input pins and values that cannot be delivered. If you change the enable values in the MacroTest patterns file to always be complementary, MacroTest would then succeed. Alternatively, if you add a MUX to make the enable inputs

independently controllable in test mode and keep the original MacroTest patterns unchanged, MacroTest would use the MUX to control one of the inputs to succeed at delivering the complementary values.

Once MacroTest is successful, you should simulate the resulting MacroTest patterns in a time-based simulator. This verifies that the conversion was correct, and that no timing problems exist. The tool does not simulate the internals of primitives, and therefore relies on the fact that the inputs produced the expected outputs given in the test file. This final simulation ensures that no errors exist due to modeling or simulation details that might differ from one simulator to the next. Normal ATPG tool considerations hold, and it is suggested that DRC violations be treated as they would be treated for a stuck-at fault ATPG run.

To prepare to MacroTest an empty (TieX) macro that needs to be driven by a write control (to allow pulsing of that input pin on the black box), issue the [set_macrotest_options](#) command. This command prevents a G5 DRC violation and allows you to proceed. Also, if a transparent latch (TLA) on the control side of an empty macro is unobservable due to the macro, the [set_macrotest_options](#) command prevents it from becoming a TieX, as would normally occur. Once it becomes a TieX, it is not possible for MacroTest to justify macro values back through the latch. If in doubt, when preparing to MacroTest any black box, issue the [set_macrotest_options](#) command before exiting setup mode. No errors will occur because of this, even if none of the conditions requiring the command exist.

ATPG commands and options apply within MacroTest, including cell constraints, ATPG constraints, clock restrictions (it only pulses one clock per cycle), and others. If MacroTest fails and reports that it aborted, you can use the [set_abort_limit](#) command to get MacroTest to work harder, which may allow MacroTest to succeed. Mentor Graphics recommends that you set a moderate abort limit for a normal MacroTest run, then increase the limit if MacroTest fails and issues a message saying that a higher abort limit might help.

ATPG effort should match the simulation checks for bus contention to prevent MacroTest patterns from being rejected by simulation. Therefore, if you specify [set_contention_check On](#), you should use the [-Atpg](#) option. Normally, if you use [set_contention_check Capture_clock](#), you should use the [-Catpg](#) option instead. Currently, MacroTest does not support the [-Catpg](#) option, so this is not advised. Using the [set_decision_order Random](#) is strongly discouraged. It can mislead the search and diagnosis in MacroTest.

In a MacroTest run, as each row is converted to a test, that test is stored internally (similar to a normal ATPG run). You can save the patterns to write out the tests in a desired format (perhaps Verilog to allow simulation and WGL for a tester). The tool supports the same formats for MacroTest patterns as for patterns generated by a normal ATPG run. However, because MacroTest patterns cannot be reordered, and because the expected macro output values are not saved with the patterns, it is not possible to read macrotest patterns back into the ATPG tool. You should generate Macrotest patterns, then save them in all desired formats.

Note

 The macro_output node in the netlist must not be tied to Z (floating).

Macro Test Examples

The following are examples of using MacroTest.

Example 1 — Basic 1-Cycle Patterns

Verilog Contents:

```
RAM mem1 (.Dout ({ Dout [7] ,Dout [6] ,Dout [5] ,Dout [4] ,Dout [3] ,
Dout [2] , Dout [1] , Dout [0] }), .RdAddr ({ RdAddr[1] , RdAddr[0] } ),
.RdEn ( RdEn ), .Din ({ Din[7] , Din[6] , Din[5] , Din[4] , Din[3] ,
Din[2] , Din[1] , Din[0] } ) , .WrAddr ({ WrAddr[1] , WrAddr[0] } ), .WrEn ( WrEn ));
```

ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
    input (RdAddr,WrAddr) (array = 1 : 0;)
    input (RdEn,WrEn) ()
    input (Din) (array = 7 : 0;)
    output (Dout) (
        array = 7 : 0;
        data_size = 8;
        address_size = 2;
        read_write_conflict = XW;
        primitive = _cram(,
            _write {,,} (WrEn,,WrAddr,Din),
            _read {,,,} (,RdEn,,RdAddr,Dout)
        );
    )
)
```

Note

 Vectors are treated as expanded scalars.

Because Dout is declared as “array 7:0”, the string “Dout” in the port list is equivalent to “Dout<7> Dout<6> Dout<5> Dout<4> Dout<3> Dout<2> Dout<1> Dout<0>”. If the declaration of Dout had been Dout “array 0:7”, then the string “Dout” would be the reverse of the above expansion. Vectors are always allowed in the model definitions. Currently, vectors are not allowed in the macrotest input patterns file, so if you redefine the pin order in the header of that file, scalars must be used. Either “Dout<7>”, “Dout(7)”, or “Dout[7]” can be used to match a bit of a vector.

Dofile Contents:

```
set_system_mode analysis
macrotest mem1 ram_patts2.pat
write_patterns results/pattern2.f -replace
```

Test File Input (ram_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn) (
//   input (RdAddr,WrAddr) (array = 1 : 0;
//   input (RdEn,WrEn) ()
//   input (Din) (array = 7 : 0;
//
//   output (Dout) (
//     array = 7 : 0;
//     data_size = 8;
//     address_size = 2;
//
//   .....
// Write V1 (data vector 1) to address 0.  Data Outputs
// and Read Address are Don't Cares.
XXXXXXXX XX 0 10101010 00 P
// Read V1 from address 0.  Data Inputs and Write Address
// are Don't Cares.
HLHLHLHL 00 1 XXXXXXXX XX 0
XXXXXXXX XX 0 0x010101 01 P    // Write V2 to address 1.
LXLHLHLH 01 1 xxxxxxxx xx 0    // Read V2 from address 1.
```

Converted Test File Output (results/pattern2.f) Contents:

```

... skipping some header information ...
SETUP =
    declare input bus "PI" = "/clk", "/Datsel",
                    "/scanen_early", "/scan_in1", "/scan_en",
.... skipping some declarations ...

declare output bus "PO" = "/scan_out1";
.... skipping some declarations ...

CHAIN_TEST =
    pattern = 0;
    apply "grp1_load" 0 =
        chain "chain1" = "0011001100110011001100";
    end;
    apply "grp1_unload" 1 =
        chain "chain1" = "0011001100110011001100";
    end;
end;

SCAN_TEST =
    pattern = 0 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "011010101000000000000000";
    end;
    force "PI" "001X0XXXXXXXXX" 1;
    pulse "/scanen_early" 2;
    measure "PO" "1" 3;
    pulse "/clk" 4;
    apply "grp1_unload" 5 =
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXXXX";
    end;

    pattern = 1 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "100000000000000000000000";
    end;
    force "PI" "001X0XXXXXXXXX" 1;
    measure "PO" "1" 2;
    pulse "/clk" 3;
    apply "grp1_unload" 4=
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXX10101010";
    end;
... skipping some output ...

SCAN_CELLS =
    scan_group "grp1" =
        scan_chain "chain1" =
            scan_cell = 0  MASTER  FFFF  "/rden_reg/ffdpb0"...
            scan_cell = 1  MASTER  FFFF  "/wren_reg/ffdpb0"...
            scan_cell = 2  MASTER  FFFF  "/datreg1/ffdpb7"...
... skipping some scan cells ...
            scan_cell = 20  MASTER  FFFF  "/doutreg1/ffdpb1"...
            scan_cell = 21  MASTER  FFFF  "/doutreg1/ffdpb0"...
        end;
    end;
end;

```

Example 2— Synchronous Memories (1- & 2-Cycle Patterns)

Verilog Contents:

For this example, the RAM is as before, except a single clock is connected to an edge-triggered read and edge-triggered write pin of the macro to be tested. It is also the clock going to the MUX scan chain. There is also a separate write enable. As a result, it is possible to write using a one-cycle pattern, and then to preserve the data written during shift by turning the write enable off in the shift procedure. However, for this example, a read must be done in two cycles—one to pulse the RAM’s read enable and make the data come out of the RAM, and another to capture that data into the scan chain before shifting changes the RAM’s output values. There is no independent read enable to protect the outputs during shift, so they must be captured before shifting, necessitating a 2-cycle read/observe.

ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (
    input (RdAddr,WrAddr) (array = 1 : 0;)
    input (RdClk,WrEn, WrClk) ()
    input (Din) (array = 7 : 0;)
    output (Dout) (
        array = 7 : 0;
        data_size = 8;
        edge_trigger = rw;
        address_size = 2;
        read_write_conflict = XW;
        primitive = cram(,
            _write {,,} (WrClk,WrEn,WrAddr,Din),
            _read {,,,} (,RdClk,,RdAddr,Dout)
        );
    )
)
```

Note that because the clock is shared, it is important to only specify one of the macro values for RdClk or WrClk, or to make them consistent. X means “Don’t Care” on macro inputs, so it will be used to specify one of the two values in all patterns to ensure that any external embedding can be achieved. It is easier to not over-specify MacroTest patterns, which allows using the patterns without having to discover the dependencies and change the patterns.

Dofile Contents:

```
set_system_mode analysis
macrotest mem1 ram_patts2.pat
write_patterns results/pattern2.f -replace
```

Test File Input (ram_patts2.pat) Contents:

```
// model RAM (Dout, RdAddr, RdClk, Din, WrAddr, WrEn, WrClk) (
//   input (RdAddr,WrAddr) (array = 1 : 0;)
//   input (RdClk,WrEn, WrClk) ()
//   input (Din) (array = 7 : 0;)
//
//   output (Dout) (
//     array = 7 : 0;
//     data_size = 8;
//     edge_trigger = rw;
//     .....
// Write V1 (data vector 1) to address 0.
XXXXXXXXXX XX X 10101010 00 1 P
// Read V1 from address 0 -- next 2 rows (1 row per cycle).
XXXXXXXXX 00 P XXXXXXXXX XX 0 X + // + indicates another cycle.
HLHLHLHL XX X XXXXXXXXX XX 0 X // Values observed this cycle.
XXXXXXXXX XX X 01010101 01 1 P // Write V2 to address 1.
XXXXXXXXX 01 P XXXXXXXXX XX 0 X + // Read V2, address 1, cycle 1.
LHLHLHLH XX X XXXXXXXXX XX 0 X // Read V2, address 1, cycle 2.
```

Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ...
SETUP =
    declare input bus "PI" = "/clk", "/Datsel",
                    "/scanen_early", "/scan_in1", "/scan_en",
.... skipping some declarations ...

declare output bus "PO" = "/scan_out1";
.... skipping some declarations ...

CHAIN_TEST =
    pattern = 0;
    apply "grp1_load" 0 =
        chain "chain1" = "0011001100110011001100";
    end;
    apply "grp1_unload" 1 =
        chain "chain1" = "0011001100110011001100";
    end;
end;

SCAN_TEST =

    pattern = 0 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "011010101000000000000000";
    end;
    force "PI" "001X0XXXXXXXXX" 1;
    pulse "/scanen_early" 2;
    measure "PO" "1" 3;
    pulse "/clk" 4;
    apply "grp1_unload" 5 =
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXXXX";
    end;

    pattern = 1 macrotest ;
    apply "grp1_load" 0 =
        chain "chain1" = "100000000000000000000000";
    end;
    force "PI" "001X0XXXXXXXXX" 1;
    pulse "/clk" 2;
    force "PI" "001X0XXXXXXXXX" 3;
    measure "PO" "1" 4;
    pulse "/clk" 5;
    apply "grp1_unload" 6=
        chain "chain1" = "XXXXXXXXXXXXXXXXXXXX10101010";
    end;
... skipping some output ...
```

Example 3 — Using Leading Edge & Trailing Edge Observation Only

Assume that a clock with an off value of 0 (positive pulse) is connected through buffers to a rising edge read input of a macro, and also to both rising and falling edge D flip-flops. Either of the flip-flops can capture the macro's output values for observation. If you specify that the outputs should be captured in the same cycle as the read pulse, then this will definitely occur if you invoke MacroTest with the -Te_observation_only switch because only the trailing edge (TE) flip-flops will be selected for observation. The rising edge of the clock triggers the macro's

read, the values propagate to the scan cells in that same cycle, and then the falling edge of the clock captures those values in the TE scan cells.

On the other hand, if you invoke MacroTest with the -Le_observation_only switch and indicate in the MacroTest patterns that the macro's outputs should be observed in the cycle after pulsing the read pin on the macro, the rising edge of one cycle would cause the read of the macro, and then the rising edge on the next cycle would capture into the TE scan cells.

These two command switches (-Te_observation_only and -Le_observation_only) ensure that MacroTest behaves in a manner that is compatible with the particular macro and its embedding. In typical cases, only one kind of scan cell is available for observation and the MacroTest patterns file would, of course, need to be compatible. These options are only needed if both polarities of scan cells are possible observation sites for the same macro output pin.

For additional information on the use of these switches, refer to the [macrotest](#) description in the *Tessent Shell Reference Manual*.

Verifying Test Patterns

After testing the functionality of the circuit with a simulator, and generating the test vectors with the ATPG tool, you should run the test vectors in a timing-based simulator and compare the results with predicted behavior from the ATPG tools. This run will point out any functionality discrepancies between the two tools, and also show timing differences that may cause different results. The following subsections further discuss the verification you should perform.

Design Simulation with Timing	457
Potential Causes of Simulation Mismatches	465

Design Simulation with Timing

At this point in the design process, you should run a full timing verification to ensure a match between the results of golden simulation and ATPG. This verification is especially crucial for designs containing asynchronous circuitry.

You should have already saved the generated test patterns with the [write_patterns](#) command. The tool saved the patterns in parallel unless you used the -Serial switch to save the patterns in series. You can reduce the size of a serial pattern file by using the -Sample switch; the tool then saves samples of patterns for each pattern type, rather than the entire pattern set (except MacroTest patterns, which are not sampled nor included in the sampled pattern file). This is useful when you are simulating serial patterns because the size of the sampled pattern file is reduced and thus, the time it takes to simulate the sampled patterns is also reduced.

Note

 Using the -Start and -End switches will limit file size as well, but the portion of internal patterns saved will not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns will more closely approximate the results you would obtain from the entire pattern set.

The Verilog Test Bench.....	458
Parallel Versus Serial Patterns.....	464

The Verilog Test Bench

If you selected -Verilog as the format in which to save the patterns, the application automatically creates a test bench that you can use in a timing-based simulator such as ModelSim to verify that the tool-generated vectors behave as predicted by the ATPG tools.

For example, assume you saved the patterns generated as follows:

ANALYSIS> write_patterns pat_parallel.v -verilog -replace

The tool writes the test patterns out in one or more pattern files and an enhanced Verilog test bench file that instantiates the top level of the design. These files contain procedures to apply the test patterns and compare expected output with simulated output.

After compiling the patterns, the scan-inserted netlist, and an appropriate simulation library, you simulate the patterns in a Verilog simulator. If there are no miscompares between the ATPG tool's expected values and the values produced by the simulator, a message reports that there is "no error between simulated and expected patterns." If any of the values do not match, a simulation mismatch has occurred and must be corrected before you can use the patterns on a tester.

Verilog Simulation	458
Verilog Plusargs	458
Clock Monitoring During Simulation	460

Verilog Simulation

If you are using ModelSim as your Verilog simulator, you'll want to invoke with the "+acc=npr" option in order to get visibility into nets, ports, and registers while decreasing run time.

For example:

```
% vsim -c -voptargs+=+acc=npr
```

The visibility provided by the "+acc=npr" option should be sufficient for debugging advanced test benches. For even more visibility but with less performance gain, you can modify the above example as follows:

```
% vsim -c -voptargs+=+acc
```

Verilog Plusargs

Verilog plusargs are Verilog command line arguments used to provide information, including values, to the simulation run. The name plusarg stems from the fact that the argument name is preceded by a plus sign (+). The default Verilog test bench supports several Verilog plusargs.

- STARTPAT — Sets the starting pattern number for the simulation.
- ENDPAT — Sets the ending pattern number for the simulation.
- CHAINTEST — Makes the STARTPAT and ENDPAT plusargs apply to the chain test patterns instead of scan test.
- END_AFTER_SETUP — Causes the test bench to simulate only the test_setup vectors and then finish without simulating any of the other patterns.
- SKIP_SETUP — Causes the test bench to skip simulating the test_setup vectors and start simulation with the first pattern (either chain or scan test, whichever is present).
- CONFIG — Specifies a name of the .cfg file, which controls which .vec files are simulated. For information about using this plusarg, refer to [CONFIG Usage](#).

When the [SIM_CHANGE_PATH](#) parameter keyword is used, this Verilog plusarg is available:

- NEWPATH — Sets a new file path for the Verilog test bench. The default length of the filename is 512. You can override this value using the [SIM_TMP_REG_LENGTH](#) parameter.

STARTPAT, ENDPAT, and CHAINTEST Usage

By default (CHAINTEST=0), STARTPAT and ENDPAT specify the starting and ending scan test patterns. When CHAINTEST=1, STARTPAT and ENDPAT specify the starting and ending chain test patterns. So, specifying a STARTPAT of 0 without specifying CHAINTEST=1 skips the chain test patterns and simulates only the scan test patterns. Specifying an ENDPAT of a certain number and specifying CHAINTEST=1 causes simulation up to that number of chain test patterns and no simulation of scan test patterns. Specifying a STARTPAT and CHAINTEST=1 causes simulation starting from the chain test specified, and also simulation of all scan test patterns. What is not supported is the ability to start simulation at a certain chain test pattern and then end at a certain scan test pattern.

CONFIG Usage

If you have multiple test benches for the same design but each for different pattern sets, you need compile only one of the test benches and then specify the .cfg files from the other pattern sets in order to simulate those patterns. From the same design and setup information, you can create patterns within the tool and then write these out as a Verilog test bench. You can then create different pattern sets for the same design and setup information (that is, do not change clocks, scan chains, or edt logic) and then also write out as Verilog test benches. You can save the first test bench and compile in ModelSim, and then delete the test bench and .name files for all of the other pattern sets, saving only the .cfg and .vec files. Then, using the one test bench, you can use the CONFIG plusarg to specify which pattern set to simulate and use the other plusargs to control the range of patterns to simulate.

Verilog Simulation Dump Files

By default, the Verilog test bench contains a section to create a dump file. This uses the **SIM_DUMPFILE_PATH** parameter keyword, which defaults to true. As a result, the Verilog test bench includes the following:

```
 `ifdef VCD
    initial begin
        $dumpfile("patname.v.dump");
        $dumpvars;
    end
 `endif

 `ifdef UTVCD
    initial begin
        $dumpfile("patname.v.dump");
        $vtDump;
        $dumpvars;
    end
 `endif

 `ifdef debussy
    initial begin
        $fsdbDumpfile("patname.v.fsdb");
        $fsdbDumpvars;
    end
 `endif
```

To instruct the test bench to write a Verilog dump file, you must define in the in the test bench one of the three formats, and use the **SIM_INCLUDE** parameter keyword to specify the file that defines the format in the test bench.

Plusargs Usage Example

As an example of how to use plusargs, the following command line invokes ModelSim and simulates four scan test patterns:

```
vsim <testbench_top> -c -do "run -all" +STARTPAT=5 +ENDPAT=8
```

Clock Monitoring During Simulation

You can instruct the tool to generate the Verilog test bench with the capability to monitor clocks for both ATPG and IJTAG patterns during simulation. Additionally, you can monitor fast input clocks of all OCC instances that are in fast capture mode.

ATPG and IJTAG Patterns Specifics

For ATPG and IJTAG patterns, the tool reports during test bench pattern simulation a summary of each clock, and if the clock is running at the correct period/frequency for the following patterns:

- **ATPG Patterns** — The Verilog test bench monitors all the clocks associated with all the iCalls from the `test_setup` and all the external/internal asynchronous clocks with a period.
- **IJTAG Patterns** — The Verilog test bench monitors the clock destinations for IJTAG-based patterns. For each iClock, you can enable clock monitoring at the time when the iClock is executed to ensure each iClock is running at the correct period.

The tool independently monitors each clock to ensure that the clock is running with the correct period for a duration of n cycles of the clock. By default n equals 10, but you can change this number.

Limitations

There is no support for monitoring clocks derived from TCK as these clocks may not be continuous during the clock monitoring window because of unforeseen inject_cycles that can turn off TCK.

OCC Fast Input Clock Specifics

For OCC fast input clock inputs, the test bench monitors the fast clock inputs of all OCC instances that are in fast capture mode. The clock monitoring mechanism checks to ensure the fast clock inputs of all OCC instances that are in fast capture mode are running after `test_setup`.

Monitoring Window

The test bench monitors the fast clock inputs of all OCC instances in fast capture mode right after `test_setup` up to the start of the unloading of the next pattern encompassing at least 10 tester cycles or the end of the simulation, whichever comes first. The first pattern can be chain pattern, scan pattern, or whichever come first. If the fast clock inputs of these OCC instances are running during this monitor window, the test bench assumes that all they are free running until the end of simulation.

OCC Monitoring and LogicBIST Simulation

Limitations

- The test bench only detects whether the OCC fast clock inputs are running or not. There is no detection of whether the OCC fast input clock is running at a correct frequency.
- No support for detecting if OCC fast clock inputs that periods that are multiples of each other are synchronized. The test bench only detects if the clocks are running or not.
- OCC fast input clock monitoring is disabled for logicBIST simulation.

Clock Monitoring Code Test Bench Generation

The **SIM_CLOCK_MONITOR** parameter file keyword instructs the tool to generate the clock monitoring controls in the Verilog test bench when you create patterns with the **write_patterns** command. By default, this keyword is set to ON.

You can disable this by specifying in your external parameter file or interactively on the command line the **SIM_CLOCK_MONITOR** keyword with a value of OFF in conjunction with the **write_patterns** command as in the following example:

```
write_patterns test_patterns.v -verilog -parameter_list {SIM_CLOCK_MONITOR off}
```

Clock Monitoring Control During Simulation

During simulation, you can turn off clock monitoring using the following methods:

```
`define MGC_DISABLE_CLOCK_MONITOR  
`define MGC_DISABLE_CLOCK_MONITOR_PatternName
```

where *PatternName* is the prefix of the test bench leaf name.

You can find these controls in the Verilog test bench before the clock monitoring code.

Clock Precision Margin

By default, the test bench considers the clock correct if the clock's running period is within 1 percent of the expected period. During simulation, you can change the precision margin to another value using the following methods:

```
`define MGC_CLOCK_MONITOR_PERIOD_MARGIN_PERCENT margin
```

where *margin* is an integer that specifies a percentage value.

```
`define MGC_CLOCK_MONITOR_PERIOD_MARGIN_PERCENT_PatternName margin
```

where *PatternName* is the prefix of the test bench leaf name, and *margin* is an integer that specifies a percentage value.

When using these methods, you do not need to regenerate the test bench. You can find these controls in the Verilog test bench before the clock monitoring code.

Clock Monitoring Duration

By default, the test bench monitors each clock for a duration of 10 cycles. During simulation, you can change the duration using the following methods:

```
`define MGC_CLOCK_MONITOR_CYCLES number_of_cycles
```

where *number_of_cycles* is the new duration specified in cycles.

```
`define MGC_CLOCK_MONITOR_CYCLES_PatternName number_of_cycles
```

where *PatternName* is the prefix of the test bench leaf name, and *number_of_cycles* is the new duration specified in cycles.

When using these methods, you do not need to regenerate the test bench. You can find these controls in the Verilog test bench before the clock monitoring code.

Abort on Error

During simulation, you can abort the simulation if the computed period of the clock does not fall within the expected margin. By default, the test bench stops simulation when there are errors detected in the clock period. You can change this using the following methods:

```
`define MGC_CONTINUE_ON_CLOCK_ERROR
```

overrides the default abort.

```
`define MGC_CONTINUE_ON_CLOCK_ERROR_PatternName
```

overrides the default abort by prefix of the test bench leaf name.

You can find these controls in the Verilog test bench before the clock monitoring code.

Clock Monitor Report

At the beginning of each clock monitoring, the tool reports the monitored clock using a format similar to the following:

```
# 4700ns: Start Clock Monitoring on:  
# core_rtl_tessent_mbist_c1_controller_inst.BIST_CLK
```

At the conclusion, the tool issues one of the following reports for each monitored clock:

- Clocks that are running correctly within the expected margin. For example:

```
# 111600ns: Clock Monitoring passed:  
# core_rtl_tessent_mbist_c1_controller_inst.BIST_CLK Period =  
# 9.000 ns as expected (within 1.0% margin of 9.000 ns)
```

- Clocks that are running incorrectly because the periods are not within the expected margin. For example:

```
# 111600ns: Clock Monitoring failed:  
# core_rtl_tessent_mbist_c1_controller_inst.BIST_CLK Period  
# expected = 9.000 ns actual = 9.400 ns.
```

- Clocks that are not pulsing at all.

```
# 111600ns: Clock Monitoring failed:  
# core_rtl_tessent_mbist_c1_controller_inst.BIST_CLK Period  
# expected = 9.000 ns actual = n/a (no transition detected).
```

- Clocks that did not receive enough duration to compute the periods.

```
# 111600ns:      Clock Monitoring failed:  
# core_rtl_tessent_mbist_c1_controller_inst.BIST_CLK  
# Period expected = 9.000 ns actual = 9.000 ns ( did not receive  
# the expected 10 cycles)
```

OCC Clock Monitoring Report Specifics

At the end of the beginning of the monitoring window, the test bench issues the following message for each OCC fast clock:

```
#5440 Start OCC Fast Clock Input Monitoring on: <fast_clock_input_pin>.
```

At the end of the monitoring window, the test bench issues the following message for each OCC fast clock input that is running during the monitoring window. The message includes the period and the number of clock cycles detected during the monitoring window:

```
#1040 OCC Fast Clock Input Monitoring Passed: <fast_clock_input_pin>. Measured period = <period>. Number of Cycles = <num_cycles>
```

At the end of the monitoring window, the test bench issues the following message for each OCC fast clock input that is not running during the monitoring window:

```
#1040 OCC Fast Clock Input Monitoring Failed: <fast_clock_input_pin>. No transitions detected.
```

Parallel Versus Serial Patterns

Be sure to simulate parallel patterns and at least a few serial patterns. Parallel patterns simulate relatively quickly, but do not detect problems that occur when data is shifted through the scan chains. One such problem, for example, is data shifting through two cells on one clock cycle due to clock skew. Serial patterns can detect such problems. Another reason to simulate a few serial patterns is that correct loading of shadow or copy cells depends on shift activity. Because parallel patterns lack the requisite shift activity to load shadow cells correctly, you may get simulation mismatches with parallel patterns that disappear when you use serial patterns. Therefore, always simulate at least the chain test or a few serial patterns in addition to the parallel patterns.

For a detailed description of the differences between serial and parallel patterns, refer to “[Serial Versus Parallel Scan Chain Loading](#)” on page 546 and “[Parallel Scan Chain Loading](#)” on page 546. See also “[Reduce Serial Loading Simulation Time with Sampling](#)” on page 548 for information on creating a subset of sampled serial patterns. Serial patterns take much longer to simulate than parallel patterns (due to the time required to serially load and unload the scan chains), so typically only a subset of serial patterns is simulated.

Potential Causes of Simulation Mismatches

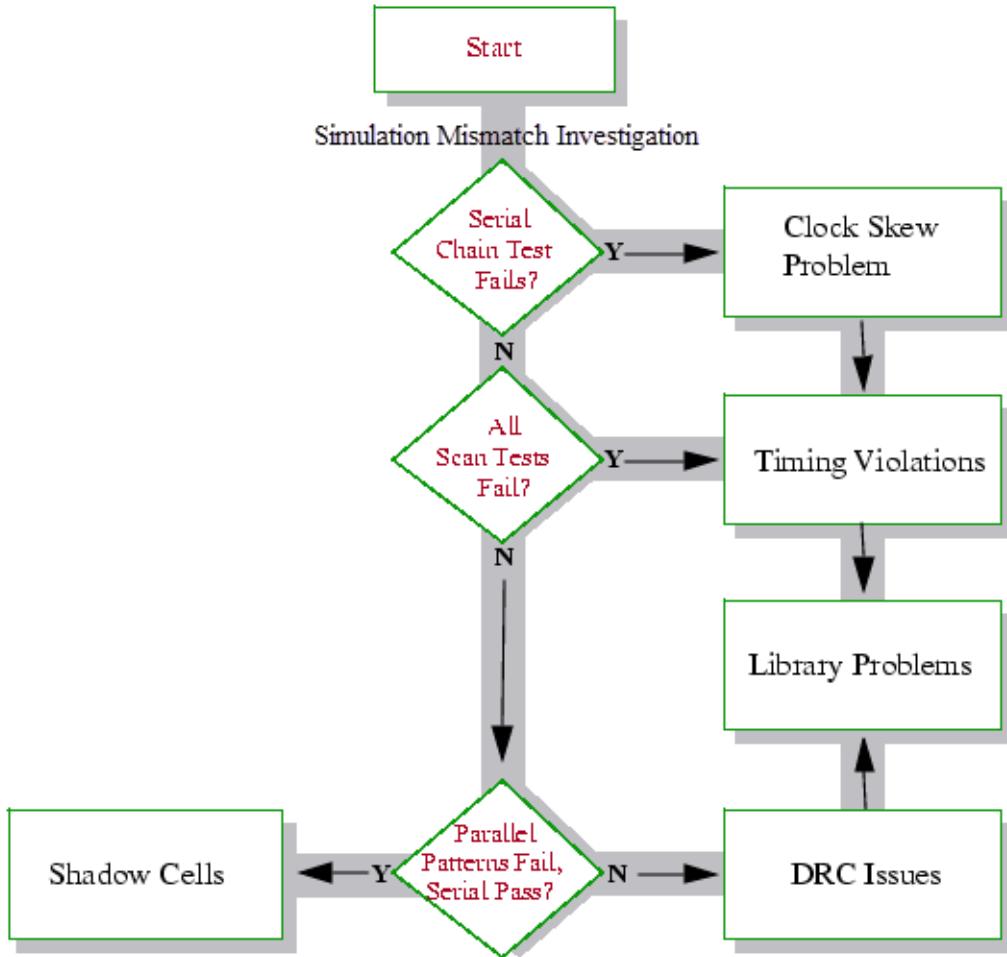
Simulation mismatches can have any number of causes; consequently, the most challenging part of troubleshooting them is knowing where to start. Because a lot of information is available, your first step should be to determine the likeliest potential source of the mismatch.

Note

Before troubleshooting simulation mismatches, be sure that the chain test has run without error. You should resolve any problem with the scan chain before investigating simulation mismatches.

Figure 8-86 is a suggested flow to help you investigate causes of simulation mismatches.

Figure 8-86. Mismatch Diagnosis Guidelines



For clock skew problems, refer to the topic “[Clock-Skew Problems with Mux-DFF Designs](#)” on page 480.

The remaining issues, along with analysis approaches, are discussed in the following sections:

Simulation Mismatch Investigation	466
DRC Issues	467
Shadow Cells	467
Library Problems.....	467
Timing Violations.....	468
Simulation Data Analysis	468
Simulation Mismatch Analysis	471

Simulation Mismatch Investigation

If DRC violations do not seem to be a problem, you need to take a closer look at the mismatches. You need to determine when, where, and how many mismatches occur.

- **Are the mismatches reported on primary outputs (POs), scan cells or both?** Mismatches on scan cells can be related to capture ability and timing problems on the scan cells. For mismatches on primary outputs, the issue is more likely to be related to an incorrect value being loaded into the scan cells.
- **Are the mismatches reported on just a few or most of the patterns?** Mismatches on a few patterns indicates a problem that is unique to certain patterns, while mismatches on most patterns indicate a more generalized problem.
- **Are the mismatches observed on just a few pins/cells or most pins/cells?** Mismatches on a few pins/cells indicates a problem related to a few specific instances or one part of the logic, while mismatches on most patterns indicate that something more general is causing the problem.
- **Do both the serial and the parallel test bench fail or just one of them?** A problem in the serial test bench only, indicates that the mismatch is related to shifting of the scan chains (for example, data shifting through two cells on one clock cycle due to clock skew). The problem with shadows mentioned in “[Parallel Versus Serial Patterns](#)” on page 464, causes the serial test bench to pass and the parallel test bench to fail.
- **Does the chain test fail?** As described above, serial pattern failure can be related to shifting of the scan chain. If this is true, the chain test (which simply shifts data from scan in to scan out without capturing functional data) also fails.
- **Do only certain pattern types fail?** If only ram sequential patterns fail, the problem is most certainly related to the RAMs (for instance incorrect modeling). If only clock_sequential patterns fail, the problem is probably related to non-scan flip-flops and latches.

DRC Issues

The DRC violations that are most likely to cause simulation mismatches are C6 and T24.

For details on these violations, refer to “[Design Rule Checking](#)” in the *Tessent Shell Reference Manual* and Support Center describing each of these violations. For most DRC-related violations, you should be able to see mismatches on the same flip-flops where the DRC violations occurred.

You can avoid mismatches caused by the C6 violation by enabling the [set_clock_off_simulation](#) command.

Shadow Cells

Another common problem that produces simulation mismatches is shadow cells.

Such cells do not cause DRC violations, but the tool issues the following message when going into analysis mode:

```
// 1 external shadows that use shift clocking have been identified.
```

A shadow flip-flop is a non-scan flip-flop that has the D input connected to the Q output of a scan flip-flop. Under certain circumstances, such shadow cells are not loaded correctly in the parallel test bench. If you see the above message, it indicates that you have shadow cells in your design and that they may be the cause of a reported mismatch. For more information about shadow cells and simulation mismatches, consult the online Support Center—see “[Mentor Support Services](#)” on page 710.

Library Problems

A simulation mismatch can be related to an incorrect library model; for example, if the reset input of a flip-flop is modeled as active high in the analysis model used by the tool, and as active low in the Verilog model used by the simulator. The likelihood of such problems depends on the library. If the library has been used successfully for several other designs, the mismatch probably is caused by something else. On the other hand, a newly developed, not thoroughly verified library could easily cause problems. For regular combinational and sequential elements, this causes mismatches for all patterns, while for instances such as RAMs, mismatches only occur for a few patterns (such as RAM sequential patterns).

Another library-related issue is the behavior of multi-driven nets and the fault effect of bus contention on tristate nets. The ATPG tool is conservative by default, so non-equal values on the inputs to non-tristate multi-driven nets, for example, always results in an X on the net. For additional information, see the [set_net_resolution](#) and [set_net_dominance](#) commands.

Timing Violations

Setup and hold violations during simulation of the test bench can indicate timing-related mismatches. In some cases, you see such violations on the same scan cell that has reported mismatches; in other cases, the problem might be more complex. For instance, during loading of a scan cell, you may observe a violation as a mismatch on the cell(s) and PO(s) that the violating cell propagates to. Another common problem is clock skew.

This is discussed in “[Clock-Skew Problems with Mux-DFF Designs](#)” on page 480.

Another common timing related issue is that the timeplate and/or test procedure file has not expanded. By default, the test procedure and timeplate files have one “time unit” between each event. When you create test benches using the -Timingfile switch with the [write_patterns](#) command, the time unit expands to 1000 ns in the Verilog test benches. When you use the default -Procfile switch and a test procedure file with the [write_patterns](#) command, each time unit in the timeplate is translated to 1 ns. This can easily cause mismatches.

Simulation Data Analysis

If you still have unresolved mismatches after performing the preceding checks, examine the simulation data thoroughly and compare the values observed in the simulator with the values expected by the tool. The process you would use is very similar in the Verilog test benches.

Mismatch Resolution Using Simulation Data

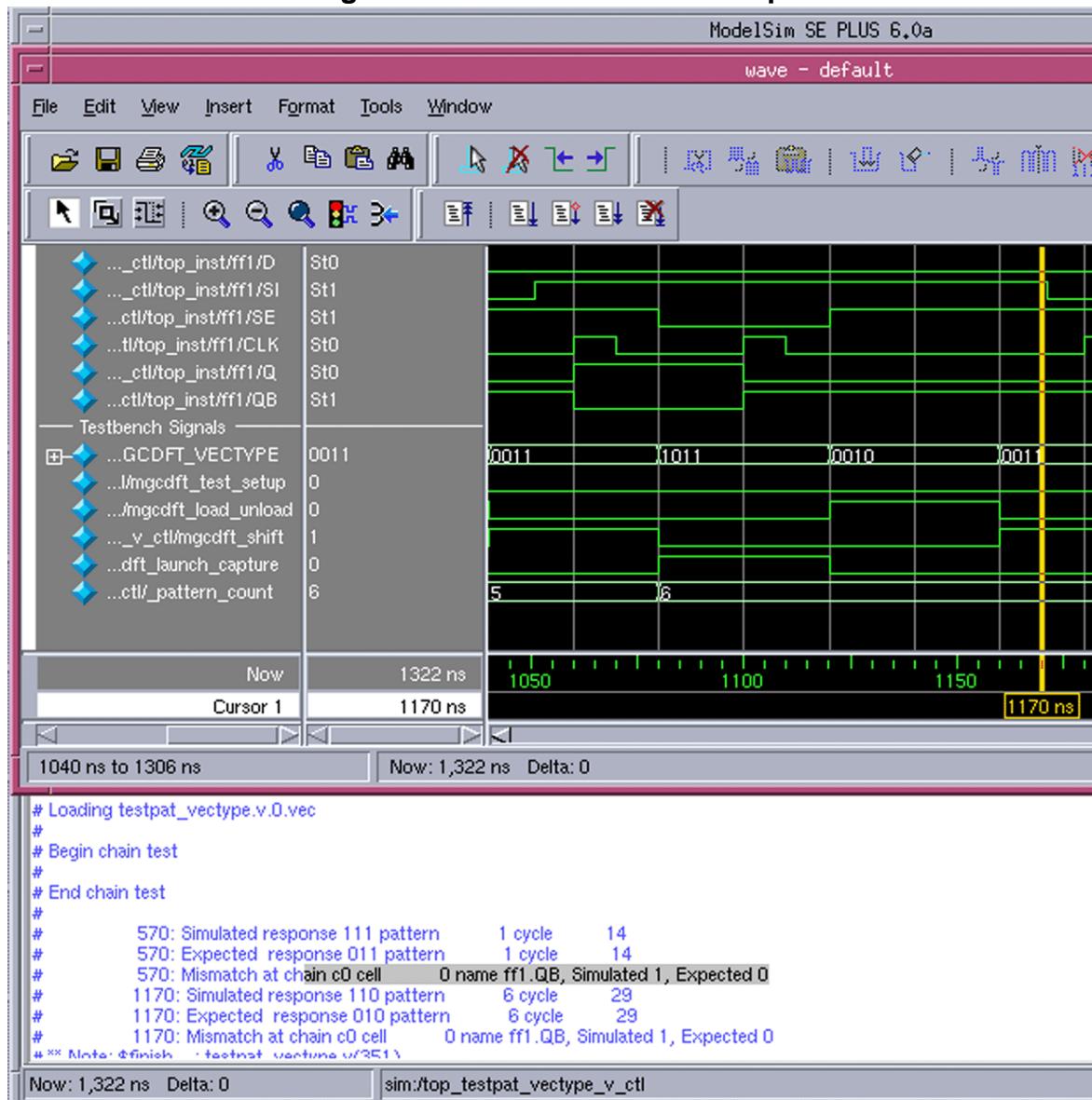
When simulated values do not match the values expected by the ATPG tool, the enhanced Verilog parallel test bench reports the time, pattern number, and scan cell or primary output where each mismatch occurred. The serial test bench reports only output and time, so it is more challenging to find the scan cell where the incorrect value has been captured. It is recommended to run the parallel test bench with a few serial shift cycles. This enables you to catch both capture and shift problems with a single pattern set. To apply serial post shift cycles, write out the patterns with the “SIM_POST_SHIFT” parameter. For example, to define five post-shift cycles, use “[write_patterns <pat.v> -verilog -parameter_list {SIM_POST_SHIFT 5}](#)”.

Based on the time and scan cell where the mismatch occurred, you can generate waveforms or dumps that display the values just prior to the mismatch. You can then compare these values to the values that the tool expected. With this information, you can trace back in the design (in both the ATPG tool and the simulator) to see where the mismatch originates.

When comparing Verilog simulation data to ATPG tool data, it is helpful to use the “_procedure_string”, “_pattern_count”, and “_cycle_count” signals at the top level of your test bench. Make sure to display the “_procedure_string” signal in ASCII format to see the simulated procedure. When you use the “_procedure_string” signal, the Verilog test bench will include the test procedure simulated by the tool, which makes it easier for you to understand the sequence of events in the test bench, and also how to compare data between the ATPG tool and

the simulator. In the example simulation transcript in Figure 8-87, a mismatch is reported for patterns 1 and 6. For pattern 6, the mismatch is reported at time 1170.

Figure 8-87. Simulation Transcript



Note that the waveforms include the values for the flip-flop with mismatches, as well as the signals *mgcdft_shift*, *mgc_dft_launch_capture*, and *_pattern_count*. Note that the *_pattern_count* variable will increment just prior to the capture procedure. That means that when *_pattern_count* variable is 6 and *mgcdft_shift* is 1, data is shifted out for pattern 6 (and shifted in for pattern 7). By using these signals as a guide, you can see that the time of the mismatch is during the shift procedure after pattern 6. Note that the capture for pattern 6 occurs between time 1080 and 1120, when the *mgcdft_launch_capture* signal is high.

To see the corresponding data in the ATPG tool, you can use the [set_gate_report <pattern_index>](#) command to see data for pattern 6:

```
// command: read_patterns testpat.ascii
// command: set_gate_report pattern_index 6 -external
// command: report_gates ff1
// ff1 dfsc
// CK I (010) /clk
// SC I (000) /scan_en
// D0 I (000) /sub/reg_ws5/Q
// D1 I (111) /sub/reg_w4/QB
// Q O (100) /sub/reg_ws7/D
// QB O (011) /sub/reg_w8/D1
```

Simulation Mismatch Analysis

By default, the simulation VCD file, test patterns, and design data are all analyzed and the mismatch sources are identified. Once the analysis is complete, you can use DFTVisualizer to graphically display the overlapping data and pinpoint the source of each mismatch.

The ATPG simulation mismatch analysis functionality is enhanced to optimize the debugging of large (100K gates and more) designs and to support all simulators and distributed processing. Once analysis is complete, DFTVisualizer graphically displays the source of the mismatches for easy identification.

The Simulation Mismatch Analysis Flow	471
Automatically Analyzing Simulation Mismatches	472
Manually Analyzing Simulation Mismatches	475
Patterns Analysis	479
Clock-Skew Problems with Mux-DFF Designs.....	480

The Simulation Mismatch Analysis Flow

The simulation mismatch analysis flow includes several stages.

You can analyze simulation mismatches using the following methods:

- [Automatically Analyzing Simulation Mismatches](#) — Using this method, the tool runs the entire flow, from [Stage 1 — ATPG](#) through [Stage 5 — Mismatch source identification](#) using a single command invocation— see [Figure 8-88](#).
- [Manually Analyzing Simulation Mismatches](#) — Using this method, you can run mismatch analysis flow in steps. For example, you correct known issues in your failure file and, instead of re-running [Stage 2 — Verilog test bench simulation](#), you can proceed to [Stage 3 — Debug test bench generation](#) using the modified failure file.

Both the automatic and manual flows follow the identical stages: the key difference is the automatic flow runs the *entire* flow while the manual flow allows you to use the flow in steps. The following sections cover the simulation mismatch analysis flow in detail.

Note

 This procedure does not support debugging MacroTest or chain test patterns.

Stage 1 — ATPG

Generate the flattened model, test patterns, and Verilog test bench (*mentor_default.v*). The test patterns must be saved in a format that can be read back into the ATPG tool (binary, ASCII, STIL, or WGL), and the test bench must be able to generate a failure file.

Stage 2 — Verilog test bench simulation

Simulating the Verilog test bench generates a failure file (*mentor_default.v.fail*). If no simulation mismatches are found, the automatic simulation mismatch analysis stops.

Stage 3 — Debug test bench generation

Generate a test bench for just the failing patterns (*mentor_default.v_vcdtb.v*) by using the ATPG tool to read the flattened netlist and read in the test patterns generated in Stage 1 and the failure file from Stage 2. The test bench is set up to output a simulation results VCD file.

Stage 4 — Debug test bench simulation

In the same ATPG tool session, simulate the debug test bench generated in Stage 3. This simulation produces the VCD file (*mentor_default.v_debug.vcd*).

Stage 5 — Mismatch source identification

Load the VCD file and trace each mismatch to its source. Then report the mismatch sources.

Note

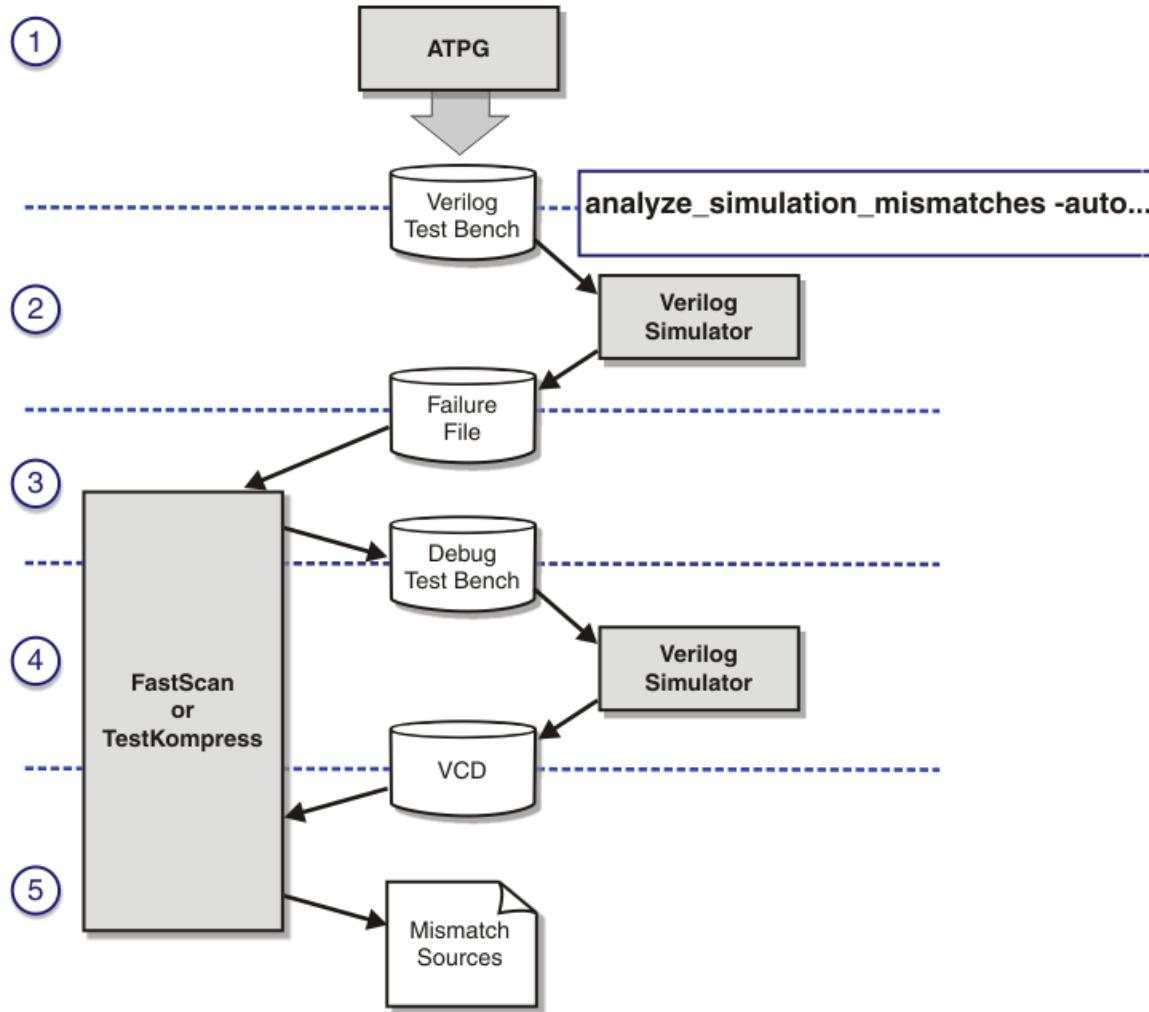
 All generated files are placed in the *work_dft_debug* directory inside your working directory. This directory is created if it does not already exist.

Automatically Analyzing Simulation Mismatches

You can use the tool to automatically analyze simulation mismatches.

Figure 8-88 shows the automatic simulation mismatch analysis flow. Using this procedure, you invoke the tool using the `analyze_simulation_mismatches` command with the `-auto` switch, and the tool automatically runs the all stages of the flow. This procedure supports using a third-party simulator and distributing processing to a remote server.

Figure 8-88. Automatic Simulation Mismatch Analysis Flow



Prerequisites

- (Optional) If you are using an external third-party simulator, you must create a script to invoke, set up, and run the simulator. See [Example](#).
- (Optional) If you want to distribute the simulation part of the analysis, you must have access to a remote server that can be accessed through rsh. For more information, see the [analyze_simulation_mismatches](#) description in the *Tessent Shell Reference Manual*.

Procedure

Note

All generated files are placed in the *work_dft_debug* directory inside your working directory. This directory is created if it does not already exist.

1. Use the ATPG tool to read a design netlist or flat model. For example:

```
$ tessent -shell  
SETUP> set_context patterns -scan  
SETUP> read_verilog data/design.v
```

2. Specify the scan data for the scan cells in the design and switch to analysis system mode. For example:

```
SETUP> add_scan_groups ...  
SETUP> add_scan_chains ...  
SETUP> add_clocks  
...  
SETUP> set_system_mode analysis
```

3. Specify the source test patterns for the design. For example:

```
ANALYSIS> read_patterns pats/testpat.bin
```

4. If you are using a third-party simulator, specify your simulator invoke script. For example:

```
ANALYSIS> set_external_simulator -simulation_script runsim
```

For more information, see [Example](#) and the [set_external_simulator](#) description in the *Tessent Shell Reference Manual*.

5. Run the automatic mismatch analysis. For example:

```
ANALYSIS> analyze_simulation_mismatches -auto -external_patterns
```

By default, the analysis runs on the local server. To run the simulation portion of the analysis on a remote server, use the -host option. For example:

```
ANALYSIS> analyze_simulation_mismatches -auto -external_patterns -host abc_test
```

For more information, see the [analyze_simulation_mismatches](#) description in the *Tessent Shell Reference Manual*.

By default, the analysis compares the specified failure file to the current test pattern source to verify that both are generated from the same version of the design. If files do not match, an error displays and the process aborts.

Once the test patterns and failure file pass the verification, a test bench is created specifically for the mismatches in the failure file and simulated. The simulation results are compared with the test patterns and design data to determine the source of simulation mismatches listed in the failure file.

6. Open DFTVisualizer to view and further debug the mismatches. For example:

```
ANALYSIS> open_visualizer
```

7. Click **Tools > Analyze Mismatch**. The Select a Mismatch ID dialog box displays.

8. Select the ID for the mismatch to debug, and click **Analyze**.

DFTVisualizer displays and highlights overlapping design, simulation, and test pattern data for the selected simulation mismatch.

Tip

 Optionally, you can use the [report_mismatch_sources](#) command to replace steps 6 through 8 and automatically display the mismatches in DFTVisualizer. If you issue `report_mismatch_sources` with the `-display` option, the Wave window displays the following signals in addition to the instances associated with the displayed mismatch source: `mgcdft_test_setup`, `mgcdft_load_unload`, `mgcdft_single_shift`, `mgcdft_shift_extra`, `mgcdft_launch_capture`

You can use the waveform of these signals to easily determine the active test procedure in any displayed cycle. The signal value will be 1 to designate the current active procedure.

Note

 In the Flat Schematic window, pin names assigned to the value “.” indicate that the VCD debug test bench did not capture the VCD value for that location. It is not possible to capture VCD values for every node in the design due to very large file sizes and run time; the test bench only captures values in the combination cone behind the failure location, which in most cases provides sufficient information to explain the mismatch.

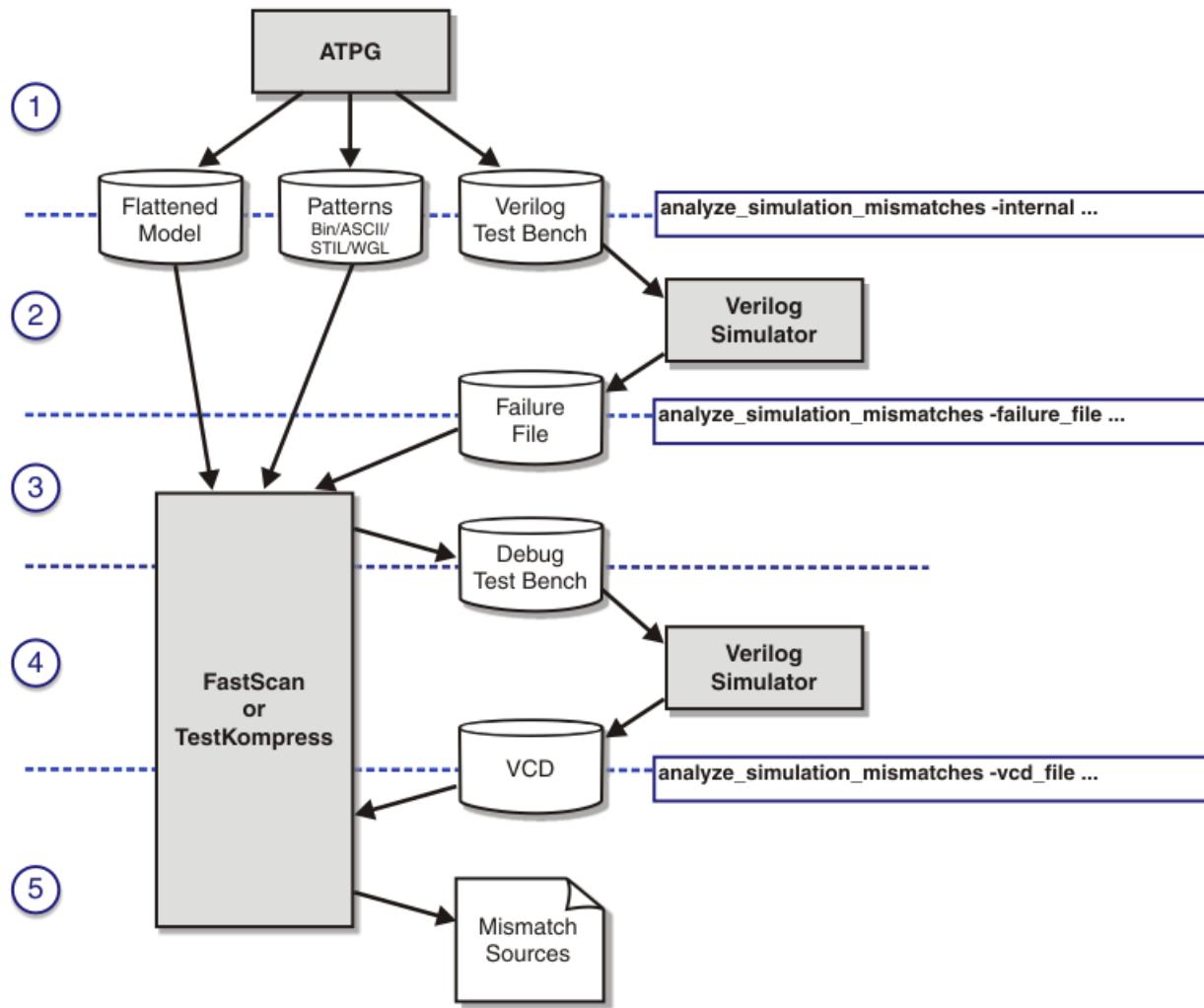
For a complete list of possible pin name values resulting from simulation mismatches, see the [report_mismatch_sources](#) description in the *Tessent Shell Reference Manual*.

Manually Analyzing Simulation Mismatches

You can manually debug simulation mismatches using the `analyze_simulation_mismatches` command. This procedure supports using a third-party simulator and distributing processing to a remote server.

[Figure 8-89](#) shows the manual simulation mismatch analysis flow.

Figure 8-89. Manual Simulation Mismatch Analysis Flow



Prerequisites

- (Optional) If you are using an external third-party simulator, you must create a script to invoke, set up, and run the simulator. See [Example](#).
- (Optional) If you want to distribute the simulation part of the analysis, you must have access to a remote server that can be accessed through rsh. For more information, see the [analyze_simulation_mismatches](#) description in the *Tessent Shell Reference Manual*.
- A design netlist or flat model and the associated test patterns are available.
- Test patterns must have been verified and mismatches exist.
- In order to generate a failure file for a manually generated test bench, you must set the “[SIM_DIAG_FILE](#)” parameter file keyword to 2 or 1(default) prior to ATPG.

- For a manual simulation, you must set the “_write_DIAG_file” parameter to 1 in the Verilog test bench to generate the failure file. This is done automatically if you set the SIM_DIAG_FILE parameter file keyword to 2 prior to ATPG.

Procedure

Note

 All generated files are placed in the *work_dft_debug* directory inside your working directory. This directory is created if it does not already exist.

1. Use the ATPG tool to read a design netlist or flat model. For example:

```
$ tessent -shell  
SETUP> set_context patterns -scan  
SETUP> read_verilog data/design.v
```

2. Specify the scan data for the scan cells in the design and switch to analysis system mode. For example:

```
SETUP> add_scan_groups ...  
SETUP> add_scan_chains  
...  
SETUP> add_clocks  
...  
SETUP> set_system_mode analysis
```

3. Specify the source test patterns for the design. For example:

```
> read_patterns pats/testpat.bin
```

4. If you are using a third-party simulator, specify your simulator invoke script. For example:

```
> set_external_simulator -simulation_script runsim
```

For more information, see [Example](#) and the `set_external_simulator` description in the *Tessent Shell Reference Manual*.

5. Run the mismatch analysis. For example:

```
> analyze_simulation_mismatches -external_patterns
```

By default, the analysis runs on the local server. To run the simulation portion of the analysis on a remote server, use the `-host` option. For example:

```
ANALYSIS> analyze_simulation_mismatches -external_patterns -host abc_test
```

For more information, see the `analyze_simulation_mismatches` description in the *Tessent Shell Reference Manual*.

By default, the analysis compares the specified failure file to the current test pattern source to verify that both are generated from the same version of the design. If files do not match, an error displays and the process aborts.

Once the test patterns and failure file pass the verification, a test bench is created specifically for the mismatches in the failure file and simulated. The simulation results are compared with the test patterns and design data to determine the source of simulation mismatches listed in the failure file.

6. Simulate the Verilog test bench using ModelSim or a third-party simulator to create a failure file—see the [set_external_simulator](#) command for details.

You must also create a script to set up and run an external simulator for the subsequent steps of this procedure.

7. After simulation, perform a simulation mismatch analysis using the [analyze_simulation_mismatches](#) command with the -FAilure_file switch and argument to generate a test bench for just the failing patterns. For example:

```
ANALYSIS> analyze_simulation_mismatches -failure_file pat.fs.v.fail
```

You must use the flattened netlist and test patterns from the initial ATPG session.

This step creates the test bench for just the failing patterns (*mentor_default.v_vcdtb.v*). The test bench is set up to output the simulation results to a VCD file.

8. In the same ATPG tool session, simulate the test bench for the failing patterns using the [analyze_simulation_mismatches](#) command with the -TEstbench_for_vcd switch argument. For example:

```
ANALYSIS> analyze_simulation_mismatches \
           -testbench_for_vcd mentor_default.v_vcdtb.v
```

This step produces a VCD file (*mentor_default.v_debug.vcd*).

9. After simulation, load the VCD file. For example:

```
ANALYSIS> analyze_simulation_mismatches -vcd_file mentor_default.v_debug.vcd
```

10. Report the mismatch sources using the [report_mismatch_sources](#) command.

You can also view and further debug the mismatches using DFTVisualizer.

11. Open DFTVisualizer to view and further debug the mismatches. For example:

```
ANALYSIS> open_visualizer
```

12. Click **Debug Simulation Mismatches**. The Select a Mismatch ID dialog box displays.
13. Select the ID for the mismatch to debug, and click **Analyze**.

DFTVisualizer displays and highlights overlapping design, simulation, and test pattern data for the selected simulation mismatch.

Examples

The following example shows a simulation script for ModelSim where the design netlist and model library have been pre-compiled into the my_work directory. The script will be used to compile and simulate the test bench that was saved using the write_patterns command as well as the debug test bench created by the tool. Prior to execution of the script, the \${1} variable will be replaced by the tool with the name of the test bench being compiled and \${2} will be replaced by the test bench top level being simulated.

```
#!/bin/csh -f
vlog ${1} -work my_work
vsim ${2} -c -lib my_work -do "run -all" -sdfmax \
/${2}/design_inst=~/design/good.sdf
```

If there are simulation mismatches in the first execution of the saved test bench, the analyze_simulation_mismatches command creates a new version of the test bench for the analysis. To create a simulation script that modifies this test bench, such as adding fixed force statements, you must substitute the variable \$ENTITY for the test bench name.

For example: force -freeze sim:\$entity/instance/pin0 1

The script must be executable otherwise the following error is returned:

```
analyze_simulation_mismatches -simulation_script vsim_scr
sh: line 1: ./vsim_scr: Permission denied
// Error: Error when running script:
//   vsim_scr work_dft_debug/mentor_default.v
//   circle_mentor_default_v_ctl > /dev/null
// No mismatch source is found.
```

To correct this error, use the following Linux command in the tool before running the analyze_simulation_mismatches command:

```
!chmod +x vsim_scr
```

Patterns Analysis

Sometimes, you can find additional information that is difficult to access in the Verilog test benches in other pattern formats. When comparing different pattern formats, it is useful to know that the pattern numbering is the same in all formats. In other words, pattern #37 in the ASCII pattern file corresponds to pattern #37 in the WGL or Verilog format.

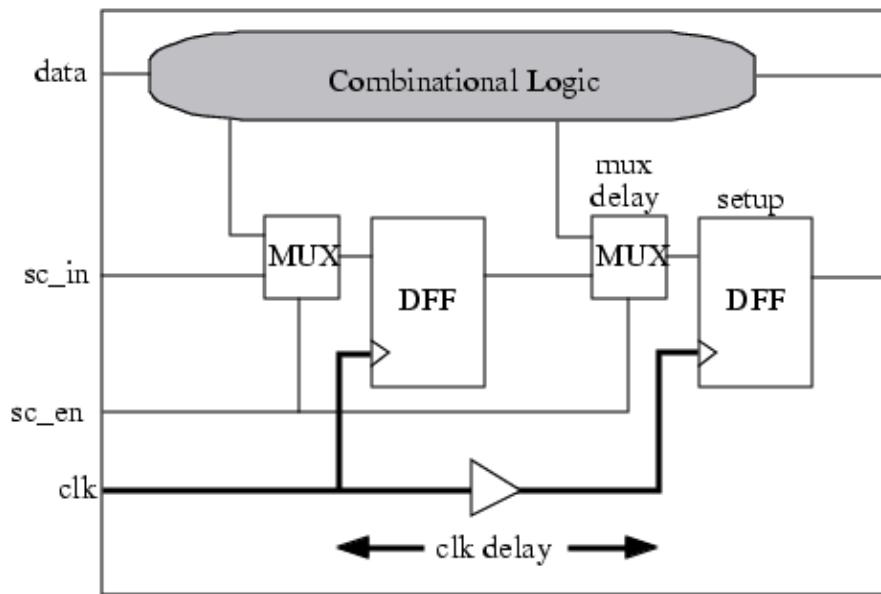
Each of the pattern formats is described in detail in the section, “[Basic Test Data Formats for Patterns](#),” beginning on [page 549](#).

Clock-Skew Problems with Mux-DFF Designs

If you have mux-DFF scan circuitry in your design, you should be aware of, and thus test for, a common timing problem involving clock skew.

Figure 8-90 depicts the possible clock-skew problem with the mux-DFF architecture.

Figure 8-90. Clock-Skew Example



You can run into problems if the clock delay due to routing, modeled by the buffer, is greater than the mux delay minus the flip-flop setup time. In this situation, the data does not get captured correctly from the previous cell in the scan chain and therefore, the scan chain does not shift data properly.

To detect this problem, you should run both critical timing analysis and functional simulation of the scan load/unload procedure. You can use ModelSim or another HDL simulator for the functional simulation, and a static timing analyzer such as SST Velocity for the timing analysis. Refer to the *ModelSim SE/EE User's Manual* or the *SST Velocity User's Manual* for details on performing timing verification.

Chapter 9

Multiprocessing for ATPG and Simulation

This chapter explains multiprocessing functionality for accelerating ATPG and simulation. Multiprocessing is a combination of distributed processing and multithreading. Multiprocessing functionality allows you to create slave processes with multiple threads to efficiently use additional processors.

Definition of Multiprocessing Terms	481
Multiprocessing to Reduce Runtime	483
Multiprocessing Requirements	483
Procedures for Multiprocessing	486
Using Multiprocessing for ATPG and Simulation	486
SSH Environment and Passphrase Errors	487
How to Disable Multithreading Functionality	489
Threads Addition to the Master	489
Processor Addition in Manual Mode	490
Processor Addition in Grid Mode	492
Processor Addition to the LSF Grid	493
Manual Mode Processor Deletion	494
Grid Mode Processor Deletion	494

Definition of Multiprocessing Terms

This manual uses the following multiprocessing terminology.

- **Distribution or Distributed Processing** — The dividing up and simultaneous execution of processing tasks on one or multiple machines.
- **Multithreading** — The dividing up and simultaneous execution of processing tasks within a single process running on one machine. Threads can run on separate cores in parallel to speed up execution. This also minimizes memory use per thread by sharing the design information across all threads running, compared to using the same number of processors on multiple single-threaded slaves.
- **Multiprocessing** — A general term that can refer either to distribution or multithreading, or to a combination of both.
- **Manual Mode** — Starting additional processors by explicitly specifying the host(s) on which the processors run.
- **Master Process** — The master process is the process started when you invoke the tool.

- **Slave Process** — A separate process that does not share memory with the master process. Slave processes typically run on a machine other than the master host. If the -multithreading option is off, slave processes can also run on the master host.
- **Process** — An instance of the executable running on the master or slave host, regardless of the number of threads in use.
- **Processor** — A resource that executes a thread, which is added with the `add_processors` command or by starting the tool.
- **Grid Mode** — Starting additional processors using the grid engine. You have limited control in specifying the host upon which the job is started because that is handled by the grid engine, subject to any constraints you place on the grid job submission.
- **Thread** — The smallest independent unit of a process. A process can consist of multiple threads, all of which share the same allocated memory and other resources on the same host.
- **Worker Thread** — Threads other than the main thread.

Multiprocessing to Reduce Runtime

A growing number of Tesson tools can optionally use multiprocessing through distribution, multithreading, or both, to reduce runtime.

For example, these are some of the commands and processes that support multiprocessing:

- [compress_patterns](#) (fault simulation)
- [create_patterns](#) (ATPG)
- [identify_redundant_faults](#) (ATPG)
- [order_patterns](#) (fault simulation)
- [simulate_patterns](#) (fault simulation)
- Static SDC (false and multicycle path) analysis
- A limited subset of power metrics calculations

For large designs, distributing portions of these commands' processing load using multiprocessing can reduce their runtime, sometimes significantly. The reduction in runtime depends on several factors and is not directly proportional to the number of additional threads doing the processing. The particular design, the tool setups you use, and the type of patterns you generate, for example, determine the kinds of processes the tool must run and the proportion of them it can distribute to additional threads. Generally, the ATPG runtime improvement will be greater for transition patterns than for stuck-at.

Slave processors or threads can be on the machine on which the tool is running or on remote machines, wherever you can access additional processors on your network. ATPG results (coverage and patterns) using multiprocessing are the same as without multiprocessing, regardless of the number of processors.

Multiprocessing Requirements..... 483

Multiprocessing Requirements

To enable the tool to establish and maintain communication with multiple processors on multiple host machines and run slave processing jobs on them, you need to inform the tool of the network names of available machines (manual specification) or direct the tool to use an automated job scheduler to select machines for you. The tool supports Load Sharing Function (LSF), Sun Grid Engine (SGE), or custom job schedulers.

The following prerequisites must be satisfied for whichever you use:

- **Manual Specification (does not require a job scheduler)**

You can specify hosts manually, without using the SGE, LSF or a custom job scheduler. The master host must be able to create processes on the slave host via the **rsh** or **ssh** shell command.

- **rsh** — This requires that the network allow connection via **rsh**, and that your *.rhosts* file allow **rsh** access from the master host without specifying a password. This is the default.

Note

 The *.rhosts* file on host machines must have read permission set for user. Write and execute permission can optionally be set for user, but must *not* be set for other and group.

rsh access is not required for the tool to create additional processes on the master host.

- **ssh** — This requires that the network allow connection via **ssh**. To enable use of **ssh**, issue a [set_multiprocessing_options](#) command within the tool to set the multiprocessing “remote_shell” variable to **ssh**. Do this prior to issuing an [add_processors](#) command.

Master and slave machines must be correctly specified in the global DNS name server for reliable network operation, and you will need to know either the network name or IP address of each remote machine you plan to use. Consult the System Administrator at your site for additional information.

- **Job Scheduler** — You must have available at your site at least one of the following methods of network job scheduling. Whichever you use, it must allow the master process (the process started when you invoke the tool) to create slave processes on different host machines.
 - **Load Sharing Function (LSF)** — To use LSF, ensure your environment supports use of the LSF scheduler before you invoke the tool. For example, the **LSF_BINDIR** environment variable must be set appropriately, in addition to other requirements. An appropriate setup can often be performed by sourcing a configuration file supplied with the scheduler installation.
 - **Sun Grid Engine (SGE)** — To use SGE, ensure your environment supports use of the SGE scheduler before you invoke the tool. For example, the **SGE_ROOT** environment variable must be set appropriately, in addition to other requirements. An appropriate setup can often be performed by sourcing a configuration file supplied with the scheduler installation.
 - **Custom Job Scheduling** — For the tool to use a custom job scheduler, you need to inform the tool of the command used at your site to launch the custom job scheduler. You do this by issuing a [set_multiprocessing_options](#) command within the tool to set the “generic_scheduler” variable to the appropriate site-specific command.

- **Job Scheduling Options** — You can control certain aspects of the job scheduling process with the switches you set with the [set_multiprocessing_options](#) command. For complete information about these switches, refer to the [set_multiprocessing_options](#) description in the *Tessent Shell Reference Manual*.
- **Tessent Software Tree Installation** — It must be possible to execute the Mentor Graphics Tessent executables (via fully specified paths) from any new processes.
- **Tool Versions** — All multiprocessing hosts must run the same version of the tool. This is not an issue when using a single executable for all slave hosts, but depending on the installation, may become an issue when the path to the executable points to a different physical disk location on a slave host than on the master host. The tool installation tree must be accessible via the same path on all slave hosts.

Procedures for Multiprocessing

The following sections describe how to use multiprocessing functionality.

Using Multiprocessing for ATPG and Simulation	486
SSH Environment and Passphrase Errors	487
How to Disable Multithreading Functionality	489
Threads Addition to the Master	489
Processor Addition in Manual Mode	490
Processor Addition in Grid Mode	492
Processor Addition to the LSF Grid	493
Manual Mode Processor Deletion	494
Grid Mode Processor Deletion	494

Using Multiprocessing for ATPG and Simulation

The following describes the basic procedure for using multiprocessing for ATPG and fault simulation.

These steps assume you have satisfied the “[Multiprocessing Requirements](#)” on page 483.

Procedure

1. Use the [set_multiprocessing_options](#) command to specify the values of any multiprocessing variables, such as the job scheduler you will be using or the remote shell. Manual specification requires this step only if you want the tool to use [ssh](#). Specific LSF and SGE options will depend on how SGE and LSF are configured at your site. It is a good idea to consult your System Administrator, as you may not need any options.
2. Use the [add_processors](#) command to define host machines and the number of processors that the tool can use for slave processes. For example:

SETUP> add_processors lsf:4 machineB:2

specifies for the tool to distribute the processing load among any four available LSF processors and two processors on the machine named “machineB.” Note that with the default settings there would be one slave with four threads from LSF, and one slave with two threads from machineB. And without multithreading enabled, there would be four separate slaves from LSF, and two from machineB.

Note

 If there are no slave processes or additional threads running, the session consumes just one license. However, when you initiate additional processors using the [add_processors](#) command, the tool acquires additional licenses for these processors, with each additional license allowing up to four additional processors. The previous example adds six processors, so two additional licenses would be used.

3. Perform any other tool setups you need for ATPG or fault simulation, then issue the command to be multiprocessed. The tool displays a message indicating the design is being sent to slave processors and then let you know when the slave processors start participating.
4. The following is a list of related commands:
 - [add_processors](#) — Runs multiple processors in parallel on multiple machines to reduce ATPG or fault simulation runtime.
 - [delete_processors](#) — Removes processors previously defined using the add_processors command.
 - [get_multiprocessing_option](#) — Returns the value of a single specified variable previously set with the set_multiprocessing_options command. This is an introspection command that returns a Tcl result.
 - [report_multiprocessing_options](#) — Displays the values of all variables used when executing multiprocessing commands for ATPG or fault simulation.
 - [report_processors](#) — Displays information about the processors used for the current multiprocessing environment.
 - [set_multiprocessing_options](#) — Sets the values of one or more multiprocessing variables.

SSH Environment and Passphrase Errors

Use the information in this section to resolve problems that can occur if you attempt to use the SSH protocol when your environment has not set up the secure shell agent and password.

Error: Cannot change from rsh to ssh in the same tool session

SETUP> report_processors

Error: Cannot change from rsh to ssh in the same tool session.

This error message indicates the [set_multiprocessing_options](#) command was not used to set the remote_shell variable to ssh *prior* to use of the [report_processors](#) or [add_processors](#) command.

Error: Not running a secure shell agent (SSH_AUTH_SOCK does not exist, use ssh-agent)

SETUP> set_multiprocessing_options -remote_shell ssh

```
Permission denied (publickey,password,keyboard-interactive).  
// Error: Not running a secure shell agent ( SSH_AUTH_SOCK does not  
// exist, use ssh-agent ).
```

The SSH functionality checks for the presence of two environment variables, **SSH_AGENT_PID** and **SSH_AUTH_SOCK**, that describe your secure shell session. If not present, it indicates the required ssh-agent daemon is not running.

To fix this problem, suspend the tool session with Control-Z and run the **ssh-agent** shell program. This will start the agent and echo to the screen the required settings for the environment variables. For example:

```
ssh-agent  
setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171;  
setenv SSH_AGENT_PID 13172;
```

Note

 Ensure you remove the trailing semicolons (;) if you copy and paste the ssh-agent output from the shell environment when you resume the tool session.

You can then resume the suspended session and set the environment variables:

SETUP> setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171

SETUP> setenv SSH_AGENT_PID 13172

Then attempt to set **remote_shell** again.

Error: ssh-agent requires passphrase (use ssh-add)

SETUP> set_multiprocessing_options -remote_shell ssh

```
Permission denied (publickey,password,keyboard-interactive).  
// Error: ssh-agent requires passphrase (use ssh-add).  
// Note: SSH protocol not engaged.
```

This error message indicates the SSH agent is running but has not been told the passphrase to allow SSH operations.

To fix this problem, suspend the tool session with Control-Z and run the **ssh-add** shell program:

```
SETUP> ^Z
Stopped (user)
% ssh-add
Could not open a connection to your authentication agent.
% setenv SSH_AUTH_SOCK /tmp/ssh-yXm13171/agent.13171
% setenv SSH_AGENT_PID 13172
% ssh-add
Enter passphrase for /user/.ssh/id_dsa:
```

Enter a passphrase:

```
Enter passphrase for /user/.ssh/id_dsa: xxxxxxxxxxxx
Identity added: /user/.ssh/id_dsa (/user/royb/.ssh/id_dsa)
%
```

Then resume the session and attempt to set remote_shell again:

```
SETUP> set_multiprocessing_options -remote_shell ssh
// Note: SSH protocol is engaged.
```

How to Disable Multithreading Functionality

Multithreading functionality is enabled by default. However, if the tool reads a flat model produced by the pre-v2013.2 simulation kernel, the tool is able to use multithreading functionality only for ATPG and not simulation. In this case, the tool uses only one thread per host to simulate the patterns generated.

You can disable multithreading functionality with the following command:

```
> set_multiprocessing_options -multithreading off
```

Threads Addition to the Master

If you add processors to the master host with multithreading enabled, the master process increases its thread count instead of adding additional slaves. The command transcript lists how many threads were started.

```
> add_processors localhost:4 ;
// adds 4 threads to the master process for a total thread count of 5

// Adding 4 threads to birdeye (master)
// Master with 5 threads running.

> report_processors
```

```
// hosts          threads   arch      CPU(s)    %idle    free RAM process size
// -----  -----  -----  -----  -----  -----
// birdeye (master)      5  x86-64  8 x 2.9 GHz  89%  113.91 MB  232.56 MB
// master with 5 threads running.
```

Note that you can specify the master host by name or IP address.

Processor Addition in Manual Mode

With multithreading enabled, the add_processors command can start only one process per slave host, and create additional processors (to master or slave) as additional threads.

The following example starts a distributed slave process on the host “odin” that runs with 4 total threads.

```
> add_processors odin:4

// Adding 4 threads to odin (new slave)
// Master with 1 thread and 1 slave with 4 threads running (5 total threads).
```

In the following example, the first command starts a distributed slave process on the host “odin” that runs with 2 threads. The second command increases the number of threads in the slave process from 2 to 4 rather than starting a new slave with 2 threads.

```
# odin is not the master host

> add_processors odin:2

// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).

> add_processors odin:2

// Adding 2 threads to odin (existing slave now using 4 threads)
// Master with 1 thread and 1 slave with 4 threads running (5 total threads).
```

Note that if you do not specify a number after the host name, the default is 1.

If you add more threads to a host than it has CPU cores, the command adds the threads but issues a warning that the maximum number of available CPUs has been exceeded. If you specify the string “maxcpu” instead of a number, the tool fills up the process with threads until the number of threads equals the number of processors available to the host. If the host already uses the maxcpu number of threads, the tool issues a warning that no more processors were added. Note that this is true for the master host as well as slave hosts.

For example:

```
# Assume odin has 8 processors and is not the master host

> add_processors odin:2
```

```
// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).
```

> add_processors odin:maxcpu

```
// Adding 6 threads to odin(existing slave now using 8 threads)
// Master with 1 thread and 1 slave with 8 threads running (9 total threads).
```

The “add_processors maxcpu” command always adds enough threads to reach a total of “maxcpu” threads for a host (that is, the second **add_processors** command in the preceding example would always add enough threads to total 8 on the host).

The following is a more detailed example of adding processors in manual mode:

```
# 3 different hosts used: thor, odin, and localhost,
# each with 16 CPUs
```

```
# First show the options used (defaults)
```

> report_multiprocessing_options

```
// Multiprocessing options:
// Option           Type   Value  Description
// -----
// generic_delete   string
// generic_scheduler string
// license_timeout  number 5      # mins to acquire license ( 0: infinite )
// lsf_options      string
// multithreading   on/off on    turn on/off multithreading flow
// processors_per_grid_request number -1     # processors grouped for one grid request
//                               (default: 1 for SGE and GENERIC, 4 for LSF requests)
// remote_shell     string rsh   rsh or ssh remote_shell setting
// result_time_limit float  45    time limit (min) used to detect
//                               non-responsive slaves.
// scheduler_timeout number 10   # mins for job scheduler
// sge_options       string
```

```
# add a slave process with 2 threads on odin
```

> add_processors odin:2

```
// Adding 2 threads to odin (new slave)
// Master with 1 thread and 1 slave with 2 threads running (3 total threads).
```

```
# add a slave process with 1 thread on thor
```

> add_processors thor

```
// Adding 1 thread to thor (new slave)
// Master with 1 thread and 2 slaves with 3 threads running (4 total threads).
```

```
# add 4 additional threads to the master
```

> add_processors localhost:4

```
// Adding 4 threads to masterhost (master)
// Master with 5 threads running.
```

```
> add_processors odin:maxcpu thor:maxcpu
//      Adding 6 threads to odin (existing slave now using 8 threads)
//      Adding 7 threads to thor (existing slave now using 8 threads)
//      Master with 5 threads and 2 slaves with 16 threads running
//      (21 total threads).

# do it again just to get a warning that no more threads are added

> add_processors odin:maxcpu thor:maxcpu
// Warning: Max number of processors on odin already in use (8 exist).
// No processors added.
// Warning: Max number of processors on thor already in use (8 exist).
// No processors added.

> report_processors
//   hosts          threads    arch      CPU(s)    %idle    free RAM  process size
//   -----  -----  -----  -----  -----  -----
//   masterhost (master)      5     x86-64  1 x 2.6 GHz    99%    59.28 MB  256.79 MB
//   odin                  8     x86-64  8 x 2.9 GHz    86%   6525.36 MB  154.53 MB
//   thor                  8     x86-64  8 x 2.8 GHz   100%  58782.55 MB  153.59 MB
//   master with 5 threads and 2 slaves with 16 threads running.
```

Processor Addition in Grid Mode

As in manual mode, the tool doesn't start a new slave process on a host given by the grid engine if there is already a running slave on that machine. Instead, the thread count of the existing slave (or master, if the grid request returns the master host) is increased.

Requesting one processor at a time from the grid engine is inefficient in many cases, as the tool has to request enough memory for a full slave process with every grid request. Therefore, the tool groups a number of processors together into one grid request for a host with enough processors and the memory needed by one slave with the specified number of threads.

So, if the processors_per_grid_request variable is set to 4, then an “add_processors sge:8” command results in two grid requests for 4 processors (slots) each.

The threads started are associated with the requested grid resource, and the resource is freed back to the grid only if all threads are removed.

The following is an example of adding processors in grid mode:

```
# group 4 processors per grid request

> set_multiprocessing_options -processors_per_grid_request 4...
# add 20 processors via LSF in 5 groups of 4 processors each

> add_processors lsf:20
```

```
# this may result in a situation like this:
```

> report_processors

hosts	threads	arch	CPU(s)	%idle	free RAM	process size
// masterhost (master)	5	x86-64	16 x 2.8 GHz	100%	4655.45 MB	189.70 MB
// kraken	4	x86-64	8 x 2.8 GHz	100%	49828.50 MB	151.86 MB
// brighty	8	x86-64	8 x 2.9 GHz	100%	9461.83 MB	152.66 MB
// joker111	4	x86-64	16 x 2.8 GHz	100%	41892.00 MB	166.05 MB
// master with 5 threads and 3 slaves with 16 threads running.						

Note that for LSF, the processors_per_grid is set to 4, and the requested processors_per_grid is automatically provided to the LSF scheduler. However, for SGE or the generic scheduler, the default is set to 1 because bundling of slot requests is site-specific. You have to configure the options specific to that grid system before using this option on non-LSF grids. For more information about how to do this, refer to the [set_multiprocessing_options](#) description in the *Tessent Shell Reference Manual*.

Note

 Take care to exclude any unsupported platforms that exist in your grid environment by specifying appropriate grid job submission constraints. For a list of supported platforms, refer to “[Supported Hardware and Operating Systems](#)” in *Managing Mentor Graphics Tessent Software*.

Processor Addition to the LSF Grid

In addition to the general features described in the previous section, the tool provides some extra assistance for submitting jobs to the LSF grid.

Whereas SGE has predefined machine types and architectures for which the tool can constrain grid requests to hosts that are supported by the tool, LSF requires individual grid installations to classify hosts using site-specific “model” and “type” designations. If you specify “type” constraints in the lsf_options variable (with “[set_multiprocessing_options -lsf_options](#)”), then the tool uses those constraints. Otherwise, the tool attempts to determine which type/model combinations it can use as described below.

By default, the tool uses both LSF scheduler learning and LSF heuristics learning to determine suitable type/model combinations when submitting jobs to the grid. You can enable and disable these two features using the [set_multiprocessing_options](#) switches: `-lsf_learning` and `-lsf_heuristics`. When adding processors to the LSF grid, the sequence of events is as follows:

1. When LSF scheduler learning is enabled, the tool uses the LSF system to determine what machines and type/model combinations are available. It then runs the `lrun` command to log onto likely machines to determine which are compatible with the tool. LSF scheduler learning cannot succeed if your LSF system is configured to prevent use of `lrun`.

Tip

If your system does not support **lsrun**, you may want to disable LSF scheduler learning to avoid the resulting overhead and warning messages.

2. If LSF scheduler learning is disabled or fails, the tool then uses LSF heuristics learning unless you have disabled this feature with “-lsf_heuristics off.” LSF heuristics learning is an attempt to automatically choose machines on the LSF grid that are compatible with the tool. It attempts to determine tool compatibility based on the names of the types and models available and successfully identify some type/model combinations that the tool can use; however, this may also result in specifying incompatible combinations or miss some compatible combinations.

Tip

If you are sure that your slave requests will always result in the tool obtaining compatible hosts (for example, because you specified an LSF host queue with -lsf_options that contains only compatible hosts), then you may want to disable LSF heuristics learning to avoid learning potentially incorrect or incomplete results.

3. If LSF heuristics learning is disabled or fails, then the tool submits the job anyway without specifying any type/model restrictions.

Manual Mode Processor Deletion

The delete_processors command decreases the number of running threads or stops an entire slave process if the number of threads equals zero (or less) after deducting the number of processors to delete from the number of running threads.

If you do not specify a number, the command removes all threads and the slave process on the related host. If you do not specify a number or a host, the command removes all slaves and their threads, as well as all additional threads on the master.

If you issue a delete_processors command for the master, the tool has to make sure that the master is kept running with at least one thread. If a thread is deleted from its process, the process immediately frees all the related thread data.

Grid Mode Processor Deletion

The delete_processors command releases grid slots only if all threads belonging to a grid request were deleted.

For example:

```
# (assume that sge_options have already been configured properly.)  
> set_multiprocessing_options -processors_per_grid_request 4
```

```
# add 20 processors to SGE

> add_processors sge:20

# this may end up in a situation like this:
# each slave has at least 4 threads

> report_processors

#   hosts          threads  arch      CPU(s)    %idle    free RAM  process size
#   -----  -----  -----  -----  -----  -----  -----
# localhost (master)      5  x86-64  16 x 2.8 GHz  100%  4655.45 MB  189.70MB
# kraken                  4  x86-64  8 x 2.8 GHz  100%  49828.50 MB  151.86MB
# brighty                 8  x86-64  8 x 2.9 GHz  100%  9461.83 MB  152.66MB
# joker111                4  x86-64  16 x 2.8 GHz  100%  41892.00 MB  166.05MB
# master with 5 threads and 3 slaves with 16 threads running.

# delete 4 threads from brighty, which frees exactly one grid slot
# (since 4 processors were requested with each grid request)

> delete_processors brighty:4

# delete 2 threads on kraken, which does not free any grid resource

> delete_processors kraken:2

# delete everything from kraken, all grid resources (1 request) are freed

> delete_processors kraken

# What is left?

> report_processors

#   hosts          threads  arch      CPU(s)    %idle    free RAM  process size
#   -----  -----  -----  -----  -----  -----
# localhost (master)  5  x86-64  16 x 2.8 GHz  100%  4655.45 MB  189.70 MB
# brighty            4  x86-64  8 x 2.9 GHz  100%  9461.83 MB  152.66 MB
# joker111           4  x86-64  16 x 2.8 GHz  100%  1892.00 MB  166.05 MB
# master with 5 threads and 2 slaves with 8 threads running.
```


Chapter 10

Scan Pattern Retargeting

Scan pattern retargeting improves efficiency and productivity by enabling you to generate core-level test patterns and *retarget* them for reuse at the top level. Patterns for multiple cores can be merged and applied simultaneously at the chip level. This capability can be used for cores that include any configuration the tool supports for ATPG; this includes multiple EDT blocks and/or uncompressed chains, pipeline stages, low power, and cores with varying shift lengths.

The scan pattern retargeting capability provides these features:

- Design and develop cores in isolation from the rest of the design.
- Generate and verify test patterns of the cores at the core level.
- Generate test patterns for identical cores just once.
- Divide and conquer a large design by testing groups of cores.
- Have pipeline stages and/or inversion between the core boundary and chip level.
- Broadcast stimuli to multiple instances of the same core.
- Merge patterns generated for multiple cores and apply them simultaneously.
- Automatically trace core-level scan pins to the chip level to extract connections, pipelining, and inversion outside the cores.
- Perform reverse mapping of silicon failures from the chip level to the core level to allow diagnosis to be performed at the core level.



Caution

You should verify the chip-level patterns through simulation since the tool's DRCs may not detect every setup error or timing issue.

If you would like to use Tessent Diagnosis to diagnose your retargeted patterns, refer to “Reverse Mapping Top-Level Failures to the Core” in the *Tessent Diagnosis User’s Manual*.

Tools and Licensing Required for Scan Retargeting	498
Scan Pattern Retargeting Process Overview.....	498
Core-Level Pattern Generation	501
Scan Pattern Retargeting of Test Patterns	504
Chip-Level Test Procedures	504
External_capture Procedure	505
Test Procedure Retargeting For Scan Pattern Retargeting	505

Scan Pattern Retargeting Without a Netlist	508
Generating Patterns at the Core Level	512
Retargeting Patterns in Internal Mode	513
Extracting Connectivity Between the Top Level and the Core Boundary	515
Retargeting Patterns Without a Top-Level Netlist	516
Generating Patterns in External Mode	516
Retargeting Example	518
Core-level Pattern Generation for CPU Cores	519
Core-level Pattern Generation for NB Core	520
Retargeting of Patterns at the Chip Level	520
Retargeting of Patterns at the Chip Level Without a Netlist	521
Core-in-Core Pattern Retargeting	523
Core-in-Core Pattern Retargeting Flow	524
Generation of Child Core Internal Mode Patterns	526
Retargeting of Child Core Internal Mode Patterns	526
Generation of Parent Internal Mode Patterns	527
Retargeting of Parent Internal Mode Patterns	528
Scan Pattern Retargeting Limitations	528

Tools and Licensing Required for Scan Retargeting

Scan pattern retargeting requires a Tesson TestKompress license. Reverse mapping of chip-level failures, as well as diagnosis of those failures, requires a Tesson Diagnosis license.

Scan pattern retargeting and the generation of core-level retargetable patterns must be done within the Tesson Shell tool. Instructions for invoking Tesson Shell are provided in the following section.

For complete information about using Tesson Shell, refer to the *Tesson Shell User's Manual*.

Scan Pattern Retargeting Process Overview

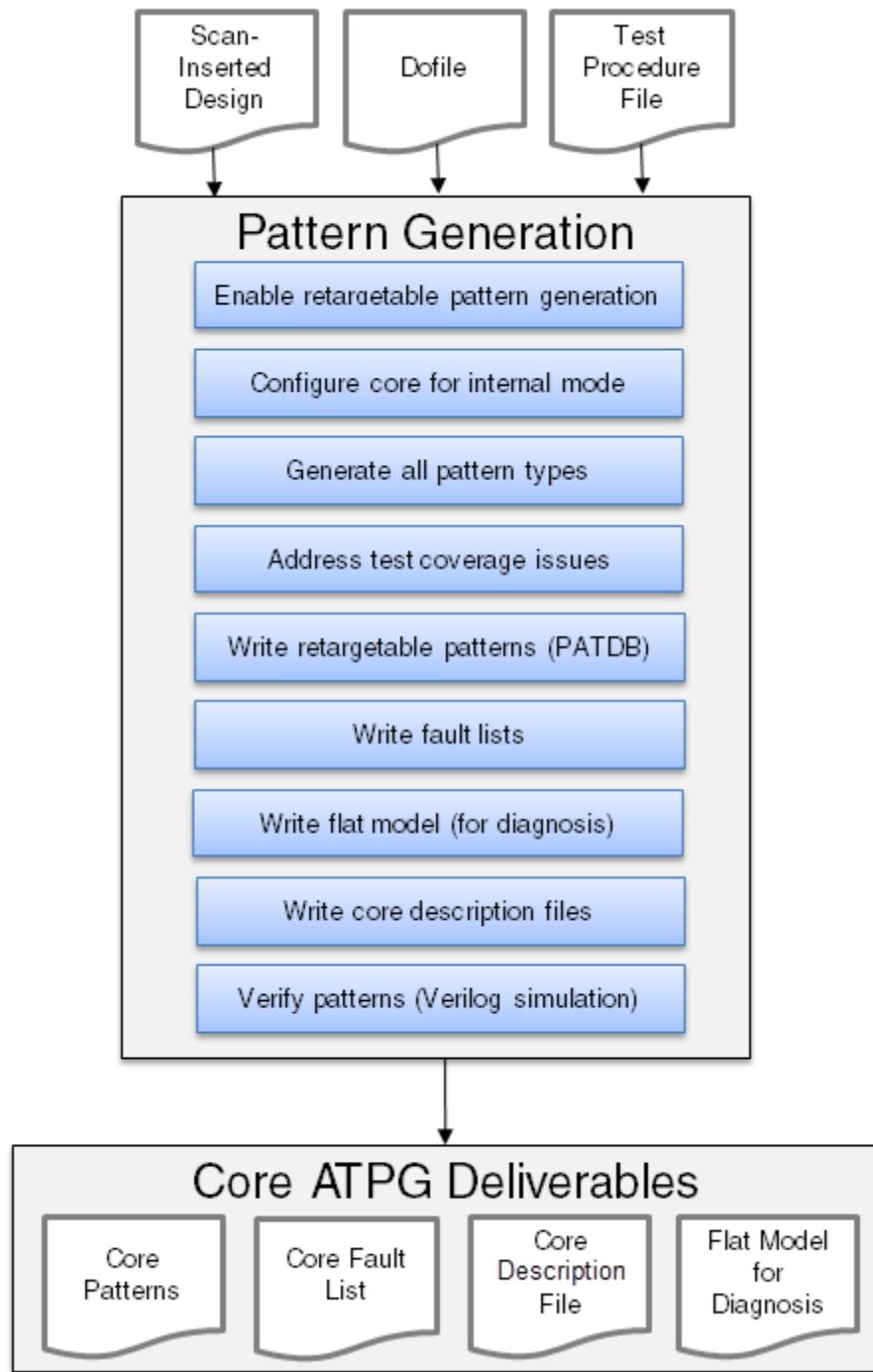
The scan retargeting process consists of generating patterns for all cores, retargeting the core-level test patterns to the chip level, and generating patterns for the top-level chip logic only.

An example of the scan pattern retargeting process is presented in “[Retargeting Example](#)” on page 518.

This is a description of the scan pattern retargeting process:

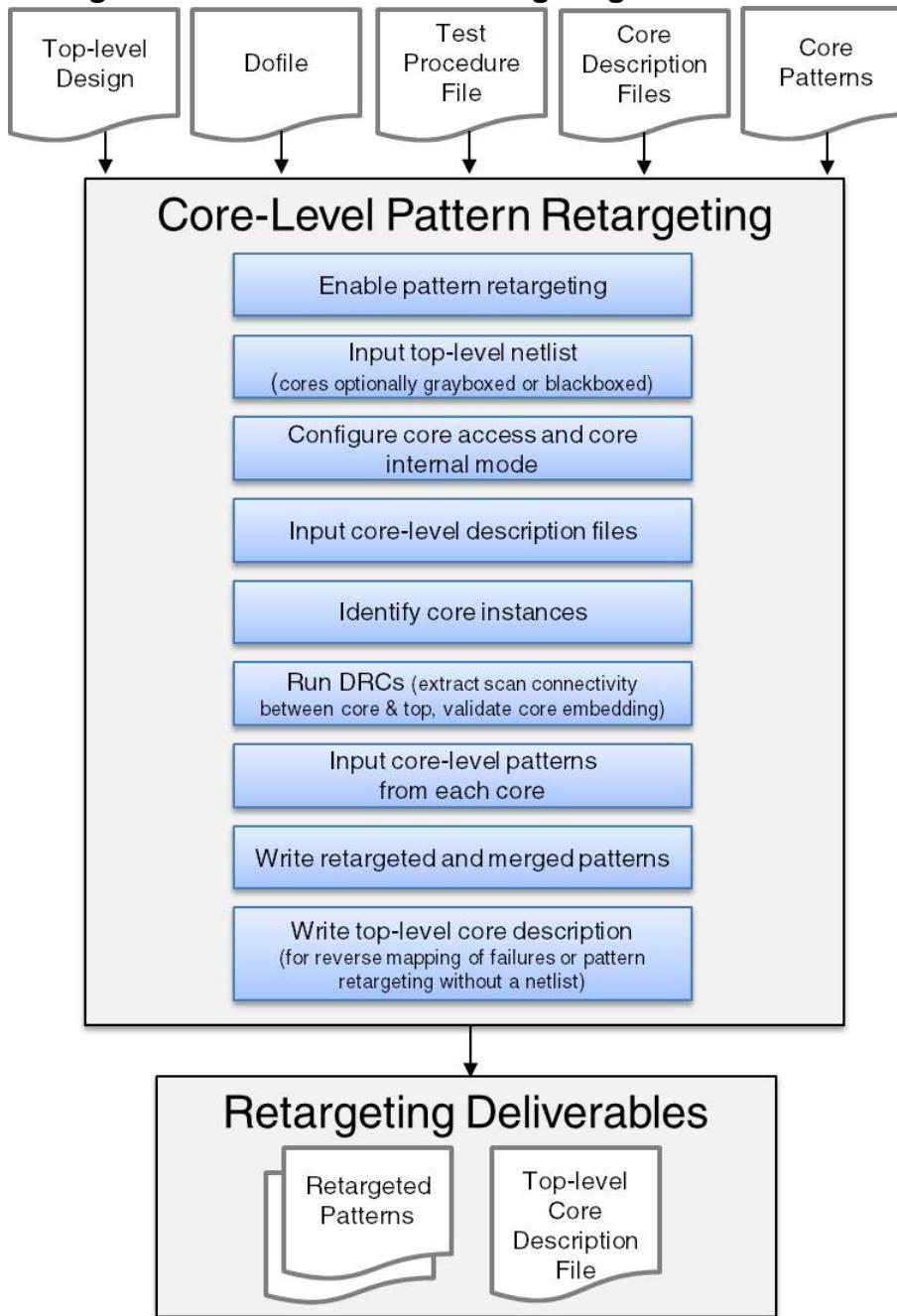
1. Generate patterns for all cores. This process is illustrated in Figure 10-1. For more information, see “[Generating Patterns at the Core Level](#)” on page 512.”

Figure 10-1. Core-level Pattern Generation Process

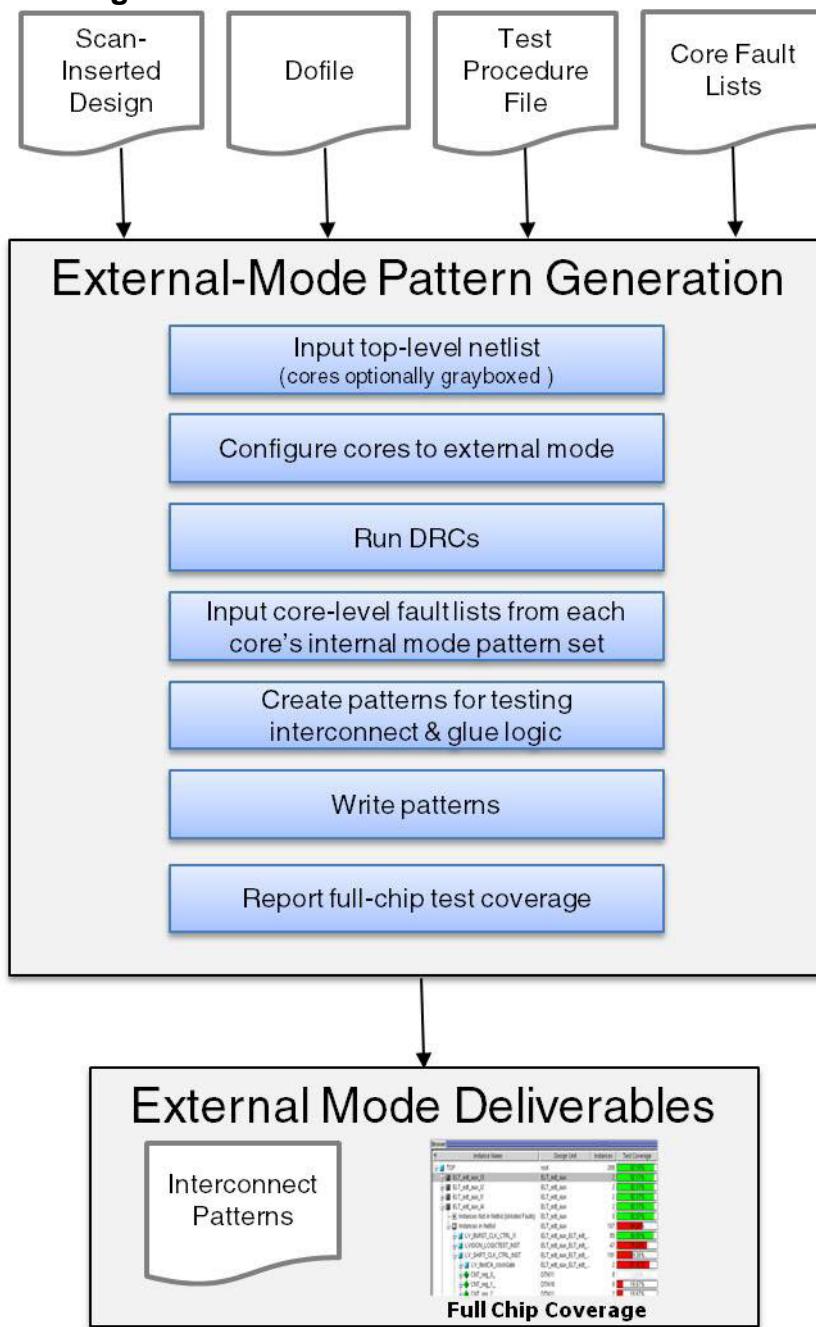


2. Retarget core-level test patterns to the chip level. This process is illustrated in Figure 10-2. For more information, see “[Retargeting Patterns in Internal Mode](#)” on page 513.”

Figure 10-2. Scan Pattern Retargeting - Internal Mode



3. Generate patterns for the top-level chip logic only. This process is illustrated in Figure 10-3. For more information, see “[Generating Patterns in External Mode](#)” on page 516.

Figure 10-3. Pattern Generation - External Mode

To enable core-level generation of retargetable patterns, you must execute “`set_current_mode [mode_name]` -type internal” in setup mode.

When later retargeting core-level patterns to the top level, the tool maps only scan data from the core level to the top level. The tool does not map core-level PIs and POs to the top level. Nor does the tool map core-level capture cycle events from the core level to the top level. Therefore, primary inputs must be constrained during capture, outputs must be masked, and clocking must meet certain requirements.

The PIs and POs at the core level are not valid fault sites when the core is embedded at the next level up, which creates inconsistencies when accounting for faults at the core level and the faults at the top level. The PIs and POs will be treated by the tool as no-fault sites when it runs in the internal mode type.

During core-level pattern generation, the tool automatically adds X constraints on most input ports and adds output masks on all output ports so that generated patterns can be mapped to the chip level for retargeting. Pulse-always clocks, pulse-in-capture clocks, and constrained pins are not forced to X. The tool also omits internal primary inputs (added with `add_clocks` or `add_primary_inputs` commands) since those cut points are for modeling and are usually controlled by Named Capture Procedures (NCPs) or clock control definitions.

You can exclude additional primary inputs from X constraints; this is necessary if the pins are controlled by an NCP that models chip-level constraints such as a clock controller located outside of the core. All inputs must be constrained or controlled (for example, using all enabled NCPs).

Note

 Even `scan_enable`, like any other input, must either be constrained to 0 or forced using an NCP.

If a pin is not a pulse-always or pulse-in-capture clock, is not constrained, and is excluded from isolation constraints (that is, the tool did not constrain it to X), the `create_patterns` command checks that the pin is controlled. If the pin is not controlled, the tool issues an R7 DRC violation.

Be careful when you apply the `is_excluded_from_isolation_constraints` attribute to a port, and never ignore an R7 DRC violation. Doing this can result in ATPG controlling the unconstrained port to any arbitrary value which can vary from pattern to pattern, even though pattern retargeting only retargets the scan data and does NOT retarget values on primary inputs from the core level to the top.

Core-level test patterns can be saved in the pattern database (PatDB) format when using the `write_patterns` command. While patterns saved in PatDB or ASCII format can be retargeted, PatDB is the recommended format. You can also use the `write_patterns` command, as in the normal ATPG flow, to generate simulation test benches or write the patterns in other formats.

You can use existing commands to write out core-level fault list and fault detection information.

Caution

 When generating the EDT IP, if output channel pipeline stages will be added later, you must specify “set_edt_pins -change_edge_at_compactor_output trailing_edge” to ensure that the compactor output changes consistently on the trailing edge of the EDT clock. Output channel pipeline stages should then start with leading-edge sequential elements.

Clocking Architecture

The scan pattern retargeting functionality imposes the following restrictions on clock architecture:

- The tool does not perform per-pattern capture cycle mapping. The capture clock conditions at the core boundary must be identical for every pattern in the pattern set.
- The recommended methodology is to have a programmable clock chopper inside each core that is programmed by scan cell values. The scan cells used to program the clock chopper must be inside the core. This provides ATPG with the flexibility to generate the required clocking sequences, yet ensures that the clock conditions at the boundary of the cores are the same (pulse-always clocks, or clocks that are pulsed only during setup and/or shift but constrained off during capture).
- If the clock controller is outside the core, it must have been programmed statically during test_setup to deliver the same clock sequence during the capture phase of every pattern. During core-level ATPG, the clocking at the core boundary must be enforced by defining an NCP. In addition, any unconstrained pins on the core boundary must have the is_excluded_from_isolation_constraints attribute set; otherwise, the tool will constrain them to X.
- No support is provided if the clocking on the boundary of the core is different for each pattern, such as when the capture clocks are programmed using a scan chain outside the core, or when the capture clocks are driven directly by chip-level pins.
- The standard OCC is recommended for use in hierarchical cores. For more information, see “[Core OCC Recommendation](#)”.

The Tessent Core Description File

The pattern generation step produces a Tessent core description (TCD) file. The TCD file contains the information the tool needs to map the core-level patterns to the next level of hierarchy. This includes information such as description of the EDT hardware and scan chains.

Scan Pattern Retargeting of Test Patterns

To retarget core-level test patterns, Tessent Shell maps the scan path pins from the cores to the top level, adjusts the core-level patterns based on the additional retargeting pipeline stages learned between the core and chip level (to ensure that all patterns from different cores have the same shift length via padding), and maps them to the chip-level pins they connect to.

Note that retargeting of core-level patterns does *not* require you to provide a full netlist for every core. Alternatively, you can specify a core as either a graybox or blackbox in the following ways:

- If you have generated a graybox and want to handle the core as a graybox model, use the `read_verilog` command to read the Verilog model that was written out from the graybox generation step. Although the wrapper scan chains in the graybox model are not needed for the retargeting mode, feedthroughs are preserved in the graybox model. If feedthroughs are used for connecting cores to the chip level and blackboxing the cores would break that path from the core boundary to the top, it is recommended to use the graybox model.
- If you have the full core netlist in a separate file from the top-level module(s) and just want to read it in and preserve the boundary but discard the contents, use the “`read_verilog design -interface_only`” command. This method can only be used if there are no feedthroughs or control logic within the core that are needed for the current mode of operation.
- If you want to read the full design and blackbox specific core instances during design flattening, use the `add_black_box` command. This method can only be used if there are no feedthroughs or control logic within the core that are needed for the current mode of operation.

Chip-Level Test Procedures	504
External_capture Procedure.....	505
Test Procedure Retargeting For Scan Pattern Retargeting.....	505
Scan Pattern Retargeting Without a Netlist	508

Chip-Level Test Procedures

You must provide the chip-level `test_setup` procedure needed for the retargeting step.

You must also either provide the chip-level `load_unload` and `shift` procedures, or allow the tool to create them automatically by mapping and merging the core-level `load_unload` and `shift` procedures as described in “[Test Procedure Retargeting For Scan Pattern Retargeting](#)” on page 505.

External_capture Procedure

You specify the capture cycle in an external_capture procedure, using the set_external_capture_options -capture_procedure switch. The -capture_procedure switch specifies the external_capture procedure to be used. The tool uses this external capture procedure for all capture cycles between each scan load, even when the pattern is a multi-load pattern.

An example external_capture procedure is shown here:

```
procedure external_capture ext_procedure_1 =
    timeplate tmp_1;
    cycle =
        force_pi;
        pulse clk_trigger 1;
        pulse reference_clock;
    end;
    cycle =
        pulse reference_clock;
    end;
end;
```

The external_capture procedure has the same syntax as an NCP, except it does not have internal and external modes. In comparison to an NCP, the external_capture procedure has the following restrictions:

- Can only have one force_pi statement.
- Cannot have any measure_po statements.
- Cannot have any load_cycles as it is used between each scan load of a pattern.
- Cannot contain any events on internal signals.

Pin constraints defined at the top level are used in this generic capture sequence. No pattern-specific capture information is mapped from the patterns.

Test Procedure Retargeting For Scan Pattern Retargeting

The automatic mapping (transfer) of test procedure information from the core level to the top level of the design occurs when the tool transitions to analysis mode.

load_unload and shift Procedures

When core-level patterns are retargeted to the chip-level, chip-level load_unload and shift test procedures are needed. If chip-level load_unload and shift test procedures do not exist, the tool automatically generates them based on the information in the core-level TCD files. This

retargeting step merges all of the core test procedures (if there is more than one core), and maps them to the top-level test procedures.

Clocks and Pin Constraints

By default, the tool maps all clocks and pin constraints that are specified at the core-level to the top. Clock types are mapped based on the core-level specification. Note, the tool maps any permanently tied constraints (CT) in those files as constraints that can be overridden (C).

Even though clocks and pin constraints are automatically mapped to the top level by default, you can still define them at the top, or drive the core-level input constraints through logic such as a TAP.

Any clocks that are mapped to the top are removed when the tool exits analysis mode.

test_setup and test_end Procedures

You will typically need to provide top-level test_setup and test_end procedures to configure instruments such as PLLs, OCCs, EDT IPs, channel multiplexer access, and so on in preparation for scan test. These instruments may be at the top level and/or within the cores. The test_setup and test_end procedures can consist of a cyclized sequence of events, and/or IJTAG iCall statements.

If any of the instruments used for scan test are inside a core, a test_setup and/or test_end procedure at the core level must have been used to initialize them. If IJTAG was used within a core-level test_setup or test_end procedure, this information is retained in the TCD file. When an instance of this core is added for mapping, the iCalls that were used at the core level are automatically mapped to the top level and added to the top-level test_setup or test_end procedure. In other words, if you use IJTAG to set up the core at the core level, the tool automatically maps those sequences to the top-level test_setup and test_end procedures without you doing anything. Of course, the top-level test_setup and test_end procedures may have additional events to configure top-level instruments.

In order for the tool to perform automatic IJTAG mapping, you must have extracted the chip-level ICL using the [extract_icl](#) command. You must also have sourced the core-level PDL. This step-by-step procedure is described in the “[Performing ICL Extraction](#)” section in the *Tessent IJTAG User’s Manual*.

Note

 If you have not previously extracted ICL in a separate step, you can still execute the normal scan pattern retargeting flow with one change. After entering the patterns -scan context, you need to read ICL for each of the cores, extract the ICL using the [extract_icl](#) command, read the PDL file for each module, and then proceed with the rest of the scan retargeting flow.

Make sure to call the [extract_icl](#) command before you issue the [set_procfile_name](#) command otherwise it will be understood that the test_setup proc located in the specified proc file is

actually needed to extract the ICL. The test_setup proc will be simulated and used to establish the back ground simulation values when tracing and extracting the ICL network. Because this test_setup was simulated to do ICL extraction, its name will end up in the extracted ICL and the [process_patterns_specification](#) command will give you an error if you do not refer to it using the procfile_name property inside the [AdvancedOptions](#) wrapper.

Several DRCs validate the presence of both ICL and PDL. Note that the tool does not store ICL and PDL in the core-level TCD file; it only stores references to their usage in the test_setup procedure.

Note

 You can disable load_unload and shift procedure retargeting by executing the “[set_procedure_retargeting_options -scan off](#)” command. If test procedure retargeting is disabled and chip-level load_unload and shift test procedures are missing, the tool generates an error.

You can disable the automated mapping of core-level iCalls stored in the TCD file to the top level by executing the “[set_procedure_retargeting_options -ijtag off](#)” command. (This will not have any impact on iCalls you explicitly added into the top-level setup procedures either explicitly or indirectly using the [set_test_setup_icall](#) command.) If IJTAG retargeting is disabled, you must provide the needed test_setup and test_end procedures.

Constraint of the Top-Level Port During the Load_Unload Test Procedure

You can set a constant value for a top-level port during the simulation of the load_unload test procedure, when needed, to enable tracing of signals from the core level to the chip level. You can do this by setting the value of the constraint_value_during_load_unload attribute using the [set_attribute_value](#) command.

If, for example, scan_enable must be 1 to trace signals needed for load_unload and shift procedure retargeting, you would need to define this load_unload constraint in the tool for test procedure retargeting to work; especially, because this value will likely conflict with the 0 input constraint typically added to scan_enable for capture.

For more information see the [set_attribute_value](#) command and the [Port](#) data model attributes in the *Tessent Shell Reference Manual*.

Scan Pattern Retargeting Without a Netlist

Before describing the process of scan pattern retargeting without a netlist, it is important to clarify that the retargeting flow consists of two distinct steps.

The two steps for scan pattern retargeting without a netlist are extraction and retargeting:

- **Extraction** — This step is the process of determining the connectivity from a lower-level wrapped core to the current top level of hierarchy. It validates that the core is embedded properly with respect to the signals and constrained values necessary for retargeting (for example, scan chains/channels, clocks, and constrained inputs). This step is also where we perform optional mapping of the test procedures. Once the extraction step is complete, a TCD file can be written out that captures the information needed for retargeting core-level patterns to the current design level.
- **Retargeting** — This is the step in which the core-level patterns that have been read in are then mapped to the top-level pins, and multiple pattern sets are merged together for simultaneous application.

These steps are discrete and you can perform them separately in different sessions or together.

[Figure 10-2](#) shows retargeting as a single pass procedure but you can separate these two steps out of the process that is illustrated. The extraction step occurs when you execute the “set_system_mode analysis” command and perform DRC. All connectivity between the core and the top is extracted and validated. After extraction, you use the write_core_description command to write out the top-level TCD file which includes the extracted connectivity and top-level scan procedures. This top-level TCD file includes all of the core-level TCD information in this mode so that it is self-contained. You can stop at this point, and not read and write the core-level retargetable patterns, and you will have completed the extraction step only. At this point, you can write out the top-level TCD file. If, instead, you continue and read and write the core-level retargetable patterns before writing out the top-level TCD file, you will have completed the pattern retargeting step.

When performing the extraction step, you are not required to provide a full netlist for every core; instead, you can specify a core as either a graybox or blackbox. The tool retrieves all of the information necessary for retargeting its patterns from the core-level TCD file. These are the only reasons that you would not want to blackbox a core:

- If you have a core-level test_setup procedure. The R5 DRC (test_setup validation) cannot be performed if the setup logic in the core is not present during extraction DRC.
- If the logic in the core is involved in the top-level procedure simulation and access from the top to the core. For example, if the core includes feed-throughs of currently active cores, or includes the TAP for the design which is needed for simulating the test_setup procedure.

You are able to perform pattern retargeting without performing extraction again because you can utilize the previously generated information in the top-level TCD file. The top-level TCD

file includes all of the information needed to perform retargeting including the core-level descriptions, connectivity of the scan pins, and top-level scan procedures used and validated during extraction. You only need to read in the top-level TCD file and the core-level patterns to perform pattern retargeting. This enables efficient retargeting of new core-level patterns if the design and procedures have not changed since DRC and core connectivity extraction were performed; if this is not the case, you will need to rerun extraction again.

If anything changes with the setup of the core, such as procedures or constraints, you must rerun extraction and DRC because the previously extracted top-level TCD may no longer be valid.

Retargeting Support For Patterns Generated at the Top Level 509

Retargeting Support For Patterns Generated at the Top Level

Scan patterns are typically retargeted with respect to lower-level cores instantiated in the design; in this case, the core is a lower-level core. However, in some cases, you can test part of the top-level logic in parallel with lower-level cores by treating this partial view of the top level as a core instance; in this case, the core is actually the top level.

During pattern retargeting, you use the `add_core_instances -current_design` command and option to add the top-level logic as a core instance. This option allows you to add the core instance with respect to the current design; this core instance then becomes the top level. The `-current_design` option also allows patterns for the current design to be merged with patterns of other core instances in the hierarchy as shown in “[Example 2](#)” on page 511.

A core can have multiple modes. A *mode* of a core is a specific configuration or view of that design with a specific set of procedures and scan definitions. Two modes of a design can be mutually exclusive (such as the internal and external test of a wrapper core), or independent of each other (such as when we generate patterns for non-overlapping parts of a core and then merge those patterns as in “[Example 1](#)” on page 509).

Example 1

The following example demonstrates the use of the `-current_design` switch to merge two modes of a core; in this case, the “core” is the top level. In the example, patterns are generated separately for two sub-blocks of a design from the top-level ports. Sub-Block1 is tested as mode M1 of the top level, and Sub-Block2 is tested as mode M2 of the top level. The result, shown in [Figure 10-4](#), is the merging of the two pattern sets originally generated for each individual mode into a single pattern set applied at the top level.

The first dofile reads in the top-level design and writes out the patterns and TCD files for mode M1 targeting Sub-Block1:

```
// Invoke tool to generate patterns for each mode of top_level
set_context pattern -scan
read_verilog top_level.v //Read netlist of top_level
// Mode M1 consists of the Sub-Block1 design; the user applies constraints to access
// Sub-Block1 and can optionally blackbox Sub-Block2 to reduce memory consumption
set_current_mode M1 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM1.patdb -patdb
write_core_description top_levelM1.tcd
```

The second dofile reads in the design and writes out the patterns and TCD files for mode M2 targeting Sub-Block2:

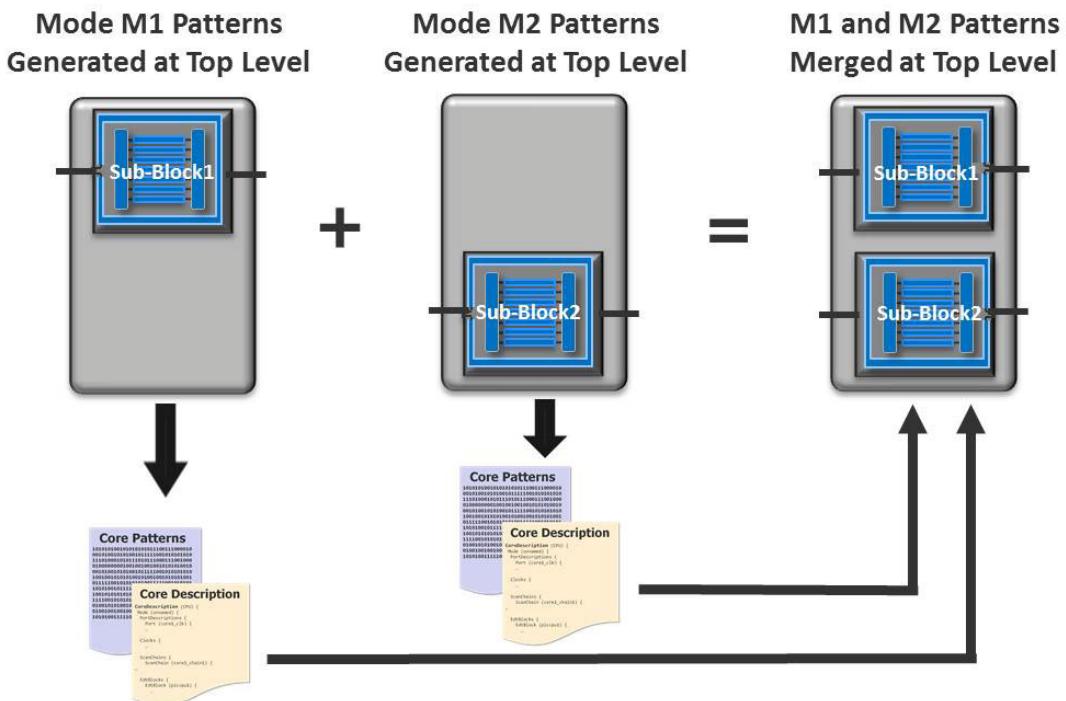
```
// Mode M2 consists of the Sub-Block2 design; the user applies constraints to access
// Sub-Block2 and can optionally blackbox Sub-Block1 to reduce memory consumption
set_current_mode M2 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM2.patdb -patdb
write_core_description top_levelM2.tcd
```

The third dofile merges the two separately generated top-level pattern sets (modes M1 and M2) into a single pattern set:

```
// Invoke the tool again to retarget patterns to top level
set_context patterns -scan_retargeting
...
read_core_descriptions top_levelM1.tcd
read_core_descriptions top_levelM2.tcd
add_core_instances -core top_level -mode M1 -current_design
add_core_instances -core top_level -mode M2 -current_design
set_system_mode analysis

// Mode extracted from PATDB file
read_patterns top_levelM1.patdb
read_patterns top_levelM2.patdb
write_patterns
...
```

Figure 10-4. Merging Patterns for Multiple Views of a Core at the Top Level



Example 2

The following example demonstrates the use of the `add_core_instances -current_design` switch to merge mode M1 of the current design and mode internal of CoreB. In the example, one pattern set is generated at the top level (mode M1 of the top level) to test Sub-Block1, and a second pattern set is generated at the core level of CoreB. The result, as shown in Figure 10-5, is the integration of the two pattern sets, originally generated for mode M1 of the top level and mode internal of CoreB, into the top level.

The first dofile reads in the top level design and writes out the patterns and TCD files for mode M1 targeting Sub-Block1.

```
// Invoke tool to generate patterns for mode M1 of the top level
set_context patterns -scan
read_verilog top_level.v // Read netlist of top_level
// Mode M1 consists of the Sub-Block1 design; the user applies constraints to access
// Sub-Block1 and can optionally blackbox any other blocks not targeted to reduce memory
// consumption
set_current_mode M1 -type internal
...
set_system_mode analysis
...
create_patterns
write_patterns top_levelM1.patdb -patdb
write_core_description top_levelM1.tcd
```

The second dofile reads in the CoreB design and writes out the patterns and TCD files.

```

set_system_mode setup
delete_design
read_verilog CoreB.v
set_current_mode internal -type internal
...
set_system_mode analysis
create_patterns
write_patterns CoreB.patdb -patdb
write_core_description CoreB.tcd

```

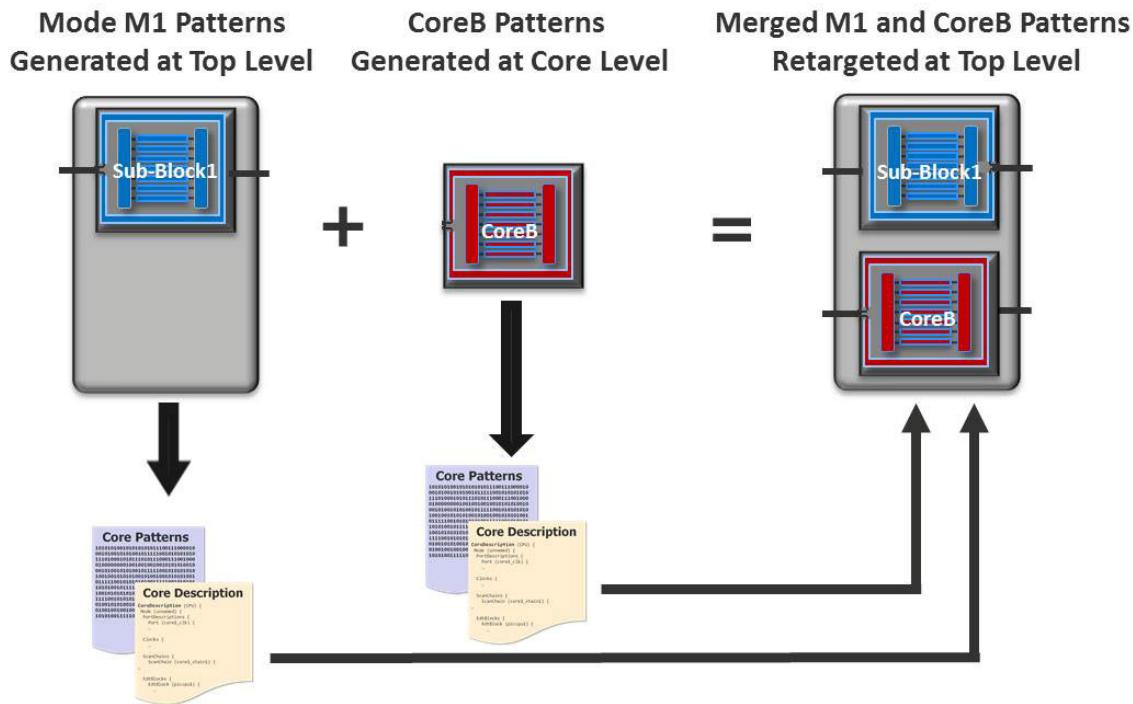
The third dofile merges the top-level pattern set (mode M1) with the retargeted CoreB patterns.

```

// Invoke tool to retarget patterns to top level
set_context patterns -scan_retargeting
...
read_core_descriptions top_levelM1.tcd
read_core_descriptions CoreB.tcd
add_core_instances -core top_level -mode M1 -current_design
add_core_instances -core CoreB -mode internal -instances CoreB_i
...
set_system_mode analysis
read_patterns top_levelM1.patdb
read_patterns CoreB.patdb
write_patterns
...

```

Figure 10-5. Merging Patterns for Top Level and Core at the Top Level



Generating Patterns at the Core Level

This procedure describes how to generate core-level test patterns.

Prerequisites

- Core-level netlist
- Cell library

Procedure

1. Enable ATPG using the “[set_context](#) patterns -scan” command.
2. Read the core-level netlist and cell library.
3. Configure the core for internal mode.
4. Generate all pattern types using the [create_patterns](#) command.

If you are performing ATPG when the mode type is internal and you run [create_patterns](#) on the IDDQ fault model, the tool will issue this warning:

```
// Warning: Creating IDDQ patterns with the mode type set to
// 'internal'. IDDQ patterns cannot be retargeted.
```

5. Address all test coverage issues.
6. Verify the test patterns (test benches, STA).
7. Write the retargetable core test patterns using the [write_patterns](#) command.
8. Write the core fault lists using the [write_faults](#) command.
9. Write the TCD file using the [write_core_description](#) command.
10. Write the flat model for diagnosis using the [write_flat_model](#) command.

Retargeting Patterns in Internal Mode

This procedure describes how to retarget core-level test patterns to the chip level.

Prerequisites

- TCD file for each of the core types.
- Retargetable patterns resulting from core-level pattern generation.
- Top-level netlist (with cores optionally blackboxed or grayboxed).
- Top-level test procedures.

Procedure

1. Enable the retargeting of patterns using the “[set_context](#) patterns -scan_retargeting” command.

Note

-  A full netlist for cores whose patterns are being retargeted is *not* required. Retargeting only requires a graybox or blackbox model of the cores to both retarget the patterns and to generate chip-level serial and parallel simulation test benches.
-

2. Read the TCD file for each of the core types.
3. Bind each core description to the core instances it represents in the design using the [add_core_instances](#) command.
4. Read a dofile and test procedure file to configure core access and to configure the cores whose patterns are being retargeted into their internal mode.
5. Run DRCs when “[set_system_mode analysis](#)” is invoked, including extracting scan connections from the core instances to the top level and validating the embedding of the cores.
6. Read in the retargetable patterns for each core type using the [read_patterns](#) command.

Be aware that IDDQ patterns are not supported for scan pattern retargeting. If they are read in, the tool will issue the following message:

```
// Error: IDDQ patterns are not supported for scan pattern
// retargeting
```

7. Write the retargeted patterns using the [write_patterns](#) command. The tool automatically performs pin mapping (between core and chip-level) and pattern merging as it writes the chip-level patterns.
8. Write the top-level TCD file. This file includes the core-level core description information as well as scan connectivity information from the cores to the top. This information is later used for reverse mapping of silicon failures back to the core level for diagnosis, or for pattern retargeting without a netlist.

Note

-  The top-level TCD file also enables generation of top-level patterns without a netlist. For more information, see “[Retargeting of Patterns at the Chip Level Without a Netlist](#).”
-

Chip-level patterns cannot be read back into Tessent Shell since the internal clocking information is lost. However this is not needed since diagnosis is done at the core-level. The generated simulation test bench can be simulated for validation. In addition, the design objective of this functionality is to eliminate the need to load the full design into the tool.

Extracting Connectivity Between the Top Level and the Core Boundary

When you want to separate the extraction and retargeting steps, you can extract the connectivity from the core level to the top level and create the top-level TCD file.

You can then perform the retargeting. See “[Retargeting Patterns Without a Top-Level Netlist](#)” on page 516.

Prerequisites

- Core-level netlists. The core can optionally be blackboxed or grayboxed.
- Core-level TCD files
- Top-level netlist

Procedure

1. Enable the retargeting of patterns using the “[set_context patterns -scan_retargeting](#)” command.

Note

 A full netlist for cores whose patterns are being retargeted is *not* required. Retargeting only requires a graybox or blackbox model of the cores to both retarget the patterns and to generate chip-level serial and parallel simulation test benches.

2. Read the TCD file for each of the core types.
3. Bind each core description to the core instances it represents in the design using the [add_core_instances](#) command.
4. Read a dofile and test procedure file to configure core access and to configure the cores whose patterns are being retargeted into their internal mode.
5. Perform extraction using the “[set_system_mode analysis](#)” command. This command runs DRCs, extracts scan connections from the core instances to the top level and validates the embedding of the cores.
6. Write the top-level TCD file using the [write_core_description](#) command. This file includes the core-level core description information as well as scan connectivity information from the cores to the top. This information is later used for reverse mapping of silicon failures back to the core level for diagnosis, or for pattern retargeting without a netlist.

Results

Top-level TCD file.

Related Topics

- [Scan Pattern Retargeting Without a Netlist](#)
- [Retargeting Patterns Without a Top-Level Netlist](#)

Retargeting Patterns Without a Top-Level Netlist

You can retarget core-level patterns at the top level without a netlist by using the information contained in the top-level TCD file.

Prerequisites

- Top-level TCD file. The top-level TCD file results from performing the extraction step described in “[Extracting Connectivity Between the Top Level and the Core Boundary](#)” on page 515.
- Retargetable patterns resulting from core-level pattern generation.

Procedure

1. Enable the retargeting of patterns using the “`set_context patterns -scan_retargeting`” command.
2. Read the top-level TCD file using the `read_core_descriptions` command. In the absence of a netlist, the tool recreates what is needed for the design using this file.
3. Change to analysis mode using “`set_system_mode analysis`”. This command automatically sets the current design, adds the cores, sets pin constraints, and provides any other information needed for the design.
4. Read in the retargetable patterns for each core type using the `read_patterns` command.
5. Report core instances using the `report_core_instances` command.
6. Retarget and merge patterns as they are written out with respect to the chip-level boundary using the `write_patterns` command.

Generating Patterns in External Mode

This procedure describes how to generate patterns in external mode.

Prerequisites

- Complete top-level netlist or top-level netlist with cores grayboxed.
- Core-level fault lists.
- Top-level dofile.

- Top-level test procedure file.

Procedure

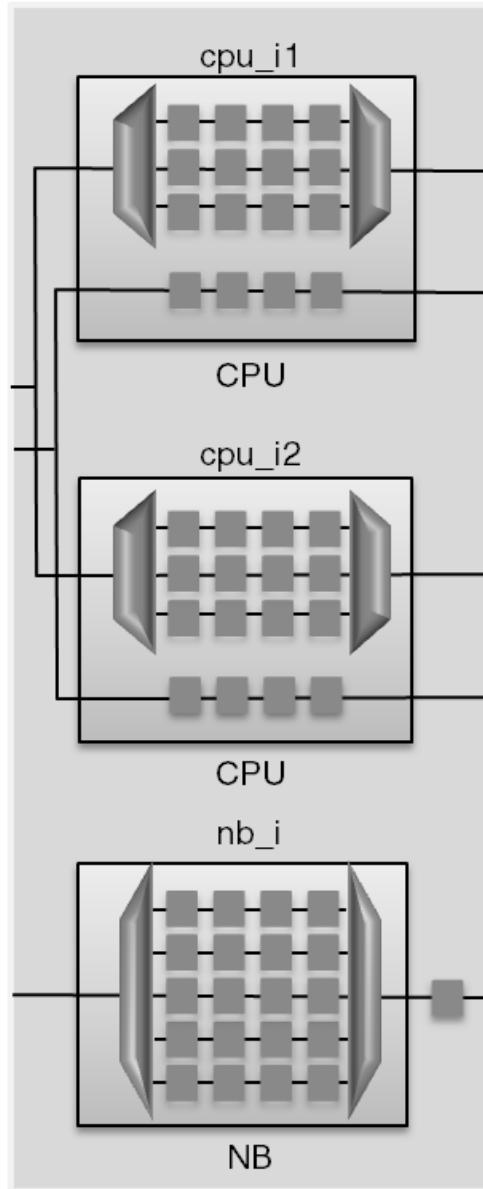
1. Enable the generation of patterns using the “[set_context](#) patterns -scan” command.
2. Read a complete top-level netlist, or one where each core is replaced by a light-weight graybox model, and the cell library.
3. Read a dofile and test procedure file to configure cores to external mode.
4. Run DRCs when “[set_system_mode analysis](#)” is invoked.
5. Create patterns for testing interconnect and top-level glue logic.
6. Write the generated patterns using the [write_patterns](#) command.
7. Read the core-level fault lists from each core’s internal mode pattern set. If the netlist is not complete due to the use of the graybox models, add the -graybox switch to the “`read_faults fault_file -instance instances -merge`” command.
8. Report the full chip test coverage.

Retargeting Example

This example demonstrates the retargeting use model.

This example is based on the design shown in [Figure 10-6](#). The design has three cores of two types: two identical instances of the CPU core and one instance of the NB core. TOP is the chip level of the design. In the example, the input is broadcast to identical core instances.

Figure 10-6. Retargeting of Core-level Patterns



Core-level Pattern Generation for CPU Cores	519
Core-level Pattern Generation for NB Core	520
Retargeting of Patterns at the Chip Level.....	520

Core-level Pattern Generation for CPU Cores

Core-level pattern generation for the CPU core is shown here.

```
# Set the context for ATPG
set_context patterns -scan

# Read cell library (ATPG library file)
read_cell_library atplib

# Read the core netlist
read_verilog CPU.v

# Specify to generate patterns that are retargetable for the internal mode
# of the core. Because the mode name was not specified, the default is
# the same as the mode type "internal".
set_current_mode -type internal

# Specify pins the tool should not constrain to X because they will be
# explicitly controlled by a Named Capture Procedure that models a
# chip-level clock controller driving core-level clocks
set_attribute_value {clk1 clk2} \
    -name is_excluded_from_isolation_constraints

# Add commands to set up design for ATPG. To avoid coverage loss due to
# auto isolation constraints on scan_enable signals, constrain scan_enable
# signals to a non-X state
...
add_input_constraints input_scan_enable -C1
add_input_constraints output_scan_enable -C0
add_input_constraints core_scan_enable -C0

# Change to analysis mode
set_system_mode analysis

# Generate TCD file
write_core_description CPU.tcd -replace

# Specify the fault model type and generate patterns
set_fault_type stuck
create_patterns

# Write patterns for subsequent retargeting; PatDB is recommended format.
write_patterns CPU_stuck.pat -patdb -replace

# Write fault list
write_faults CPU_stuck.faults.gz -replace

# Write flat model for diagnosis
write_flat_model CPU_stuck.flat_model.gz -replace
```

Core-level Pattern Generation for NB Core

Core-level pattern generation for the NB core is the same as for the CPU core.

```
# Set the context for ATPG
set_context patterns -scan

# Read cell library (ATPG library file)
read_cell_library atpglib

# Read core netlist
read_verilog NB.v

# Specify to generate patterns that are retargetable for the internal mode
# of the core. Because the mode name was not specified, the default is the
# same as the mode type "internal".
set_current_mode -type internal
...
```

Retargeting of Patterns at the Chip Level

The steps for scan pattern retargeting are shown here. The tool performs design rule checking as it changes to analysis mode.

```

# Specify that this is the scan pattern retargeting phase
set_context patterns -scan_retargeting

# Read cell library (ATPG library file)
read_cell_library atpglib

# Read all netlists and the current design. You can replace the core
# netlists with graybox models, or blackbox them to reduce memory usage
# and run time. You can blackbox them using the
# read_verilog -interface_only, or add_black_box commands.
read_verilog TOP.v
read_verilog CPU.v -interface_only
read_verilog NB.v -interface_only
set_current_design TOP

# Specify the top-level test procedure file
set_procfile_name TOP.testproc

# Read all TCD files
read_core_descriptions CPU.tcd
read_core_descriptions NB.tcd

# Bind core descriptions to cores
add_core_instances -core CPU -modules CPU
add_core_instances -core NB -instances /nb_i

# Change to analysis mode
set_system_mode analysis

# Read core (retargetable) patterns
read_patterns CPU_stuck.retpat
read_patterns NB_stuck.retpat

# Specify generic capture window
set_external_capture_options -capture_procedure \
    <external_capture_procedure_name>

# Report core instances, then retarget and merge patterns as they are
# written out with respect to the chip-level boundary. The STIL format
# used here can later be read back into the tool to reverse map top level
# failures to the core.
report_core_instances
write_patterns TOP_stuck.stil -stil -replace

# Write top-level core description. Used for reverse mapping of silicon
# failures back to the core level for diagnosis, or for pattern
# retargeting without a netlist
write_core_description TOP_stuck.tcd -replace

```

Retargeting of Patterns at the Chip Level Without a Netlist

When a logic change is made to a core that requires the core-level patterns to be regenerated, if the logic change does not affect the ports of the core or the test setup conditions, you can

retarget those patterns without loading the chip-level netlist. Instead, you can use the top-level TCD file that was written out during the original retargeting run. This allows you to avoid the runtime required to load in a chip-level netlist and perform the extraction again.

The steps for scan pattern retargeting *without a netlist* are shown here. Prior to performing the retargeting in this example, you must have already run the extraction step to generate the top-level TCD file.

Note

 The extraction step is identical to the steps performed in “[Retargeting of Patterns at the Chip Level](#)” on page 520 except that the reading and writing of patterns is omitted.

In the absence of a netlist, the tool recreates what is needed for the design from the top-level TCD file. The tool performs design rule checking as it changes to analysis mode.

```
# Specify that this is the scan pattern retargeting phase
set_context patterns -scan_retargeting

# Read the top-level TCD file that resulted from the extraction step
read_core_descriptions TOP_stuck.tcd

# Changing to analysis mode automatically sets the current design, adds
# the cores, sets pin constraints and any other necessary information for
# the design
set_system_mode analysis

# Read core (retargetable) patterns
read_patterns CPU_stuck.retpat
read_patterns NB_stuck.retpat

# Report core instances (created automatically), then retarget and merge
# patterns as they are written out with respect to the chip-level boundary
report_core_instances
write_patterns TOP_stuck.stil -stil -replace
```

Core-in-Core Pattern Retargeting

Core-in-core pattern retargeting is the process of retargeting patterns through multiple levels of embedded wrapped cores to the top level of the design. That is, if the top design contains one or more cores and any one of those cores contains any number of nested cores, you can retarget patterns from the lowest level to the top level of the design.

The core-in-core pattern retargeting process provides two different options: you can use flat extraction that retargets patterns from the child to the chip level in one step, or you can perform extraction in multiple steps (one for each intermediate (typically wrapped) core level).

Flat Extraction Flow — The flat extraction flow requires you to provide the blackbox, graybox, or full netlist for the core being retargeted and the full netlist for all levels of hierarchy above it. While the flat extraction flow may require less file management (fewer TCD files to track) and fewer extraction steps, it does mean loading large netlists that need more memory. It also is less consistent with a hierarchical methodology in which everything is completed at each (typically) wrapped level of hierarchy.

Multiple Extraction Step Flow — Instead of the flat extraction flow, you can retarget patterns by performing multiple extraction steps (one for each intermediate (typically) wrapped level). For example, consider a design in which there is a low-level wrapped core, a child core, instantiated inside another wrapped core, a parent core, which is then instantiated in the top level. You perform extraction from the child to the parent at the parent level, store the associated TCD file, perform extraction from the parent level to the top, and then perform pattern retargeting. In addition to requiring multiple extraction steps, this flow requires more file management. However, the multiple extraction step flow provides the following benefits:

- Allows you to, optionally, perform verification at each level as you traverse the design.
- Compatible with a hierarchical flow methodology because at each level you extract everything that is needed for the parent level including the TCD file which allows you to forget about what is inside the parent.
- More efficient because it does not require you to provide the full content of your netlist from the child to the top level. You need only load a blackbox (or optionally graybox) for the core being extracted and the netlist of the level to which you are extracting.

Core-in-Core Pattern Retargeting Flow	524
Generation of Child Core Internal Mode Patterns.....	526
Retargeting of Child Core Internal Mode Patterns	526
Generation of Parent Internal Mode Patterns	527
Retargeting of Parent Internal Mode Patterns	528

Core-in-Core Pattern Retargeting Flow

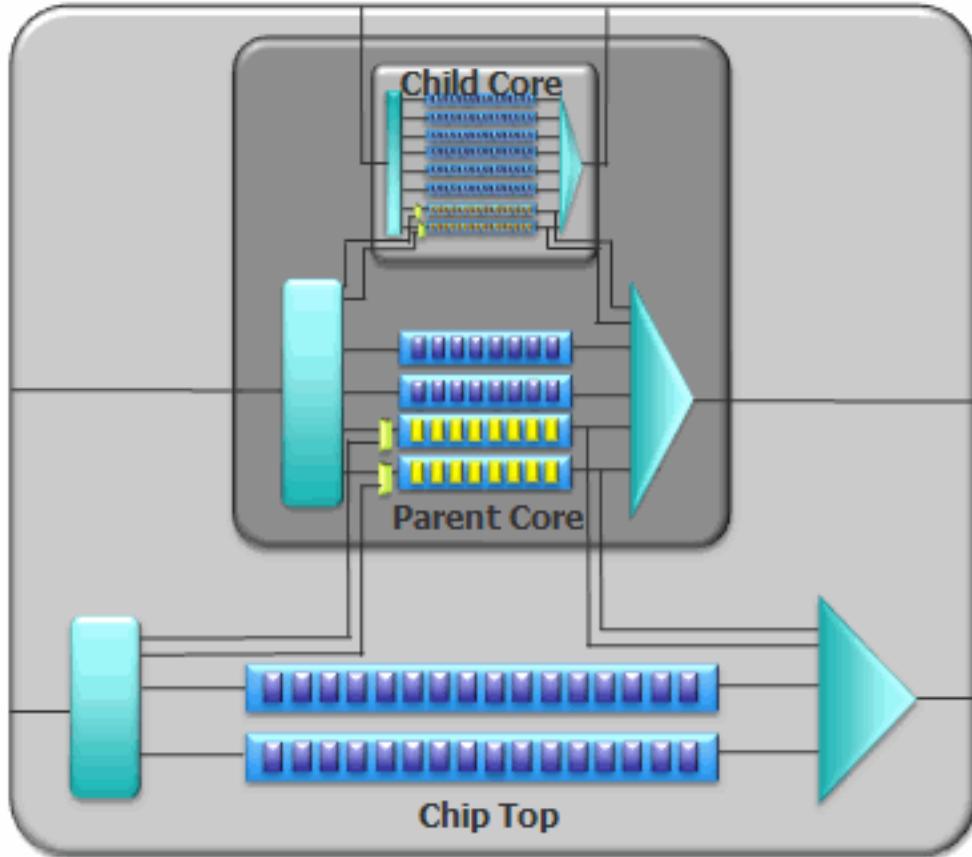
You perform core-in-core pattern retargeting by, first, doing the incremental extraction step for each wrapped level in the design. Then, the actual retargeting is performed once at the chip level at which time the core-level patterns are retargeted to the chip level.

Because your design may contain multiple levels between the child and chip level, the process can be extended to multiple levels. Your design may have multiple parent cores at the same level and/or multiple child cores at the same level.

In describing the process and steps required for generating and retargeting patterns from the child level to the chip level, this section uses three design levels, as shown in [Figure 10-7](#), and refers to them as the “chip level”, “parent level”, and “child level”. However, as mentioned before, this is a recursive flow that can be expanded to any number of levels.

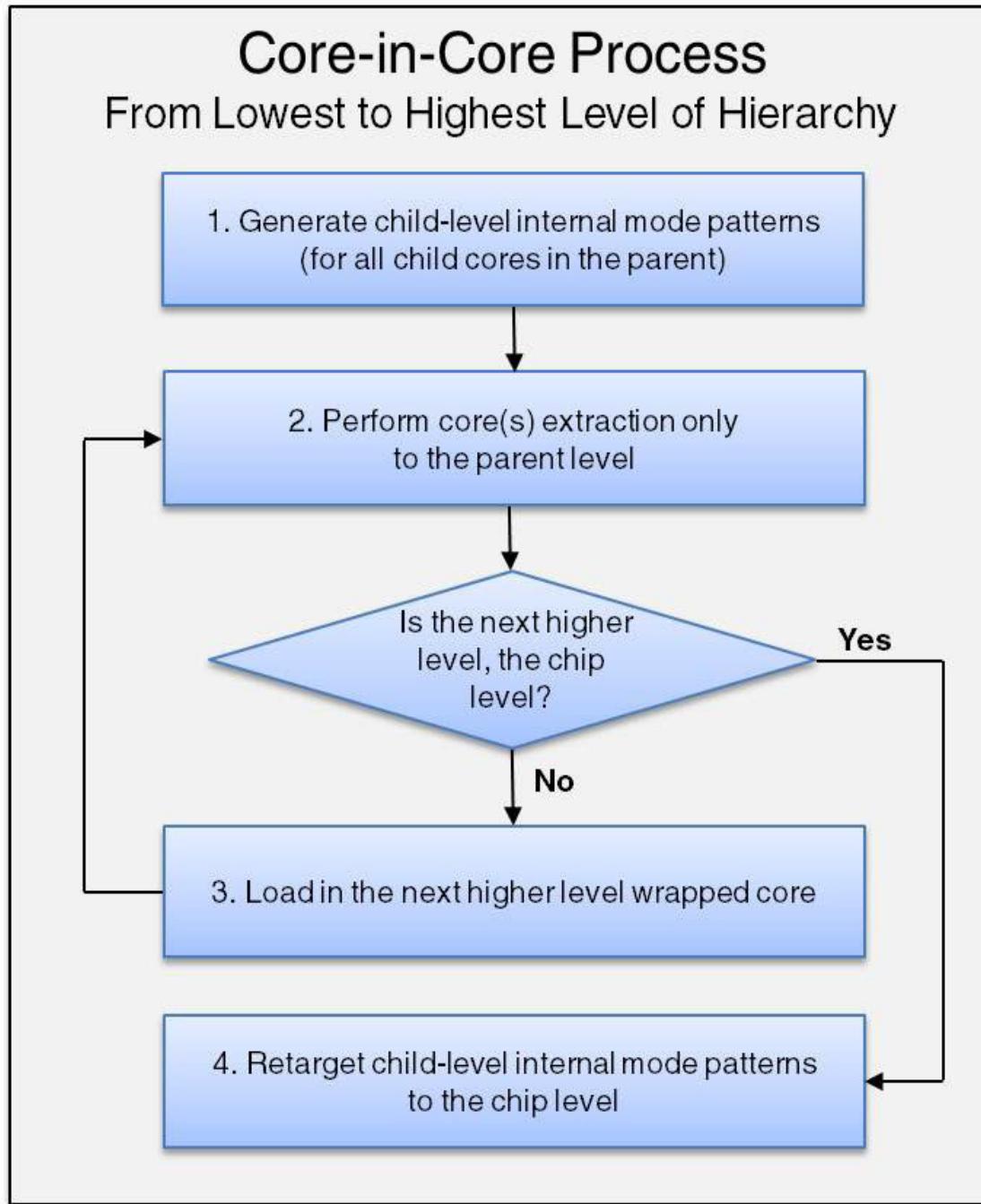
For example, the parent core shown in [Figure 10-7](#) could have multiple child cores inside it, or the chip could have multiple parent cores inside it.

Figure 10-7. Three Levels of Wrapped Cores



The core-in-core scan pattern retargeting process begins at the lowest level of design hierarchy and traverses up the design one level at a time as shown in [Figure 10-8](#):

Figure 10-8. Core-in-Core Retargeting Process Overview



Generation of Child Core Internal Mode Patterns

You generate child core internal mode patterns by performing the following steps at the child core(s) level.

Procedure

1. Read full child netlist.
2. Set child mode to internal.
3. Generate child internal mode patterns.
4. Write out the child patterns.
5. Write out the child TCD file (`write_core_description`).
6. Optionally, write out the flat model for diagnosis.
7. Write out the fault list.
8. Verify scan patterns at the child level.

Related Topics

[Core-in-Core Pattern Retargeting Flow](#)

Retargeting of Child Core Internal Mode Patterns

You retarget child core internal mode patterns by first performing intermediate extraction steps prior to the retargeting step.

The extraction of the child core(s) to the parent level is performed in steps one through seven.

Procedure

1. Read the child blackbox.
2. Read the full parent netlist.
3. Configure the parent core to provide access to the child core and configure the child core as it was configured when its patterns were generated.
4. Read the child TCD file.
5. Extract the child core connectivity and verify the child core embedding. Extraction is performed as part of DRC when you transition to analysis mode (“[set_system_mode analysis](#)”).
6. Write out the parent TCD file (the TCD file will include all child internal mode TCD info and new parent TCD info).

7. Optionally, retarget child core patterns to parent level for verification of core embedding only.

Once you get to the top level (below which the highest wrapped level is instantiated), you only need to incrementally extract connectivity from the parent level to the top. The connectivity from the child to the parent was previously extracted. Following this incremental extraction step, pattern retargeting can be performed directly from the child level to the top. This is described in steps 8 through 17.

8. Read the parent blackbox.
9. Read the full chip netlist.
10. Configure the chip to provide scan access from the top to the parent.
11. Configure the parent as it was configured in the previous procedure which configured the child for internal test and provided scan access from the parent to the child.
12. Read the parent TCD file from step 6 (which includes the extracted child TCD file).
13. Add the parent core as a core instance using [add_core_instances](#).
14. Extract the parent core connectivity and verify the parent core embedding. Extraction is performed as part of DRC when you transition to analysis mode (“`set_system_mode analysis`”).
15. Write the chip-level TCD file. (You will need this file if you will be performing pattern retargeting without a netlist as described in “[Retargeting Patterns Without a Top-Level Netlist](#)” on page 516.)
16. Read the child core internal mode patterns.
17. Write the retargetable patterns.

Related Topics

[Core-in-Core Pattern Retargeting Flow](#)

Generation of Parent Internal Mode Patterns

You can generate parent-level internal mode patterns, by performing the following steps at the parent level.

Procedure

1. Read the child graybox.
2. Read the full parent netlist.
3. Set the child to external mode.
4. Set the parent to internal mode.

5. Write the parent TCD file.
6. Generate parent internal mode patterns.
7. Write the parent internal mode patterns.
8. Optionally, write out the flat model for diagnosis.
9. Write out the fault list.
10. Verify the scan patterns.

Related Topics

[Core-in-Core Pattern Retargeting Flow](#)

Retargeting of Parent Internal Mode Patterns

Because you now want to retarget the patterns that were generated at the level of the parent to the top, you perform extraction and retargeting of the parent internal mode patterns in one step without the intermediate extraction step.

Procedure

1. Read the parent blackbox.
2. Read in the full chip set.
3. Configure the chip to parent internal mode.
4. Read the parent TCD file from step 5.
5. Add the parent core as a core instance using [add_core_instances](#).
6. Extract the parent core connectivity to the top level and verify embedding of the parent core. Extraction is performed as part of DRC when you transition to analysis mode (“[set_system_mode analysis](#)”).
7. Write the top-level TCD file.
8. Retarget the parent core internal mode patterns to the chip level.

Related Topics

[Core-in-Core Pattern Retargeting Flow](#)

Scan Pattern Retargeting Limitations

The limitations of the scan pattern retargeting functionality are listed here.

- Only parallel STIL and parallel WGL chip-level patterns can be read back into the tool. To enable reverse mapping of top level failures to the core for diagnosis, you need to have chip-level parallel STIL or WGL patterns.
- DRCs exist to validate that the design setup at the top level is consistent with the setup that was used for core-level ATPG. But this validation is not complete. For example, the capture cycle clocking is not validated. Consequently, you should perform chip-level serial simulation of a small number of scan patterns to verify the setup is correct.
- Core-level ATPG has limited flat model support. You can save a flat model in analysis mode for performing diagnosis or rerunning ATPG with that same configuration, but you must perform the initial design setup (setup mode) and design rule checking on a Verilog design and not a flat model. Note, you cannot access the `is_excluded_from_isolation_constraints` port attribute when running on a flat model.
- During the retargeting phase, flat models are not supported. A Verilog design must be read in. A flat model cannot be read in or written out because it lacks information necessary for retargeting.
- Scan pattern retargeting does not support IDDQ, so IDDQ ATPG must be created flat. Flat ATPG ensures that `measure_iddq` is performed at the same time for all cores. Scan retargeting does not perform any capture procedure mapping and merging across cores, which would be required if the tool were to retarget and merge capture cycles with IDDQ measures.

Since the current is measured for the entire chip, IDDQ ATPG ensures quiescence across the entire chip. It would not be safe to only retarget patterns for a subset of the design without having DFT/DRC ensure that the rest of the design does not draw high current that impacts the quality of the IDDQ test.

- Scan pattern retargeting does not support multiple scan groups.
- The recommended format for patterns for retargeting is pattern database (PatDB).
- When generating retargetable core-level patterns, native launch-off-shift is not supported. Native launch-off-shift is one of two methods for generating launch-off-shift patterns. For more information on launch-off-shift patterns, see “[Native Launch-Off-Shift Patterns](#)” on page 329.
- The `shadow_control`, `shadow_observe`, and `master_observe` procedures are not supported. You should not use these procedures during core-level ATPG or in the top-level test procedure file.
- Do not use STARTPAT and ENDPAT when simulating retargeted patterns that contain multi-load core patterns. When multi-load patterns from a core are retargeted to top-level chip patterns, the top-level pattern count does not align with the original core-level patterns. Each retargeted load is marked as a pattern. When saving these retargeted patterns in a Verilog testbench, if you used the STARTPAT and ENDPAT plusargs,

they refer to the top-level pattern numbers, and it might be possible to erroneously start a simulation in the middle of a multi-load core pattern.

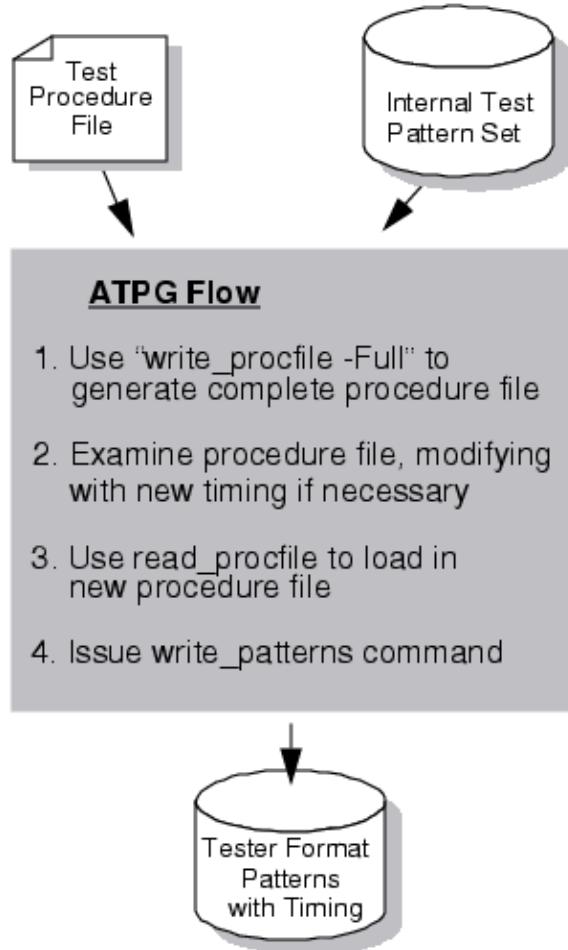
Chapter 11

Test Pattern Formatting and Timing

This chapter explains test pattern timing and procedure files.

[Figure 11-1](#) shows a basic process flow for defining test pattern timing.

Figure 11-1. Defining Basic Timing Process Flow



Test Pattern Timing Overview	532
Timing Terminology	532
General Timing Issues.....	533
Generating a Procedure File.....	534
Defining and Modifying Timeplates.....	534

Delaying Clock Pulses in Shift and Capture to Handle Slow Scan Enable Transitions	536
Delaying Clocks After All Scan Enable Transitions (Recommended).....	536
Transitions From Shift to Capture	538
Transition from Capture to Shift.....	541
Saving Timing Patterns.....	544
Features of the Formatter	544
Pattern Formatting Issues	546
Basic Test Data Formats for Patterns	549
ASIC Vendor Data Formats	554
Vector Creation and Modification.....	556
Port List Modification.....	556
Vector Data Callbacks	557
Used Ports and Vector Callback Examples	562

Test Pattern Timing Overview

Test procedure files contain both scan and non-scan procedures. All timing for all pattern information, both scan and non-scan, is defined in this procedure file.

While the ATPG process itself does not require test procedure files to contain real timing information, automatic test equipment (ATE) and some simulators do require this information. Therefore, you must modify the test procedure files you use for ATPG to include real timing information. “[General Timing Issues](#)” on page 533 discusses how you add timing information to existing test procedures.

After creating real timing for the test procedures, you are ready to save the patterns. You use the `write_patterns` command with the proper format to create a test pattern set with timing information. For more information, refer to “[Saving Timing Patterns](#)” on page 544.

Test procedures contain groups of statements that define scan-related events. See “[Test Procedure File](#)” on the *Tessent Shell User’s Manual*.

Timing Terminology

The following list defines some timing-related terms.

- **Non-return Timing** — Primary inputs that change, at most, once during a test cycle.
- **Offset** — The timeframe in a test cycle in which pin values change.
- **Period** — The duration of pin timing—one or more test cycles.
- **Return Timing** — Primary inputs, typically clocks, that pulse high or low during every test cycle. Return timing indicates that the pin starts at one logic level, changes, and returns to the original logic level before the cycle ends.

- **Suppressible Return Timing** — Primary inputs that can exhibit return timing during a test cycle, although not necessarily.

General Timing Issues

ATEs require test data in a cycle-based format. The patterns you apply to such equipment must specify the waveforms of each input, output, or bidirectional pin, for each test cycle.

Within a test cycle, a device under test must abide by the following restrictions:

- At most, each non-clock input pin changes once in a test cycle. However, different input pins can change at different times.
- Each clock input pin is at its off-state at both the start and end of a test cycle.
- At most, each clock input pin changes twice in a test cycle. However, different clock pins can change at different times.
- Each output pin has only one expected value during a test cycle. However, the equipment can measure different output pin values at different times.
- A bidirectional pin acts as either an input or an output, but not both, during a single test cycle.

To avoid adverse timing problems, the following timing requirements satisfy some ATE timing constraints:

- **Unused Outputs** — By default, test procedures without measure events (all procedures except **shift**) strobe unused outputs at a time of $\text{cycle}/2$, and end the strobe at $3*\text{cycle}/4$. The **shift** procedure strobos unused outputs at the same time as the scan output pin.
- **Unused Inputs** — By default, all unused input pins in a test procedure have a force offset of 0.
- **Unused Clock Pins** — By default, unused clock pins in a test procedure have an offset of $\text{cycle}/4$ and a width of $\text{cycle}/2$, where cycle is the duration of each cycle in the test procedure.
- **Pattern Loading and Unloading** — During the **load_unload** procedure, when one pattern loads, the result from the previous pattern unloads. When the tool loads the first pattern, the unload values are X. After the tool loads the last pattern, it loads a pattern of X's so it can simultaneously unload the values resulting from the final pattern.
- **Events Between Loading and Unloading (“patterns -scan” context only)** — If other events occur between the current unloading and the next loading, in order to load and

unload the scan chain simultaneously. The tool performs the events in the following order:

- a. **Observe Procedure Only** — The tool performs the observe procedure before loading and unloading.
- b. **Initial Force Only** — The tool performs the initial force before loading and unloading.
- c. **Both Observe Procedure and Initial Force** — The tool performs the observe procedures followed by the initial force before loading and unloading.

Generating a Procedure File

This procedure showed the basic process flow for defining test pattern timing.

Procedure

1. Use the “[write_procfile -Full](#)” command and switch to generate a complete procedure file.
2. Examine the procedure file, modify timeplates with new timing if necessary.
3. Use the [read_procfile](#) command to load in the revised procedure file.
4. Issue the [write_patterns](#) command.
5. The “[Test Procedure File](#)” section of the *Tessent Shell User’s Manual* gives an in depth description of how to create a procedure file.
6. There are three ways to load existing procedure file information into the tool:
 - During SETUP mode, use the [add_scan_groups procedure_filename](#) command. Any timing information in these procedure files will be used when [write_patterns](#) is issued if no other timing information or procedure information is loaded.
 - Use the [read_procfile](#) command. This is only valid when not in SETUP mode. Using this command loads a new procedure file that will overwrite or merge with the procedure and timing data already loaded. This new data is now in effect for all subsequent [write_patterns](#) commands.
 - If you specify a new procedure file on the [write_patterns](#) command line, the timing information in that procedure file will be used for that [write_patterns](#) command only, and then the previous information will be restored.

Defining and Modifying Timeplates

This section gives an overview of the test procedure file timeplate syntax, to facilitate Step 2 in the process flow listed previously.

For a more detailed overview of timeplates, see the “[Timeplate Definition](#)” section of the *Tessent Shell User’s Manual*.

After you have used the “`write_procedure -full`” command to generate a procedure file, you can examine the procedure file, modifying timeplates with new timing if necessary. Any timing changes to the existing timeplates, cannot change the event order of the timeplate used for scan procedures. The times may change, but the event order must be maintained.

The following example shows the contents of a timeplate, where there are two timing edges happening at time 20, and both are listed as timing edge 4. These can be skewed, but they cannot cross any other timing edge. The timing edges must stay in the order listed in the comments:

```
force_pi      0; // timing edge 1
bidi_force_pi 12; // timing edge 3
measure_po    31; // timing edge 7
bidi_measure_po 32; // timing edge 8
force InPin    9; // timing edge 2
measure OutPin 35; // timing edge 9
pulse Clk1    20 5; // timing edge 4 & 5, respectively
pulse Clk2    20 10; // timing edge 4 & 6, respectively
period 50;           // all timing edges have to happen in period
```

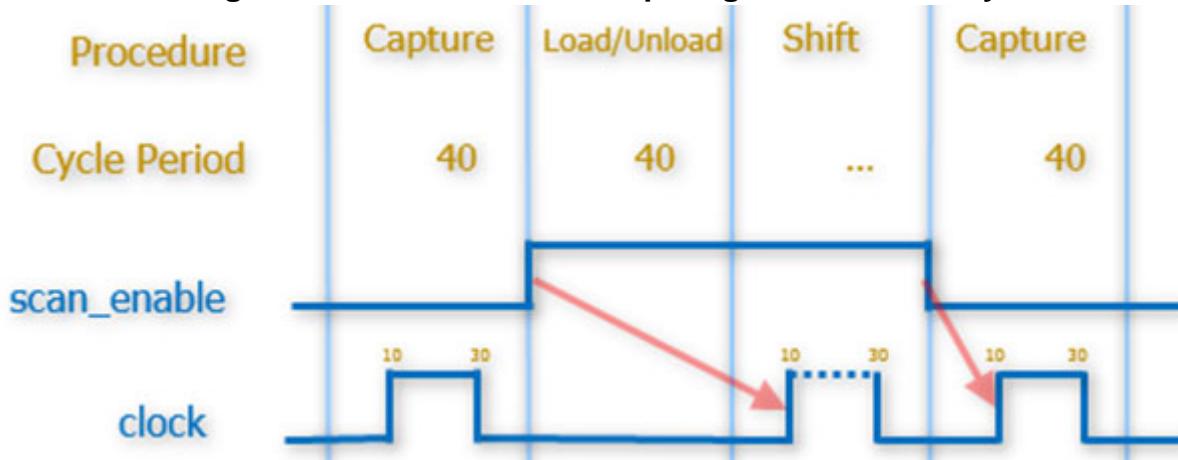
The test procedure file syntax and format is explained in the “[Timeplate Definition](#)” section in the *Tessent Shell User’s Manual*. Keep in mind that the timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed. You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. All clocks must be defined in the timeplate definition.

Delaying Clock Pulses in Shift and Capture to Handle Slow Scan Enable Transitions

There are various techniques to handle slow scan enable transitions.

During scan test, the frequency of shift and capture clocks may not allow sufficient time for scan enable (SE) to change state before the occurrence of the first clock pulse. Since SE is a global signal that is not timed for fast operation similar to clocks, there may not be enough time for it to propagate to all scan cells within one tester cycle. Although SE is the most common case of such a signal, any other high-fanout signal may have the same timing issue. The waveform in [Figure 11-2](#) highlights, in red, the edge transitions that may require additional delay.

Figure 11-2. Clock Pulses Requiring Additional Delay



These sections describe various techniques for creating procedures that avoid DRC violations and result in correct simulation results and patterns. For all the techniques described here, SE is assumed to be constrained to its off state in the dofile.

Delaying Clocks After All Scan Enable Transitions (Recommended)	536
Transitions From Shift to Capture	538
Transition from Capture to Shift	541

Delaying Clocks After All Scan Enable Transitions (Recommended)

The recommended method for delaying clock pulses after all transitions of SE is to stretch the timeplate used in the load_unload and capture procedures. Doing so addresses delay requirements when transitioning from shift to capture, as well as from capture to shift.

Timeplate Examples

For example, shift cycles may use a 40ns period with a 50 percent duty cycle as shown in this timeplate:

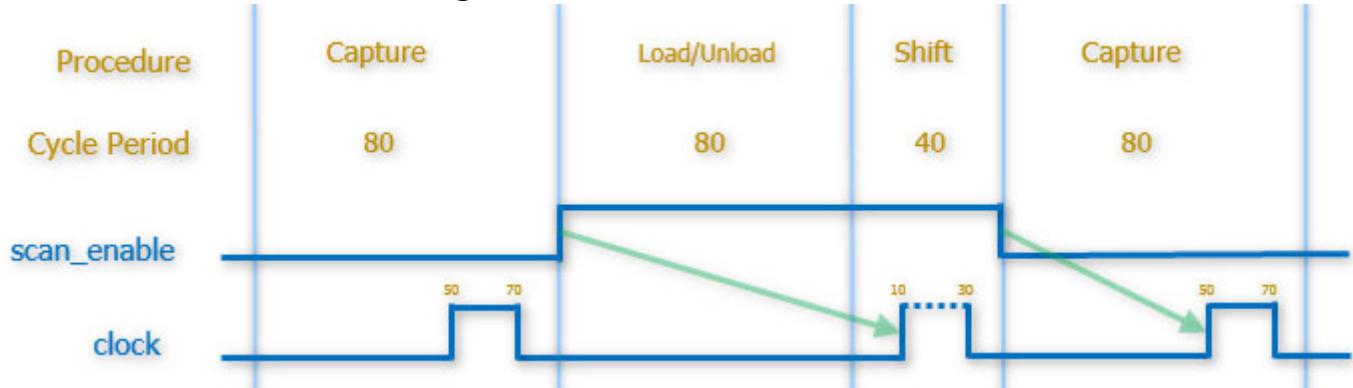
```
timeplate tp_shift =
    force_pi 0;
    measure_po 5;
    pulse_clock 10 20;
    period 40;
end;
```

This stretched timeplate can be used to delay the clock pulse by 40ns while maintaining the same 20ns duty cycle:

```
timeplate tp_load_and_capture =
    force_pi 0;
    measure_po 5;
    pulse_clock 50 20;
    period 80;
end;
```

As shown in the waveform in [Figure 11-3](#), the stretched waveform adds delay to address transitions from capture to shift and from shift to capture.

Figure 11-3. Stretched Waveform



Default Timeplate

You can explicitly specify the timeplate used by the tool as shown in the timeplate code examples shown in “[Timeplate Examples](#)” on page 537 or by changing the default timeplate using the “set default_timeplate” statement in the procedure file. It is recommended to not change the default timeplate because the tool may create different procedures automatically and will use the shift procedure’s timeplate to do so. Defining the default timeplate may result in the tool using the stretched timeplate which is not always ideal.

Transitions From Shift to Capture

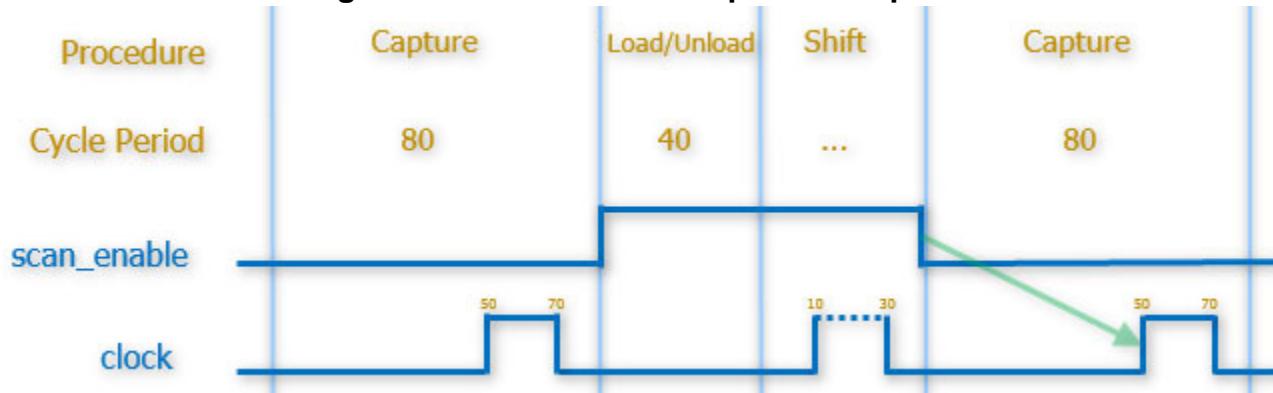
Many designs have test hardware that uses SE to disable asynchronous control signals (such as sets and resets) during shift. The logic inserted by DFT signals in Tessent tools also uses SE for this purpose. Adding post-shift cycles to the load_unload procedure cannot be used because it forces SE to transition to 0 before the end of load-unload. Unless the set/reset signals are gated off during both shift and capture, scan cell data will be disturbed and the tool will report D1 DRC violations.

There are four methods that allow the first clock pulse in capture to be delayed without the need to add post-shift dead cycles in the load-unload procedure.

Stretched Timeplate in Capture (Recommended)

The recommended method for delaying the capture pulse after SE transitions to 0 is the stretched timeplate described in [Delaying Clocks After All Scan Enable Transitions \(Recommended\)](#). The waveform in Figure 11-4 shows the same solution if only applied to capture to address shift-to-capture transitions.

Figure 11-4. Stretched Timeplate in Capture



Extended Clock Sequential and Capture Procedures

If tester restrictions prevent changing the clock period between shift and capture, the clock sequential and capture procedures can be extended to more than one cycle to achieve a similar effect as the stretched timeplate method described in the [Stretched Timeplate in Capture \(Recommended\)](#) section.

Procedures in capture can be extended by explicitly defining the `clock_sequential` and `capture` procedures. The following `clock_sequential` and `capture` procedures have a cycle with no clock pulse followed by another cycle that includes the clock pulse.

```

timeplate tp1 =
  force_pi 0;
  measure_po 5;
  pulse_clock 10 20;
  period 40;
end;
procedure clock_sequential =
  timeplate tp1;
  cycle =
    force_pi;
    measure_po;
end ;
cycle =
  pulse_capture_clock;
end;
end;
procedure capture =
  timeplate tp1;
  cycle =
    force_pi;
    measure_po;
end ;
cycle =
  pulse_capture_clock;
end;
end;

```

Figure 11-5. Extended Capture Procedure

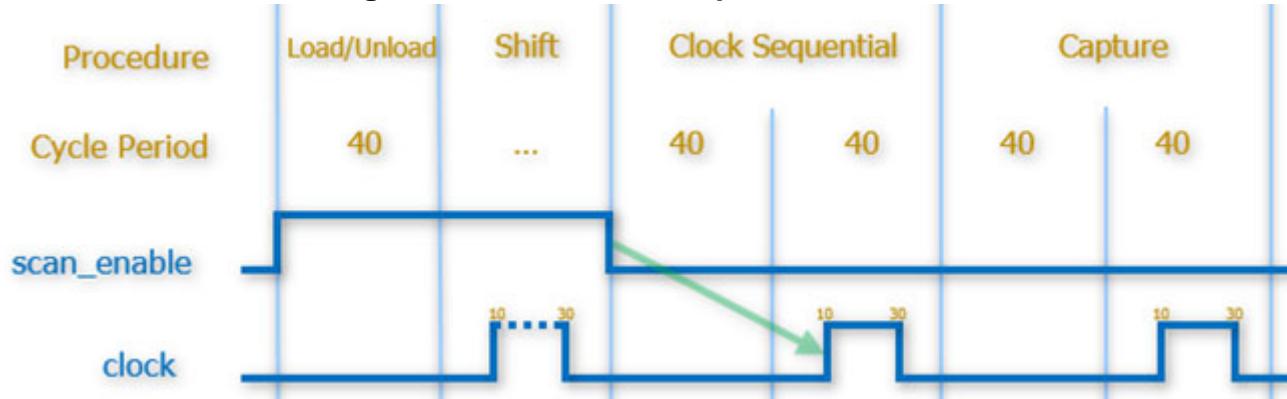
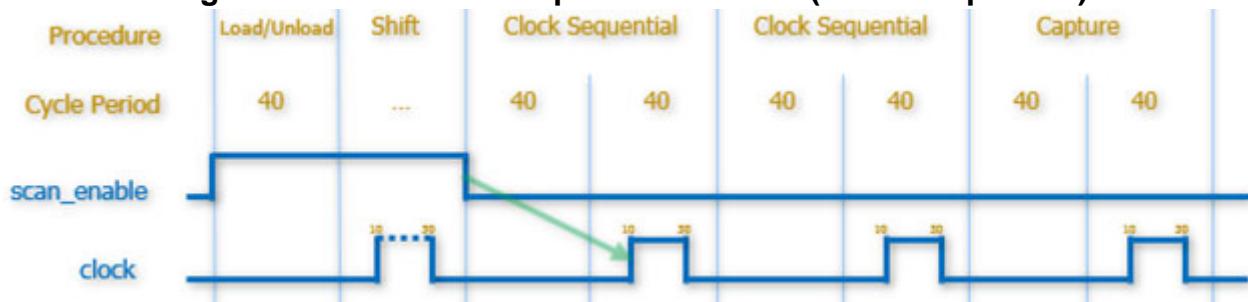


Figure 11-6. Extended Capture Procedure (Clock Sequential)

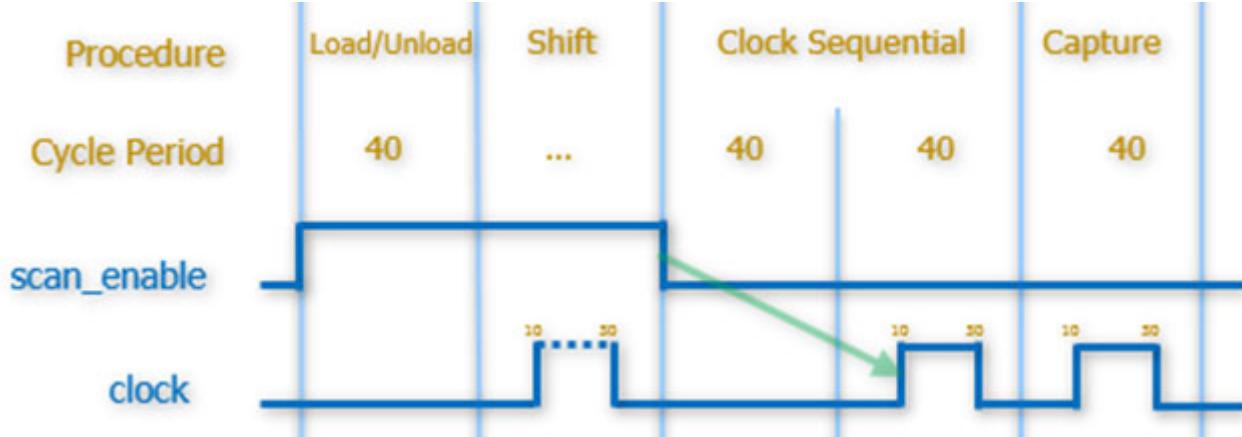


The waveforms in [Figure 11-5](#) and [Figure 11-6](#) illustrate that having a dead cycle at the start of the capture procedure may not be suitable for at-speed test if the capture clock is required to

occur immediately after the last clock pulse. If this is required for an accurate at-speed test and the tester can support creating such waveforms, it may be necessary to remove the dead cycle in the capture procedure as shown in this code and in [Figure 11-7](#):

```
procedure capture =
    timeplate tp1;
    cycle =
        force_pi;
        measure_po;
        pulse_capture_clock;
    end;
end;
```

Figure 11-7. Extended Capture Procedure (Dead Cycle Removed)



External Capture Procedures

For at-speed test, it is also possible to define additional dead cycles in an external_capture procedure to delay the first clock pulse in capture. The external capture procedure is referenced in the dofile using the [set_external_capture_options](#) command.

An example is shown in this code and in [Figure 11-8](#)

```
procedure external_capture ext_fast_cap_proc =
    timeplate tp1 ;
    cycle =
        force_pi ;
    end;
    cycle =
    end;
    cycle =
    end;
    cycle =
        pulse clock;
    end;
end;
```

Figure 11-8. External Capture Procedure (Dead Cycles Added)



Named Capture Procedures

If Named Capture Procedures (NCPs) are used, clock pulses can be delayed using the techniques described, by stretching the cycle, or by explicitly adding dead cycles

Post-shift Cycles

As explained earlier in the introduction of [Transitions From Shift to Capture](#), only designs that do not use SE to disable asynchronous controls (e.g. sets and resets) can add post-shift dead cycles to delay the clock pulse. This allows SE enough time to transition from 1 to 0 before clock pulses in capture. However since this solution is not appropriate for many designs, it is not recommended.

Transition from Capture to Shift

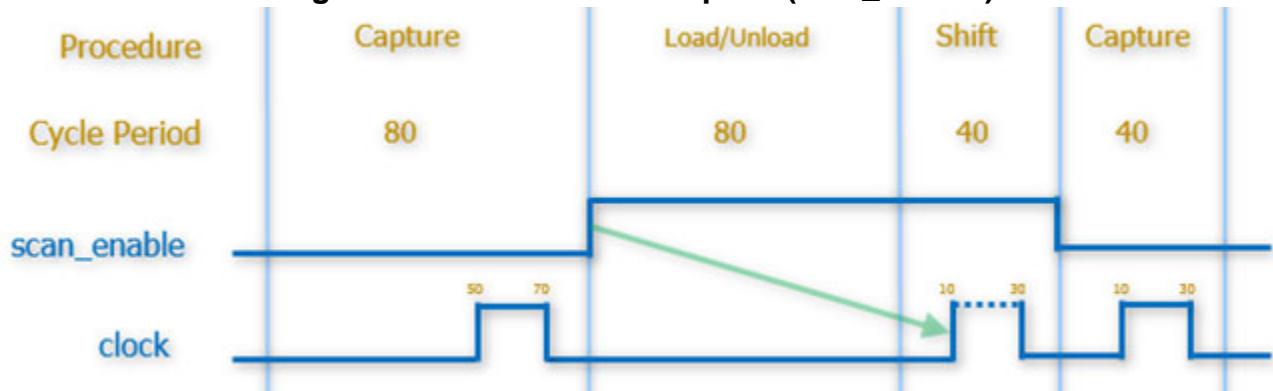
After scan enable (SE) transitions to 1, there are two method to handle slow scan enable transitions.

Stretched Timeplate in load_unload (Recommended)

The recommended method for delaying the capture pulse after SE transitions to 1 is the stretched timeplate described in "[Delaying Clocks After All Scan Enable Transitions](#)

(Recommended)" on page 536. The waveform in [Figure 11-9](#) shows the same solution if only applied to load_unload to address capture-to-shift transitions:

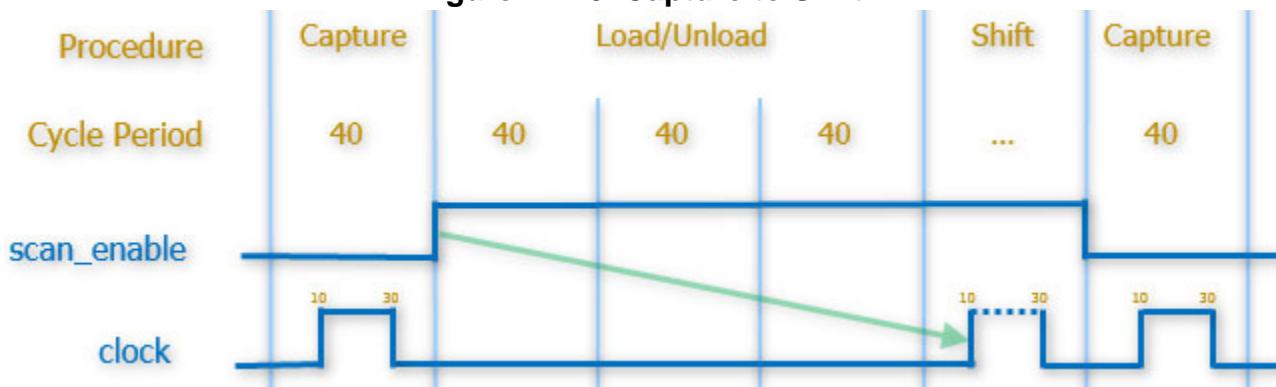
Figure 11-9. Stretched Timeplate (load_unload)



Dead Cycles Before Shift

In order to allow SE enough time to transition from 0 to 1 before the first clock pulse in shift, add any number of dead cycles with no clock pulses to the load_unload procedure before the 'apply shift' statement. An example is shown in this code and in [Figure 11-10](#):

```
timeplate tp1 =
    force_pi 0;
    measure_po 5;
    pulse_clock 10 20;
    period 40;
end;
procedure load_unload =
    timeplate tp1;
    cycle =
        force clk 0;
        force scan_en 1;
    end;
    cycle =
    end;
    cycle =
    end;
    apply shift 100;
end;
```

Figure 11-10. Capture to Shift**Delaying Other Signals in load_unload That Require Additional Delay**

The examples discussed in “[Transition from Capture to Shift](#)” on page 541 are simple procedures for a design without compression. A similar approach can be used for designs with EDT where the `edt_update` signal may also have a high fanout and require additional cycles to propagate. This example adds a dead cycle after each transition of `edt_update` and before the pulses of EDT and shift clocks.

```
procedure load_unload =
  timeplate tp1;
  cycle =
    force RST 0;
    force CLK 0;
    force scan_en 1;
    force edt_update 1;
  end;
  cycle =
  end;
  cycle =
    pulse edt_clock;
  end;
  cycle =
    force edt_update 0;
  end;
  cycle =
  end;
  apply shift 100;
end;
```

Saving Timing Patterns

You can write the patterns generated during the ATPG process both for timing simulation and use on the ATE.

Once you create the proper timing information in a test procedure file, the tool uses an internal test pattern data formatter to generate the patterns in the following formats:

- Text format (ASCII)
- Binary format
- Wave Generation Language (WGL)
- Standard Test Interface Language (STIL)
- Verilog
- Texas Instruments Test Description Language (TDL 91)
- Fujitsu Test data Description Language (FTDL-E)
- Mitsubishi Test Description Language (MITDL)
- Toshiba Standard Tester interface Language 2 (TSTL2)

Features of the Formatter	544
Pattern Formatting Issues.....	546
Basic Test Data Formats for Patterns	549
ASIC Vendor Data Formats	554

Features of the Formatter

Here are the main features of the test pattern data formatter.

- Generating basic test pattern data formats: text, Verilog, and WGL (ASCII and binary).
- Generating ASIC Vendor test data formats: TDL 91, FTDL-E, MITDL, and TSTL2.
- Supporting parallel load of scan cells (in Verilog format).
- Reading in external input patterns and output responses, and directly translating to one of the formats.
- Reading in external input patterns, performing good or faulty machine simulation to generate output responses, and then translating to any of the formats.
- Writing out just a subset of patterns in any test data format.
- Facilitating failure analysis by having the test data files cross-reference information between tester cycle numbers and pattern numbers.

- Supporting differential scan input pins for each simulation data format.

Pattern Formatting Issues

The following subsections describe issues you should understand regarding the test pattern formatter and pattern saving process.

Serial Versus Parallel Scan Chain Loading	546
Parallel Scan Chain Loading	546
Reduce Serial Loading Simulation Time with Sampling.....	548
Test Pattern Data Support for IDQ	548

Serial Versus Parallel Scan Chain Loading

When you simulate test patterns, most of the time is spent loading and unloading the scan chains, as opposed to actually simulating the circuit response to a test pattern. You can use either serial or parallel loading, and each affects the total simulation time differently.

The primary advantage of simulating serial loading is that it emulates how patterns are loaded on the tester. You thus obtain a very realistic indication of circuit operation. The disadvantage is that for each pattern, you must clock the scan chain registers at least as many times as you have scan cells in the longest chain. For large designs, simulating serial loading takes an extremely long time to process a full set of patterns.

The primary advantage of simulating parallel loading of the scan chains is it greatly reduces simulation time compared to serial loading. You can directly (in parallel) load the simulation model with the necessary test pattern values because you have access, in the simulator, to internal nodes in the design. Parallel loading makes it practical for you to perform timing simulations for the entire pattern set in a reasonable time using popular simulators like ModelSim that utilize the Verilog format.

Parallel Scan Chain Loading

You accomplish parallel loading through the scan input and scan output pins of scan sub-chains (a chain of one or more scan cells, modeled as a single library model) because these pins are unique to both the timing simulator model and the Tessent Shell internal model. For example, you can parallel load the scan chain by using Verilog force statements to change the value of the scan input pin of each sub-chain.

After the parallel load, you apply the **shift** procedure a few times (depending on the number of scan cells in the longest subchain, but usually only once) to load the scan-in value into the sub-chains. Simulating the **shift** procedure only a few times can dramatically improve timing simulation performance. You can then observe the scan-out value at the scan output pin of each sub-chain.

Parallel loading ensures that all memory elements in the scan sub-chains achieve the same states as when serially loaded. Also, this technique is independent of the scan design style or type of

scan cells the design uses. Moreover, when writing patterns using parallel loading, you do not have to specify the mapping of the memory elements in a sub-chain between the timing simulator and Tesson Shell. This method does not constrain library model development for scan cells.

Note

 When your design contains at least one stable-high scan cell, the **shift** procedure period must exceed the shift clock off time. If the **shift** procedure period is less than or equal to the shift clock off time, you may encounter timing violations during simulation. The test pattern formatter checks for this condition and issues an appropriate error message when it encounters a violation.

For example, the test pattern timing checker would issue an error message when reading in the following **shift** procedure and its corresponding timeplate:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 300; // Period same as shift clock off time
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK; // Force shift clock on and off
    end;
end;
```

The error message would state:

```
// Error: There is at least one stable high scan cell in the design.
// The shift procedure period must be greater than the shift clock off
// time to avoid simulation timing violations.
```

The following modified timeplate would pass timing rules checks:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 400; // Period greater than shift clock off time
end;
```

Reduce Serial Loading Simulation Time with Sampling

When you use the `write_patterns` command, you can save a sample of the full pattern set by using the `-Sample` switch. This reduces the number of patterns in the pattern file(s), reducing simulation time accordingly. In addition, the `-Sample` switch allows you to control how many patterns of each type are included in the sample. By varying the number of sample patterns, you can fine-tune the trade-off between file size and simulation time for serial patterns.

Note

 Using the `-Start` and `-End` switches limits file size as well, but the portion of internal patterns saved does not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns more closely approximate the results you would obtain from the entire pattern set.

After performing initial verification with parallel loading, you can use a sampled pattern set for simulating series loading until you are satisfied test coverage is reasonably close to desired specification. Then, perform a series loading simulation with the unsampled pattern set only once, as your last verification step.

Note

 The `set_pattern_filtering` command serves a similar purpose to the `-Sample` switch of the `write_patterns` command. The `set_pattern_filtering` command creates a temporary set of sampled patterns within the tool.

Test Pattern Data Support for IDDQ

For best results, you should measure current after each non-scan cycle if doing so catches additional IDDQ faults. However, you can only measure current at specific places in the test pattern sequence, typically at the end of the test cycle boundary. To identify when IDDQ current measurement can occur, the pattern file adds the following command at the appropriate places:

```
measure IDDQ ALL;
```

Several test pattern data formats support IDDQ testing. There are special IDDQ measurement constructs in TDL 91 (Texas Instruments), MITDL (Mitsubishi), TSTL2 (Toshiba), and FTDL-E (Fujitsu). The tools add these constructs to the test data files. All other formats (WGL and Verilog) represent these statements as comments.

Basic Test Data Formats for Patterns

The write_patterns command saves the patterns in the basic test data formats including text, binary, Verilog, and WGL (ASCII and binary). You can use these formats for timing simulation.

Text Format	549
Comparing the Text Format with Other Test Data Formats	549
Binary	552
Verilog.....	552
Wave Generation Language (ASCII).....	552
Standard Test Interface Language (STIL)	553

Text Format

This is the default format that the tool generates when you run the write_patterns command. The tool can read back in this format in addition to WGL, STIL, and binary format.

This format contains test pattern data in a text-based parallel format, along with pattern boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a basic text format file, enter the following at the application command line:

ANALYSIS> write_patterns filename -ascii

The formatter writes the complete test data to the file named *filename*.

For more information on the write_patterns command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Note

 This pattern format does not contain explicit timing information. For more information about this test pattern format, refer to “[Test Pattern File Formats](#)” on page 567.

Comparing the Text Format with Other Test Data Formats

The text format describes the contents of the test set in a human readable form. In many cases, you may find it useful to compare the contents of a simulation or test data format with that of the text format for debugging purposes. This section provides detailed information necessary for this task.

Often, the first cycle in a test set must perform certain tasks. The first test cycle in all test data formats turns off the clocks at all clock pins, drives Z on all bidirectional pins, drives an X on all other input pins, and disables measurement at any primary output pins.

The test pattern set can contain two main parts: the *chain test* block, to detect faults in the scan chain, and the *scan test* or *cycle test* block, to detect other system faults.

The Chain Test Block

The chain test applies the **test_setup** procedure, followed by the **load_unload** procedure for loading scan chains, and the **load_unload** procedure again for unloading scan chains. Each **load_unload** procedure in turn calls the **shift** procedure. This operation typically loads a repeating pattern of “0011” into the chains. However, if scan chains with less than four cells exist, then the operation loads and unloads a repeating “01” pattern followed by a repeating “10” pattern. Also, when multiple scan chains in a group share a common scan input pin, the chain test process separately loads and unloads each of the scan chains with the repeating pattern to test them in sequence.

The test procedure file applies each event in a test procedure at the specified time. Each test procedure corresponds to one or more test cycles. Each test procedure can have a test cycle with a different timing definition. By default, all events use a timescale of 1 ns.

Note

 If you specify a capture clock with the `set_capture_clock` command, the test pattern formatter does not produce the chain test block. For example, the formatter does not produce a chain test block for IEEE 1149.1 devices in which you specify a capture clock during tool setup.

The Scan Test Block

The scan test block in the pattern set starts with an application of the **test_setup** procedure. The scan test block contains several test patterns, each of which typically applies the **load_unload** procedure, forces the primary inputs, measures the primary outputs, and pulses a capture clock. The **load_unload** procedure translates to one or more test cycles. The force, measure, and clock pulse events in the pattern translate to the ATPG-generated capture cycle.

Each event has a sequence number within the test cycle. The sequence number’s default time scale is 1 ns.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. Therefore the last pattern in the test set contains an extra application of the **load_unload** sequence.

More complex scan styles (for example, like LSSD) use **master_observe** and **skewed_load** procedures in the pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may have several test cycles in it to operate the sequential scan controller.

Some pattern types (for example, RAM sequential and clock sequential types) are more complex than the basic patterns. RAM sequential patterns involve multiple loads of the scan chains and multiple applications of the RAM write clock. Clock sequential patterns involve multiple capture cycles after loading the scan chains. Another special type of pattern is the `clock_po` pattern. In these patterns, clocks may be held active throughout the test cycle and without applying capture clocks.

If the test data format supports only a single timing definition, the tool cannot save both `clock_po` and non-`clock_po` patterns in one pattern set. This is so because the tester cannot reproduce one clock waveform that meets the requirements of both types of patterns. Each pattern type (combinational, `clock_po`, `ram_sequential`, and `clock_sequential`) can have a separate timing definition.

General Considerations

During a test procedure, you may leave many pins unspecified. Unspecified primary input pins retain their previous state.

Note

 If you run ATPG after setting pin constraints, you should also ensure that you set these pins to their constrained states at the end of the `test_setup` procedure. The `add_input_constraints` command constrains pins for the non-scan cycles, not the test procedures. If you do not properly constrain the pins within the `test_setup` procedure, the tool does it for you, internally adding the extra force events after the `test_setup` procedure. This increases the period of the `test_setup` procedure by one time unit. This increased period can conflict with the test cycle period, potentially forcing you to re-run ATPG with the modified test procedure file.

All test data formats contain comment lines that indicate the beginning of each test block and each test pattern. You can use these comments to correlate the test data in the text format with other test data formats.

These comment lines also contain the *cycle count* and the *loop count*, which help correlate tester pattern data with the original test pattern data. The cycle count represents the number of test cycles, with the shift sequence counted as one cycle. The loop count represents the number of all test cycles, including the shift cycles. The cycle count is useful if the tester has a separate memory buffer for scan patterns, otherwise the loop count is more relevant.

Note

 The cycle count and loop count contain information for all test cycles—including the test cycles corresponding to test procedures. You can use this information to correlate tester failures to a pattern for fault diagnosis.

Binary

This format contains test pattern data in a binary parallel format, which is the only format (other than text format) that the tool can read. A file generated in this format contains the same information as text format, but uses a condensed form. You should use this format for archival purposes or when storing intermediate results for very large designs.

To create a binary format file, enter the following command:

ANALYSIS> write_patterns filename -binary

The tool writes the complete test data to the file named *filename*.

For more information about the `write_patterns` command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Verilog

This format contains test pattern data and timing information in a text-based format readable by both the Verilog and Verifault simulators. This format also supports both serial and parallel loading of scan cells. The Verilog format supports all Tessent Shell timing definitions because Verilog stimulus is a sequence of timed events.

To generate a basic Verilog format test pattern file, use the following arguments with the `write_patterns` command:

write_patterns filename [-Parallel | -Serial] -Verilog

The Verilog pattern file contains procedures to apply the test patterns, compare expected output with simulated output, and print out a report containing information about failing comparisons. The tools write all patterns and comparison functions into one main file (*filename*), while writing the primary output names in another file (*filename.po.name*). If you choose parallel loading, they also write the names of the scan output pins of each scan sub-chain of each scan chain in separate files (for example, *filename.chain1.name*). This allows the tools to report output pins that have discrepancies between the expected and simulated outputs. For more information about Verilog test benches, refer to “[The Verilog Test Bench](#)” on page 458.

For more information about the `write_patterns` command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Wave Generation Language (ASCII)

The Wave Generation Language (WGL) format contains test pattern data and timing information in a structured text-based format. You can translate this format into a variety of simulation and tester environments, but you must first read it into the Waveform database and use the appropriate translator. This format supports both serial and parallel loading of scan cells.

Some test data flows verify patterns by translating WGL to stimulus and response files for use by the chip foundry’s golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure the following:

- There is only one scan cell for each DFT library model (also called a scan subchain).
- The hierarchical scan cell names in the netlist and DFT library match those of the golden simulator (because the scan cell names in the ATPG model appear in the scan section of the parallel WGL output).
- The scan-in and scan-out pin names of all scan cells are the same.

To generate a basic WGL format test pattern file, use the following arguments with the `write_patterns` command:

`write_patterns filename [-Parallel | -Serial] -Wgl`

For more information about the `write_patterns` command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Standard Test Interface Language (STIL)

To generate a STIL format test pattern file, use the following arguments with the `write_patterns` command.

`write_patterns filename [-Parallel | -Serial] -STIL`

For more information about the `write_patterns` command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

ASIC Vendor Data Formats

The ASIC vendor test data formats include Texas Instruments TDL 91, Fujitsu FTDL-E, Mitsubishi MITDL, and Toshiba TSTL2. The ASIC vendor's chip testers use these formats.

All the ASIC vendor data formats are text-based and load data into scan cells in a parallel manner. Also, ASIC vendors usually impose several restrictions on pattern timing. Most ASIC vendor pattern formats support only a single timing definition. Refer to your ASIC vendor for test pattern formatting and other requirements.

The following subsections briefly describe the ASIC vendor pattern formats.

TI TDL 91	554
Fujitsu FTDL-E	554
Mitsubishi TDL	555
Toshiba TSTL2	555

TI TDL 91

This format contains test pattern data in a text-based format.

The tool supports features of TDL 91 version 3.0 and of TDL 91 version 6.0. The version 3.0 format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. TI's ASIC division imposes the additional restriction that comparison should always be done at the end of a tester cycle.

To generate a basic TI TDL 91 format test pattern file, use the following arguments with the `write_patterns` command:

```
write_patterns filename -Tltdl
```

The formatter writes the complete test data to the file *filename*. It also writes the chain test to another file (*filename.chain*) for separate use during the TI ASIC flow.

For more information about the `write_patterns` command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Fujitsu FTDL-E

This format contains test pattern data in a text-based format. The FTDL-E format splits test data into patterns that measure 1 or 0 values, and patterns that measure Z values. The test patterns divide into test blocks that each contain 64K tester cycles.

To generate a basic FTDL-E format test pattern file, use the following arguments with the write_patterns command:

write_patterns filename -Fjtdl

The formatter writes the complete test data to the file named *filename.fjtdl.func*. If the test pattern set contains IDDQ measurements, the formatter creates a separate DC parametric test block in a file named *filename.ftjtl.dc*.

For more information about the write_patterns command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Mitsubishi TDL

This format contains test pattern data in a text-based format.

To generate a basic Mitsubishi Test Description Language (TDL) format test pattern file, use the following arguments with the write_patterns command:

write_patterns filename -Mlttl

The formatter represents all scan data in a parallel format. It writes the test data into two files: the program file (*filename.td0*), which contains all pin definitions, timing definitions, and scan chain definitions; and the test data file (*filename.td1*), which contains the actual test vector data in a parallel format.

For more information about the write_patterns command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Toshiba TSTL2

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. TSTL2 represents all scan data in a parallel format.

To generate a basic Toshiba TSTL2 format test pattern file, use the following arguments with the write_patterns command:

write_patterns filename -TSTL2

The formatter writes the complete test data to the file named *filename*. For more information about the write_patterns command and its options, refer to the [write_patterns](#) description in the *Tessent Shell Reference Manual*.

Vector Creation and Modification

The Tesson tools provide a method to manipulate the Broadside vector data within a pattern set based on Boolean manipulation of the states on I/O ports or internal pins, creating new Broadside data to save instead.

This new Broadside vector data may represent a new top level of hierarchy of the design under test that is not present in the tool. Currently, this manipulation is restricted to Broadside pattern sets (such as IJTAG pattern sets) or ATPG scan pattern data written using the -serial switch of the write_patterns command when creating Broadside vectors.

You can also create broadside vector data from any source, pass the data to the write_patterns command, and have the tool write the vector data in one of the pattern formats including the test_setup and test_end procedure data, if present.

When using vector creation and modification, the terms pattern and vector have specific meaning as follows:

- **Pattern** — A pattern is an arrangement of test stimuli and responses meant to target a specific item or fault(s) to test. In ATPG, a pattern usually consists of a set of scan loads, some launch and capture stimulus, and then a single scan unload. As it is stored internally to the ATPG tool, a pattern does not yet consist of test vectors. That is, the data could not be written directly out as a set of test vectors that could be loaded onto a tester.
- **Vector** — A vector is a set of force and expect values on device ports that occur during one test cycle. Vectors can be processed and loaded into tester memory and applied to the DUT. A set of vectors can be grouped in such a way that they represent an ATPG pattern, but a pattern does not necessarily consist of a set of vectors, until the vectors are created by the write_patterns command to represent that pattern.

This pattern modification functionality consists of the following areas that work together to allow Broadside vectors in a pattern set to be modified or created:

Port List Modification	556
Vector Data Callbacks	557
Used Ports and Vector Callback Examples	562

Port List Modification

You modify the port list used to save the Broadside vectors.

You use the following commands when creating and modifying ports lists:

- [get_write_patterns_options](#)
- [report_write_patterns_options](#)

- [set_write_patterns_options](#)

These commands can be used to restrict the tool to only using a subset of the existing design ports for writing the patterns or can be used to add additional top-level ports that are not present in the design data that was loaded into the tool. Regardless of how the port list is modified, the full set of ports and force and expect values will still be available to be passed to the vector callback procs.

For example, a Tcl vector callback proc can use the full set of force and expect values that are in the Broadside vector to create new force and expect values that then only apply to the subset of ports. The commands will be able to create a set of ports to use, remove the existing set, report on the existing set, and introspect the set of ports to use.

See “[Used Ports and Vector Callback Examples](#)” on page 562 for complete usage examples.

Vector Data Callbacks

Using callbacks procs, you programmatically modify the force and expect values for ports in the Broadside vector data.

Vector Data Callback Procs

Another set of command options is used to enable Tcl callback procs that you can apply to all Broadside vectors. You can only use this option when writing serial scan patterns, or other pattern sets that do not have parallel load scan data (such as IJTAG patterns). These callback procs are inserted into the data flow for the [write_patterns](#) command. The write_patterns command takes a specified pattern set from the tool’s internal pattern storage and processes it or expands it into a set of Broadside tester vectors, one pattern at a time. These tester vectors are then written to the proper pattern file format. The callback is inserted into this flow after the tester vectors are created and before they are written, allowing them to be modified prior to being written to the pattern file.

In the process of writing out the modified vectors, the write_patterns command forward fills any input ports with an N value with the last forced value on that port from previous vectors. Thus, if a callback proc places an N on an input port, it will be replaced with a known value (0 or 1) if any previous vectors have forced a known value on that port. It is also possible that a vector that is passed to a callback proc may have an N value on an input port if this port was not explicitly forced in a procedure or by the data in the pattern set being written.

Tcl procs can be defined in a dofile, at the command prompt, or sourced from an external file. You must define the Tcl proc before the command is used to add the Tcl proc as a vector callback proc.

These callbacks will be passed a vector set from the write_patterns command that contains force and expect values on the full set of ports and pins, or an optionally specified set of ports and pins. The callback will return a vector set that contains new force and expect values for the full

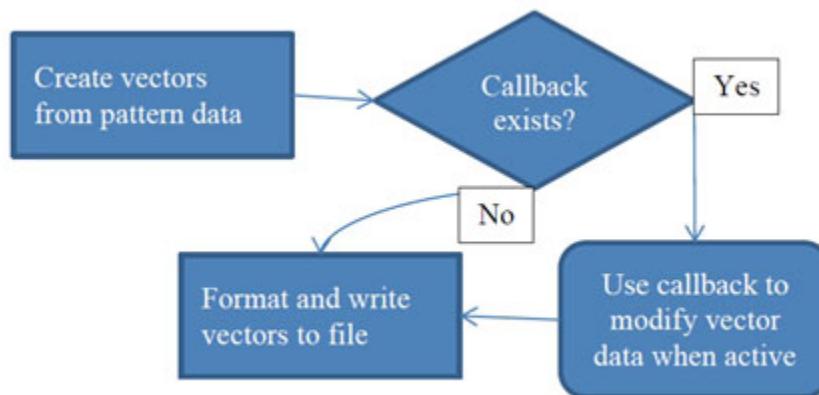
set of ports and pins, or an optionally specified reduced set of ports and pins. The callback can also return an empty vector set, or a vector set with multiple vectors in it, multiplying the number of vectors passed to it. The write_patterns command will then use the returned vector set in place of the passed one when it writes the vectors to the pattern file. When a callback returns an empty vector set, this indicates that no vectors will be written for the vector which was passed to the callback. Thus, the callback can remove vectors from the list of vectors to be written. For example, a Tcl callback to would look similar to the following:

```
proc myCallback { receivedVecSet } {  
    # code here to create returnedVecSet from receivedVecSet  
    return $returnedVecSet  
}
```

Callback procs will be executed by your Tcl interpreter in a mode that only allows introspection commands.

Figure 11-11 provides an example of the data flow within the write_patterns command when a callback is used.

Figure 11-11. Example Vector Callback



Tcl Vector Set Format

A vector_set is a a Tcl dictionary.

```

vector_set is '{` `vector_list' `{' <vector_statement> <vector_statement>
... '}' '}''

vector_statement is <vector> | <timeplate_statement> | <repeat_statement>

vector is '{` [ `annotation' <ann_string> ] [ `type' <type_string> ]
[ `port_values' <value_string> ] '}''

timeplate_statement is '{` `timeplate' <tp_string> '}''

repeat_statement is '{` `repeat' <repeat_num> '{` <vector_statement> ... '}''
'}`'

ann_string is any string including new lines, quotes, and curly braces
type_string is one of 'shift', 'load_unload', 'test_setup', 'test_end',
'shadow_control', 'master_observe', 'shadow_observe', 'ijtag_shift',
'ijtag_update', 'ijtag_hold', 'ijtag_reset', and 'capture'. tp_string is
the name of a valid timeplate to use for this vector and all subsequent
vectors in this vector_set, unless another timeplate_statement is used.
value_string is any sequence of the characters 0 1 N Z P L H X T without
spaces or newlines.

```

A vector_list is a Tcl dictionary of items where each item contains a list of more dictionaries. Syntactically it looks like this:

```

'{` `vector_list' `{' ( `{' `timeplate' string '}`
| `{' `annotation' string '}`
| `{' `type' string 'port_values' string '}`
| `{' `repeat' string '{` ... '}' '}`)*
`}`,
`}'

```

The vector_list is followed by its item is the only top-level dictionary entry. From a Tcl processing point of view, the following code snippet shows how to access and modify every port_value entry in a received_vector_set:

```

proc my_callback {received_vector_set} {
    foreach vector [dict get $received_vector_set vector_list] {
        if {[dict exists $vector port_values]} {
            dict set vector port_values //some modification of port values
        }
        lappend vector_list $vector
    }
    return [list vector_list $vector_list]
}

```

The vector_set repeat statement can be used to repeat a single vector or a block of vector statements. The repeat_num string specifies that the vector is repeated a certain number of times. The number of characters in value_string is equal to the number of ports used. Both the string pair of annotation and the ann_string and the port_values and value_string are optional.

It is possible to have a vector that is completely empty and a vector_set that consists of nothing but empty vectors or completely empty. When used with callback procs, if no optional port lists

were specified when the callback proc was declared, then the vector port_values is expected to have enough data for the full port list, in the order of the full port list. If port lists are specified when the callback proc is added, then the port_values in the vector set will contain values for only those ports in the port list in the order specified. Vector sets passed to the callback procs and returned from the callback procs all have the same format but may contain different port values based on the received_port and returned_port lists (see “[Port List Modification](#)” on page 556).

Each string for the port values should only consist of the following characters:

- 0 1 N Z — Force low, force high, force unknown, force Z input values
- P — Pulse clock using pulse timing in timeplate
- L H X T — Measure low, measure high, measure unknown (no measure), and measure Z output values

The force unknown value (N) on an input pin becomes a force of the previous value when combined with other vectors. The annotation keyword is used to include an annotation that is associated with a vector. These annotations can come from the “annotate” statement in the test procedures, annotations generated from IJTAG iNotes and IJTAG solver statements, and implicit annotations that are created when writing patterns, such as the “Pattern” or “Chain Pattern” annotations. For these implicit annotations, since these are normally generated by the pattern format writer, and are not always consistent, a new consistent annotation is created and passed to the vector callbacks in their place. These annotations take the form of “MGC: Chain Pattern” and “MGC: Pattern”, representing the start of a chain test pattern and start of a scan test pattern respectfully. If these annotations are returned from the vector callback proc, then they are removed before being passed to the pattern format writer as the pattern format writer will still insert its implicit annotation.

The type keyword is used to attach a vector type to a vector. The strings used to identify the type correspond to the procedures which produced the vector, or if ‘capture’ is the type string, this corresponds to all launch and capture vectors regardless of how they were produced. In the IJTAG flow, the vector type can also correspond to IJTAG register states in the solved pattern set. These vector types will be in the vectors that are passed to the vector callback procs from the tool and can be used by the callback proc to determine what vector values to return. It is not necessary for the callback proc to return vectors with these types, any untyped vector will keep the type of whatever vector occurred previously. The timeplate keyword is used to change the timeplate used for the next vector, and all subsequent vectors for this vector set, unless another timeplate statement is encountered. At the end of the vector_set, the timeplate used reverts to the timeplate that was in effect prior to the vector_set. For example, if the timeplate specified for an IJTAG pattern set is “tp1”, and if a callback proc returns a vector_set that has three vectors, where the timeplate statement changes the timeplate to “tp2” prior to the second vector, then the final list of vectors would have the timeplate “tp1” used for the first vector, timeplate “tp2” used for the second and third vector, and any vectors after that would return to using timeplate “tp1” unless otherwise changed by any subsequent callback procs. The repeat keyword can be used to repeat one or more vectors in a vector_set N times as specified by the

`repeat_num`. All entries in a `vector_set` are optional, so it is possible for a callback to return an empty vector set to remove vectors. All entries in a vector are also optional, so it is possible to have a vector set that has an empty vector. For an empty vector, the values on the ports are assumed to be N or the previously forced value for input pins, X for output pins, and Z for bidi pins. For example, if a callback proc is passed a `vector_set` that has one vector in it, and the callback proc returns an empty `vector_set` (just { }), this means that callback proc has removed the vector in the `vector_set` that was passed to it. If on the other hand a callback proc returns a vector set that contains an empty vector ({ `vector_list` {} }), this is an empty vector that means everything remains unchanged from the previous vector, similar to an empty cycle in a procedure of the procedure file. A callback procedure that does not return anything can be used to just introspect the vector data as it will not modify the vector data being written in any way.

Here is an example of a vector set:

```
vector_list {{ annotation "my annotation" port_values 000 }
    { timeplate tp1 }
    { type load_unload port_values 101 }
    { repeat 200 { port_values 100 } } // repeat this 200 times
    { port_values 110 }
    { repeat 1000 { {port_values 100} } // start of loop for 1000 times
        { port_values 101 }
        { port_values 100 } } // end of loop for 1000 times
}
```

Note that when a vector set is passed to a vector callback procedure when writing patterns, if there are any repeat blocks in the vectors to be written (from ijtag patterns or sub procedures) then the “repeat” statement and all of vector data within the repeat is passed to the callback in one vector set. There is no syntax for specifying the begin repeat, followed by calling the callback for each individual vector within the repeat.

Flow Example

The following example demonstrates using a vector callback procedure to invert the value on an internal TDI port to the external TDI port. In this case, the external port is already in the netlist so a pseudo port is added for the internal TDI, and `delete_connections` and `create_connections` are used to attach the pseudo port to the internal cut point. This is not a complete example, but it shows the creation of the pseudo ports, the definition of the callback procs, and the commands which add the callback proc and set the existing used ports.

```
SETUP> set_context dft
SETUP> set_system_mode insertion
INSERTION> create_port my_tdi -on_module top -dir input
INSERTION> delete_connections U1/Y
INSERTION> create_connections my_tdi internal_inst/tdi
INSERTION> set_system_mode setup
SETUP> set_context patterns -ijtag
SETUP> set_system_mode analysis
ANALYSIS> proc ModifyPortValue { port_values } {
    set port_val_length [string length $port_values]
    for {set i 0} {$i < $port_val_length} {incr i} {
        switch [string index $port_values $i ]{
            "0" { set port_values [string replace $port_values $i $i 1 ]}
            "1" { set port_values [string replace $port_values $i $i 0 ]}
            "L" { set port_values [string replace $port_values $i $i H ]}
            "H" { set port_values [string replace $port_values $i $i L ]}
            "X" { set port_values [string replace $port_values $i $i X ]}
            Default { puts [string index $port_values $i] }
        }
    }
    return $port_values
}
ANALYSIS> proc myCallback {received_vector_set} {
    set cnt 0
    foreach i $received_vector_set {
        set port_values_idx [lsearch $i port_values]
        if {$port_values_idx ne -1} {
            incr port_values_idx
            set port_values [lindex $i $port_values_idx]
            set port_values [ModifyPortValue $port_values]
            set tmp [lreplace $i $port_values_idx $port_values_idx $port_values]
            set received_vector_set [lreplace $received_vector_set $cnt $cnt $tmp]
        }
        incr cnt
    }
    return $received_vector_set
}
ANALYSIS> proc RemovePortFromList {port_list port} {
    set idx [lsearch $port_list $port]
    set port_list [lreplace $port_list $idx $idx]
    return $port_list
}
ANALYSIS> set_write_pattern_options -vector_callback myCallback -trigger every_cycle
ANALYSIS> set write_ports [get_write_pattern_options -returned myCallback ]
ANALYSIS> set write_ports [RemovePortFromList $write_ports "my_tdi" ]
ANALYSIS> set_write_pattern_options -existing_used_ports $write_ports
ANALYSIS> set_write_pattern_options -vector_callback myCallback -trigger every_cycle \
           -received {my_tdi} -returned_port {tdi} -repl
```

Used Ports and Vector Callback Examples

This section provides a series of simple examples for using used ports and vector callbacks.

Example 1

The following example shows how to work with the set of used ports and vector callbacks in:

```
ANALYSIS> set_write_pattern_options -reset_used_ports
ANALYSIS> set my_ports [get_write_pattern_options -all_used_port_list ]
# store a TCL list of all ports
# now set the port list to be a subset of the original, plus one
# more input port "top tdi"
ANALYSIS> set_write_pattern_options -existing_used_ports $my_subset_ports \
-additional_port_list {top_tdi input}
# now write out a Verilog testbench from an ijttag pattern set called pat_set1 using only the
# used ports defined above
ANALYSIS> write_patterns my_pat.v -verilog -pattern_sets
# pat_set1
```

In the above example, since we were writing an ijttag pattern set, there is no scan pattern data within the pattern set, so it was not necessary to use the –serial switch on the write_patterns command. In the following example, we will write ATPG scan patterns, so it will be necessary to use the –serial switch in order to only produce broadside vector data. Using the –parallel switch would cause the tool to issue an error, as that would produce both broadside vectors and scan vectors.

Example 2

The following example shows how to setup a vector callback proc to modify the vectors in a set of scan patterns written as broadside vectors using the –serial switch. Assume the same commands to modify the set of used ports was also used for this.

```
# add a vector callback to modify all of the vectors
ANALYSIS> set_write_pattern_options -vector_callback protocol_A \
-returned_port [concat $my_ports [list top_tdi] ]
# now write a Verilog testbench using the modified port list and broadside vectors
# modified by protocol_A
ANALYSIS> write_patterns pat1_protocol_A.v -verilog -serial -replace
```

Example 3

The following example shows how to create a user-defined vector set which then can be written out instead of a pattern set.

```
# create a new vector and append to a vector_list
ANALYSIS> set ann "some annotation string here"
ANALYSIS> set port_vals 0101HZN
ANALYSIS> set my_vector [list annotation $ann port_values $port_vals ]
ANALYSIS> lappend vector_list $my_vector
# assume other vectors have been created
ANALYSIS> lappend vector_list $second_vector
ANALYSIS> lappend vector_list $third_vector
ANALYSIS> set my_vector_set [list vector_list $vector_list]
# now write a Verilog testbench using the vector_set
ANALYSIS> write_patterns pat1_vector_set1.v -verilog -serial -vector_set $my_vector_set
```

Example 4

The following example shows how to use a callback proc to set a value on an additional fake port to indicate that an analogue measurement should be performed when a particular iNote is found in an iJtag pattern set that has the “analogue” keyword.

```
ANALYSIS> set_write_pattern_options -reset_used_ports
# now set the port list to be the default plus one more input port "ana_measure"
ANALYSIS> set_write_pattern_options -additional_port_list {{ana_measure input}}
ANALYSIS> proc myCallback {received_vector_set} {
    set port_index [lsearch [get_write_pattern_options -received_callback_port_list \
        myCallback] ana_measure]
    array set vector_element [lindex $received_vector_set 0]
    if {[string first "analogue" $vector_element(annotation)] != -1 } {
        set vector_element(port_values)
        [string replace $vector_element(port_values) port_index port_index 1]
    } else {
        set vector_element(port_values)
        [string replace $vector_element(port_values) port_index port_index 0]
    }
    return [list vector_list [array get $vector_element]]
}
ANALYSIS> set_write_pattern_options -vector_callback myCallback \
    -returned_ports [concat $some_ports [list ana_measure]] \
    -received_ports [concat $some_ports [list ana_measure]]
ANALYSIS> write_patterns pat_analogue.stil -stil -pattern_set ijtag_analogue_test
```

In the above example, the code that sets the variable “port_index” could be moved outside the callback proc to improve performance. This however does create a dependency that “port_index” must be properly set by user Tcl code prior to using the write_patterns command with this callback, and if multiple port indexes are needed, then they all must be defined with unique variable names and properly set before using the callback proc.

Example 5

The following example shows how to use a callback to echo all of the vector data being saved to the log file without modifying the vector data in any way.

```
ANALYSIS> set_write_pattern_options -reset_used_ports
ANALYSIS> proc myCallback {received_vector_set} {
    puts $received_vector_set
}
ANALYSIS> set_write_pattern_options -vector_callback myCallback
ANALYSIS> write_patterns pat_echo.stil -stil
```

Example 6

The following example shows to add additional ports that have the clock attribute and how to use the pulse_additional statement in the procedure file to specify timing for the additional port.

In the procedure file, the timeplate definition could look like this:

```
timeplate tp1 =  
    force_pi 0;  
    measure_po 10;  
    pulse_clock 20 20;  
    pulse_additional NewClock 25 10;  
    period 50;  
end;
```

The command to add the additional clock ports is:

```
ANALYSIS> set_write_pattern_options \  
-additional_port_list { { NewClock input clock } { NewClock2 input clock 1 } }
```

Two additional_ports are added. The port “NewClock” is an input port that is a clock, it has an offstate of 0, and it uses the timing described in the timeplate for “NewClock”. The port “NewClock2” is an input port that is a clock, it has an offstate of 1, and it uses the timing described in the timeplate by the pulse_clock statement.

Chapter 12

Test Pattern File Formats

This chapter provides information about the ASCII and BIST pattern file formats.

ASCII File Format.....	568
BIST Pattern File Format.....	577

ASCII File Format

The ASCII file that describes the scan test patterns is divided into five sections, which are named header_data, setup_data, functional_chain_test, scan_test, and scan_cell. Each section (except the header_data section) begins with a section_name statement and ends with an end statement. Also in this file, any line starting with a double slash (//) is a comment line.

Header_Data	568
ASCII Setup_Data	568
Functional_Chain_Test	571
Scan_Test	572
Scan_Cell	575

Header_Data

The header_data section contains the general information, or comments, associated with the test patterns. This is an optional section that requires a double slash (//) at the beginning of each line in this section. The data printed may be in the following format:

```
// model_build_version - the version of the model build program that was
// used to create the scan model.

// design_name - the design name of the circuit to be tested.

// date - the date in which the scan model creation was performed.

// statistics - the test coverage, the number of faults for each fault
// class, and the total number of test patterns.

// settings - the description of the environment in which the ATPG is
// performed.

// messages - any warning messages about bus contention, pins held,
// equivalent pins, clock rules, and so on are noted.
```

ASCII Setup_Data

The setup_data section contains the definition of the scan structure and general test procedures that will be referenced in the description of the test patterns.

Note

 Additional formats are added to the setup_data section for BIST patterns. For information on these formats, see “[BIST Pattern File Format](#)” on page 577”.

The data printed will be in the following format:

```
SETUP =
    <setup information>
END;
```

The setup information will include the following:

declare input bus “PI” = <ordered list of primary inputs>;

This defines the list of primary inputs that are contained in the circuit. Each primary input will be enclosed in double quotes and be separated by commas. For bidirectional pins, they will be placed in both the input and output bus.

declare output bus “PO” = <ordered list of primary outputs>;

This defines the list of primary outputs that are contained in the circuit. Each primary output will be enclosed in double quotes and be separated by commas.

```
CLOCK "clock_name1" =
    OFF_STATE = <off_state_value>;
    PULSE_WIDTH = <pulse_width_value>;
END;
CLOCK "clock_name2" =
    OFF_STATE = <off_state_value>;
    PULSE_WIDTH = <pulse_width_value>;
END;
```

This defines the list of clocks that are contained in the circuit. The clock data will include the clock name enclosed in double quotes, the off-state value, and the pulse width value. For edge-triggered scan cells, the off-state is the value that places the initial state of the capturing transition at the clock input of the scan cell.

```
WRITE_CONTROL "primary_input_name" =
    OFF_STATE = <off_state_value>;
    PULSE_WIDTH = <pulse_width_value>;
END;
```

This defines the list of write control lines that are contained in the circuit. The write control line will include the primary input name enclosed in double quotes, the off-state value, and the pulse width value. If there are multiple write control lines, they must be pulsed at the same time.

```
PROCEDURE TEST_SETUP "test_setup" =
    FORCE "primary_input_name1" <value> <time>;
    FORCE "primary_input_name2" <value> <time>;
    ...
    ...
END;
```

This is an optional procedure that can be used to set nonscan memory elements to a constant state for both ATPG and the load/unload process. It is applied once at the beginning of the test pattern set. This procedure may only include force commands.

```
SCAN_GROUP "scan_group_name1" =
    <scan_group_information>
END;
SCAN_GROUP "scan_group_name2" =
    <scan_group_information>
END;
....
```

This defines each scan chain group that is contained in the circuit. A scan chain group is a set of scan chains that are loaded and unloaded in parallel. The scan group name will be enclosed in double quotes and each scan group will have its own independent scan group section. Within a scan group section, there is information associated with that scan group, such as scan chain definitions and procedures.

```
SCAN_CHAIN "scan_chain_name1" =
    SCAN_IN = "scan_in_pin";
    SCAN_OUT = "scan_out_pin";
    LENGTH = <length_of_scan_chain>;
END;
SCAN_CHAIN "scan_chain_name2" =
    SCAN_IN = "scan_in_pin";
    SCAN_OUT = "scan_out_pin";
    LENGTH = <length_of_scan_chain>;
END;
....
```

The scan chain definition defines the data associated with a scan chain in the circuit. If there are multiple scan chains within one scan group, each scan chain will have its own independent scan chain definition. The scan chain name will be enclosed in double quotes. The scan-in pin will be the name of the primary input scan-in pin enclosed in double quotes. The scan-out pin will be the name of the primary output scan-out pin enclosed in double quotes. The length of the scan chain will be the number of scan cells in the scan chain.

```
PROCEDURE <procedure_type> "scan_group_procedure_name" =
    <list of events>
END;
```

The type of procedures may include shift procedure, load and unload procedure, shadow-control procedure, master-observe procedure, shadow-observe procedure, and skew-load procedure. The list of events may be any combination of the following commands:

```
FORCE "primary_input_pin" <value> <time>;
```

This command is used to force a value (0,1, X, or Z) on a selected primary input pin at a given time. The time values must not be lower than previous time values for that procedure. The time for each procedure begins again at time 0. The primary input pin will be enclosed in double quotes.

```
APPLY "scan_group_procedure_name" <#times> <time>;
```

This command indicates the selected procedure name is to be applied the selected number of times beginning at the selected time. The scan group procedure name will be enclosed in double quotes. This command may only be used inside the load and unload procedures.

```
FORCE_SCI "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be placed on the scan chain inputs. The scan chain name will be enclosed in double quotes.

```
MEASURE_SCO "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be measured on the scan chain outputs. The scan chain name will be enclosed in double quotes.

Functional_Chain_Test

The functional_chain_test section contains a definition of a functional scan chain test for all scan chains in the circuit to be tested. For each scan chain group, the scan chain test will include a load of alternating double zeros and double ones (00110011...) followed by an unload of those values for all scan chains of the group.

The format is as follows:

```
CHAIN_TEST =
    APPLY "test_setup" <value> <time>;
    PATTERN = <number>;
    APPLY "scan_group_load_name" <time> =
        CHAIN "scan_chain_name1" = "values....";
        CHAIN "scan_chain_name2" = "values....";
        ....
        ....
    END;
    APPLY "scan_group_unload_name" <time> =
        CHAIN "scan_chain_name1" = "values....";
        CHAIN "scan_chain_name2" = "values....";
        ....
        ....
    END;
END;
```

The optional “test_setup” line is applied at the beginning of the functional chain test pattern if there is a test_setup procedure in the Setup_Data section. The number for the pattern is a zero-based pattern number where a functional scan chain test for all scan chains in the circuit is to be tested. The scan group load and unload name and the scan chain name will be enclosed in double quotes. The values to load and unload the scan chain will be enclosed in double quotes.

During the loading of the scan chains, each value of the corresponding scan chain will be placed at its scan chain input pin. The shift procedure will shift the value through the scan chain and continue shifting the next value until all values for all the scan chains have been loaded. Since the number of shifts is determined by the length of the longest scan chain, X’s (don’t care) are placed at the beginning of the shorter scan chains. This will ensure that all the values of the scan chains will be loaded properly.

During the unloading of the scan chains, each value of the corresponding scan chain will be measured at its scan chain output pin. The shift procedure will shift the value out of the scan chain and continue shifting the next value until all values for all the scan chains have been unloaded. Again, since the number of shifts is determined by the length of the longest scan chain, X’s (don’t measure) are placed at the end of the shorter scan chains. This will ensure that all the values of the scan chains will be unloaded properly.

Here is an example of a functional scan chain test:

```
CHAIN_TEST =
    APPLY "test_setup" 1 0;
    PATTERN = 0;
    APPLY "g1_load" 0 =
        CHAIN "c2" = "XXXXXXXXX0011001100110011001100";
        CHAIN "c1" = "XXXXXXXXXXXXXX001100110011001100";
        CHAIN "c0" = "0011001100110011001100110011001";
    END;
    APPLY "g1_unload" 1 =
        CHAIN "c2" = "0011001100110011001100XXXXXXXX";
        CHAIN "c1" = "001100110011001100XXXXXXXXXXXX";
        CHAIN "c0" = "0011001100110011001100110011001";
    END;
END;
```

Scan_Test

The scan_test section contains the definition of the scan test patterns that were created by Tesson Shell.

A scan pattern will normally include the following:

```
SCAN_TEST =
    PATTERN = <number>;
    FORCE "PI" "primary_input_values" <time>;
    APPLY "scan_group_load_name" <time> =
        CHAIN "scan_chain_name1" = "values....";
        CHAIN "scan_chain_name2" = "values....";
    ....
    ....
    END;
    FORCE "PI" "primary_input_values" <time>;
    MEASURE "PO" "primary_output_values" <time>;
    PULSE "capture_clock_name1" <time>;
    PULSE "capture_clock_name2" <time>;
    APPLY "scan_group_unload_name" <time> =
        CHAIN "scan_chain_name1" = "values....";
        CHAIN "scan_chain_name2" = "values....";
    ....
    ....
    END;
    ....
    ....
    ....
END;
```

The number of the pattern represents the pattern number in which the scan chain is loaded, values are placed and measured, any capture clock is pulsed, and the scan chain is unloaded. The pattern number is zero-based and must start with zero. An additional force statement will be applied at the beginning of each test pattern, if transition faults are used. The scan group load and unload names and the scan chain names will be enclosed by double quotes. All the time values for a pattern must not be lower than the previous time values in that pattern. The values to load and unload the scan chain will be enclosed in double quotes. Refer to the “[Functional Chain Test](#)” on page 571 section on how the loading and unloading of the scan chain operates.

The primary input values will be in the order of a one-to-one correspondence with the primary inputs defined in the setup section. The primary output values will also be in the order of a one-to-one correspondence with the primary outputs defined in the setup section.

If there is a test_setup procedure in the Setup_Data section, the first event, which is applying the test_setup procedure, must occur before the first pattern is applied:

```
APPLY "test_setup" <value> <time>;
```

If there are any write control lines, they will be pulsed after the values have been applied at the primary inputs:

```
PULSE "write_control_input_name" <time>;
```

If there are capture clocks, then they will be pulsed at the same selected time, after the values have been measured at the primary outputs. Any scan clock may be used to capture the data into the scan cells that become observed.

Scan patterns will reference the appropriate test procedures to define how to control and observe the scan cells. If the contents of a master is to be placed into the output of its scan cell where it may be observed by applying the unload operation, the master_observe procedure must be applied before the unloading of the scan chains:

```
APPLY "scan_group_master_observe_name" <value> <time>;
```

If the contents of a shadow is to be placed into the output of its scan cell where it may be observed by applying the unload operation, the shadow_observe procedure must be applied before the unloading of the scan chains:

```
APPLY "scan_group_shadow_observe_name" <value> <time>;
```

If the master and slave of a scan cell are to be at different values for detection, the skew_load procedure must be applied after the scan chains are loaded:

```
APPLY "scan_group_skew_load_name" <value> <time>;
```

Each scan pattern will have the property that it is independent of all other scan patterns. The normal scan pattern will contain the following events:

1. Load values into the scan chains.
2. Force values on all non-clock primary inputs.
3. Measure all primary outputs not connected to scan clocks.
4. Exercise a capture clock (optional).
5. Apply observe procedure (if necessary).
6. Unload values from scan chains.

When unloading the last pattern, the tool loads the last pattern a second time to completely shift the contents of the last capture cycle so that the output matches the calculated value. For more information, see “[Handling of Last Patterns](#)” in the *Tessent TestKompress User’s Manual*.

Although the load and unload operations are given separately, it is highly recommended that the load be performed simultaneously with the unload of the preceding pattern when applying the patterns at the tester.

For observation of primary outputs connected to clocks, there will be an additional kind of scan pattern that contains the following events:

1. Load values into the scan chains.

2. Force values on all primary inputs including clocks.
3. Measure all primary outputs that are connected to scan clocks.

Scan_Cell

The scan_cell section contains the definition of the scan cells used in the circuit.

The scan cell data is in the following format:

```
SCAN_CELLS =
    SCAN_GROUP "group_name1" =
        SCAN_CHAIN "chain_name1" =
            SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
                        <relsciinv> <relscoinv> <instance_name>
                        <model_name> <input_pin> <output_pin>;
            ....
        END;
        SCAN_CHAIN "chain_name2" =
            SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
                        <relsciinv> <relscoinv> <instance_name>
                        <model_name> <input_pin> <output_pin>;
            ....
        END;
        ....
    END;
    ....
END;
```

The fields for the scan cell memory elements are the following:

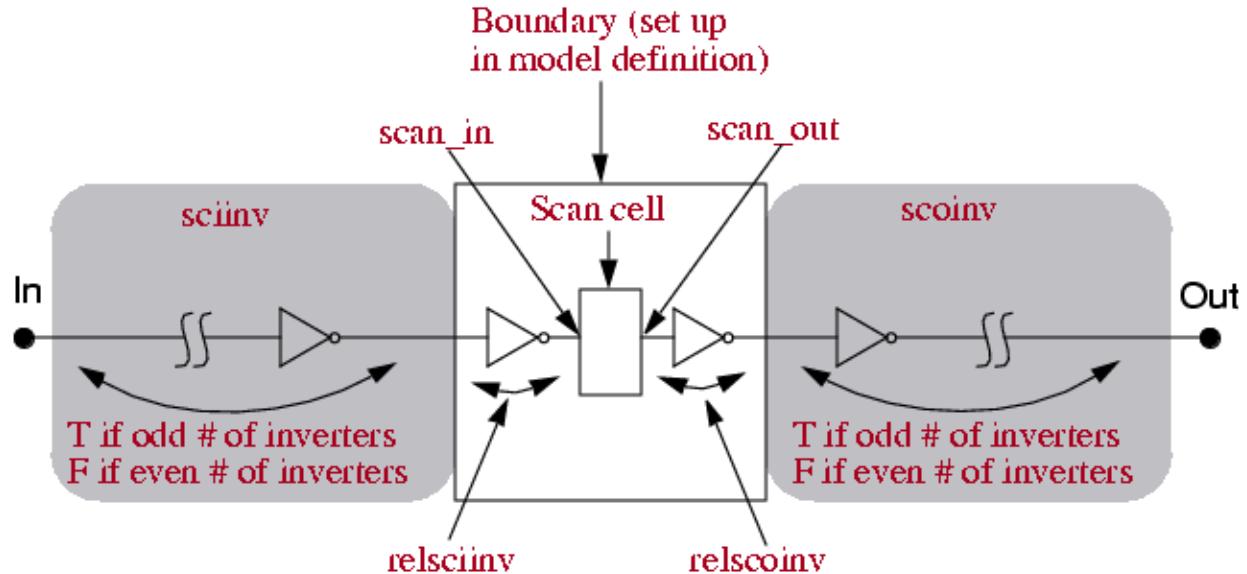
- **cellid** — A number that identifies the position of the scan cell in the scan chain. The number 0 indicates the scan cell closest to the scan-out pin.
- **type** — The type of scan memory element. The type may be MASTER, SLAVE, SHADOW, OBS_SHADOW, COPY, or EXTRA.
- **sciinv** — Inversion of the library input pin of the scan cell relative to the scan chain input pin. The value may be T (inversion) or F (no inversion).
- **scoinv** — Inversion of the library output pin of the scan cell relative to the scan chain output pin. The value may be T (inversion) or F (no inversion).
- **relsciinv** — Inversion of the memory element relative to the library input pin of the scan cell. The value may be T (inversion) or F (no inversion).
- **relscoinv** — Inversion of the memory element relative to the library output pin of the scan cell. The value may be T (inversion) or F (no inversion).
- **instance_name** — The top level boundary instance name of the memory element in the scan cell.

- **model_name** — The internal instance pathname of the memory element in the scan cell (if used - blank otherwise).
- **input_pin** — The library input pin of the scan cell (if it exists, blank otherwise).
- **output_pin** — The library output pin of the scan cell (if it exists, blank otherwise).

Example Circuit

Figure 12-1 illustrates the scan cell elements in a typical scan circuit.

Figure 12-1. Example Scan Circuit



BIST Pattern File Format

All of the test pattern file formats described in the previous section directly apply to BIST patterns also. However, an additional set of BIST-specific formats are added to the ASCII pattern file format.

The BIST-specific formats are described in this section.

BIST Pattern Setup_Data	577
Scan_Test for BIST	578

BIST Pattern Setup_Data

The setup_data section for BIST contains a subsection that defines the BIST pattern-specific configuration. This subsection includes a file version identifier and definitions for signature registers (prpg_register and misr_register) in the BIST pattern.

A BIST-specific statement is used to identify that the following configuration is BIST pattern-specific. The BIST-specific statement has the following format:

```
declare bist pattern specific configuration =
```

A version tag is used to indicate the BIST pattern version. The version tag has the following format:

```
BIST_ASCII_PATTERN_FILE_SUBVERSION
```

Here is an example of a BIST pattern-specific statement combined with a version tag statement:

```
declare bist pattern specific configuration =
BIST_ASCII_PATTERN_FILE_SUBVERSION = 1;
```

Note

 The “bist pattern specific configuration” statement disappears if a LFSM value is not included.

A *pattern_internal_view* switch that indicates whether the BIST pattern internal view is written to the pattern file; its possible values are “ON” or “OFF”. If the switch is set to on, as shown in the following example, BIST patterns are included in the pattern file.

```
pattern_internal_view = "on"
```

The *prpg_register* and *mistr_register* identifiers define signature registers such as PRPG and MISR in the BIST pattern. The signature register statements use the following format:

```
signature_register <name of signature register> =
    length = <length of the register>;
    type = <type of the register>;
    init_value = <initial state of the register>;
end;
```

Here are some examples of signature register statements:

```
prpg_register "decomp1" =
    length = 22;
    type = PRPG;
    init_value = "000000000000000000000000";
end;

prpg_register "decomp2" =
    length = 12;
    type = PRPG;
    init_value = "000000000000";
end;

prpg_register "prpg1" =
    length = 16;
    type = PRPG;
    init_value = "0000000000000000";
end;

prpg_register "prpg2" =
    length = 16;
    type = PRPG;
    init_value = "0000000000000000";
end;

mistr_register "mistr1" =
    length = 32;
    type = MISR;
    init_value = "11111111111111111111111111111111";
end;

mistr_register "mistr2" =
    length = 24;
    type = MISR;
    init_value = "11111111111111111111111111111111";
end;
```

Scan_Test for BIST

Each BIST pattern includes one *l fsm_snapshot* statement. Within the pattern statement, the keyword *pre_load*, *pre_unload*, or *post_unload* is followed by a register name to indicate when the snapshot for the register is to be taken.

The keywords are applied as follows:

- *pre_load* — Used for PRPG type registers.
- *pre_unload* and *post_unload* — Used for MISR type registers.

The snapshot statement is composed so that the simulation of each BIST pattern is independent from all others:

- Given the *pre_load* value of PRPG registers, the loading data for each BIST pattern can be computed.
- Given the *pre_unload* value of MISR registers, the MISR registers after unload can be computed and thus verified.

Here is an example of a pattern containing a `snapshot` statement:

```
pattern = 0;
l fsm_snapshot =
    pre_load "decomp1" = "000000000000000000000000";
    pre_load "decomp2" = "000000000000";
    pre_load "prpg1" = "111111111111111";
    pre_load "prpg2" = "111111111111111";
    pre_unload "misr1" = "11111111111111111111111111111111";
    pre_unload "misr2" = "11111111111111111111111111111111";
    post_unload "misr1" = "111111110001111101111111011111";
    post_unload "misr2" = "1111111000000001111111";
end;

apply "grp1_load" 0 =
...
end;

force "PI" "...." 1;
measure "PO" "...." 2;

apply "grp1_unload" 3 =
...
end;
```


Chapter 13

Power-Aware DRC and ATPG

This chapter describes the power-aware DRC and ATPG flow for use with the ATPG tool.

This chapter contains the following sections:

Power-Aware Overview	582
Assumptions and Limitations	582
Multiple Power Mode Test Flow	583
Power-Aware ATPG for Traditional Fault Models	583
CPF and UPF Parser	584
Power-Aware ATPG Procedure	586
Power-Aware Flow Examples	588

Power-Aware Overview

The electronics industry has adopted low-power features in major aspects of the design continuum. In response, EDA vendors and major semiconductor companies have defined the commonly-used power data standard formats to describe the power requirements: UPF and CPF.

Tessent Shell supports the following versions of the UPF and CPF formats:

- IEEE 1801 standard / UPF 2.0
- Common Power Format (CPF) 1.0 and 1.1

You load this power data directly into the tool to collect the power information. Once loaded, the tools perform the necessary DRCs to ensure that the DFT logic is inserted properly with respect to the design's power domains and, if the design passes the rule checks, perform ATPG with the given power mode configuration. For information about the power-aware DRC (V) rules, refer to the [Tessent Shell Reference Manual](#).

The tool's low-power functionality provides you with a method to do the following:

- Provide DRCs to trace the active power mode and ensure that the scan operation works under the current power configuration.
- Provide capability to generate test for the traditional fault models while aware of the power mode configuration.

Assumptions and Limitations	582
Multiple Power Mode Test Flow	583
Power-Aware ATPG for Traditional Fault Models	583

Assumptions and Limitations

The power-aware functionality comes with a few assumptions and limitations.

- Circuit hierarchy is preserved in the CPF or UPF file and is the same as the netlist. Note that for modular EDT design, the module hierarchy that includes EDT logic needs to be preserved, otherwise the tool may not be able to associate an EDT block with a power domain. Consequently, some power DRC rules related to EDT logic may not be performed.
- The power-aware DRC rules and reporting are based on the loaded power data (UPF or CPF). The DRC rules do not include testing that crosses different power modes. Additionally, the DRC rules do not cover the testing of the shutoff power domain such as the retention cells testing.

Multiple Power Mode Test Flow

For a design with multiple test modes, the scan chain configuration may be different for each power mode.

You should perform the DRC and ATPG separately for each power mode using the following steps:

1. Configure the power mode to be tested using the `test_setup` procedure.
2. Load the CPF file. The tool automatically identifies the power mode that the system is currently configured to and report it to user.

In addition, new V rules are defined and are checked when you switch to a non-SETUP mode. (See “[Power-Aware Rules \(V Rules\)](#)” in the *Tessent Shell Reference Manual*.) This ensure that the loaded UPF/CPF file contains consistent power information and the inserted DFT logic (for example, scan insertion) has considered the design power domains properly.

Note

 For the purpose of DRC, you must load to load a UPF/CPF file in the SETUP mode. When loading a UPF/CPF file in a non-SETUP mode, the tool does not perform V DRC until you switch the tool from the SETUP mode to a non-SETUP mode.

Pattern Generation

ATPG generates patterns only for the current power mode. If there are different power modes which enable the identical power domains (just in different voltage configuration), then the pattern set can be reused by loading the pattern file and performing fault grading. Reuse of the pattern set is at your discretion.

Despite the pattern reuse, you should still write one `test_setup` procedure for every power mode to be tested, and perform DRC check for each power mode to ensure the scan chains can operate properly in the power mode. In addition, a new pattern set should be stored to reflect the updated `test_setup` for the corresponding power mode. Finally, when writing out patterns after ATPG, the tool should also save the current power mode information (as a comment) to the pattern file for the user’s reference.

Power-Aware ATPG for Traditional Fault Models

During the traditional fault model test stage, the circuit is configured into a static power mode by the test procedure at the beginning of the capture cycle and stays at the same power mode for the entire capture cycle. The ATPG engine explicitly prevents the power control logic from changing the active power mode.

A low-power DRC is performed before ATPG to check if the active power mode can be disturbed and the analysis result is used by ATPG to determine if any additional ATPG effort is

needed to keep the static power mode. This is similar to E10 rule for the bus contention check which is used by ATPG to enable the extra justification to prevent the bus contention. If the circuit contains the power mode that powers on all power domains (called ALL_ON state), you can use this state for the traditional fault models. An ALL_ON state allows the circuit to be tested for logic faults in one test set run. In addition, this state also allows the tool to perform DRC for entire circuit in this run.

Power Partitioning

If the circuit needs to partition with partial power domains powered in a given time, then you must perform multiple ATPG runs; specifically, each run with its procedure file and the scan configuration. You must ensure that every power domain is covered by at least one run. Additionally, the chip- level test coverage can be computed manually from each separate run.

In the case of multiple power partition flow, the always on power domain, the faults may be targeted multiple times. To reduce the creation of patterns for same faults in the always-on domains, you can use the following command:

add_faults –power_domains always_on -delete

This explicitly removes the faults in always_on domain if the faults have been targeted by other run.

The other way to avoid creation of duplicated patterns is to load the previous generated fault list (using [read_faults –merge](#)) so that the detected faults by previous runs will not be re-targeted again.

CPF and UPF Parser

The following table lists the power data commands relevant to the power-aware DRC and ATPG process.

The other CPF and UPF commands are parsed by the tool but discarded.

Table 13-1. Power Data Commands Directly Related to ATPG and DRC

Power Features	IEEE 1801 / UPF 2.0	CPF 1.0 / CPF 1.1
Power Domains	create_power_domain add_domain_elements create_composite_domain merge_power_domain	create_power_domain update_power_domain
Power Modes (Power States)	add_power_state ¹ add_pst_state create_pst	assert_illegal_domain_configurations create_assertion_control ² create_power_mode update_power_mode

Table 13-1. Power Data Commands Directly Related to ATPG and DRC (cont.)

Power Features	IEEE 1801 / UPF 2.0	CPF 1.0 / CPF 1.1
State Transitions	describe_state_transition ¹	create_mode_transition
Power Network (to derive power-on domains and active state)	add_port_state create_power_switch map_power_switch ¹ set_power_switch create_supply_port create_supply_net connect_supply_net set_domain_supply_net	
Retention Cells	map_retention_cell set_retention set_retention_control set_retention_elements ³	create_state_retention_rule update_state_retention_rule define_state_retention_cell ²
Isolation Cells	map_isolation_cell set_isolation set_isolation_control use_interface_cell ³	create_isolation_rule update_isolation_rule define_isolation_cell ²
Level Shifters	map_level_shifter_cell set_level_shifter use_interface_cell ³	create_level_shifter_rule update_level_shifter_rule define_level_shifter_cell ²
Design Scope	set_design_top set_scope upf_version load_upf load_upf_protected find_objects ⁴ Tcl support	Include get_parameter set_design set_instance set_cpf_version set_time_unit set_power_mode_control_group set_array_naming_style set_macro_model end_macro_model Tcl support

1. add_power_state, describe_state_transition and map_power_switch are not used currently for DRC and ATPG purpose.

2. The CPF commands for defining power cells are used for DRC but are not used by ATPG to identify the power cells that exist in the design. These commands may be needed during test synthesis to automatically insert power cells.
3. The UPF commands set_retention_elements and use_interface_cell are not used to identify the location of power cells. These commands may be needed during test synthesis to automatically insert power cells.
4. Implemented according to the IEEE 1801-2009 (UPF 2.0) standard. The “process” object type is not supported because the design data model does not have that information. The “model” object type is implemented even though it first appears in the IEEE 18012013 (UPF 2.1) standard. The query_* commands are not supported. A note in the IEEE-2009 standard states, “Compliance requirement: A tool compliant to this standard shall support the find_objects command. A tool compliant to this standard may support the query_* commands in this clause. These query_* commands do not make up the power intent of a design; they are only used for querying the design database and are included in this standard to enable portable, user-specified query procedures across tools that are compliant to this standard.”

Power-Aware ATPG Procedure

When using the power-aware ATPG flow, you should separate the test for the logic faults from the test for the low-power features. When testing the logic faults, the traditional fault models (for example, the stuck-at fault model and the transition fault model) are applied and the power switch logic need not to be presented.

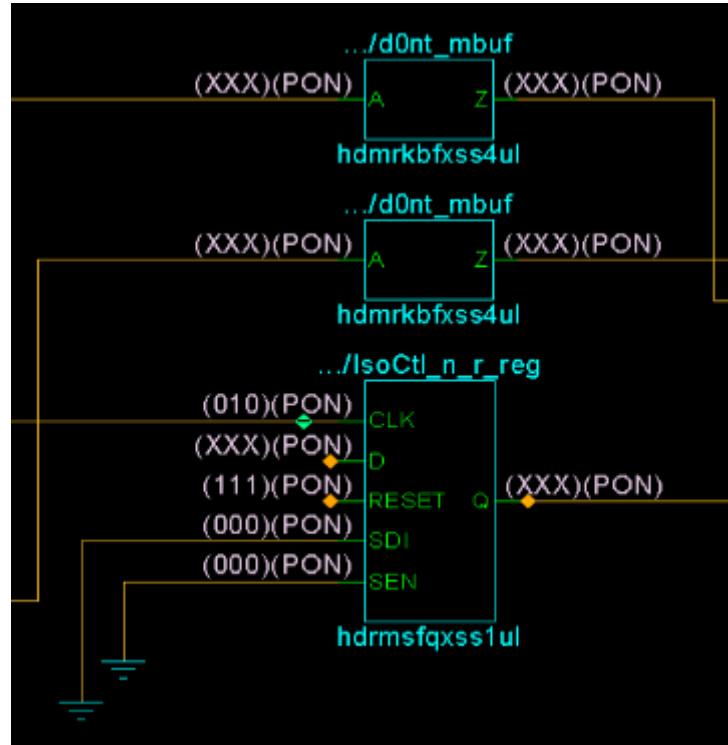
The power data in UPF or CPF format should be applied to the tool so the low-power rules (see “[Power-Aware Rules \(V Rules\)](#)”) can be checked and the circuit can be simulated according to the active power mode.

Power data is loaded into the tool using the read_upf or read_cpf commands. The tool, by default, reports power data using the [report_gates](#) command which shows the power on (PON) and power off (POFF) status of the gates. For example:

```
report_gates /lp_case_si_rst_sms/ati_rst_sync/sync_r/U3/Udff
//  /lp_case_si_rst_sms/ati_rst_sync/sync_r  sync3msfqxss1ul
//    SDI      I(PON)  /vl_sms_lp_case_si_sms_proc_sms_1_stp/
U_lp_case_si_sms_proc_bist/uu5/Z
//    D      I(PON)  /lp_case_si_rst_sms/ati_rst_sync/sync_buf/Z
//    SEN     I(PON)  /se
//    CLK     I(PON)  /lp_case_si_rst_sms/ati_rst_sync/uu1/Z
//    Q      O(PON)  /lp_case_si_rst_sms/ati_rst_sync/uu2/A  /
lp_case_si_rst_sms/and_r/B
//      in power domain PD_P2
```

In DFTVisualizer, the PON will be displayed as shown in [Figure 13-1](#).

Figure 13-1. PON Reporting in DFTVisualizer



Procedure

1. In general, you perform the following steps with the power-aware ATPG flow, either directly from the command line or scripted in a dofile:
2. Invoke the Tesson Shell, set the context the “patterns -scan,” and read in the low-power design. For example:

```
SETUP> read_verilog low-power_design.v
```

3. Add scan chains and other configuration information as appropriate.
4. Load the power data using the `read_cpf` or `read_upf` command. For example:

```
SETUP> read_upf test.upf
```

5. Set the mode to analysis. For example:

```
SETUP> set_system_mode analysis
```

At this point, the tool performs the DRC checks. See “[Power-Aware Rules \(V Rules\)](#)” in the *Tesson Shell Reference Manual*.

6. Add faults using the applicable power-aware switch to the `add_faults` command. For example:

```
SETUP> add_faults -on_domains
```

7. Optionally write the faults to a fault list for multiple power-aware test methodologies—see “[Multiple Power Mode Test Flow](#).” For example:

SETUP> write_faults on_domain_fault_list.txt

8. Create the patterns using the `create_patterns` command.
9. The power-aware ATPG flow supports reporting mechanisms tailored to the power information. See the following commands for more information:
 - [report_faults](#)
 - [report_power_data](#)

Power-Aware Flow Examples

This section contains examples of the power-aware flow.

Example 1

[Table 13-2](#) shows one example design with four power domains and four power modes.

Table 13-2. Example Design With Four Power domains and Power Modes

Power Modes	CPU	Power Domains			test_setup file
		MEM1	CTL	Radio	
active	Active	Active	ON	Tx, Rx	test_setup_active
standby	Idle	Sleep	ON	Rx	test_setup_standby
idle	Sleep	Sleep	ON	Rx	test_setup_idle
sleep	Sleep	Sleep	OFF	Off	N/A
scan chains	chain1 chain2	chain3	chain4	none	

The design contains four scan chains arrayed as follows:

- **Chain1 and Chain 2** — Located in the CPU power domain.
- **Chain3** — Located in MEM1 power domain.
- **Chain4** — Located in the CTL power domain.

To test the design requires multiple `test_setup` procedures, one for each power mode except for the one turning off all power domains. To reduce the maintenance overhead of test procedure files, `test_setup` procedures are written in the following separate files:

- `test_setup_active`

- test_setup_standby
- test_setup_idle

The main test procedure file, which contains the rest of procedures, uses an include statement to include the test_setup for the power mode to be tested. For example, using the following include statement:

```
# include test_setup_active"
```

will test the “active” power mode.

The recommended flow is to test the power mode with all power domains active first, if such power mode exists (power mode “active” in “[Example Design With Four Power domains and Power Modes](#)” on page 588). This allows the tool to view all scan chains and check the “[Power-Aware Rules \(V Rules\)](#)” rules crossing all power domains. Note that when testing different power modes, the scan chains need to be defined accordingly to prevent DRC violations. For example when testing “standby” mode where “MEM1” power domain is off, chain3 should not be defined, otherwise it is a [V8](#) DRC violation.

See “[Power-Aware Rules \(V Rules\)](#)” in the *Tessent Shell Reference Manual* for a complete discussion.

Example 2

In this example, the design contains the following:

- Three power domains {D1, D2, D3}
- Two power mode
 - S1: (D1=ON, D2=ON, D3=OFF)
 - S2: (D1=OFF, D2=ON, D3=ON)

Additionally, assume that the design holds PS1 property (specifically, during the shift cycles and capture cycles), and the design can be kept in the same power mode.

To test the design, you must perform the following *multiple* tool runs:

- **run 1** — ATPG for power mode S1
 - a. Configure the design to power mode S1 by the end of test_setup.
 - b. Add scan chains only in power domain D1 and D2.
 - c. Add faults only in the power ON domains (D1 and D2) using the [add_faults -on_domains](#) command.
 - d. Create patterns and write patterns.

- e. Write faults to a file named *flist_S1.txt* using the write_faults command.
- f. Write isolation faults to a file named *flist_S1_iso.txt* using write_faults -isolation to save for later use.
- g. Write level-shifter faults to a file named *flist_S1_ls.txt* using write_faults -level_shifter to save for later use.
- **run 2 — ATPG for power mode S2**
 - a. Configure the design to power mode S2 by the end of test_setup.
 - b. Add scan chains only in power domain D2 and D3.
 - c. Add faults only in the power ON domains (D2 and D3) using **add_faults -on_domains** command.
 - d. Load the previously-saved S1 fault list using the following command:

ANALYSIS> read_faults flist_S1.txt -merge -Power_check on

This updates the fault status of faults in D2 but discard faults in D1 as they are not power on faults. See the **read_faults** command for more information.

- e. Create patterns and write patterns.
- f. Write faults to a file named *flist_S2.txt* using the write_faults command.
- g. Write isolation faults to a file named *flist_S2_iso.txt* using write_faults -isolation to save for later use.
- h. Write level-shifter faults to a file named *flist_S2_ls.txt* using write_faults -level_shifter to save for later use.

After run 2, you can calculate the entire fault coverage by loading multiple fault lists into the tool. You do this by using the **read_faults -Power_check OFF** command and switch as shown in the following steps:

- **run 3 — Overall test coverage report**
 - a. Load the previously-saved fault list for D1:
ANALYSIS> read_faults flist_S1.txt -merge -Power_check off
 - b. Load the previously-saved fault list for D2:
ANALYSIS> read_faults flist_S2.txt -merge -Power_check off
 - c. Issue the **report_statistics** command to report the overall test coverage.
 - d. Write the faults to a file named *faults flist_all.txt* using the write_faults command.

The final tool run reports isolation fault test coverage ATPG and level-shifter fault test coverage and saves these in separate fault lists.

1. Report the isolation fault test coverage:

ANALYSIS> report_statistics -isolation

2. Report the level-shifter fault test coverage:

ANALYSIS> report_statistics -level_shifter

3. Write all isolation faults to a file:

ANALYSIS> write_faults flist_all_iso.txt -isolation

4. Write all level-shifter faults to a file:

ANALYSIS> write_faults flist_all_ls.txt -level_shifter

Chapter 14

Low-Power Design Test

You can use the low-power scan insertion and ATPG flow for use with Tesson Scan and the ATPG tool.

Low-Power Testing Overview	594
Low-Power Assumptions and Limitations	594
Low-Power CPF/UPF Parameters	594
Test Insertion	596
Low-Power Test Flow	596
Scan Insertion with Tesson Scan	598
Power-Aware Design Rule Checks	603
Low-Power DRCs	603
Low-Power DRC Troubleshooting	603
Power State-Aware ATPG	606
Power Domain Testing	606
Low-Power Cell Testing	608

Low-Power Testing Overview

Tessent Scan supports operations that enable low-power testing.

- Insertion of dedicated wrapper cells in the presence of isolation cells.
- Assigning dedicated wrapper cells to power domains based on the logic it is driving and the priority of the power domains.

The low-power design flow includes the following steps:

1. Specify low-power data specifications in the CPF/UPF file. See “[Low-Power CPF/UPF Parameters](#)”.
2. Insert scan cells and EDT logic in the design. See “[Test Insertion](#)”.
3. Validate low-power data, scan, and EDT logic. See “[Power-Aware Design Rule Checks](#)”.
4. Generate power domain-aware test patterns. See “[Power State-Aware ATPG](#)”.
5. Test low-power design components.

Low-Power Assumptions and Limitations	594
Low-Power CPF/UPF Parameters	594

Low-Power Assumptions and Limitations

The power-aware functionality comes with several assumptions and limitations.

- Tessent Scan does not write or update CPF/UPF files. Tessent Scan assumes that everything that belongs to a power domain is explicitly listed in the UPF/CPF file. Because of this assumption, you must make the following changes to the UPF/CPF file after the scan insertion step and before ATPG:
 - Add any inserted input wrapper cells to the correct power domain in the CPF/UPF file.
 - Explicitly define all isolation cells and their control signals, and level shifters in the CPF/UPF file. Tessent Scan checks for their presence but does not add them. For more information, see “[Scan Insertion with Tessent Scan](#).”

Low-Power CPF/UPF Parameters

Tessent Scan uses CPF/UPF files to identify the power domains in the design and to constrain scan insertion within the power domain boundaries.

Tessent Scan supports power structure and configuration data from the following formats:

- CPF 1.0 and 1.1
- UPF 2.0 (IEEE1801)

If you are a member of Si2, you can access complete information about CPF standards at the following URL:

https://www.si2.org/openeda.si2.org/project/showfiles.php?group_id=51

You can purchase complete information on UPF standards at the following URL:

http://www.techstreet.com/standards/ieee/1801_2009?product_id=1744966

In the CPF/UPF file, you must specify power domains and information related to the power domains, power states (modes) of the design, and isolation cells and control signals. You can use the commands listed in [Table 14-1](#) to specify this information.

Table 14-1. Power Data Commands Directly Related to ATPG

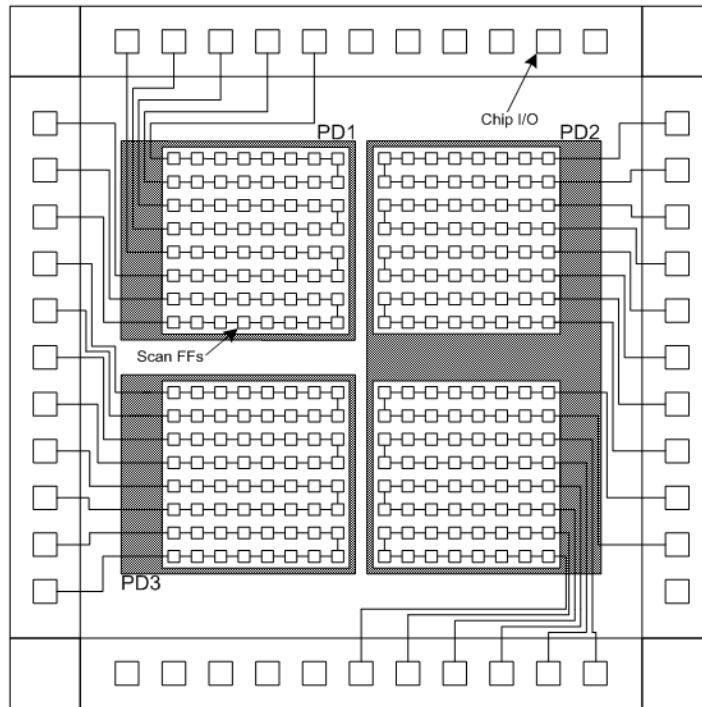
Power Features	IEEE 1801 / UPF 2.0	CPF 1.0 / CPF 1.1
Power Domains	create_power_domain add_domain_elements create_composite_domain merge_power_domain	create_power_domain update_power_domain
Power Modes (Power States)	add_power_state add_pst_state create_pst	assert_illegal_domain_configurations create_assertion_control create_power_mode update_power_mode
Isolation Cells	set_isolation_command set_isolation_control map_isolation_cell	create_isolation_rule define_isolation_cell update_isolation_cell
Level Shifters	map_level_shifter_cell	create_level_shifter_rule update_level_shifter_rule

Test Insertion

Test insertion includes scan substitution and stitching. You can insert scan chains on a power domain basis. Alternately, you can group test logic based on power states.

Figure 14-1 shows an example of chains inserted on a power domain basis. Note, no scan chains span the power domains because the tool does not insert scan across power domains.

Figure 14-1. Scan Chain Insertion by Power Domain



Low-Power Test Flow	596
Scan Insertion with Tessent Scan.....	598

Low-Power Test Flow

To begin the low power test flow, load the low-power data and then use the loaded power domain information to guide the process of scan partitioning and scan insertion.

Procedure

1. Load the design data and libraries.
2. Read the CPF or UPF file:
read_cpf filename
read_upf filename
3. Set up appropriate design constraints.

4. Enter analysis mode:

SETUP> set_system_mode analysis

5. If wrapper cells are enabled, scan chain partitioning will be affected. Refer to “[Managing Wrapper Cells with Power Domains](#)” for more information.
6. Specify scan chains based on maximum scan chain length or scan chain number using the `set_power_domain` command:

**set_power_domain {{Power_domain_name... | -All [-EDT]}
[-Number_of_chains integer | -Max_length integer]}**

For example:

> set_power_domain PD_interleaver -number 65

> set_power_domain PD_mem_ctrl -max_length 6

Any scan flip-flop not specified by the `set_power_domain` command is added to the default top-level power domain.

7. Insert the test structures into the design netlist using the [insert_test_logic](#) command.
8. Report the design statistics using the [report_statistics](#) command.
9. Write out the netlist and ATPG setup.

Scan Insertion with Tessent Scan

You can use Tessent Scan to do both scan cell substitution and scan stitching. By loading the low-power CPF or UPF file for the design before beginning the scan insertion and stitching process, the power domain information can be used to guide the scan partitioning process. Tessent Scan does not insert scan across power domains.

You can explicitly specify the number and length of scan chains in each power domain of the design using the `set_power_domain` command. If you do not use `set_power_domain`, Tessent Scan adopts the default settings based on the following command/options settings:

- **insert_test_logic** — arguments `-MAX_length` and `-Number`
- **add_scan_partition** — arguments `-NUmber` and `-MAX_length`
- **set_wrapper_chains** — arguments `-INPUT_NUMber`, `-INPUT_MAX_length`, `-OUTPUT_NUMber`, and `-OUTPUT_MAX_length`

Note

 Because Tessent Scan does not function with a concept of power mode or power state, chain balancing does not occur during low-power scan insertion.

During scan insertion, the tool does not generate a design that is clean in terms of power domains. Tessent Scan creates a design that is functionally correct in the context of power domains but it does not insert isolation cells or level shifters when routing global signals across power domains. You will need to add these objects themselves using a synthesis or physical implementation tool and then manually add them to the CPF/UPF file.

For more information on isolation cells and level shifters, see “[Low-Power Cell Testing](#)”.

Managing Scan Enable Signals for Multiple Power Domains	598
Managing Wrapper Cells with Power Domains	599
Managing Test Point Insertion with Power Domains	601

Managing Scan Enable Signals for Multiple Power Domains

The low-power test flow allows you to insert multiple scan enable signals in your design. This means you can insert a scan enable signal for each power domain in the design; you can also insert multiple scan enable signals for a single power domain. The behavior of multiple scan enable signals is the same regardless of whether your design has no power domains defined or has multiple power domains defined.

You assign one scan enable signal to a power domain by using the `set_scan_enable` command and specifying the power domain with the *partition_name* argument as shown in the example here:

```
set_scan_enable scan_en_pin_path pd1 pd2
```

For more information on assigning a scan enable signal to a power domain, refer to the [set_scan_enable](#) description in the *Tessent Shell Reference Manual*.

To assign more than one scan enable signal to a single power domain, you use the `set_scan_enable_sharing` command and specify the `-scan_partition` switch which assigns a unique scan enable signal to each of the specified power domains. By default, the types of the added scan enable signals are different for the core chains, input wrapper chains, and output wrapper chains.

```
set_scan_enable_sharing -prefix base_name -scan_partition
```

Managing Wrapper Cells with Power Domains

Newly-inserted dedicated wrapper cells are stitched into wrapper chains based on the power domain affiliation of the wrapper cells (dedicated and shared) within the wrapper chains. The tool identifies the power domain that a dedicated wrapper cell is affiliated with during wrapper cell identification; the power domain is chosen based on which power domain the PI or PO being registered is connected to. If a PI is connected to more than one power domain, the dedicated wrapper cell is associated with the first encountered power domain.

[Figure 14-2](#) shows the stitching of the input wrapper cells for each power domain. The newly added dedicated wrapper cells are stitched with the shared wrapper cells in the same power domain.

Figure 14-2. Input Wrapper Cells and Power Domains

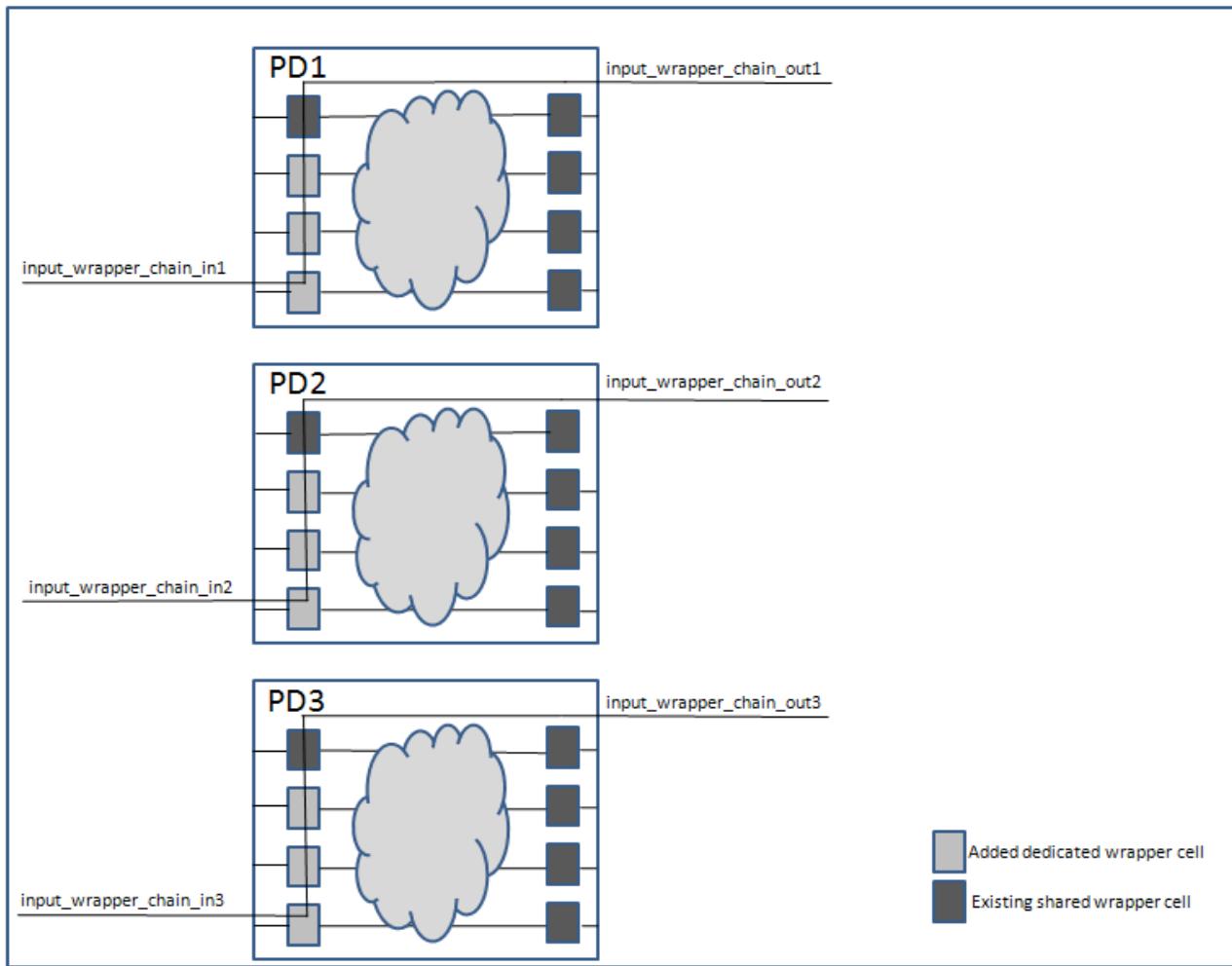
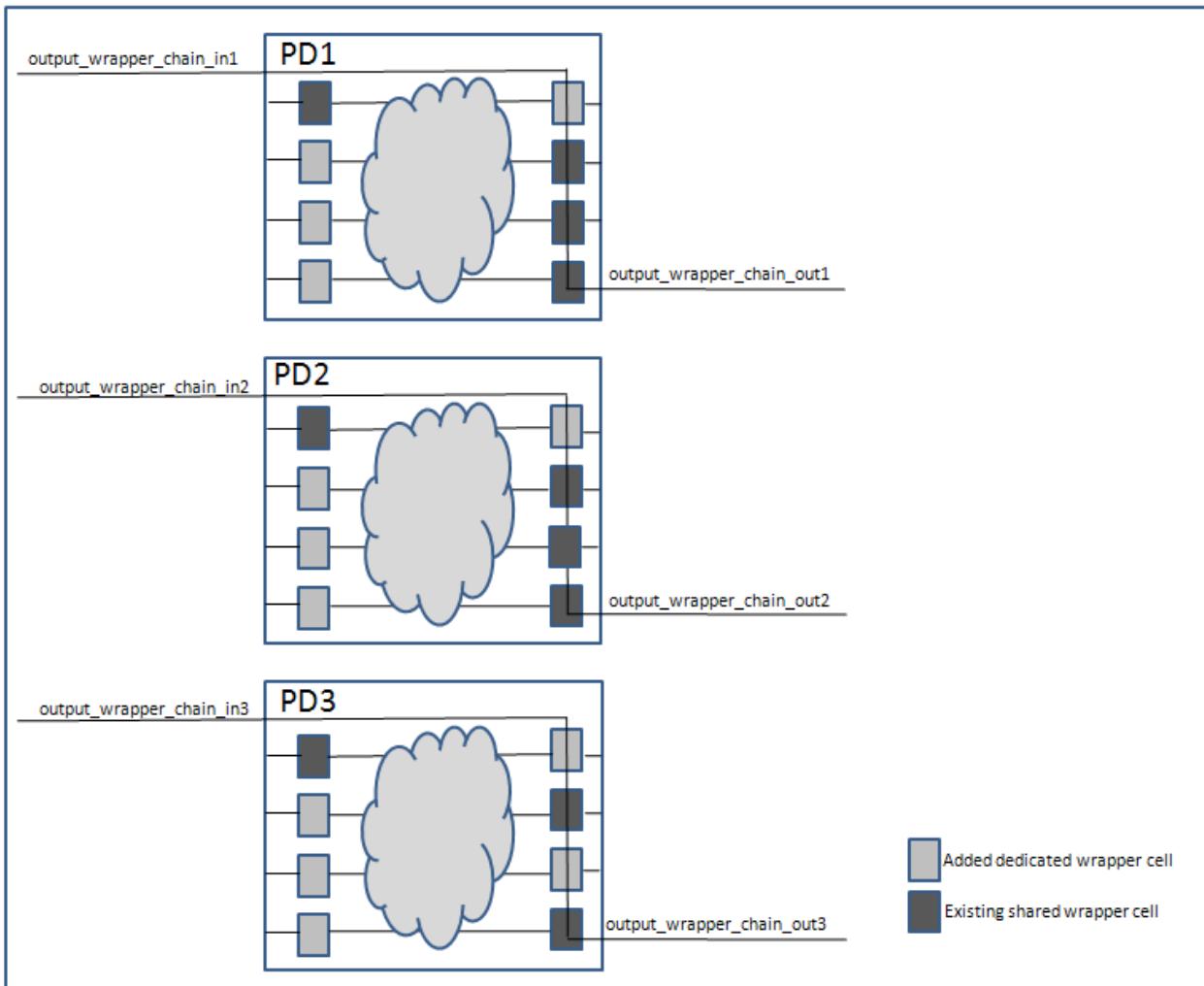


Figure 14-2 shows the stitching of output wrapper cells for each power domain. The newly added dedicated wrapper cells are stitched with the shared wrapper cells in the same power domain.

Figure 14-3. Output Wrapper Cells and Power Domains



Managing Test Point Insertion with Power Domains

When a CPF or UPF file is read into the tool, the tool marks locations near the power isolation cells and level shifters with `no.observe_point` and `no.control_point` attributes.

At the output side of a power isolation cell or level shifter, the following pins are marked with the `no.observe_point` attribute:

- The output pin of the power isolation cell or level shifter.
- The input pins of gates that are driven by the power isolation cell or level shifter.

At the input side a power isolation cell or level shifter the following pins are marked with the `no.control_point` attribute:

- The input pins of the power isolation cell or level shifter.

- The output pin of gates that drive the power isolation cell or level shifter.

The test point analysis algorithm takes this into account so that no automatically generated test points are placed at these locations. In addition, the commands to manually add test points at these locations also guard against insertion of the specified test point type (control or observe) at that location, and a warning is generated that the test point will not be inserted on that particular side of the power isolation cell or level shifter. This verification of the test points is done after the command “insert_test_logic” is issued.

```
// Warning: Control point at "u1/SP[6]" will not be inserted.  
// This location is at the input side of a power isolation cell  
// or level shifter.  
// You can insert a control point at the output side of the power  
// isolation cell/level shifter.  
  
// Warning: Observe point at "U83/Z" will not be inserted.  
// This location is at the output side of a power isolation  
// cell "U83".  
// You can insert an observe point at the input side of the power  
// isolation cell "U83".
```

Power-Aware Design Rule Checks

Tessent Scan performs power-aware design rule checking to validate power data and power-aware reporting to facilitate troubleshooting design rule violations.

Low-Power DRCs	603
Low-Power DRC Troubleshooting.....	603

Low-Power DRCs

In addition to checking design logic and the cell library, power-aware design rules (DRCs V1-V21) check the design for violations and correctness of power data provided through CPF/UPF files.

Note

 DFTVisualizer does not have any direct support for power domain-related DRC failures. However, you can use standard design tracing features in DFTVisualizer to pinpoint issues.

- DRCs V1 to V7 — Checked after reading a power data file.
- DRCs V8 to V21 — Checked when you switch from setup mode to a non-setup mode.

For information on specific V DRCs, see “[Power-Aware Rules \(V Rules\)](#)” in the *Tessent Shell Reference Manual*.

Low-Power DRC Troubleshooting

You can use the `report_scan_partitions` command to generate a scan cell report that includes power domain information as shown in the following example. Tessent Scan treats power domains as scan partitions during scan chain stitching.

The `report_scan_partitions` command extracts all updated information from the CPF/UPF files and reports each power domain (scan partition) and the pathnames of all of the sequential instances in that power domain; newly added dedicated wrapper cells and lockup cells are indicated by the string “(new cell)” following their name.

DFT> `report_scan_partitions -all -expand`

```
-----  
ScanPartitionNam    Members  
-----  
PD2                I6  
                    I10/lckup3 (new cell)  
                    I10/lckup2 (new cell)  
                    I10/lckup1 (new cell)  
                    I10/outreg2 (new cell)  
                    I10/outreg1 (new cell)  
                    I10/sf_0001_1  
                    I10/sf_0002_1  
                    I10/sf_0003_1  
PD3                I2/I2/lckup3 (new cell)  
                    I2/I2/lckup2 (new cell)  
                    I2/I2/lckup1 (new cell)  
                    I2/I2/outreg1 (new cell)  
                    I2/I2/sf_0001_1  
                    I2/I2/sf_0002_1  
                    I2/I2/sf_0003_1  
                    I2/I2/I1_2  
DEF                lckup2 (new cell)  
                    lckup1 (new cell)  
                    inreg2 (new cell)  
                    inreg1 (new cell)  
                    outreg1 (new cell)  
-----
```

Power Domains for Scan Cells Reporting

You can use the `report_scan_cells` command to generate a scan cell report that includes power domain information as shown here:

```
ANALYSIS> report_scan_cells -power_domain
```

For more information, refer to the [report_scan_cells](#) description in the *Tessent Shell Reference Manual*.

Power Domains for Gates Reporting

You can use the `set_gate_report` command to add power domain information to the results of the `report_gates` report as shown in the example here:

```
ANALYSIS> set_gate_report -power on
```

```
ANALYSIS> report_gates 154421
```

```
//  /mc0/present_state_reg_3_/inst1122_4/mlc_dff (154421)  DFF  
//    "S"      I(ON)  153682-  
//    "R"      I(ON)  153681-  
//    "CO"     I(ON)  26078-  
//    "DO"     I(ON)  43013-  
//    "OUT"    O(ON)  1115-  
//    MASTER   cell_id=0  chain=chain66  group=grp1  invert_data=FFFF  
//    in power domain PD_mem_ctrl
```

For more information, refer to the [set_gate_report](#) description in the *Tessent Shell Reference Manual*.

Power Domain Violation Tracing

You can use DFTVisualizer design tracing capabilities to pinpoint failures and issues reported during design rule checking.

Power State-Aware ATPG

The recommended method for ATPG is to have all power domains on if possible. With this method, you can manage power considerations due to switching activity during test by sequencing the test of design blocks and using low-power ATPG. This is the simplest approach and maximizes fault coverage. If you cannot use this method due to static power limitations, you should select a minimum set of power states to achieve the required fault coverage for all the blocks in the design as well as inter-block connectivity.

Power Domain Testing	606
Low-Power Cell Testing	608

Power Domain Testing

You can run ATPG in multiple power modes and power states by setting the appropriate power mode/state conditions as specified in the CPF or UPF.

To demonstrate, assume a design has the following two power modes:

- PM1 — All domains on
- PM2 — *PD_mem_ctrl* powered down when *mc_pwr* is “0”

By setting a pin constraint on *mc_pwr* as shown here, ATPG recognizes which power mode is currently set and runs appropriately:

```
SETUP> add_input_constraints mc_pwr -c1
SETUP> set_system_mode analysis
...
// -----
// Begin circuit learning analyses.
// -----
...
SETUP> create_patterns

No faults in fault list. Adding all faults...
...
// Enabling power-aware ATPG: initial power mode in capture cycle = PM2
...
```

If the power mode can change during the capture cycle, the tool will issue V12 violations. When V12 handling is a warning (default), the tool automatically adds ATPG constraints to fix the power mode during the capture cycles:

```
...
// 1 ATPG constraint is added to fix power mode during capture.
// The active power mode at the beginning of capture cycle is PM2.
// Power-aware ATPG is enabled.
...
```

When power-aware ATPG is enabled, the tool performs ATPG and fault simulation according to the power mode: the regular gates in a power-off domain will be X and the corresponding faults are AU. The fault grouping will classify the fault as AU due to power off if faults in a power domain are added:

ANALYSIS> create_patterns

ANALYSIS> report_statistics

Statistics Report	
Stuck-at Faults	
Fault Classes	#faults (total)
FU (full)	354
DS (det_simulation)	52 (14.69%)
DI (det_implementation)	40 (11.30%)
UU (unused)	26 (7.34%)
RE (redundant)	134 (37.85%)
AU (atpg_untestable)	102 (28.81%)
Fault Sub-classes	
AU (atpg_untestable)	
P OFF (power_off)	102 (28.81%)
*Use "report_statistics -detailed_analysis" for details.	
Coverage	
test_coverage	47.42%
fault_coverage	25.99%
atpg_effectiveness	100.00%
-	
#test_patterns	0
#simulated_patterns	0
CPU_time (secs)	1.9

Instead of adding all faults, you can choose to add, delete, report, and write faults on power-on domains or any user-specified power domains:

```
add_/delete_/report_/write_faults
[[-ON_domains] | [-OFF_domains] |
 [ -POWer_domains {domain_name ...} ] ]
[ -ISolation_cells] [ -LLevel_shifters ] [ -REtention_cells ]
```

Low-Power Cell Testing

Level shifters and isolation cells require special handling during scan insertion.

The handling of these low power cells is shown in these sections:

Level Shifters	608
Isolation Cells	608

Level Shifters

In most cases, you can handle level shifters as standard buffers and do not need any special handling to achieve full test coverage. However, if one or more power domains in the design can operate at more than one supply voltage, you should run ATPG for all permutations of the supply voltage for both input and output sides of any given level shifter to ensure full coverage.

Isolation Cells

Isolation cells are special low power cells used to separate power on and power off domains. They require special handling during scan insertion as described in this section.

For a full description of isolation cells, refer to IEEE 1801:*Design and Verification of Low Power Integrated Circuits*.

You can test isolation cells in one of two modes:

- **Normal transmission mode** — In this mode, when an isolation cell is disabled and stuck-at “0” or “1”, the fault can be detected on the input and output of the cell. A stuck-at fault (isolation on) for the isolation enable pin is also detected because this failure forces the cell into isolation and prevents the transmission of data.
- **Isolation mode** — In this mode, input side stuck-at faults are not detectable if the isolation cell is a clamp-style cell (output held to “0” or “1” when isolation on). If the isolation cell is a latch-style cell, input side stuck-at faults are detectable by latching the value of the driving domain before enabling isolation.

Chapter 15

Using MTFI Files

This chapter describes MTFI (Mentor Tessent Fault Information) features, which are available for use in the ATPG tool and Tessent LogicBIST. MTFI is a common and extendable file format for storing fault status information.

This chapter describes MTFI syntax, as well as the major MTFI features:

MTFI File Format	610
MTFI Features	613
Support of Fault Classes and Sub-Classes	613
Support of Stuck and Transition Fault Information	615
Support of N-Detect Values	615
Support of Different Fault Types in the Same File	617
Support for Hierarchical Fault Accounting	617
Commands that Support MTFI	619

MTFI File Format

MTFI is the default (but optional) file format for reading and writing fault information in the dft -edt and patterns -scan contexts.

MTFI is an ASCII file format for storing fault information about an instance that the ATPG tool can read and write.

Format

The following simple MTFI example shows the minimum required keywords in bold:

```
FaultInformation {
:  version : 1;
  FaultType (Stuck) {
    FaultList {
      Format : Identifier, Class, Location;
      Instance ("/i1") {
        0, DS, "F1";
        1, DS, "F2";
      }
    }
  }
}
```

Parameters

The basic syntax elements of MTFI are as follows:

- Comments
The “//” identifies the start of the comment until the end of line.
- FaultInformation { ... }
Keyword that specifies the top-level block and file type.
- version : *integer*
Keyword that specifies the syntax version of the MTFI file.
- FaultType (*type*) { ... }
Keyword that specifies the fault type of the data: Stuck, Iddq, Toggle, Transition, Path_delay, Bridge, and Udfm. The keywords are identical to those available for the [set_fault_type](#) command described in the *Tessent Shell Reference Manual*.
- FaultList { ... }
Keyword that defines a data block that stores per-fault data.
- UnlistedFaultsData { ... }
Keyword that defines a data block that stores the coverage information for specific graybox instances to allow hierarchical fault accounting. For more information, refer to “[Support for Hierarchical Fault Accounting](#)” on page 617.

- FaultCollapsing { true | false }

Keyword that specifies the fault collapsing status in the fault list. A value of “true” means that the faults in the list are collapsed; a value of “false” means that the faults in the list are uncollapsed. You can use this statement only inside the FaultList data block.

- DetectionLimit : *integer*

Keyword that specifies the detection drop limit for a DS fault.

- Format

Keyword that specifies the sequence of values stored in a specific data block (FaultList or UnlistedFaultsData). The following keywords are available for use in the Format statement:

- Class

Required keyword that specifies the fault category, and optionally the sub category (for example, DS, UC, AU.BB). For more information, refer to “[Support of Fault Classes and Sub-Classes](#)” on page 613.

- Location

Keyword that specifies the pin pathname. You can use this keyword only in the FaultList data block. In the case of a stuck-at fault, the keyword is required.

- Identifier

Required keyword that specifies the fault identifier. In the case of stuck-at or transition faults, the value can be 0 or 1. No other fault types are supported.

- Detections

Optional keyword that specifies the number of detections for the fault types stuck-at and transition. For more information, refer to “[Support of N-Detect Values](#)” on page 615.

- CollapsedFaultsCount

Required keyword that specifies the number of collapsed faults. You can use this keyword only in the UnlistedFaultsData data block.

- UncollapsedFaultsCount

Required keyword that specifies the number of uncollapsed faults. You can use this keyword only in the UnlistedFaultsData data block.

- Instance (“*pathname*“) { ... }

Required keyword that specifies the instance pathname to all of the pins in the data block. You must enclose the pathname in parentheses and double quotes. The pathname can be an empty string.

Examples

The following is an example of a typical MTFI file:

```
//  
// Tessent FastScan v9.6  
//  
// Design = test.v  
// Created = Tue Dec 20 20:08:46 2011  
//  
// Statistics:  
//   Test Coverage = 50.00%  
//   Total Faults = 6  
//     UC (uncontrolled) = 2  
//     DS (det_simulation) = 1  
//     DI (det_implementation) = 1  
//     AU (atpg_untestable) = 2  
//  
FaultInformation {  
    version : 1;  
    FaultType ( Stuck ){  
        FaultList {  
            FaultCollapsing : false;  
            Format : Identifier, Class, Location;  
            Instance ( "" ) {  
                1, DS, "/i1/IN0";  
                1, AU, "/i1/IN1";  
                1, EQ, "/i2/Y";  
                0, UC, "/i5/Z";  
                1, DI, "/i5/Z";  
                0, AU, "/i4/IN1";  
                1, UC, "/i4/IN1";  
            }  
        }  
    }  
}
```

MTFI Features

The following text explains the major features of the MTFI format.

Support of Fault Classes and Sub-Classes.....	613
Support of Stuck and Transition Fault Information	615
Support of N-Detect Values	615
Support of Different Fault Types in the Same File.....	617
Support for Hierarchical Fault Accounting	617

Support of Fault Classes and Sub-Classes

MTFI provide features that allow you to specify fault classes and sub-classes.

The following example shows how MTFI supports fault classes and sub-classes using the Format statement's Class keyword:

```
FaultInformation {
    version : 1;
    FaultType ( Stuck ){
        FaultList {
            FaultCollapsing : false;
            Format : Identifier, Class, Location;
            Instance ( "" ) {
                1, DS, "/i1/IN0";
                1, AU, "/i1/IN1";
                1, EQ, "/i2/Y";
                0, UC, "/i5/Z";
                1, DI, "/i5/Z";
                0, AU, "/i4/IN1";
                1, UC, "/i4/IN1";
            }
        }
    }
}
```

In the following example, one AU fault is identified as being AU due to black box, and one DI fault is specified as being due to LBIST. So for these two faults, the MTFI file reports both class and sub-class values.

```

FaultInformation {
    version : 1;
    FaultType ( Stuck ) {
        FaultList {
            FaultCollapsing : false;
            Format : Identifier, Class, Location;
            Instance ( "" ) {
                1, DS, "/i1/IN0";
                1, AU, "/i1/IN1";
                1, EQ, "/i2/Y";
                0, UC, "/i5/Z";
                1, DI, "/i5/Z";
                1, DI.LBIST, "/i5/Z";
                0, AU.BB, "/i4/IN1";
                1, UC, "/i4/IN1";
            }
        }
    }
}

```

In the preceding example, the sub-class information is part of the class value and separated by the dot. Any of the AU, UC, UO and DI fault classes can be further divided into different sub-classes. For all the available AU fault sub-classes, refer to the [set_relevant_coverage](#) command description in the *Tessent Shell Reference Manual*. The tool supports the following UC and UO fault sub-classes: ATPG_Abort (AAB), Unsuccessful (UNS), EDT Abort (EAB). The tool supports the pre-defined fault sub-class EDT for DI faults.

User-Defined Fault Sub-Classes

Besides the predefined fault sub-classes described previously, the tool also supports a user-defined sub-class for AU and DI faults. The user-defined sub-class can be any string based on the following character set:

- A-Z
- a-z
- 0-9
- -
- _

It is your responsibility to ensure that the name of the user-defined sub-class differs from any of the predefined sub-classes. Otherwise, the faults may be accounted for incorrectly.

Note the statistics report displays the breakdown of DI faults when you use the “`report_statistics -detailed_report DI`” command and when there are some DI faults in specific subcategories.

Support of Stuck and Transition Fault Information

MTFI is able to represent stuck fault information in a fashion similar to the classic format.

This is shown in the following example:

```
FaultInformation {
    version : 1;
    FaultType ( Stuck ) {
        FaultList {
            FaultCollapsing : false;
            Format : Identifier, Class, Location;
            Instance ( "" ) {
                1, UC, "/in1";
                0, DS, "/in1";
                0, DS, "/i1/IN0";
                1, DS, "/i1/OUT";
                1, DS, "/i1/IN0";
                1, AU, "/i1/IN1";
                1, EQ, "/i2/Y";
                0, UC, "/i5/Z";
                1, DS, "/out";
                1, DI, "/i5/Z";
            }
        }
    }
}
```

Support of N-Detect Values

The information about how often a fault has been detected can be handled for a single fault in the FaultList block, as well as for a group of faults in the UnlistedFaultsData block.

In the FaultList block, you can add the value to the list using the Format statement's Detections keyword. The value must be a positive integer that specifies the number of detections of this fault. All classes other than DS must have a value of 0.

In the following example, the detection drop limit is 4. For any DS faults that are detected four times or more, the tool reports the detection value as “4”.

```
FaultInformation {
    version : 1;
    FaultType ( Stuck ) {
        FaultList {
            FaultCollapsing : false;
            DetectionLimit : 4;
            Format : Identifier, Class, Detections, Location;
            Instance ( "" ) {
                1, DS, 3, "/i1/OUT";
                1, DS, 4, "/i1/IN0";
                1, AU.BB, 0, "/i1/IN1";
                1, EQ, 0, "/i2/Y";
                0, UC, 0, "/i5/Z";
                1, DS, 2, "/out";
                1, DI, 0, "/i5/Z";
            }
        }
    }
}
```

In general, the same rules are valid for the UnlistedFaultsData block, as shown in the following example:

```
FaultInformation {
    version : 1;
    FaultType ( Stuck ) {
        UnlistedFaultsData {
            DetectionLimit : 4;
            Format : Class,Detections,CollapsedFaultCount,
                      UncollapsedFaultCount;
            Instance ( "/CoreD/i1" ) {
                UC,      0, 1252, 2079;
                UC.EAB, 0, 452, 543;
                DS,     1, 69873, 87232;
                DS,     2, 12873, 21432;
                DS,     3, 9752, 11974;
                DS,     4, 487, 6293;
                AU.BB, 0, 8708, 10046;
                AU,     0, 2374, 3782;
            }
        }
    }
}
```

In the case of the UnlistedFaultsData, the preceding example shows the fault count after the number of detections. Typically, there is one line per detection until reaching the current detection limit. So in the preceding example, the number of faults have reached the detection limit of 4.

Also note that in the previous example, the line UC, 0, 1252, 2079 shows the collapsed and uncollapsed fault count for the UC faults that are in the unclassified sub-class (that is, those that do not fall into any of the predefined or user-defined sub-classes).

Similarly, the line AU, 0, 2374, 3782 shows the fault counts for the unclassified AU faults.

While loading the MTFI file, the n-detection number stored in the file is capped at the detection limit currently set in the tool. This applies to both the detection number in FaultList data block and that in UnlistedFaultsData block. Depending on the switch, the n-detection data in the external MTFI file can be appended to the internal detection number of the corresponding fault, or replace the detection number of the corresponding fault in the internal fault list.

Support of Different Fault Types in the Same File

MTFI enables you to manually specify multiple fault types in a single file.

When reading an MTFI file using the `read_faults` command, the current fault type (set using `set_fault_type`) must match the specified fault type within the MTFI file. In order to share fault information between the fault types Stuck and Transition, MTFI enables you to manually specify both fault types in the same file, as shown in the following example:

```
FaultInformation {
    version : 1;
    FaultType ( Stuck, Transition ) {
        FaultList {
            FaultCollapsing : false;
            Format : Identifier, Class, Location;
            Instance ( "" ) {
                1, AU.PC, "/i1/OUT";
                1, AU.PC, "/i1/IN0";
                1, AU.BB, "/i1/IN1";
                1, AU.BB, "/i2/Y";
                0, AU.TC, "/i5/Z";
                1, AU.TC, "/out";
                1, AU.TC, "/i5/Z";
            }
        }
    }
}
```

MTFI does not support combinations other than Stuck and Transition. And note that none of the tools that support MTFI can output MTFI files that contain multiple fault types.

Support for Hierarchical Fault Accounting

MTFI enables you to efficiently handle fault information for hierarchical designs, where individual small cores are tested and their fault statistics saved in individual MTFI files. You can then instantiate those small cores in a larger core using “graybox” versions of the small cores plus the fault information from their MTFI files. Using this method, the test patterns for the larger core need not deal with the internal details of the small cores, only with their graybox versions (which contain only the subset of the logic needed for testing at the next higher level).

Consider the example in [Figure 15-1](#). Core_A and Core_B are stand-alone designs with their own individual test patterns and fault lists. After running ATPG on Core_A and Core_B, you store fault information in MTFI files using the following commands:

```
ANALYSIS> write_faults Core_A.mtfi  
ANALYSIS> write_faults Core_B.mtfi
```

Note that the logic in Core_A and Core_B is fully observable and controllable, so there are no unlisted faults in the Core_A.mtfi or Core_B.mtfi files.

If there are multiple fault lists for a given core, each reflecting the coverage achieved by a different test mode, and the intent is to merge these results into a single coverage for the core, you must perform the merge during a core-level ATPG run. This is because when you use the “[read_faults -graybox](#)” command at the next level of hierarchy, the command supports only a single fault list per core. Note that you can only merge fault lists of a single fault type. You cannot, for example, merge fault lists for stuck and transition at the core level.

Taking the example of Core_A previously described, you would first merge the fault lists of any other previously-run test modes before writing out the final fault list for the core. The following sequence of commands is an example of this:

```
ANALYSIS> read_faults Core_A_mode1.mtfi -merge  
ANALYSIS> read_faults Core_A_mode2.mtfi -merge  
ANALYSIS> write_faults Core_A.mtfi
```

Next you create Core_C by instantiating graybox versions of Core_A and Core_B, and then you use the fault information you wrote previously using the following commands:

```
ANALYSIS> read_faults Core_A.mtfi-Instance Core_C/Core_A -Graybox  
ANALYSIS> read_faults Core_B.mtfi -Instance Core_C/Core_B -Graybox
```

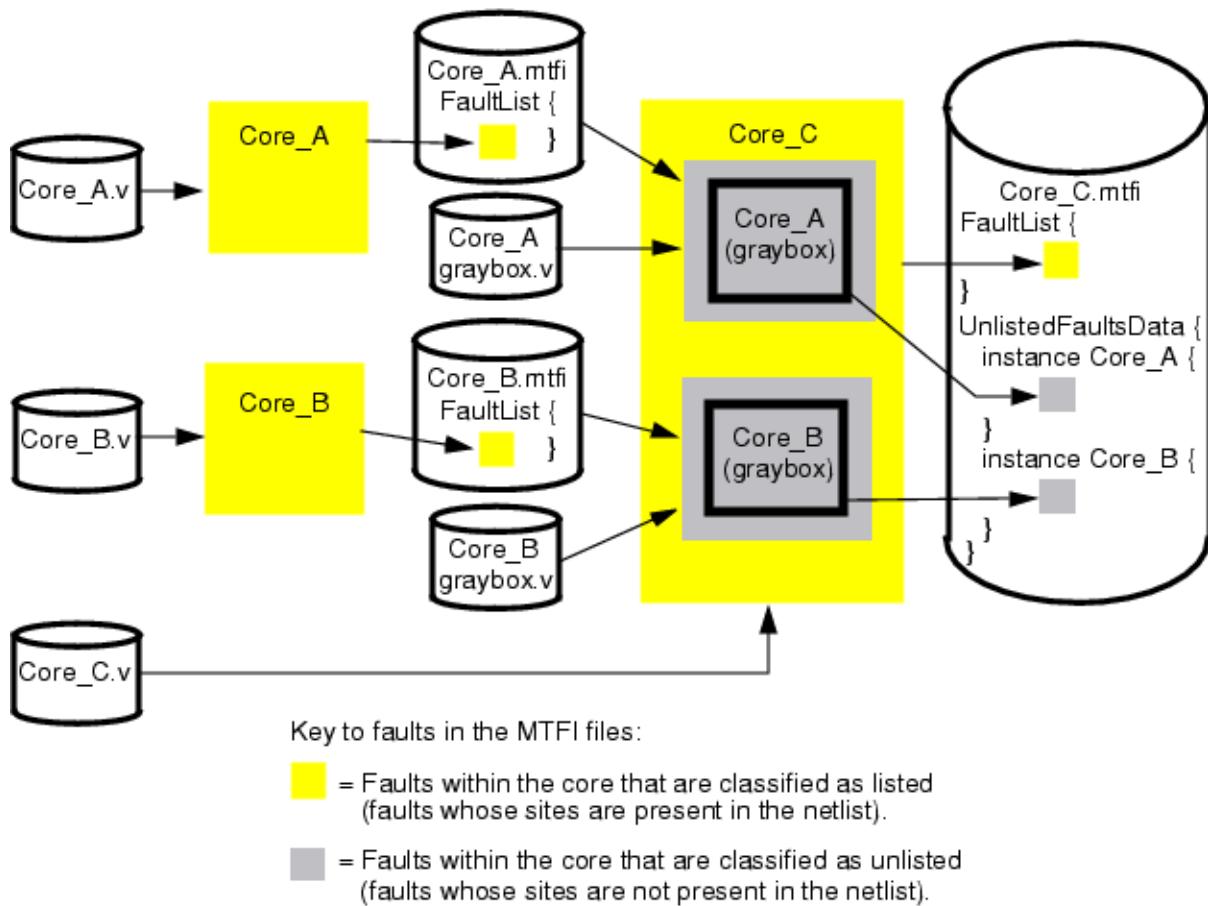
The -Graybox switch directs the tool to map any faults it can to existing nets in Core_C and to essentially discard the remaining faults by classifying them as unlisted. The tool retains the fault statistics for the unlisted faults.

After you have run ATPG on Core_C, you can write out the fault information to an MTFI file:

```
ANALYSIS> write_faults Core_C.mtfi
```

The command writes out data for both the listed and unlisted faults that were created when you issued the `read_faults` command. You could then instantiate a graybox version of Core_C in a larger core, Core_D, and load the fault information from the Core_C.mtfi file. That way, Core_A, Core_B, and Core_C will have already been fully tested and would not be tested again, but the fault statistics from all three cores would be available for viewing and analysis with the listed faults in Core_D.

Figure 15-1. Using MTFI with Hierarchical Designs



For more information about the [read_faults](#) and [write_faults](#) commands, refer to their descriptions in the *Tessent Shell Reference Manual*.

Commands that Support MTFI

The following table summarizes the commands that support MTFI.

Table 15-1. MTFI Command Summary

Command	Description
read_faults	Updates the current fault list with the faults listed in a specified file.
report_faults	Displays data from the current fault list.
write_faults	Writes data from the current fault list to a specified file.

Chapter 16

Graybox Overview

Graybox functionality streamlines the process of scan insertion and ATPG processing in a hierarchical design by enabling you to perform scan and ATPG operations on a sub-module, and then enabling you to use a simplified, graybox representation of that sub-module when performing scan and ATPG operations at the next higher level of hierarchy.

Because the graybox representation of a sub-module contains only a minimal amount of interconnect circuitry, the use of grayboxes in a large, hierarchical design can dramatically reduce the amount of memory and tool runtime required to perform scan insertion, optimize timing, analyze faults, and create test patterns.

Note

 Currently, only Mux-DFF scan architecture is supported with the graybox functionality.

[Table 16-1](#) summarizes the commands that support graybox functionality, which is available in the ATPG tool.

Table 16-1. Graybox Command Summary

Command	Description
analyze_graybox	Identifies the instances and nets to be included in the graybox netlist.
set_attribute_value	Assigns an attribute and value to specified design objects. For a list of graybox attributes, refer to the analyze_graybox command description.
report_graybox_statistics	Reports the statistics gathered by graybox analysis.
write_design	Writes the current design in Verilog netlist format to the specified file. Optionally writes a graybox netlist.

What Is a Graybox?	621
Graybox Process Overview	624
Example dofile for Creating a Graybox Netlist	625
Graybox Netlist Generation for EDT Logic Inserted Blocks	626

What Is a Graybox?

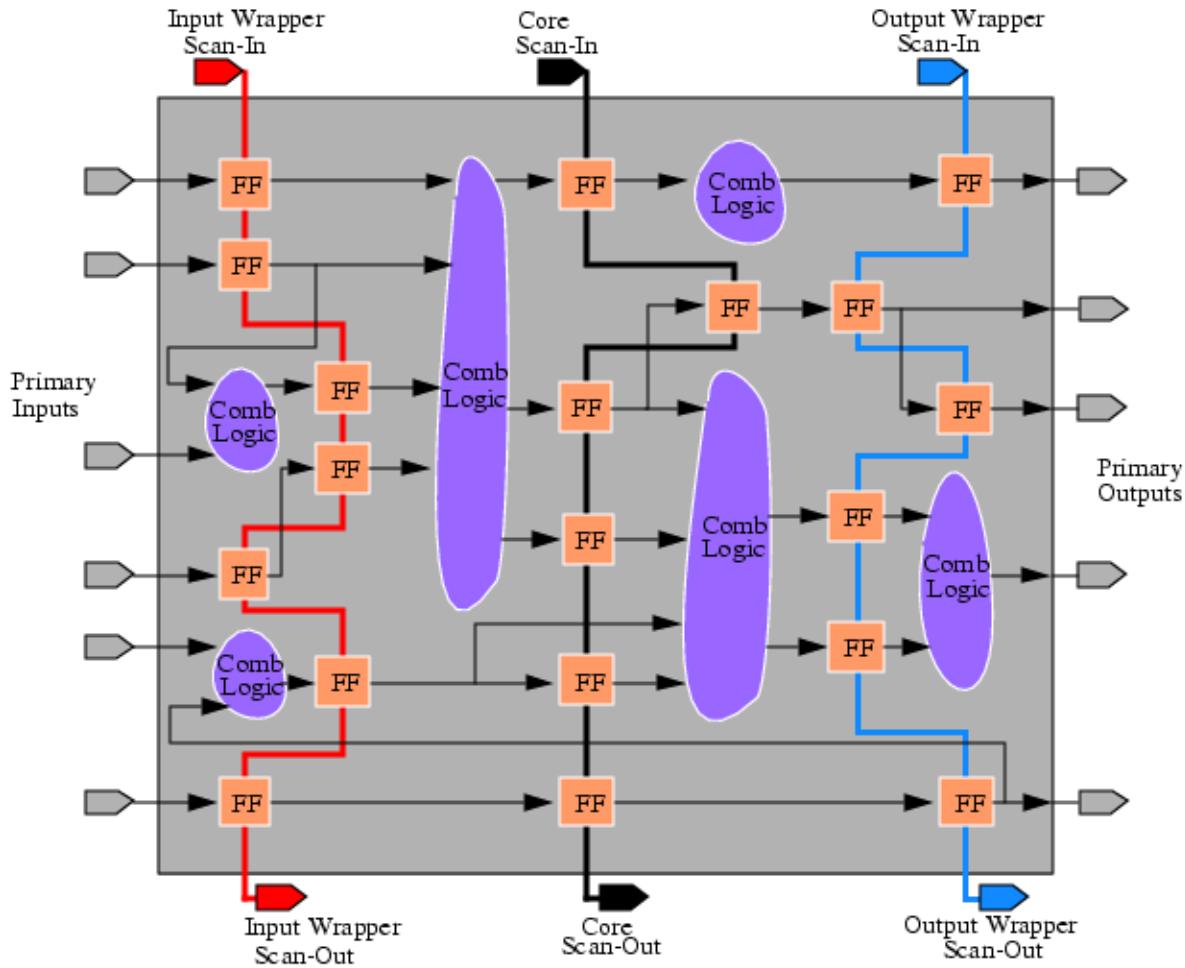
A graybox is a simplified representation of a sub-module that contains only the minimum amount of interconnect circuitry (primary inputs/outputs, wrapper chains, and the glue logic

outside of the wrapper chains) required to process the grayboxed sub-module at the next higher level of hierarchy.

To understand a graybox representation of a sub-module, first consider the full netlist representation shown in [Figure 16-1](#). This figure shows the input and output wrapper chains, the core scan chains, and the combinational logic that exists both inside and outside the wrapper chains.

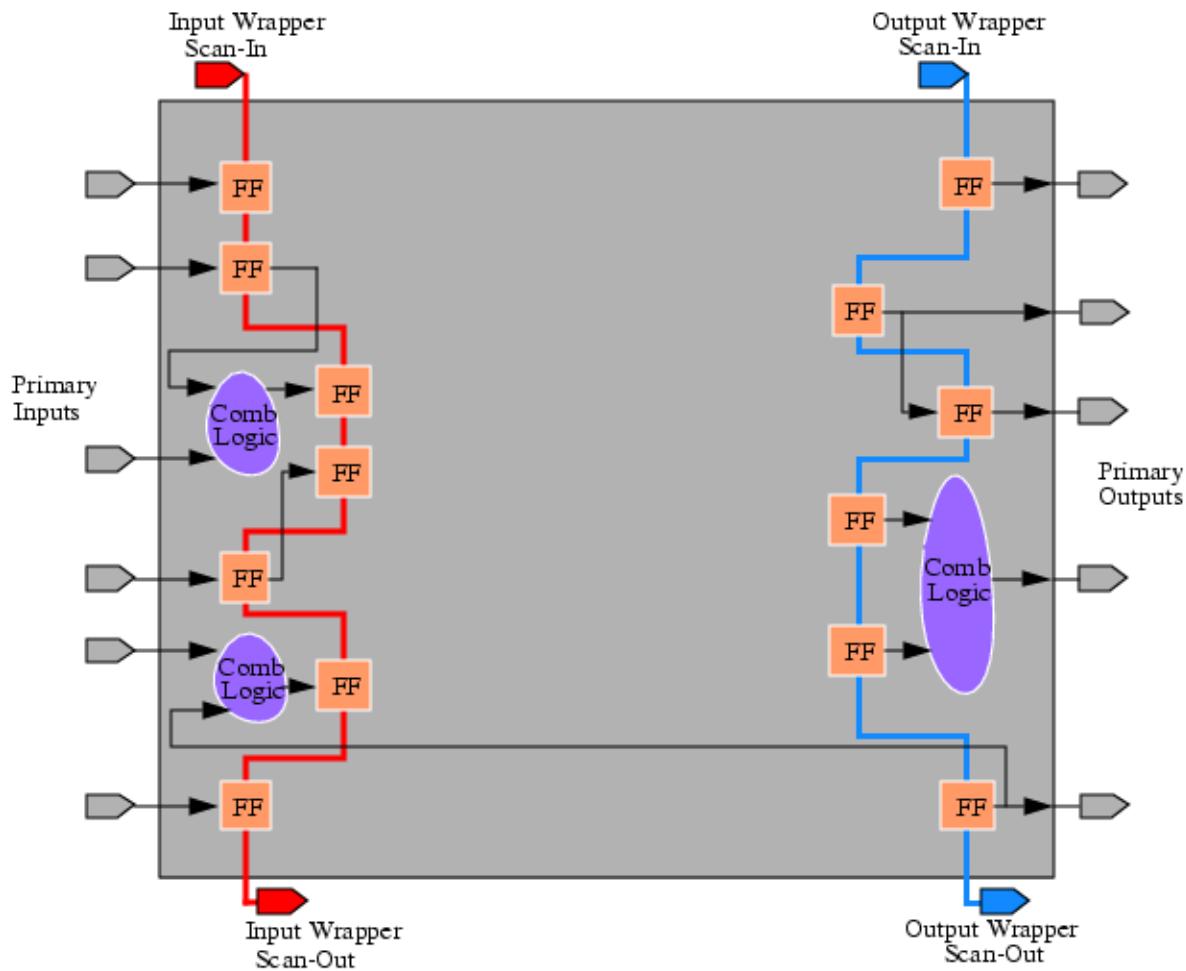
After performing scan insertion, fault accounting, and pattern creation for this sub-module, you create a graybox representation of the sub-module, as shown in [Figure 16-2](#).

Figure 16-1. Full Hierarchical Block Netlist



[Figure 16-2](#) is a graybox representation of the sub-module shown in [Figure 16-1](#). Note that the graybox contains only the primary inputs/outputs, wrapper chains, and combinational logic that exists outside of the wrapper chains (that is, any combinational logic between a primary input or output and the nearest connected flip-flop).

Figure 16-2. Graybox Version of Block Netlist



Graybox Process Overview

The following is a description of the overall process of generating a graybox netlist. Graybox functionality is available when the tool is in the analysis system mode and the design is in external mode. External mode means that the input wrapper chains are used in regular test modes (both capture and shift), whereas the output wrapper chains are used only in a non-capture mode (shift, hold or rotate). Wrapper chains inserted by Tesson Scan are configured in external mode by constraining the output wrapper chain `scan_enable` signal to active during both shift and capture phases.

The dofile used for graybox netlist generation does the following:

1. Defines clock pins used in external mode (using the `add_clocks` command).
2. Constrains test control pins that place the circuit in external mode (using the `add_input_constraints` command).
3. Defines wrapper chains (using the `add_scan_chains` command).
4. Places the circuit in external mode using a test procedure file. This test procedure file should do the following:
 - o Define a test-setup procedure for the external mode to force primary inputs that enable signal paths to wrapper cells.
 - o Define a shift and load-unload procedure to force wrapper chain scan enable signals and toggle the shift clocks for the external mode.

Other types of scan and clock procedures (such as master-observe or shadow-observe) and non-scan procedures (such as capture) might also be required to ensure that the circuit operates correctly in external mode.

Note

 Test-setup procedures are not permitted if they pulse the clocks to initialize non-scan cells to constant values in order to sensitize the control signals of external mode. In other words, the only permitted control signals that place the circuit in external mode are the primary inputs to the block (core).

5. Identifies graybox logic using the `analyze_graybox` command. The command also displays a summary to indicate the combinational and sequential logic gates identified by the analysis. The tool marks the identified graybox instances by setting their “`in_graybox`” attribute. You can also include additional instances in the graybox netlist (or exclude specific instances from the graybox netlist) by turning on/off this attribute using the `set_attribute_value` command.

The graybox analysis performs the identification by tracing backward from all primary output pins and wrapper chains. However, the scan-out pins of the core chains are excluded from the backward tracing. Since core chains are not defined with the

add_scan_chains command, you accomplish this by setting the ignore_for_graybox attribute of the scan-out pins using the set_attribute_value command.

6. The “write_design -graybox” command writes out all the instances marked with the in_graybox attribute. The tool unifies all modules that are included in the graybox netlist (except the top module). The interface (port declarations) of a unified module is preserved. The unification is required because the partial inclusion of the logic inside a module into the graybox netlist could cause conflicts between the different instances of the module, as these instances could be interacting differently with the wrapper chains.

Example dofile for Creating a Graybox Netlist **625**

Graybox Netlist Generation for EDT Logic Inserted Blocks **626**

Example dofile for Creating a Graybox Netlist

The following dofile example shows how to create a graybox netlist.

```
# Define clock pins used for external mode
add_clocks 0 NX2
add_clocks 0 NX1

# Set up for external mode
# Hold output wrapper chain scan enable active
add_input_constraints sen_out -C1

# Define wrapper chains
add_scan_groups grp1 external_mode.testproc
add_scan_chains wrapper_chain1 grp1 scan_in1 scan_out1
add_scan_chains wrapper_chain2 grp1 scan_in2 scan_out2

# Ignore core chains scan_out pins for graybox analysis to exclude the
# logic intended for internal test mode
set_attribute_value scan_out3 -name ignore_for_graybox -value true
set_attribute_value scan_out4 -name ignore_for_graybox -value true

set_system_mode analysis

# Identify graybox logic
analyze_graybox -collect_reporting_data
report_graybox_statistics -top 10

# NOTE: At this point, you can use the set_attribute_value command with
# the in_graybox attribute to include/exclude specific instances into/from
# graybox netlist.

# Write graybox netlist
write_design -graybox -output_file graybox.v -replace
```

Graybox Netlist Generation for EDT Logic Inserted Blocks

A graybox netlist can also be generated for a block whose scan chains are driven by EDT logic. Below is an example dofile to generate a graybox netlist for a block whose wrapper chain scan I/O ports are accessed directly through wrapper-extest ports, bypassing the EDT logic.

These ports are inserted to the block by Tesson Scan when you insert the wrapper chains using the “set_wrapper_chains -wrapper_extest_ports” command. The wrapper-extest ports are activated by constraining the global signal wrapper_extest to a logic 1. The EDT logic and subsequently the core logic are excluded from graybox netlist by marking the EDT channel outputs with the ignore_for_graybox attribute before performing the graybox analysis.

```
# Define clocks
add_clocks 0 clk

# Set up for external mode
# Enable access to wrapper chain extest ports and
# hold output wrapper chain scan enable active
add_input_constraints wrapper_extest -C1
add_input_constraints scan_en_out -C1

# Define wrapper chains
add_scan_groups grp1 cpu_block1_extest.testproc
add_scan_chains chain1 grp1 scan_in1 scan_out2
add_scan_chains chain2 grp1 scan_in3 scan_out4
add_scan_chains chain3 grp1 scan_in5 scan_out6
add_scan_chains chain4 grp1 scan_in7 scan_out8

# Ignore all EDT channel outputs to exclude EDT logic and core chains
# from the graybox netlist.
set_attribute_value edt_channels_out1 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out2 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out3 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out4 -name ignore_for_graybox -value true

# DRC
set_system_mode analysis

# Identify graybox logic
analyze_graybox

# Write graybox netlist
write_design -graybox -output_file graybox.v
```

In the following example, the block does not have dedicated wrapper-extest ports and therefore scan I/O ports are accessed only through EDT logic during extest. The setup of the EDT logic is skipped to simplify the DRC process as there will be no pattern generation in the same tool run. The EDT logic is inserted such a way that an exclusive subset of the channel I/O pins are used only for wrapper chains. This enables excluding the core logic from graybox netlist by marking the core chain EDT channel outputs with ignore_for_graybox attribute. Graybox analysis can identify the EDT logic that is sensitized for extest if the depth of sequential pipeline stages on

channel outputs allocated for wrapper chains is less than 2. However, in this example, the EDT block is specified as a preserve instance to include it in the graybox netlist entirely.

```

# Define clocks
add_clocks 0 clk

# No EDT setup
set_edt_options off

# Set up for external mode
add_input_constraints seno C1

# Define wrapper chains
add_scan_groups grp1 setup.testproc
add_scan_chains -internal chain1 grp1 \
    /edt_block_i/edt_scan_in[0]/edt_block_i/edt_scan_out[0]
add_scan_chains -internal chain2 grp1 \
    /edt_block_i/edt_scan_in[1] /edt_block_i/edt_scan_out[1]

# Ignore EDT channel outputs that access core logic in graybox analysis
set_attribute_value edt_channels_out3 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out4 -name ignore_for_graybox -value true

# DRC
set_system_mode analysis

# Identify graybox logic
analyze_graybox -preserve_instances edt_block_i

# Write graybox netlist
write_design -graybox -output_file graybox.v

```

The EDT logic can also be included in the graybox netlist without using the `-preserve_instances` switch explicitly. EDT Finder can identify the parts of the EDT logic that are sensitized for extest and automatically adds them as preserve instances for graybox analysis. This also enables any sequential pipeline stages on channel I/O pins to be included in the graybox netlist. The following example dofile shows a typical setup to utilize the EDT finder in graybox analysis. The EDT channel pins allocated for core chains are constrained/masked to ignore for EDT Finder. However, this is usually effective when separate EDT blocks are inserted for wrapper and core chains.

```

# Define clocks
add_clocks 0 clk

# Set up for external mode
add_input_constraints seno C1

# EDT Finder is on by default
# Ignore EDT channels that are used for core logic
add_clocks 0 edt_clock
add_input_constraints edt_clock C0
add_input_constraints edt_channels_in2 -C0
add_output_masks edt_channels_out2

# Define EDT block and all chains
add_edt_blocks edt_block
set_edt_options -channels 2 -longest_chain_range 4 28
set_edt_pins input_channel 1 edt_channels_in1 -pipeline_stages 4
set_edt_pins input_channel 2 edt_channels_in2 -pipeline_stages 4
set_edt_pins output_channel 1 edt_channels_out1 -pipeline_stages 3
set_edt_pins output_channel 2 edt_channels_out2 -pipeline_stages 3
add_scan_groups grp1 setup.testproc
add_scan_chains -internal chain1 grp1 /edt_block_i/edt_scan_in[0] \
/edt_block_i/edt_scan_out[0]
add_scan_chains -internal chain2 grp1 /edt_block_i/edt_scan_in[1] \
/edt_block_i/edt_scan_out[1]
add_scan_chains -internal chain3 grp1 /edt_block_i/edt_scan_in[3] \
/edt_block_i/edt_scan_out[3]
add_scan_chains -internal chain4 grp1 /edt_block_i/edt_scan_in[4] \
/edt_block_i/edt_scan_out[4]

# Ignore EDT channel outputs that access core logic in graybox analysis
set_attribute_value edt_channels_out3 -name ignore_for_graybox -value true
set_attribute_value edt_channels_out4 -name ignore_for_graybox -value true

# DRC
set_system_mode analysis

# Identify graybox logic
analyze_graybox

# Write graybox netlist
write_design -graybox -output_file graybox.v

```

Chapter 17

Tessent On-Chip Clock Controller

In modern designs, on-chip clock control (OCC) circuits are commonly used to manage clocks during test. Such clock controllers can generate slow-speed or at-speed clock sequences under the control of ATPG process. Tessent OCC is an implementation of a clock controller created by Tessent Shell that has been designed to meet the requirements of scan test for ATPG, Logic BIST, EDT, and Low Pin Count Test.

Tessent OCC Overview	629
Primary OCC Functions	630
On-Chip Clock Controller Design Description.....	632
The Standard OCC	633
Standard OCC With JTAG Interface.....	648
The Parent OCC	650
The Child OCC	652
Tessent OCC Insertion	654
Inserting the OCC	654
occ.dft_spec	655
post_dft_insertion_procedure.tcl	656
OCC Insertion When Using an Existing Clock as the test_clk DFT Signal.....	656
Core OCC Recommendation	658

Tessent OCC Overview

Using Tessent Shell, you can generate and insert Tessent OCCs into your design. You configure the OCCs to generate programmable clock pulses under ATPG control. Additionally, you can interface the TK/LBIST hybrid controller to a Tessent OCC that has a capture-enabled trigger and external clock control capability.

In this capacity, a Tessent OCC is used to ensure that the following requirements are met:

- Independent control by ATPG of each clock domain to improve coverage, reduce pattern count, and achieve safe clocking with minimal user intervention.
- During capture, deliver correct number of clock pulses on a per-pattern basis.
- Cleanly switch between shift and capture clocks.
- Enable slow or fast clocks during capture for application of slow and at-speed patterns.
- Scan-programmable clock waveforms generated within a wrapped core are ideal for generating patterns at the core level. These patterns can be retargeted to the top level and

merged to simultaneously test multiple cores without conflicts in how clocks are controlled within each core.

You can use the Tessent OCC with the following:

- OCC Tessent Core Description-based automation for ATPG
- TK/LBIST Hybrid Controller

Note

 The Tessent on-chip clock control methodology is intended for use on designs where all clock domains are internal. If you also have external clocks then you must divide your transition ATPG into two separate sessions: one session for the external clocks and one session for the internal clocks.

Primary OCC Functions

Typical on-chip clock controllers have three primary functions: control clock selection, clock chopping control, and clock gating.

- Clock Selection
 - Selects which always-capture or always-pulse clock is used.
 - Selection of clock based upon frequency, test type, and other criteria.
- Clock Chopping Control
 - Scan-programmable shift register controls clock pulse suppression.
 - Creates enable signal for clock gating.
- Clock Gating
 - Gates clock based upon clock chopping enable signal.

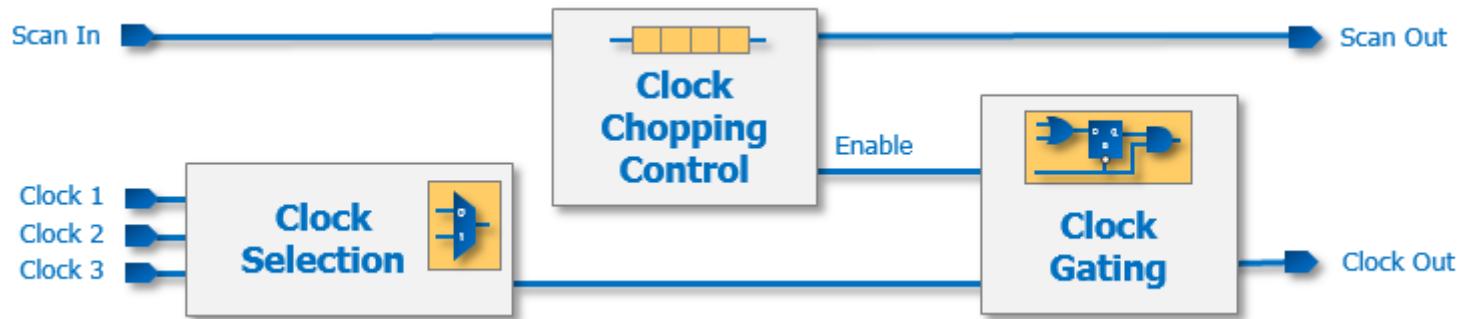
The Tessent OCC, by default, contains an internal clock gater. This option can be disabled during the generation step. When the design already contains clock gaters, the Tessent OCC can be generated with a clock enable signal. For information, see the “[OCC](#)” section in the *Tessent Shell Reference Manual*.

When a Tessent OCC with a clock enable is used, the tool detects clock gating cell controlled by the OCC. This process is automatic and does not require user input.

To enable proper detection of clock gating cells controlled by the OCC, the cells need to be properly modeled in the Cell Library with the appropriate simulation functions. For more information on library models, see “[Cell Library](#)” in the *Tessent Cell Library Manual*.

[Figure 17-1](#) shows the relationship of these three functions.

Figure 17-1. OCC Clock Control Components



On-Chip Clock Controller Design Description

There are three types of on-chip controller designs: standard, parent, and child. You select them based upon the requirements for your design. When using an OCC, you must consider the design elements discussed in this section.

The Standard OCC	633
Design Placement	633
Standard OCC Schematic	634
OCC With Capture Enable	644
Clock Control Operation Modes	644
Timing Diagrams	647
Standard OCC With IJTAG Interface	648
The Parent OCC	650
The Child OCC	652

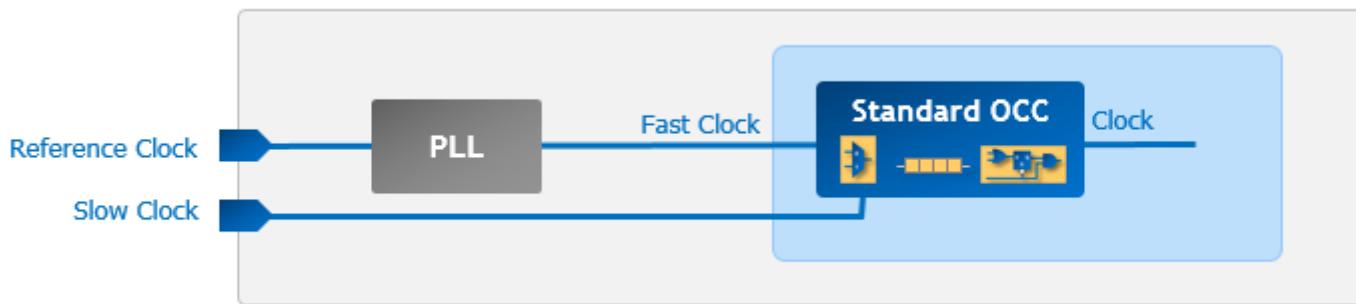
The Standard OCC

The standard OCC provides a fast clock for fast capture and a slow clock for shift and slow capture. A scan programmable shift register enables ATPG to suppress or pulse clock cycles as required. One usage for the standard OCC is Intest mode for pattern retargeting.

The standard OCC performs all three OCC functions: clock selection, clock chopping control, and clock gating. [Figure 17-2](#) shows a standard OCC implementation.

The standard OCC is the recommended OCC for use in hierarchical cores. For more information, see “[Core OCC Recommendation](#)” on page 658.

Figure 17-2. Standard OCC Example



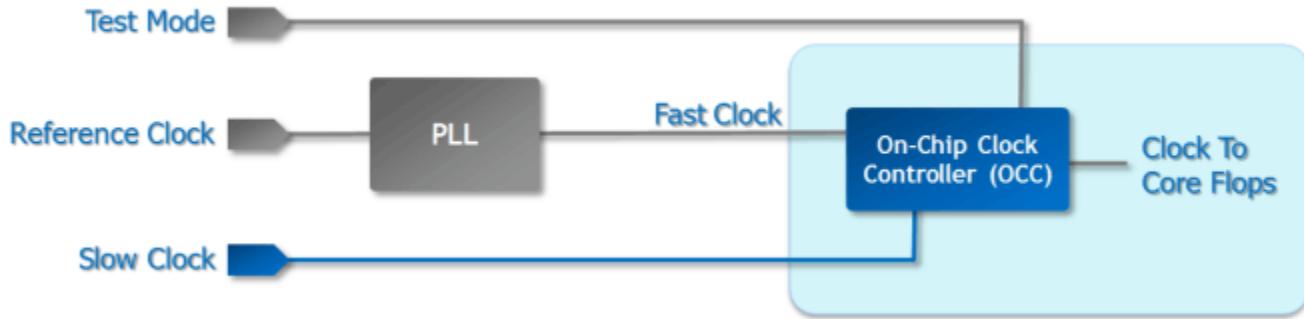
Design Placement	633
Standard OCC Schematic	634
OCC With Capture Enable	644
Clock Control Operation Modes	644
Timing Diagrams	647

Design Placement

The OCC should be inserted such that the fast clock input of the OCC is driven by the functional clock source, typically a PLL. Ideally, the OCC is placed near the clock source (PLL) but should be placed inside cores to enable local clock control as needed for pattern retargeting flows. The OCC uses a top-level slow clock for shift and slow capture as well as a test mode signal that determines if a test or functional clock is supplied to the design.

[Figure 17-3](#) shows an example of OCC placement.

Figure 17-3. Clock Control Logic Design Placement



Depending on your design style, you may need to guide the Clock Tree Synthesis (CTS) not to balance the flops and latches in the OCC with the clock tree it drives.

There is no reason for you to add a clock mux after the OCC for functional mode because the fast_clock propagates through when the test mode signal is de-asserted. Additionally, having a common path for the fast clock in functional mode and test mode simplifies clock tree synthesis and timing closure.

The clock control design is used to supply the clock when in functional and test modes. In functional mode, the fast clock is passed to the design. In test mode, the fast clock is used for at-speed capture while a top-level slow clock will be used for shift and slow capture. The reference clock supplied to the PLL is a free-running clock, typically pulse-always.

It is recommended not to flatten the clock control blocks during layout in order to ease automation of defining the clock gating logic and its operation.

Standard OCC Schematic

The schematic for the on-chip clock control circuit is shown in this figure.

The schematics shown represent an OCC that is generated without an IJTAG interface. To learn how an OCC with an IJTAG interface differs, see “[Standard OCC With IJTAG Interface](#)” on page 648.

Figure 17-4. On-Chip Clock Controller Logic Schematic

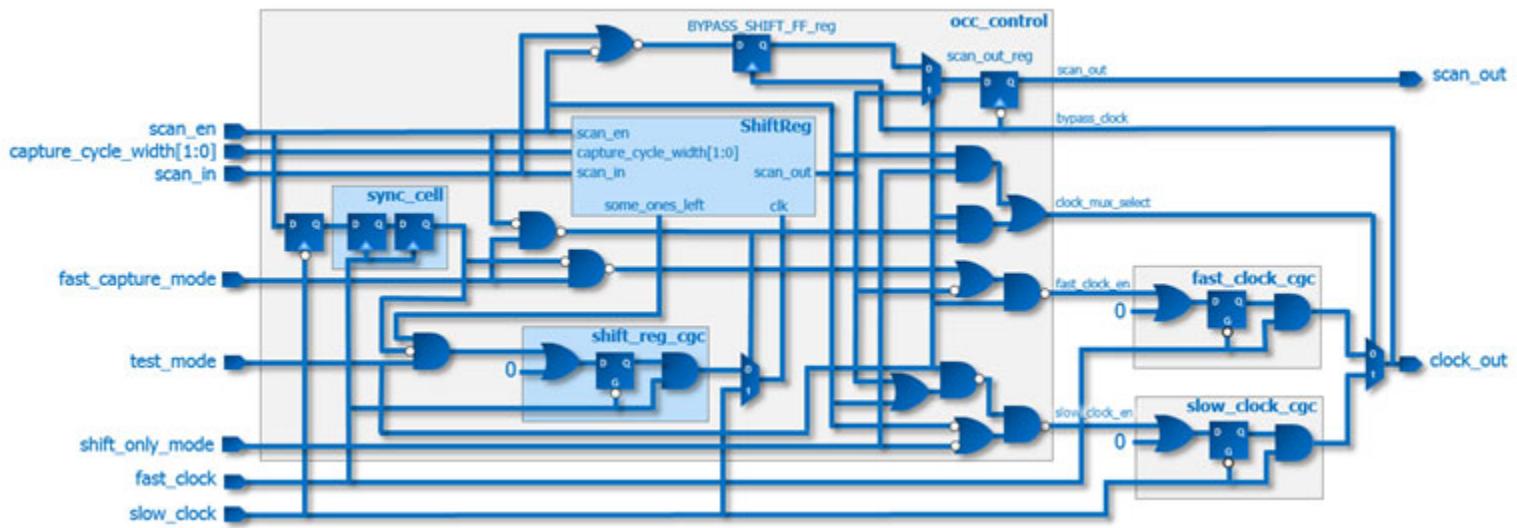


Table 17-1 describes the functionality of the clock controller I/O signals.

Table 17-1. Clock Controller I/O Signals

Name	Direction	Description
scan_en	Input	Scan enable driven by top-level pin

Table 17-1. Clock Controller I/O Signals (cont.)

Name	Direction	Description
capture_cycle_width:[1:0]	Input ¹	<p>Configures the maximum number of clock pulses during capture cycle as well as the length of the scan chain:</p> <ul style="list-style-type: none"> 00 - 1 pulse - 1 scan cell 01 - 2 pulses - 2 scan cells 10 - 3 pulses - 3 scan cells <p>If the length of the OCC scan chain is set to 4 using the capture_window_size property during OCC IP creation, this input will be configured as follows:</p> <ul style="list-style-type: none"> 11 - 4 pulses - 4 scan cells <p>For example, if the max sequential depth to be used is 2, this bus can be configured to “01” instead of default “10”. This can reduce the number of shift cycles, and more importantly reduces the number of bits which need to be encoded if the OCC chain is driven by a decompressor.</p>
scan_in	Input	Scan chain input for loading shift register
fast_capture_mode	Input ¹	Selects fast or slow capture clock (0 = slow, 1 = fast)
test_mode	Input ¹	Selects test or functional mode (0 = functional, 1 = test). When test mode is disabled, fast clock drives <code>clock_out</code> to enable functional mode. Remaining clock gaters are disabled to minimize switching activity
fast_clock	Input	Clock for fast capture (typically output of PLL, free_running or pulse_always)
slow_clock	Input	Clock for shift and slow capture
scan_out	Output	Scan chain output for unloading shift register

Table 17-1. Clock Controller I/O Signals (cont.)

Name	Direction	Description
clock_out	Output	Controlled clock output

1. Static signals that do not change during the test session should be controlled through on-chip controllers (such as IJTAG) or other means in order to reduce the need for top-level pins.

Slow Clock Driving Sequential Elements

The top-level slow clock used for shift and slow capture must not control any sequential elements directly because slow clock acts as a trigger for capture during fast-capture mode. During fast-capture, the necessary slow clock pulse(s) are not simulated by the tool and can result in simulation mismatches. This requirement is explained in more detail in the following section.

Scan Enable Synchronization

In order to synchronize the top-level scan enable signal with the fast clock (PLL output), a two flop synchronization cell is used and clocked by fast clock. This is important because scan enable is used as the trigger signal to gate the clock to the shift register. The output of the synchronization cell produces a scan enable signal which is synchronized with the fast clock and can be used during fast capture test.

Additionally, a flop is used on the input side of the synchronization cell and clocked by the trailing edge of slow clock. Since scan enable normally fans out to the entire circuit and may arrive after fast clock, the flop on slow clock ensures that scan enable is not synchronized by the fast clock until slow clock is pulsed, thus reducing the risk of a race condition.

In order to ensure proper DRC analysis and simulation, the output of the clock gater cell driven by the synchronization logic is defined as a pulse-in-capture internal clock. When using the TCD flow for pattern generation, the tool automatically defines the internal clock as a pulse-in-capture clock. This ensures correct simulation of the logic during load_unload and avoids unnecessary DRC violations.

The synchronization cell (sync_cell) shown in [Figure 17-4](#) has an asynchronous reset port that is driven by scan_en if it is active high. If the reset port of the synchronous cell is active low it is driven by ~scan_en.

In the RTL description, the synchronization cell is described as a separate module so that it can be replaced with a technology specific synchronization cell from the appropriate library.

Slow Clock Pulses in Capture

Given that slow clock is used to capture scan enable before it is synchronized by fast clock, it must not directly drive any sequential elements that may impact the scan operation. This is because the slow and fast clocks are not assumed to be synchronized thus the pulse on slow

clock at the beginning of capture is defined in the external_capture procedure used for fast capture:

```
procedure external_capture ext_fast_cap_proc =
    timeplate tmp1 ;
    cycle =
        force_pi ;
        pulse slow_clock;
    end;
end;
```

The tool does not simulate clock pulses defined in external_capture procedures when it calculates the expect values in the patterns. The clock pulses defined in this procedure are added to the created patterns after ATPG to ensure correct operation in fast capture mode. Since the output of the clock gating logic is defined as a pulse-in-capture internal clock, the tool does not need to simulate the clock pulse on the register that first captures scan enable. However, if slow clock controls any other sequential elements, it can result in simulation mismatches.

Handling Slow Scan Enable Transitions

Depending on the relation of the frequencies of slow and fast clocks, it may be necessary to add more delay before the fast capture pulses in order to enable scan enable enough time to settle to 0. Similarly, it may be desired to delay the scan enable transition from 0 to 1 after capture and before the next load/unload operation. This requires additional cycles to be added to the external_capture procedure as shown in the following example:

```
procedure external_capture ext_fast_cap_proc =
    timeplate tmp1 ;
    cycle =
        force_pi ;
    end;
    cycle =
    end;
    cycle =
    end;
    cycle =
        pulse slow_clock;
    end;
    cycle =
    end;
    cycle =
    end;
end;
```

Adding external_capture cycles with no slow_clock pulses before the cycle that pulses slow_clock, delays the fast clock pulses to give scan enable sufficient time to transition to 0.

Empty cycles after the pulse on slow_clock, delay the load/unload operation to give scan enable sufficient time to transition to 1.

Note

The scan enable synchronization operation described in this section is not used for slow capture mode which uses slow clock for shift and capture.

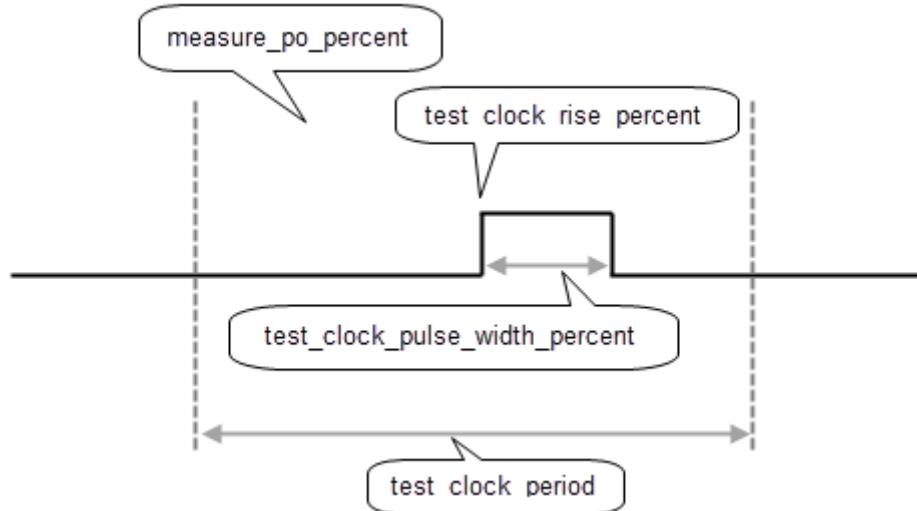
Fast Clock With Slower Frequency Than Slow Clock

If the frequency of the fast clock is slower than the frequency of the slow clock, you need to use two different timeplates for shift and for load_unload. Most situations permit you to use the same timeplate for both.

In this case, the procedure requires additional cycles using the (slower) fast clock's period, and additional post-shift cycles must be specified in the dofile to ensure correct operation of the parallel testbench.

Refer to the following files for an example of this setup. The variable settings for the sample dofile are example values only; these settings are design-specific and are included here for reference. This figure illustrates how these variables are used:

Figure 17-5. Sample Dofile Variables



Sample Dofile

```
## <Read in and set up the design>
...
##### variable settings #####
set test_clock_period 40
set scan_en_mcp 3
set slowest_fast_clock_period 45
set measure_po_percent 45
set test_clock_rise_percent 50
set test_clock_pulse_width_percent 25

import_scan_mode int_mode -fast_capture_mode on
set_current_mode int_mode_fast
source ../data/import_procedures.tcl
...
set_fault_type transition
set_external_capture_options -pll_cycles $pll_cycles 1test_capture_cycle
set_atpg_limit -pattern_count 128
create_patterns
write_tsdb_data -replace
write_patterns corea_[get_current_mode].serial.v -verilog -serial \
    -replace -end 2 -parameter_list [list SIM_TOP_NAME TB]
write_patterns -corea_[get_current_mode].parallel.v -verilog -replace \
    -parameter_list [list SIM_TOP_NAME TB SIM_POST_SHIFT $SIM_POST_SHIFT]
...
...
```

The *import_procedures.tcl* file computes the timing, pll_cycles, and SIM_POST_SHIFT:

```
import_procedures.tcl

set test_clock_period_slow [expr $scan_en_mcp * $test_clock_period]
set test_clock_period_shift_cycle $test_clock_period
set measure_po_shift_cycle [expr \
    {round(0.01*$measure_po_percent*$test_clock_period)}]
set test_clock_rise_shift_cycle [expr \
    {round(0.01*$test_clock_rise_percent*$test_clock_period)}]
set test_clock_width_shift_cycle [expr \
    {round(0.01*$test_clock_pulse_width_percent*$test_clock_period)}]
set test_clock_period_capture_cycle $test_clock_period_slow
set measure_po_capture_cycle [expr \
    {$test_clock_period_capture_cycle - ($test_clock_period_shift_cycle- \
    $measure_po_shift_cycle)}]
set test_clock_rise_capture_cycle [expr \
    {$test_clock_period_capture_cycle - ($test_clock_period_shift_cycle- \
    $test_clock_rise_shift_cycle)}]
set test_clock_width_capture_cycle $test_clock_width_shift_cycle
if {$scan_en_mcp <= 2} {
    set test_clock_period_pre_shift_cycle $test_clock_period_shift_cycle
    set measure_po_pre_shift_cycle $measure_po_shift_cycle
    set test_clock_rise_pre_shift_cycle $test_clock_rise_shift_cycle
} else {
    set test_clock_period_pre_shift_cycle [expr \
        {$test_clock_period_capture_cycle - \
        $test_clock_period_shift_cycle}]
    set measure_po_pre_shift_cycle [expr \
        {$test_clock_period_pre_shift_cycle - \
        ($test_clock_period_shift_cycle-$measure_po_shift_cycle)}]
    set test_clock_rise_pre_shift_cycle [expr \
        {$test_clock_period_pre_shift_cycle - \
        ($test_clock_period_shift_cycle-$test_clock_rise_shift_cycle)}]
}
set test_clock_width_pre_shift_cycle $test_clock_width_shift_cycle
set procfile_name ..../data/procedures.def
set capture_width 4
set pll_cycles 1
if {!{info exists slowest_fast_clock_period}} {
    display_message -warning "The variable 'slowest_fast_clock_period' is \
        not defined. Assuming the fast_clock to slow_clock period \nratio \
        is such that 2 fast_clock cycles fits inside one slow_clock cycle. \
        If you are not sure, \nspecify the period of the slowest \
        fast_clock as it exists at the input of each OCCs."
    incr pll_cycles [expr {int(ceil((2+$capture_width) * 0.5))}]
    set SIM_POST_SHIFT 1
} else {
    incr pll_cycles [expr {int(ceil((2.0+$capture_width) * \
        $slowest_fast_clock_period/$test_clock_period_slow))}]
    set SIM_POST_SHIFT [expr {int(ceil(2.0*$slowest_fast_clock_period/ \
        $test_clock_period) - 1)}]
}
puts "pll_cycles      = $pll_cycles"
puts "SIM_POST_SHIFT = $SIM_POST_SHIFT"
```

This import file uses a procedures DEF file with three different timeplates:

```
procedures.def
  timeplate ltest_shift_cycle =
    force_pi 0 ;
    measure_po $measure_po_shift_cycle ;
    pulse_clock $test_clock_rise_shift_cycle $test_clock_width_shift_cycle;
    period $test_clock_period_shift_cycle ;
  end;
  timeplate ltest_capture_cycle =
    force_pi 0 ;
    measure_po $measure_po_capture_cycle ;
    pulse_clock $test_clock_rise_capture_cycle \
      $test_clock_width_capture_cycle ;
    period $test_clock_period_capture_cycle ;
  end;
  timeplate ltest_pre_shift_cycle =
    force_pi 0 ;
    measure_po $measure_po_pre_shift_cycle ;
    pulse_clock $test_clock_rise_pre_shift_cycle \
      $test_clock_width_pre_shift_cycle ;
    period $test_clock_period_pre_shift_cycle ;
  end;
  set default_timeplate ltest_capture_cycle;
procedure shift =
  scan_group grp1 ;
  timeplate ltest_shift_cycle ;
  cycle =
    force_sci ;
    measure_sco;
  end;
end;
procedure load_unload =
  scan_group grp1 ;
  timeplate ltest_shift_cycle;
  cycle =
    timeplate ltest_pre_shift_cycle;
  end;
  apply shift 2;
end;
```

The following figures show waveform graphics for this setup. These figures include the following signals, from top to bottom:

- shift_capture_clock
- edt_clock
- edt_update
- scan_enable

Figure 17-6. Fast Capture, Serial Load

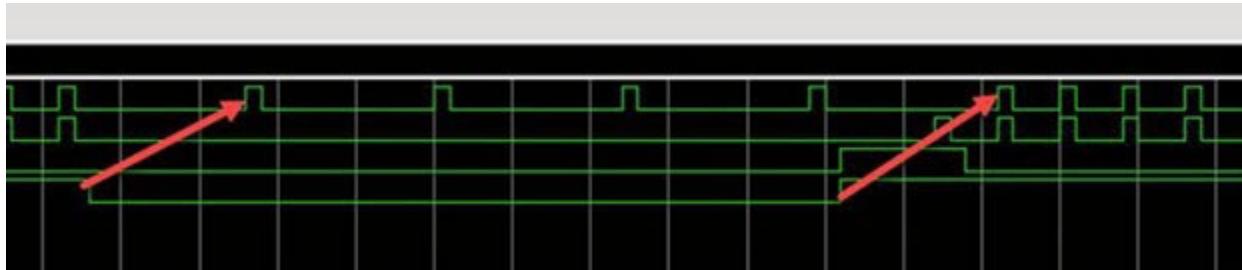


Figure 17-7. Slow Capture, Serial Load



Fast Capture Commands

When not using named capture procedures (NCP), the following three commands are automatically set by the `create_patterns` command for fast capture:

```
set_output_masks on
add_input_constraints -all -hold
set_clock_restriction domain_clock -any_interaction -compatible_clocks_between_loads on
```

The preceding three commands are not needed when using NCPs because the capture clock sequence and time for forcing PIs and measuring POs are explicitly defined by the user in NCPs.

However, you will need to specify:

```
set_external_capture_options ...
```

Bypass Shift Path

Figure 17-4 shows the standard OCC schematic has a single `BYPASS_SHIFT_FF_reg` flop. This flop is also in the parent and child OCCs. Setting the `test_mode` signal of OCCs not used for a particular test mode puts them into functional mode. The OCC sub-chain is part of the design's scan chains and, even when the OCC is not being used for a particular test mode and other parts of the design are being tested, the scan chains must continue to work. While in the functional mode, the path from `scan_in` to `scan_out` remains accessible by shifting through the `BYPASS_SHIFT_FF_reg` flop.

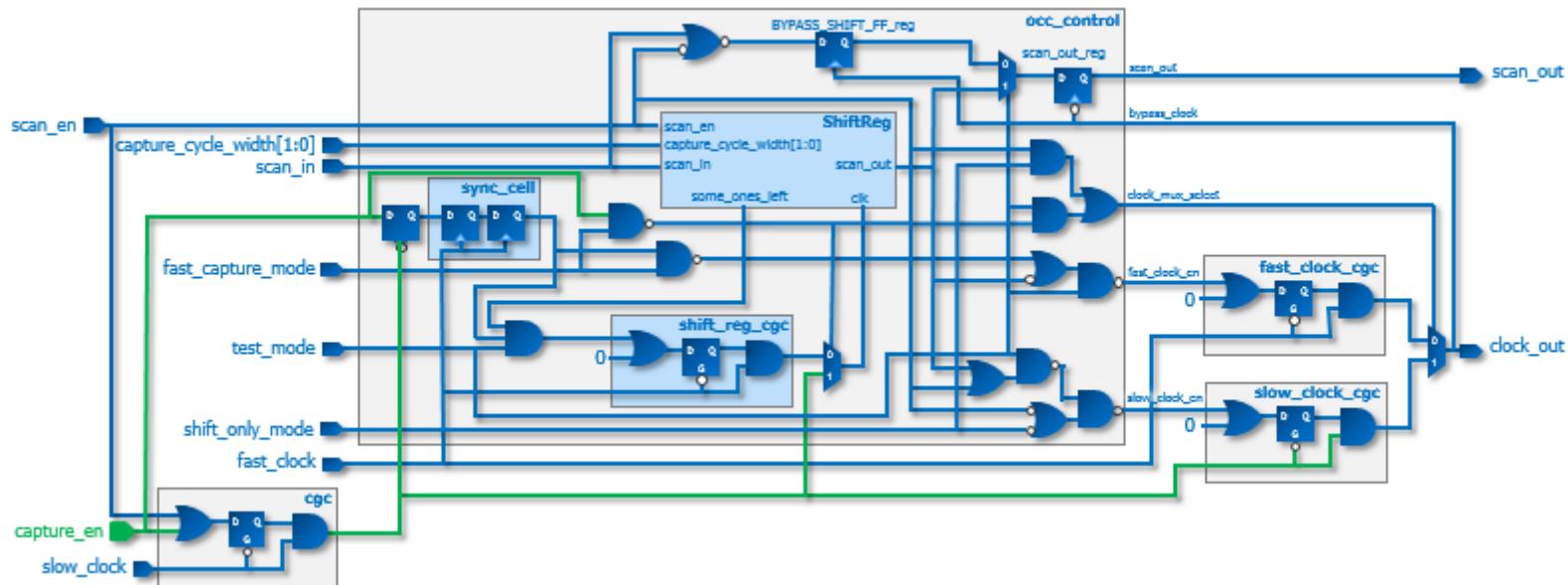
OCC With Capture Enable

There is an OCC with a capture_enable input that can be used with a free-running slow clock.

When the OCC is specified with capture enabled (capture_trigger : capture_en), a clock gater is added on the input free-running slow clock to generate appropriate clock pulses from the free running clock input. The enable signal for this clock gater is an OR gate output whose inputs are scan_en and capture_en. This enables the slow_clock pulses to be generated during shift and capture cycles respectively.

Figure 17-8 shows the OCC schematic with the clock gater at the lower left.

Figure 17-8. OCC with capture_en



This version of the OCC can also be used with a tester provided slow clock (that is not free running). In this case, the new clock gater simply lets the controlled slow clock pass through during shift and capture.

For LBIST and LPCT applications, an OCC that uses a free-running slow clock is required. This free running clock is connected to either the shift_clock of the LBIST controller or the LPCT input clock.

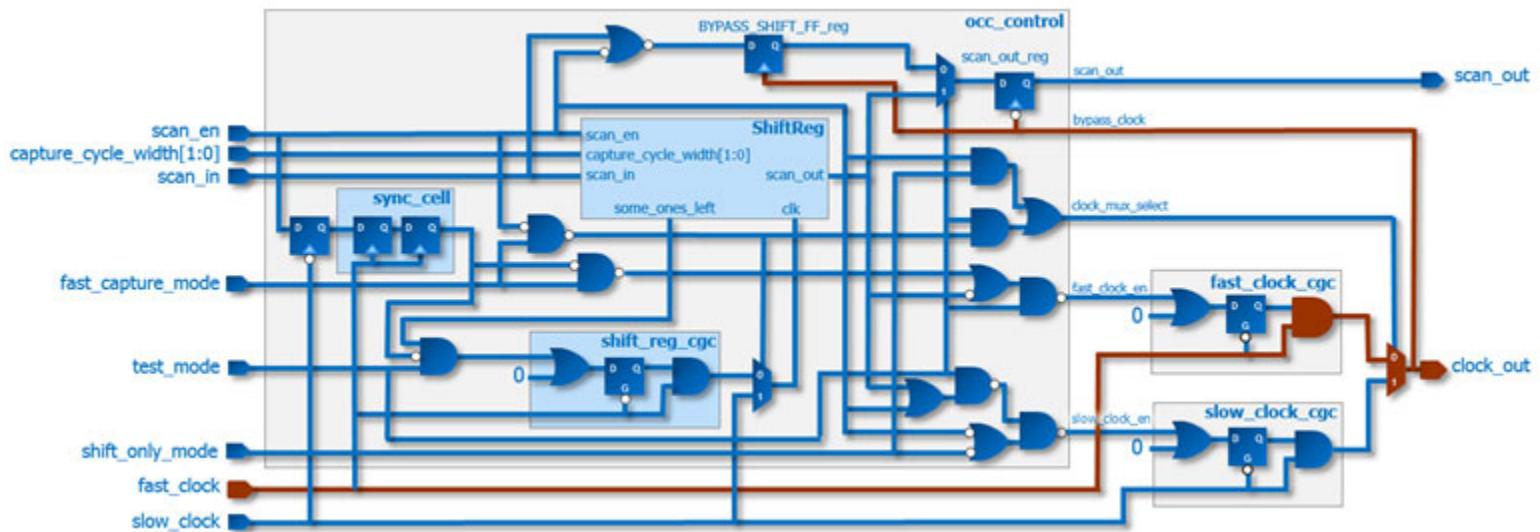
Clock Control Operation Modes

There are four modes as follows.

Functional Mode

When operating in functional mode (test_mode = 0), the fast clock gater is enabled to supply fast clock to the design. The slow clock and internal clock gaters are disabled to reduce power.

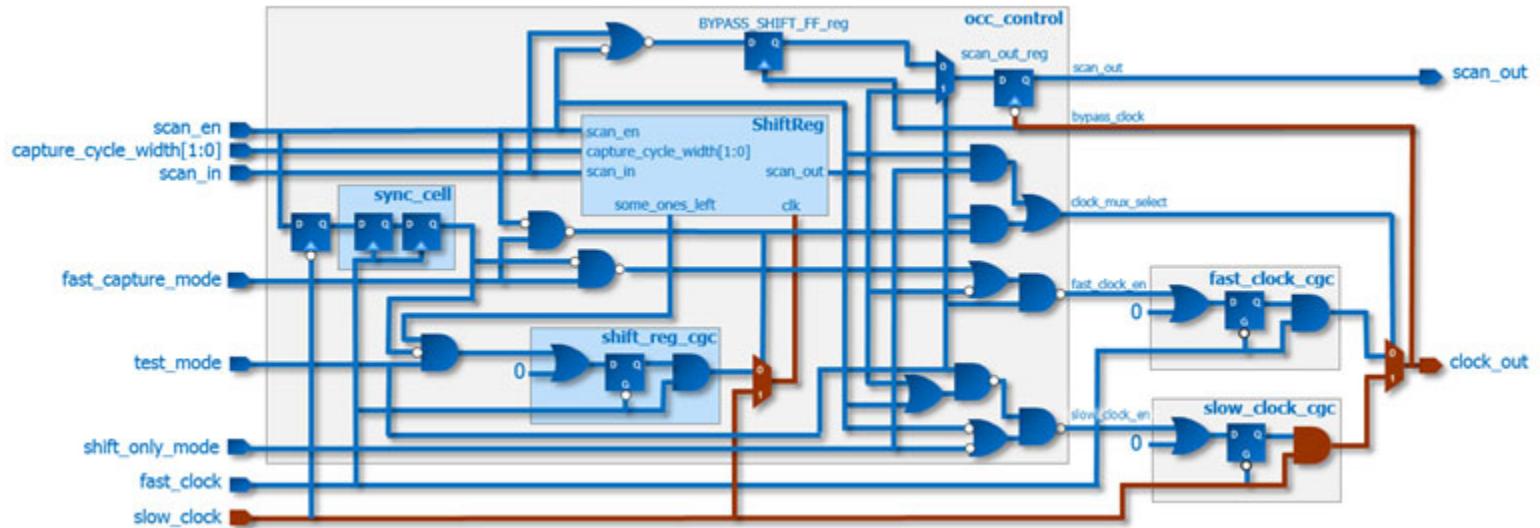
Figure 17-9. Functional Mode Operation



Shift Mode

In shift mode (**scan_en = 1**), **slow_clock** is used to load/unload scan chains which include the condition bits in **ShiftReg**.

Figure 17-10. Shift Mode Operation



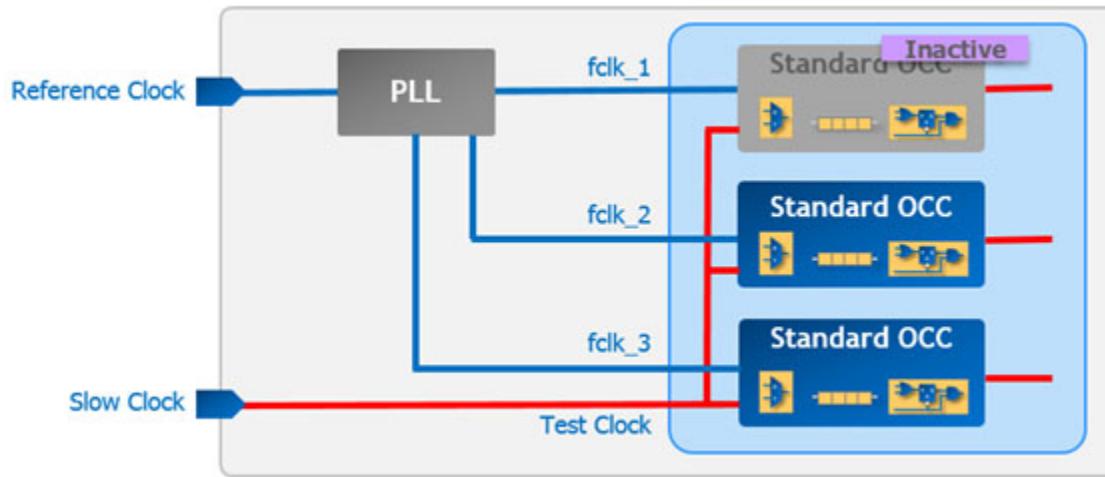
Shift-Only Mode

In shift-only mode, the OCC is disabled (**test_mode = 0**) but shift is enabled (**scan_en = 1**). In this mode, the **slow_clock** clock gater is enabled to ensure slow clock path is always used for shift. Additionally, bypass shift is enabled.

Figure 17-11 shows that *both* inactive (**test_mode = 0**) and active OCCs use **slow_clock** for shift. By default, Standard and Parent OCCs will enable the shift clock path anytime **scan_en** is

1, even if the OCC is inactive; this ensures consistent shift timing in internal and external modes.

Figure 17-11. OCC Shift Clocking

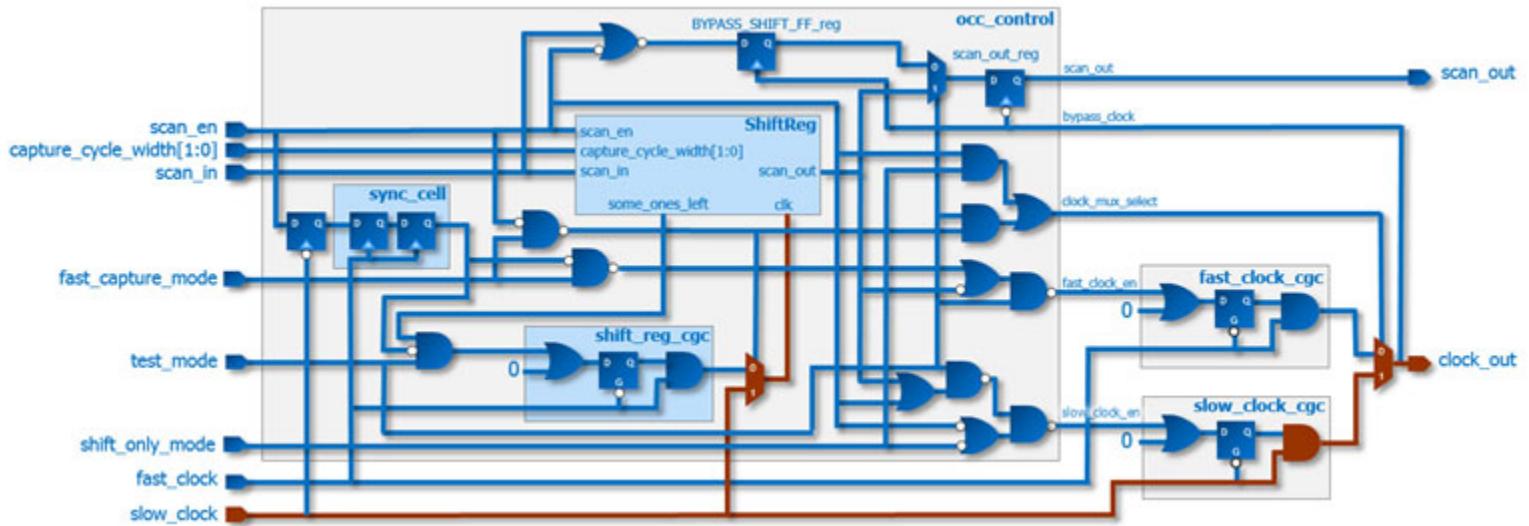


To change the default and use the functional clock instead of the test clock, set the [shift_only_mode](#) OCC wrapper property to off.

Slow Capture Mode

In slow capture mode (`fast_capture_mode = 0`), `slow_clock` is used to capture data into scan cells and to shift the condition bits in `ShiftReg`.

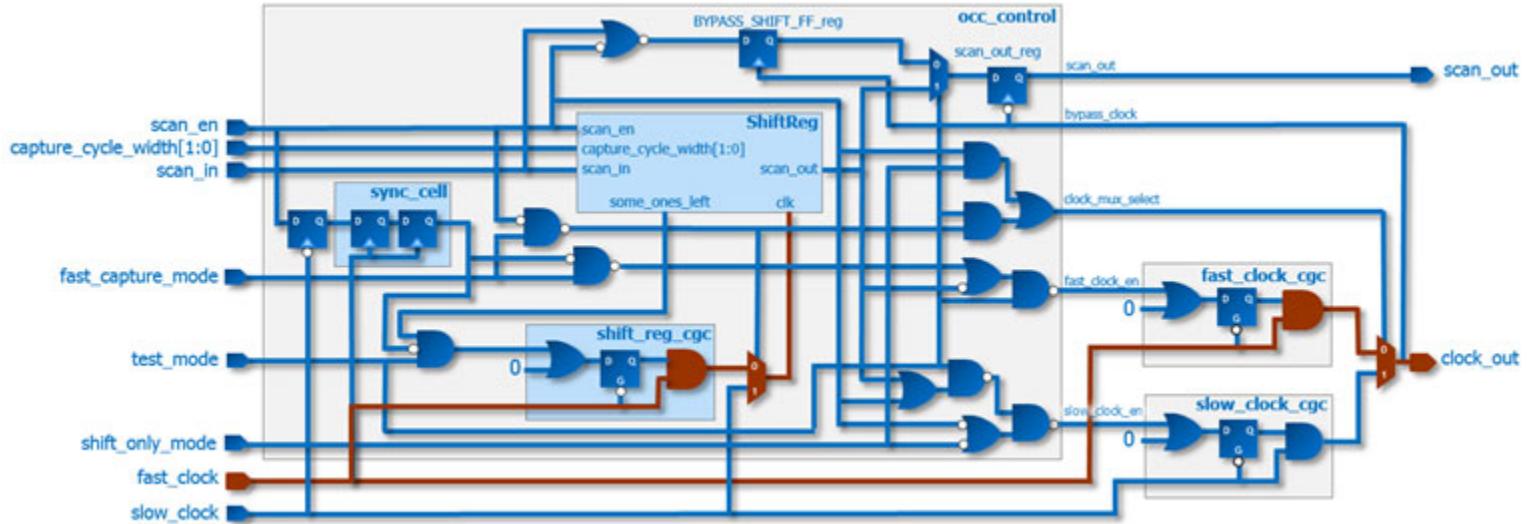
Figure 17-12. Slow Capture Mode



Fast Capture Mode

In fast capture mode (`fast_capture_mode = 1`), `fast_clock` is used to capture data into scan cells and to shift the condition bits in `ShiftReg`.

Figure 17-13. Fast Capture Mode Operation



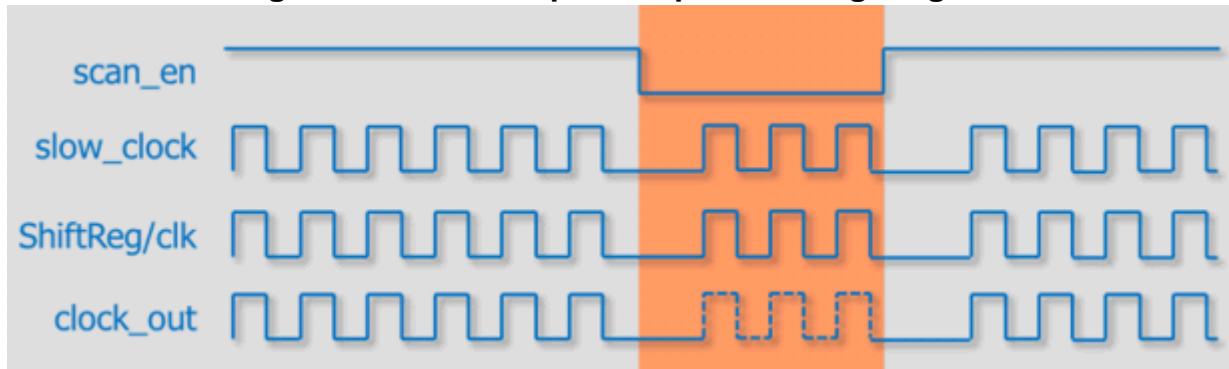
Note

- For stuck-at faults and if there is at least one OCC in fast capture mode, the tool masks all POs and holds all PIs, and allows no interaction between clocks.

Timing Diagrams

The timing diagram for slow speed capture (`fast_capture_mode = 0`) is shown in the following figure:

Figure 17-14. Slow Speed Capture Timing Diagram

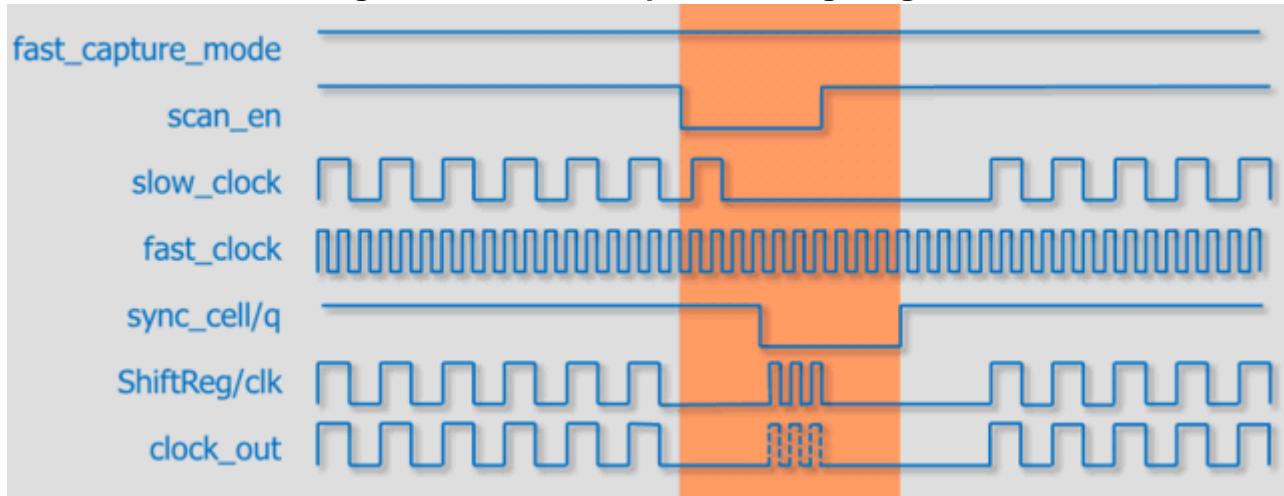


For this example, `capture_cycle_width` is set to “10” resulting in a maximum sequential depth of 3. In this mode, slow clock is used for shift as well as capture. Based on condition bits loaded

into the shift register, the `clock_out` port will generate the appropriate number of slow clock pulses.

In fast capture mode (`fast_capture_mode = 1`) the waveforms in [Figure 17-15](#) are generated. Similar to the previous example, `capture_cycle_width` is set to “10” here resulting in a maximum sequential depth of 3. In this mode, the `slow_clock` is still used for shift but the fast capture pulses on `clock_out` are based on `fast_clock`.

Figure 17-15. Fast Capture Timing Diagram



As shown, the scan enable signal which has been synchronized to the fast clock (`sync_cell/q`) is used to trigger the fast clock pulses on `ShiftReg/clk`. The `ShiftReg/clk` signal is the clock source for the shift register containing the condition bits. Based on the condition bits loaded during shift, the correct number of fast clock pulses will appear on `clock_out`.

Standard OCC With IJTAG Interface

An OCC created with an IJTAG interface has an additional multiplexer not present in an OCC without IJTAG.

As shown in [Figure 17-16](#), a multiplexer is added to the `slow_clock` path to inject `ijtag_tck` when an OCC is created with an IJTAG interface (when `ijtag_host_interface` is set to something other than none).

Figure 17-16. Standard OCC with IJTAG Interface

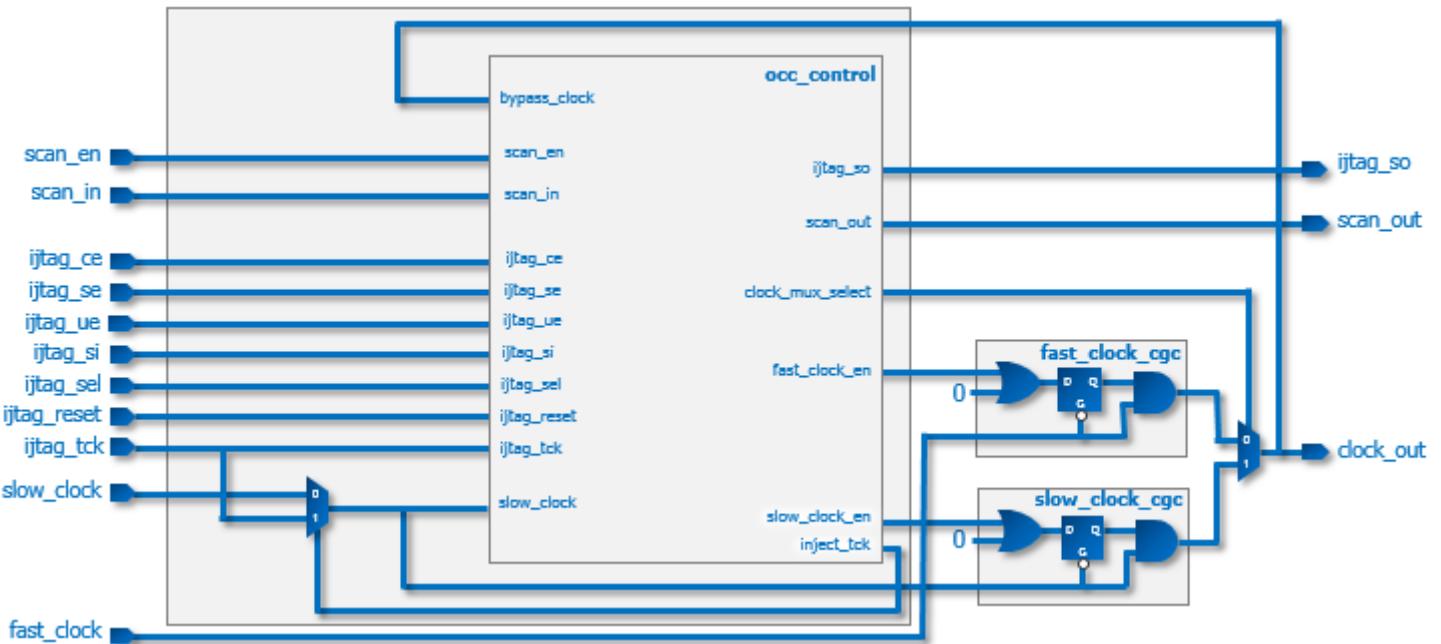
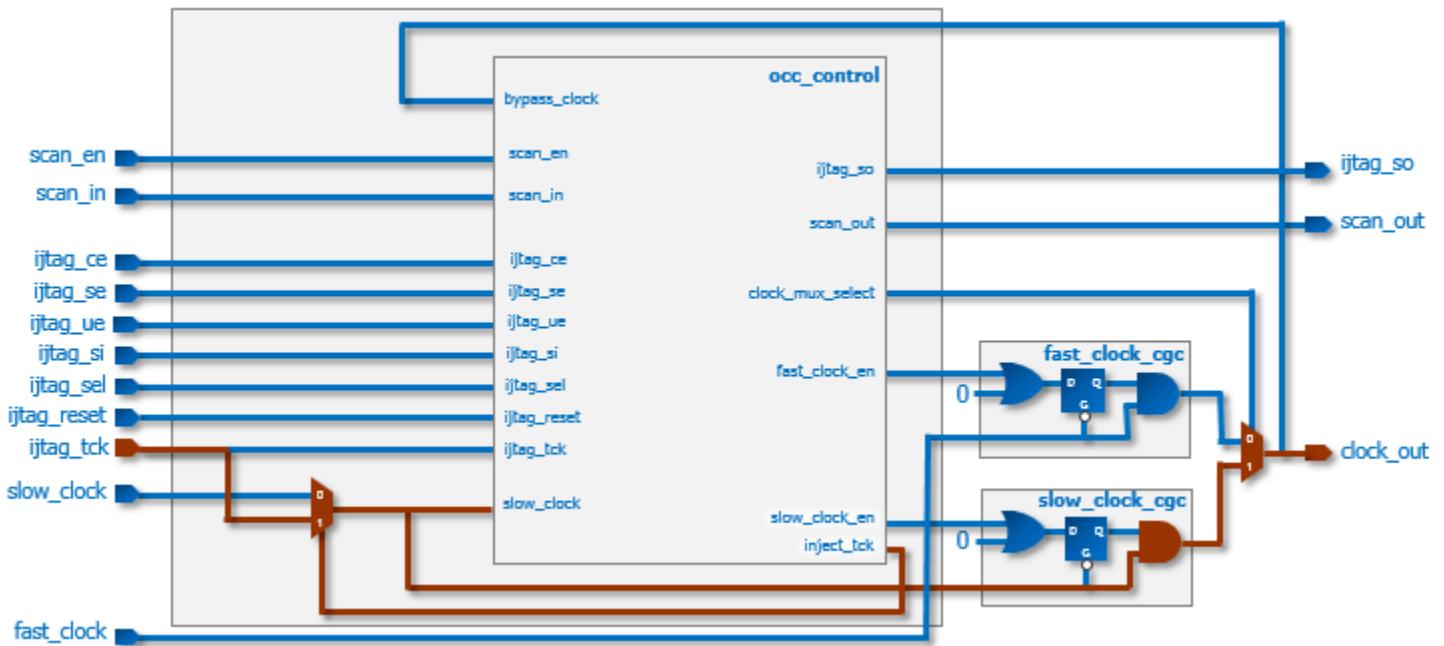


Figure 17-17 highlights the functionality of the mux added to the slow_clock input of the OCC. When inject_tck is 1, the mux selects ijttag_tck which drives clock_out through the slow clock gater and the output mux.

Figure 17-17. OCC With IJTAG Interface: ijttag_tck injected for slow clock

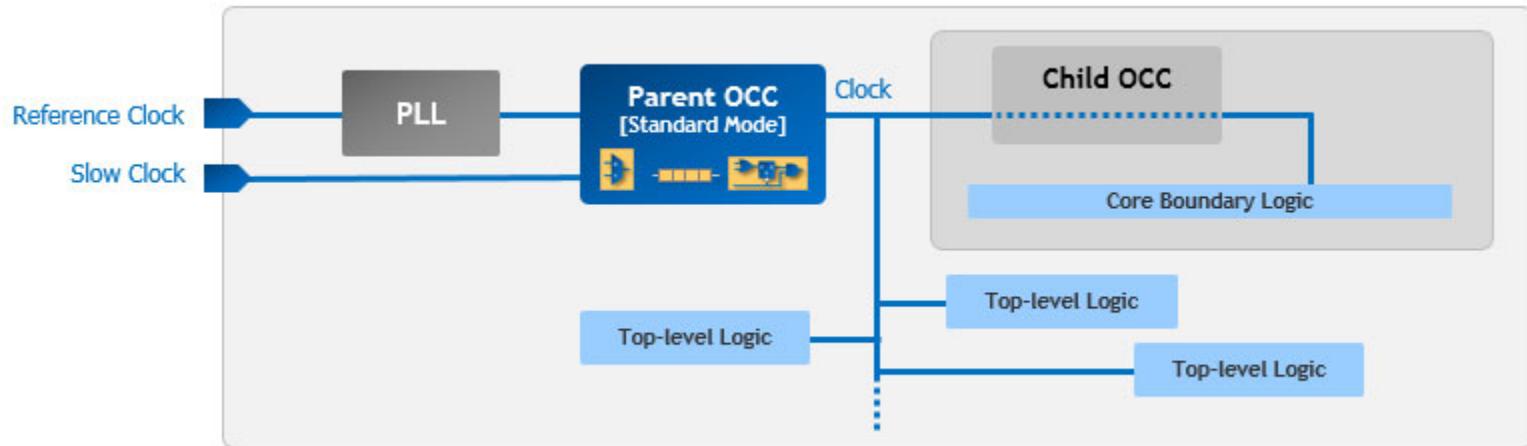


The Parent OCC

As one of the three OCC types, the parent OCC has two modes, standard mode and parent mode. In standard mode it selects and gates clocks for testing top-level logic. In parent mode, this OCC is used for clock selection only.

Parent OCC Schematic in Standard Mode Figure 17-18 shows a parent OCC used in standard mode. A typical use for this mode is pattern retargeting in Extest mode, when interactions between core-level boundary logic (wrapper chains) and top-level logic are being tested.

Figure 17-18. Parent OCC in Standard Mode (Extest Mode)



In parent mode, the OCC is used for clock selection only. The parent OCC typically feeds the clock input of a child OCC. Figure 17-19 shows a parent OCC in parent mode. An example use for this mode is pattern retargeting in Intest mode.

Figure 17-19. Parent OCC in Parent Mode (Intest Mode)

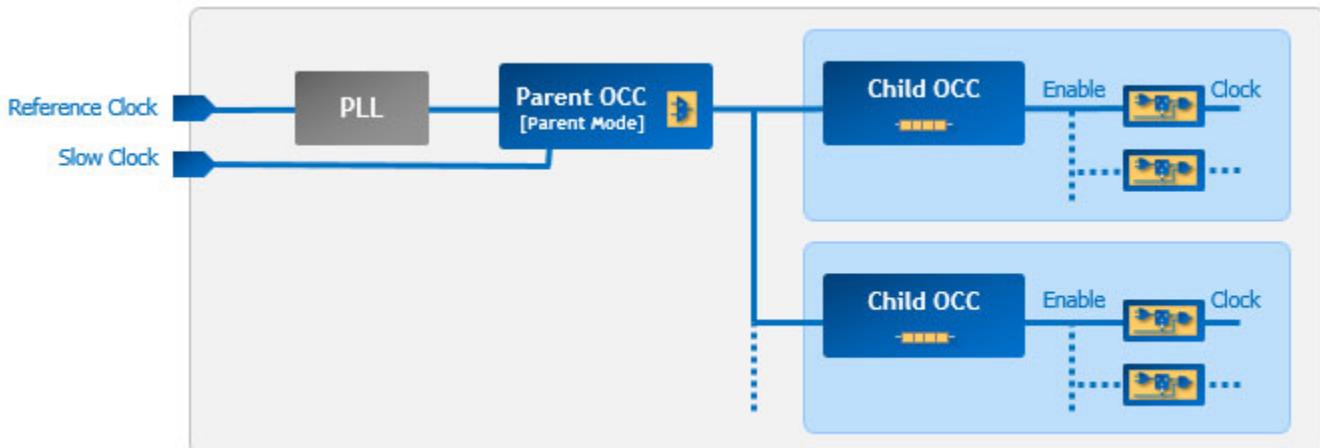


Figure 17-20 shows the parent on-chip controller logic. The shift clock does not need to be routed to the lowest-level child OCC because the parent OCC injects the shift clock at the base

of the clock in shift mode. When scan_en is low, it lets a programmable constant set of clock pulses go through.

The parent OCC usually feeds the clock input of a child OCC which, in turn, enables the scan-based test pattern generation tool to control which clock pulses go through or not.

Figure 17-20. Parent On-Chip Controller Logic Schematic

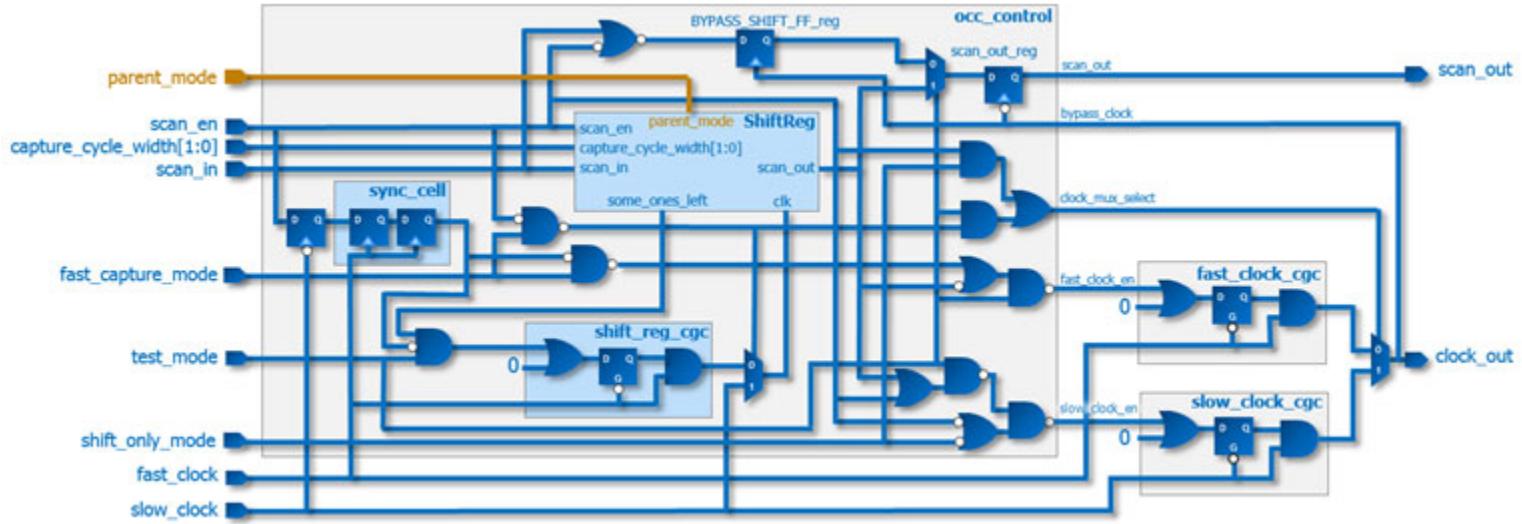


Table 17-2 describes the functionality of the clock controller I/O signals.

Table 17-2. Parent OCC I/O Signals

Name	Direction	Description
parent_mode	Input	Defines whether the OCC is used in parent mode. When the default (0) is used, the OCC functions in standard mode, selecting and gating clocks for top-level testing. When in parent mode (1), the OCC is used for clock selection only.

For a description of remaining I/O signals, see [Table 17-1](#) in the [Standard OCC Schematic](#) section.

The schematics shown represent an OCC that is generated without an IJTAG interface. To learn how an OCC with an IJTAG interface differs, see “[Standard OCC With IJTAG Interface](#).”

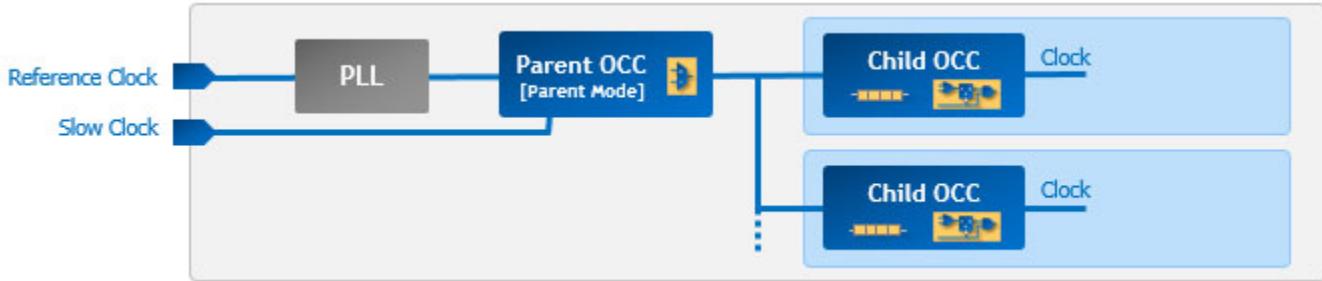
The standard OCC is the recommended OCC for use in hierarchical cores. For more information, see “[Core OCC Recommendation](#)”.

The Child OCC

The child OCC is one of the three OCC types. It performs the clock chopping control function of the OCC and, optionally, also gates the clock.

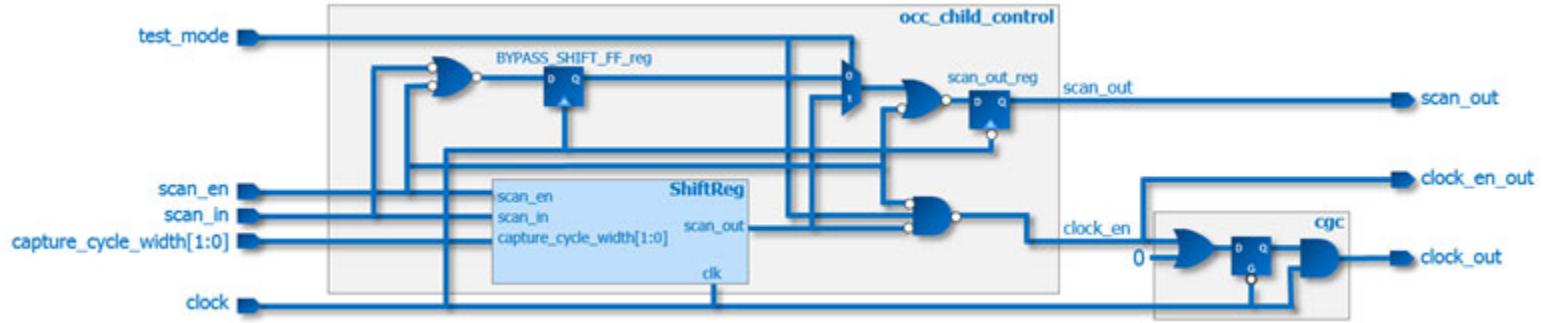
By default, the child OCC creates the clock enable signal based on values loaded into the scan-programmable shift register. This enable signal is used to control the (optional) clock gater inside the OCC as shown in [Figure 17-21](#).

Figure 17-21. Child OCC Gates and Creates Clocks (with Parent OCC)



[Figure 17-22](#) shows a detailed example schematic of the child on-chip controller with a clock gater.

Figure 17-22. Child On-Chip Controller Logic Schematic



In some cases, it is not practical to use a child OCC with an internal clock gater. Optionally, the child OCC can be created without the internal clock gater. The created signal enables layout tools to replicate and control the design's clock gaters for implementations such as a clock mesh. The handling of user provided clock gating cells is transparent to the user. This is shown in [Figure 17-23](#).

Figure 17-23. Child OCC Creates Enable for Clock Gaters (with Parent OCC)

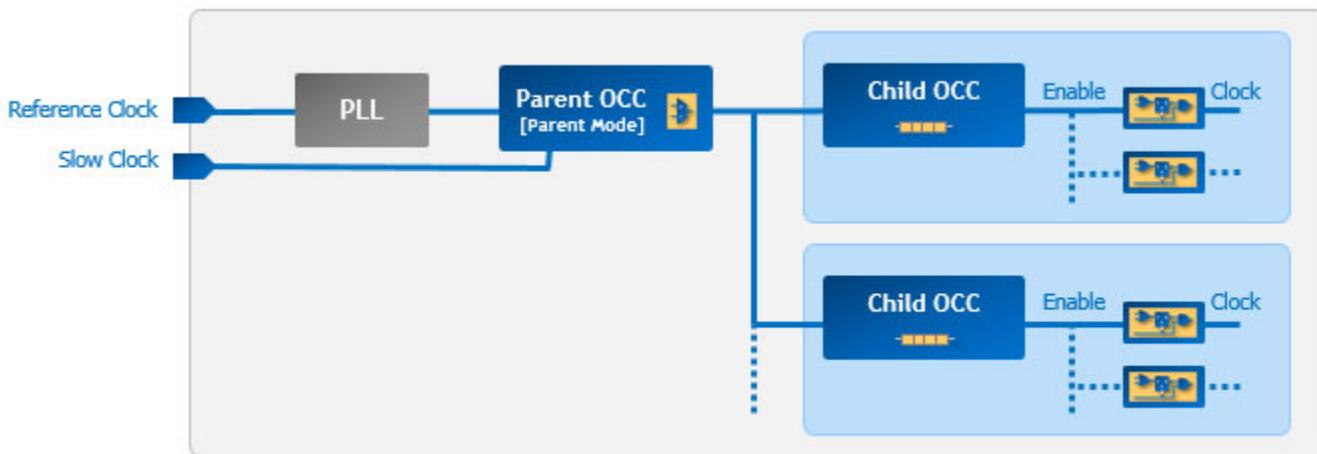
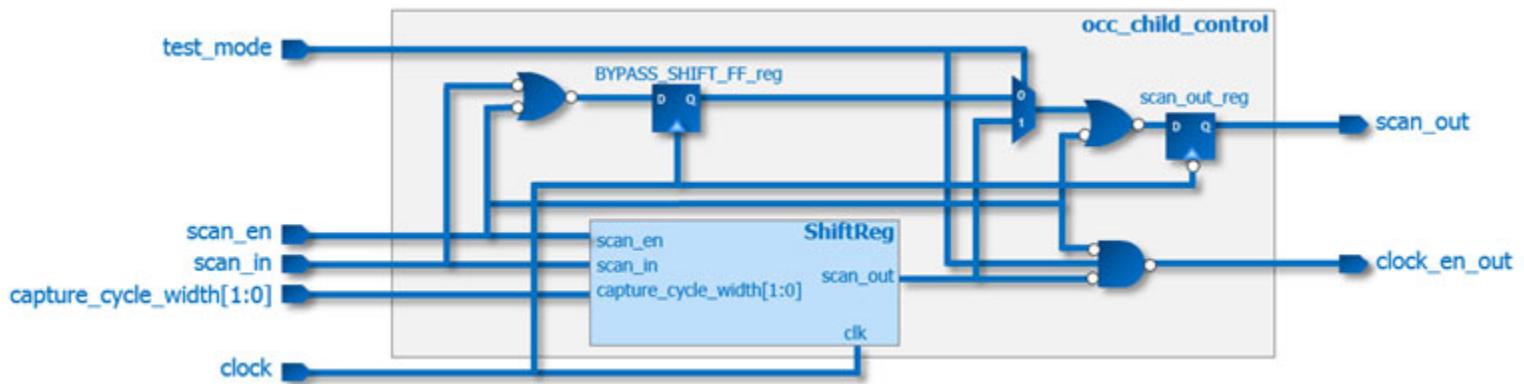


Figure 17-24 shows the child on-chip controller logic without the clock gater.

Figure 17-24. Child On-Chip Controller Logic Schematic, no Clock-Gater



To properly operate the clock gaters during capture, a sensitized path must exist between the OCC's clock enable port and the clock gaters. The tool verifies controllability for all clock gaters that have a structural connection to the OCC. If any of the structurally-connected clock gaters do not have a sensitized path from the OCC, the clock gater must be disabled during capture. In such cases, the tool verifies that the clock gater's test and functional enables are off or that its clock source is off.

The schematics shown represent an OCC that is generated without an IJTAG interface. To learn how an OCC with an IJTAG interface differs, see “[Standard OCC With IJTAG Interface](#).”

The standard OCC is the recommended OCC for use in hierarchical cores. For more information, see “[Core OCC Recommendation](#)”.

Tessent OCC Insertion

Tessent Shell is used to insert the OCC.

You define the Tessent OCC using [DftSpecification configuration syntax](#). The [OCC DftSpecification](#) defines connections for slow clock, scan enable, and static OCC signals.

To achieve the best configuration for compressed ATPG, you will need to stitch the OCC sub-chains into the scan chains. For more information, refer to “[OCC Sub-Chain Stitching](#)” in the *Tessent TestKompress User’s Manual*.

The basic procedure for inserting Tessent OCCs is described below. For more detailed examples of inserting OCCs in a flow and using them for pattern generation, refer to “[Tessent Shell Flow for Flat Designs](#)” and “[Tessent Shell Flow for Hierarchical Designs](#)” in the *Tessent Shell User’s Manual*.

Inserting the OCC	654
occ.dft_spec.....	655
post_dft_insertion_procedure.tcl	656
OCC Insertion When Using an Existing Clock as the test_clk DFT Signal	656

Inserting the OCC

This procedure uses Tessent Shell to insert an instance of the OCC logic for each internal clock.

Procedure

1. Invoke Tessent Shell from the shell prompt.

```
% tessent -shell
```

2. Set the Tessent Shell context to ‘dft’ using the [set_context](#) command.

```
SETUP> set_context dft -no_rtl -design_identifier occ
```

3. Read in the design netlist using the [read_verilog](#) command.

```
SETUP> read_verilog cpu_core.vg.gz
```

4. Read in the cell library using the [read_cell_library](#) command to create some instances such as muxes. For example:

```
SETUP> read_cell_library adk.tcelllib
```

5. Set the current design and design level using the [set_current_design](#) and [set_design_level](#) commands. For example:

```
SETUP> set_current_design cpu
```

```
SETUP> set_design_level sub_block
```

6. Read the configuration file using the [read_config_data](#) command to specify OCC insertion parameters. For example:

```
SETUP> read_config_data occ.dft_spec
```

7. Optionally define the port list and procedure to insert on-chip control for clocks and to connect condition bits of OCC into new scan chain after DFT spec is processed. See “[post_dft_insertion_procedure.tcl](#)” on page 656 for an example.
8. Validate and process the content defined in the DftSpecification wrapper using the [process_dft_specification](#) command.

```
SETUP> process_dft_specification
```

9. Synthesize the generated RTL using the [run_synthesis](#) command.

```
SETUP> run_synthesis
```

This step only applies when inserting the OCC RTL into a gate-level netlist. When inserting into an RTL design, the OCC is synthesized along with the rest of the design. The [run_synthesis](#) command also writes the updated netlist to the TSDB directory.

10. If needed, optionally write out the updated netlist using the [write_design](#) command. For example:

```
SETUP> set_current_design  
SETUP> write_design -output_file  
cpu_core_post_occ_insertion.vg.gz -replace
```

11. Exit the tool.

```
SETUP> exit
```

occ.dft_spec

The following example shows a basic OCC DftSpecification:

```
DftSpecification(cpu, occ_core) {
    reuse_modules_when_possible: on;
    OCC {
        Controller(CLK1_OCC) {
            // DEFINE ROOT OF CLOCK DOMAIN WHERE OCC SHOULD BE INSERTED
            clock_intercept_node: /BUF_OCC_1/Y;
        }
        Controller(CLK2_OCC) {
            // DEFINE ROOT OF CLOCK DOMAIN WHERE OCC SHOULD BE INSERTED
            clock_intercept_node: /BUF_OCC_2/Y;
        }
        Controller(CLK3_OCC) {
            // DEFINE ROOT OF CLOCK DOMAIN WHERE OCC SHOULD BE INSERTED
            clock_intercept_node: /BUF_OCC_3/Y;
        }
    }
}
```

post_dft_insertion_procedure.tcl

Tcl procedure for post-DFT insertion.

This OCC insertion example contains the process_dft_specification.post_insertion procedure. The procedure name is a keyword that instructs the process_dft_specification command to execute the content of the procedure immediately after inserting the specified DFT logic.

The post-insertion script creates a scan chain that concatenates the shift registers of all inserted OCCs into one uncompressed scan chain. Additionally, a dofile is created for adding the clock control scan chain in subsequent steps. Typically, the OCC sub-chains will be stitched to the design's scan chains during scan insertion. In this post-scan insertion example, the post_dft_insertion example is used to demonstrate the capability of making design edits after insertion

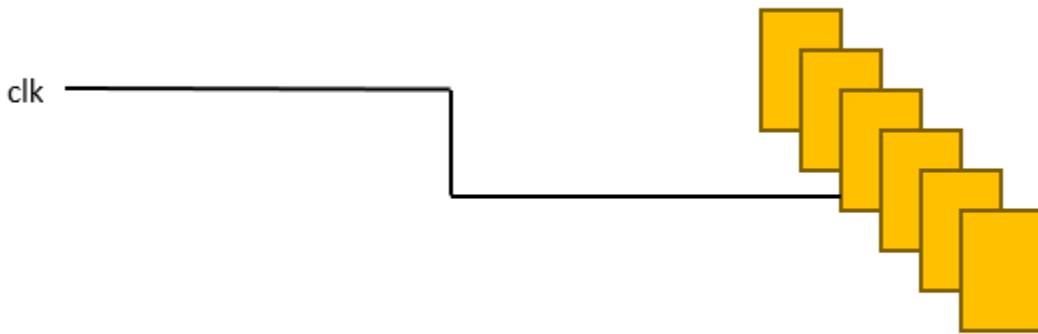
```
proc process_dft_specification.post_insertion {cpu args} {
    # Create ports for control scan chain
    create_port occ_control_scan_in -direction input
    create_port occ_control_scan_out -direction output
    # Concatenate OCC condition bit shift registers into new control scan
    #chain
    set first_inst 1
    foreach inst_name [get_name_list [get_instances -of_module
    cpu_occ_core_tessent_occ]] {
        if ($first_inst) {
            create_connection occ_control_scan_in $inst_name/scan_in
            set first_inst 0
            set last_inst_name $inst_name
        } else {
            create_connection $last_inst_name/scan_out $inst_name/scan_in
            set last_inst_name $inst_name
        }
    }
    create_connection $last_inst_name/scan_out occ_control_scan_out
}
# Create dofile to define OCC control scan chain
set file_dofile [open source/dofiles/occ_scan_setup.dofile w]
puts $file_dofile
"# Define clock control scan chain. Should be
uncompressed if number of OCCs is high to reduce specified bits."
puts $file_dofile "add_scan_chains clock_control_chain grp1
occ_control_scan_in occ_control_scan_out"
close $file_dofile
```

OCC Insertion When Using an Existing Clock as the test_clk DFT Signal

To prevent D1 DRC errors when using an existing clock as the test_clock DFT signal, you must insert an OCC in the functional clock path that does not interfere with the clock gating cells.

[Figure 17-25](#) shows a clock, created using the command “add_clocks clk -pulse_always”, driving sequential elements.

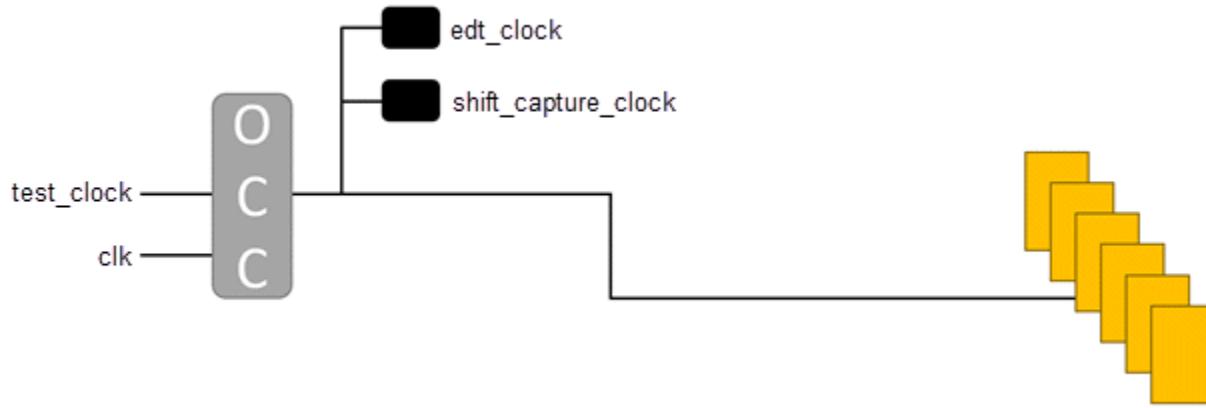
Figure 17-25. Original Clocking



You want to use the clk pin as the DFT test_clock, so you use the command, “add_dft_signals test_clock -source_nodes clk”. You also create the DFT signals edt_clock and shift_capture_clock from the test_clock DFT signal using the command “add_dft_signals {edt_clock shift_capture_clock} -create_from_other_signals”.

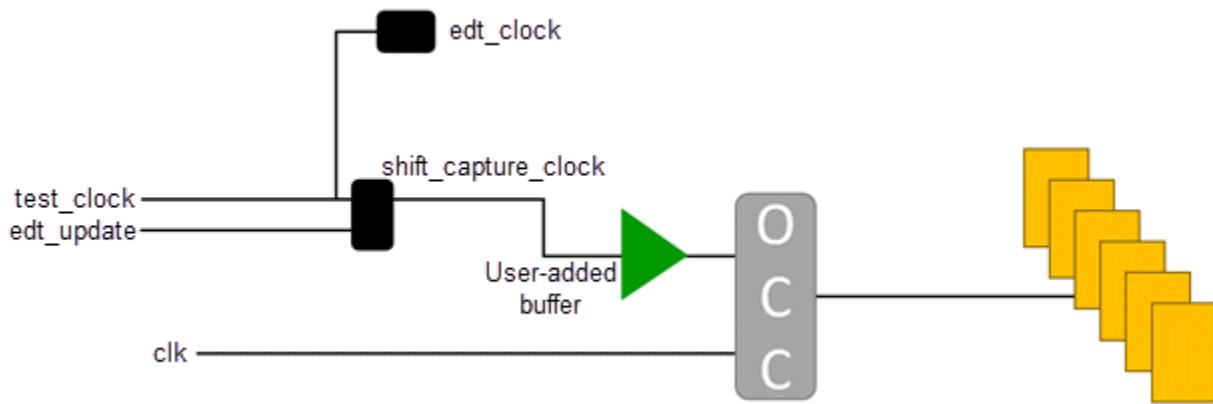
[Figure 17-26](#) shows the design with the clock gating cells that are added by the tool for edt_clock and shift_capture_clock. It also shows the OCC inserted in a location where it intercepts the test clock signal “test_clock,” which is an incorrect location for insertion.

Figure 17-26. Incorrect OCC Placement



To correct this issue, you must move the OCC to a location on the functional clock path that will not interfere with the direct, free-running clock to the clock gating cells. In some cases, you may need to add an anchor point for the OCC, such as a buffer. [Figure 17-27](#) shows the OCC in a suitable location at the output of a user-added buffer.

Figure 17-27. Correct OCC Placement



Core OCC Recommendation

The standard OCC is recommended for use in hierarchical cores to simplify timing closure and ATPG setup.

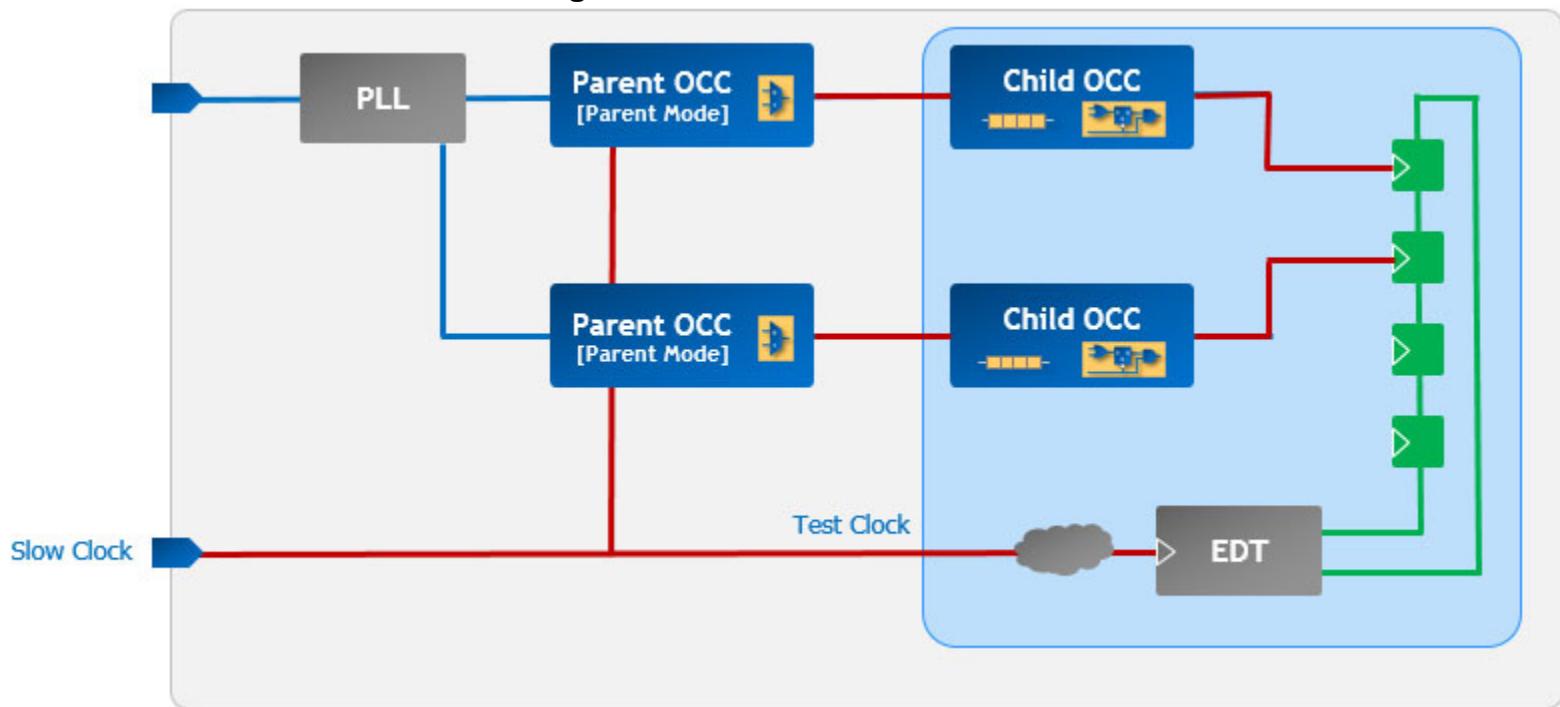
The standard OCC is preferred for core-level implementation instead of a parent/child OCC combination. It has a fast and slow clock, as well as a scan programmable shift register, which enable it to be used without the additional requirements that come with using a child OCC. See [The Standard OCC](#) section for more information.

Although it is not recommended for use in hierarchical cores, the parent/child OCC implementation is useful in designs that use a clock mesh or when a mux is not permitted inside the hierarchical core.

Shift Timing for Core-level Child OCC

The core-level child OCC has a single clock on the core boundary. The clock is used for shift as well as slow and fast capture. This feature creates additional requirements to meet shift timing in the core. [Figure 17-28](#) shows a core with two clocks, each with a child OCC that drives scan cells. There is also a test clock used during shift to control test logic, EDT logic, and other logic in the core.

Figure 17-28. Child OCC in Core



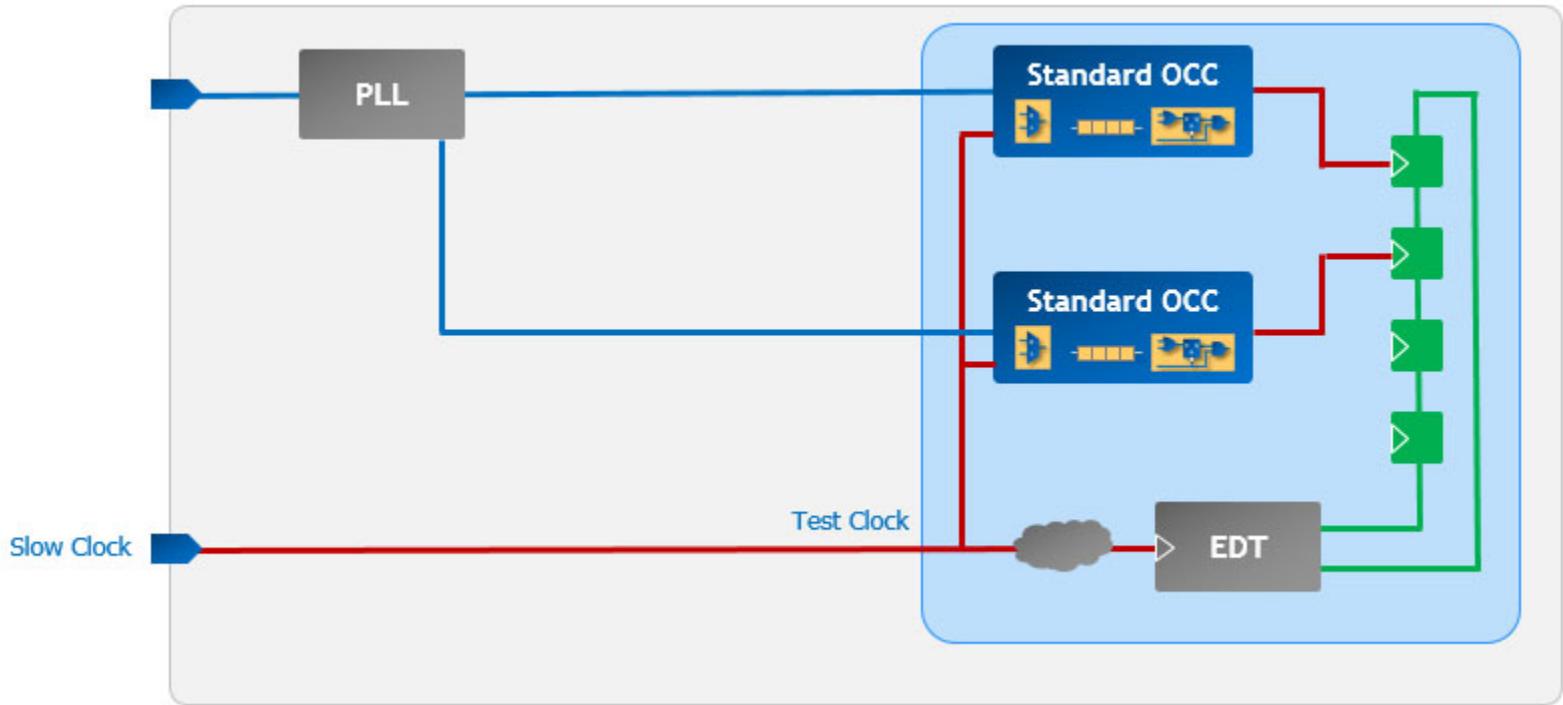
When you use a child OCC in the core, these items need to be considered and addressed during shift mode:

- There are multiple clocks on the boundary of the core that drive the scan cells.
- Timing closure at the core level must balance multiple clock paths, during shift mode, in addition to functional mode.
- The top-level clock path timing from the parent OCC to the core impacts shift and capture.
- Timing closure must balance functional clock paths for test clock.

Shift Timing for Core-level Standard OCC

The features of the standard OCC make it simpler to meet core-level shift timing requirements. [Figure 17-29](#) shows a core with a standard OCC implementation.

Figure 17-29. Standard OCC in Core



These are some of the benefits when you use a standard OCC at the core level:

- During shift, all scan cell clocks are sourced from test clock.
- Timing analysis can be closed in shift mode without skew between different clock branches.
- The top-level functional clock paths do not impact shift.
- There is no need to balance functional clock paths to test clock for shift mode.

Fast Capture ATPG Timeplate Setup

Another requirement that increases the complexity of using a child OCC for hierarchical cores is that two timeplates are required for accurate simulation of the core's fast capture patterns. This is due to the fact that the single child OCC clock is used for shift as well as slow and fast capture.

The timeplates are supplied to the tool with the following command:

```
set_procfile_name <file_with_two_timeplates>
```

By default, the first timeplate in the file is considered the default timeplate and is used for shift. If the first timeplate in the file is not for shift, use this command:

```
set_procedure_retargeting_options -timeplate <name_of_shift_timeplate>
```

The timeplate for capture is defined using this command:

```
set_external_capture_options -pll_cycles <name_of_capture_timeplate>
```

The timeplate for shift defines a single pulse on clock and for capture, it defines multiple clock pulses to match the functional frequency of the clock:

```
set time scale 1 ps;
timeplate shift_tp =
    force_pi 50;
    measure_po 100;
    pulse func_clk1 1600 3200; //156 MHz
    pulse func_clk2 1600 3200; //156 MHz
    period 6400;
end;

timeplate capture_tp =
    force_pi 50;
    measure_po 100;
    pulse func_clk1 800 1600, 4000 1600;           //312 MHz
    pulse func_clk2 400 800, 2000 800, 3600 800, 5200 800; //625 MHz
    period 6400;
end;
```


Appendix A Clock Gaters

The use of clock gaters to reduce power consumption is becoming a widely adopted design practice. Although effective for reducing power, clock gaters create new challenges for ATPG tools because of the additional complication introduced in clock paths. Among the challenges, the most frequently encountered in the tool is the C1 DRC violation. The C1 rule is fairly strict, requiring clock ports of scan cells to be at their off states when all clock primary inputs (PIs) are at their off states. Not all designs abide by this rule when there are clock gaters in clock paths.

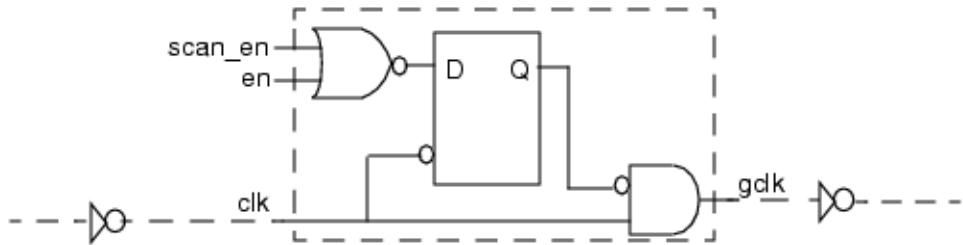
Basic Clock Gater Cell	663
Two Types of Embedding	665
Non-DRC Violation Clock Gater Embedding (Type-A)	666
Potential DRC Violation Clock Gater Embedding (Type-B)	667
Cascaded Clock Gaters	670
Level-2 Clock Gater	670
Example Combinations of Cascaded Clock Gaters	670
Clock Gater Configuration Support Summary	671

Basic Clock Gater Cell

There is one common type of clock gater cell. The cell (inside the dashed box) has three inputs: scan_en, en, and clk. The output of the cell is the gated clock, gclk.

Figure A-1 shows the structure of the cell.

Figure A-1. Basic Clock Gater Cell



The scan_en input is for test mode and is usually driven by a scan enable signal to enable shifting. The en input is for functional mode and is usually driven by internal control logic. Sometimes test_en is used to drive the scan_en input to avoid any trouble caused by the clock gater. However, this ends up keeping the clock gater always on and results in loss of coverage in the en fanin cone. The latch in the clock gater eliminates potential glitches. Depending on the embedding, there could be inversions of the clk signal both in front of clk and after gclk, which

in the figure are shown as inverters on the dashed lines representing the clk input and gelk output.

Related Topics

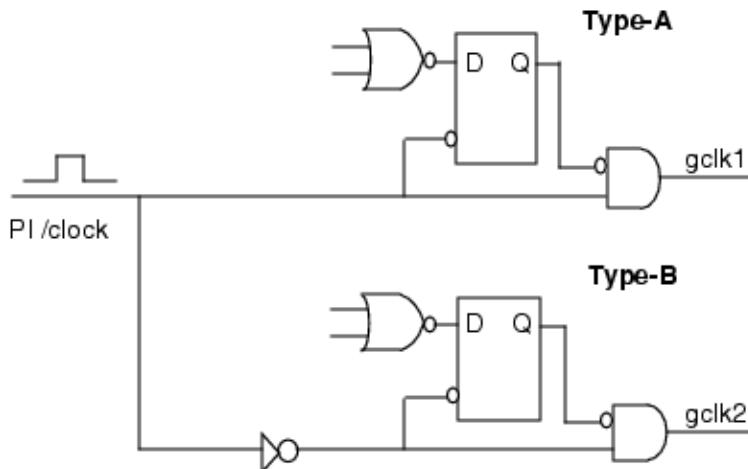
[Two Types of Embedding](#)

Two Types of Embedding

The key factor affecting whether the tool produces DRC violations is whether the clk signal gets inverted in front of the clk input to the cell.

To better understand the impact of such an inversion, it is helpful to understand two common ways designers embed the basic clock gater cell (see [Figure A-1](#)) in a design. [Figure A-2](#) shows the two possibilities, referred to as Type-A and Type-B.

Figure A-2. Two Types of Embedding for the Basic Clock Gater



In the figure, assume the PI /clock drives two clock gaters and that the off state of /clock is 0. The behavior of the two types of embeddings is as follows:

- **Type-A** — When PI /clock is at its off state, the latch in the clock gater is transparent, and the AND gate in the clock gater gets a controlling value at its input. As a result, the output gclk1 is controlled to a deterministic off state.
- **Type-B** — When /clock is at its off state, the latch in the clock gater is not transparent, and the AND gate in the clock gater gets a *non*-controlling value at its input. As a result, the output gclk2 is not controlled to a deterministic off state. Its off state value will depend on the output of the latch.

Note

 The preceding classification has nothing to do with whether there is inversion after gclk. It only depends on whether the off state of /clock can both make the latch transparent and control the AND gate.

Non-DRC Violation Clock Gater Embedding (Type-A)	666
Potential DRC Violation Clock Gater Embedding (Type-B).....	667

Non-DRC Violation Clock Gater Embedding (Type-A)

The tool fully supports the Type-A embedding for the following reasons.

- Since the latch is transparent, when scan_en is asserted at the beginning of load_unload, the clock gater is immediately turned on. Therefore, subsequent shifting is reliable and there will be no scan chain tracing DRC violations.
- Since gclk is controlled to a deterministic off state, there are no C1 DRC violations for downstream scan cells driven by it.
- The tool can pick up full fault coverage in the fanin cone logic of the en and scan_en inputs of the clock gater. This coverage is often too significant to be sacrificed.

Type-A embedding is the preferred design practice and is strongly recommended.

Tip  Type-A embedding does not necessarily mean the downstream scan cells have to be either all leading edge (LE) flip-flops or all trailing edge (TE) flip-flops. A designer can have both of them by inserting inversions after gclk.

Potential DRC Violation Clock Gater Embedding (Type-B)

Type-B embeddings have the following undesirable characteristics.

- When PI /clock is at its off state, the latch in the clock gater is not transparent. This means a change on its D input at the beginning of a cycle will not immediately be reflected at its Q output. In other words, the action of turning the clock gater on or off lags its instruction.
- The clock off state of gclk is not deterministic. It depends on what value the latch in the clock gater captures in the previous cycle. Therefore, the tool will issue a C1 DRC violation.

The kinds of problems that can arise due to these undesirable characteristics are described in the following sections.

Type-B Scan Chain Tracing Failures (T3 Violations)	667
Scan Cell Data Captures When Clocks are Off	668
Driving LE Flops With Type-B Clock Gaters.....	668

Type-B Scan Chain Tracing Failures (T3 Violations)

If the scan flip-flops downstream from the Type-B clock gater are LE triggered, the result will be scan chain tracing failures.

This is because the first shift edge in the load_unload is missed, as illustrated in [Figure A-3](#) and [Figure A-4](#). [Figure A-3](#) shows an example circuit where a Type-B clock gater drives both TE and LE scan flip-flops.

Note

 Assuming the off state of the PI /clock is 0, an inverter is needed after gclk to make the downstream flip-flop LE. The inverted gclk is indicated as gclk_b.

Figure A-3. Type-B Clock Gater Causes Tracing Failure

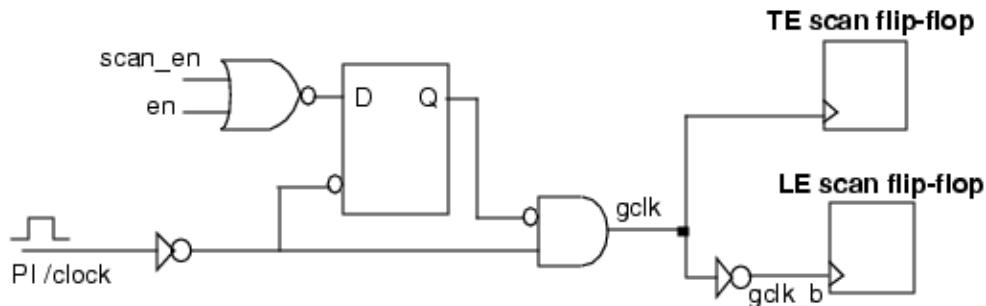


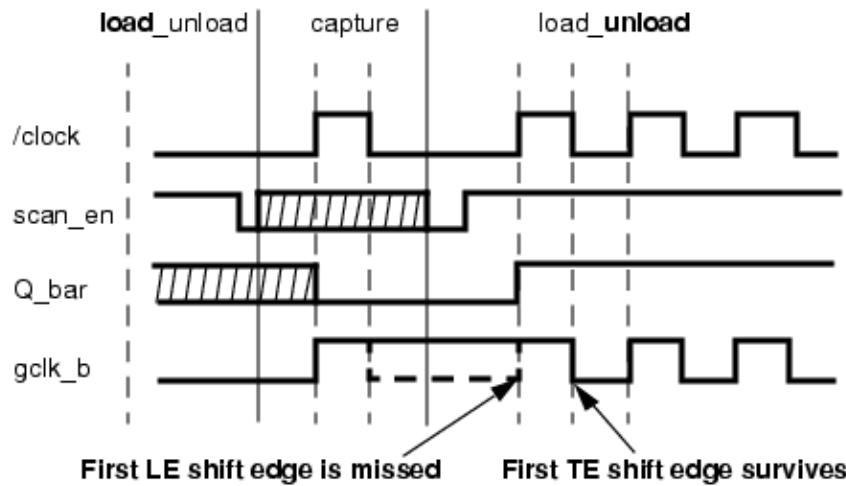
Figure A-4. Sample EDT Test Procedure Waveforms

Figure A-4 shows the timing diagram of several signals during capture and the beginning of load_unload. Suppose the latch in the Type-B clock gater captures a 1 in the last capture cycle (shown as Q_bar going low in the capture window). You can see that the first leading edge is suppressed because the latch holds its state after capture and into load_unload. Although scan_en is high at the beginning of load_unload, the output of the latch will not change until the first shift clock arrives. This lag between the “instruction” and the “action” causes the downstream LE scan flip-flops to miss their first shift edge. However, TE-triggered scan flip-flops are still able to trigger on the first trailing edge.

Tip

- i** To work around this problem, you can use test_en instead of scan_en to feed the clock gater. But be aware that this workaround loses fault coverage in the fanin cone of the clock gater’s en input.

Scan Cell Data Captures When Clocks are Off

T3 violations rarely occur in designs with Type-B clock gaters that drive LE scan flip-flops, typically because the downstream scan flip-flops in those designs are all made TE by synthesis tools. The tools handle Type-B clock gaters that drive TE scan flip-flops just fine, and change any downstream latches, LE nonscan flip-flops, RAMs, and POs to TIEX. This prevents C1 DRC violations, coverage loss in the en cone of the clock gater, and potential simulation mismatches when you verify patterns in a timing based simulator.

Driving LE Flops With Type-B Clock Gaters

When you have a Type-B clock gater driving LE flops in the chain, you must perform several steps in sequence.

Procedure

1. Initialize the clock gaters in one cycle of test_setup by driving scan_enable high and pulsing the clocks.

2. Issue the `set_stability_check` command using the following arguments:

`set_stability_check on -sim_static_atpg_constraints on`

3. Add ATPG constraints of “1” on the functional enable of the Type-B clock gaters using the `add_atpg_constraints` command using the following arguments:

`add_atpg_constraints 1 functional_enable_pin -static`

Results

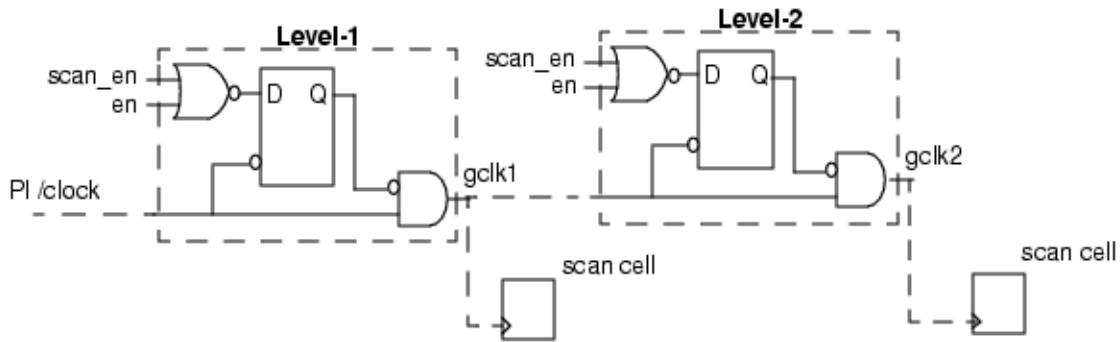
With the `-sim_static_atpg_constraints` set to on, the tool will simulate static ATPG constraints during DRC. In this case, the functional enable of the Type-B clock gater is constrained to “1” for ATPG analysis during design rules checking.

Cascaded Clock Gaters

One level of clock gating is sometimes not enough to implement a complex power controlling scheme.

Figure A-5 shows a two-level clock gating scheme using the clock gater cell. The dashed lines in the figure indicate places where inversions may exist. PI /clock is first gated by a level-1 gater and then the output gclk1 is further gated by a level-2 gater, generating the final clock gclk2.

Figure A-5. Two-level Clock Gating



Level-2 Clock Gater.....	670
Example Combinations of Cascaded Clock Gaters	670

Level-2 Clock Gater

In a two-level clock gating scheme, if the level-1 clock gater is of Type-A, then the clock off value of gclk1, being deterministic, has no problem propagating to the level-2 clock gater.

The latter can be understood based on the definition in “[Basic Clock Gater Cell](#)” on page 663.

However, when the level-1 gater is of Type-B, the tool cannot apply the basic definition to the level-2 gater directly, since the clock off value of gclk1 is undetermined. In this case, the type for the level-2 clock gater is defined by assuming the level-1 clock gater has been turned on. Therefore, even if the level-1 clock gater is of Type-B, the tool can still classify the level-2 clock gater.

Example Combinations of Cascaded Clock Gaters

There are four possible combinations of two-level cascading clock gaters and the support the tool provides for them.

- **Type-A Level-1 + Type-A Level-2** — This is the preferred combination, is fully supported by the tool, and is strongly recommended.

- **Type-A Level-1 + Type-B Level-2** — This can be reduced to a single level type-B case. All earlier discussion of type-B clock gaters applies. See “[Potential DRC Violation Clock Gater Embedding \(Type-B\)](#)” on page 667 for more information.
- **Type-B Level-1 + Type-A Level-2** — Avoid this combination. Scan chain tracing will be broken even if the downstream scan flip-flops of the level-2 gater are TE. The problem occurs when both level-1 and level-2 clock gaters capture a 1 in the last cycle of the capture window (meaning both clock gaters are turned off). Then the downstream scan flip-flops will miss both the first LE shift edge and the first TE shift edge in the subsequent load_unload. This is a design problem, not a tool limitation.
- **Type-B Level-1 + Type-B Level-2** — Supported only if all the downstream scan flip-flops are TE.

Clock Gater Configuration Support Summary

The following table summarizes the support that the tool provides for each clock gater configuration.

Table A-1. Clock Gater Configuration Support Summary

Gater Configuration	LE Scan Flip-flops Downstream	TE Flip-flops Downstream	Latches, RAM, ROM, PO Downstream
Type-A	Supported, strongly recommended.	Supported, strongly recommended.	Supported.
Type-B	T3 DRC violations. A design issue. Not supported.	Supported.	Changed to TIEX.
Type-A Level-1 + Type-A Level-2	Supported.	Supported.	Supported.
Type-A Level-1 + Type-B Level-2	Same as Type-B alone.	Same as Type-B alone.	Same as Type-B alone.
Type-B Level-1 + Type-A Level-2	T3 DRC violations. Not supported.	T3 DRC violations. Not supported.	Changed to TIEX.
Type-B Level-1 + Type-B Level-2	T3 DRC violations. Not supported.	Supported.	Changed to TIEX.

Appendix B

State Stability Issues

State stability refers to state elements that are stable, hold their values across shift cycles and patterns, and their values can be used for DRC (for example, scan chain tracing).

Debugging state stability is useful when you set up a state (for example, in a TAP controller register, in the test_setup procedure) that you use for sensitizing the shift path. Unless the state value is preserved for all shift cycles and all patterns, then scan chain tracing can fail. If you set up the correct value in the test_setup procedure but the value is changed due to the application of shift or the capture cycle, the value cannot be depended on. For such cases, you can debug state stability by identifying what caused a stable state element to unexpectedly change states.

Note

 The information in this appendix uses the `set_stability_check` command set to On (the default for this command), except as noted in Example 3 and Example 9.

Display of State Stability Data	673
State Stability Data Format	674
State Stability Examples	677
Example 1 — Basic Example	680
Example 2 — Multiple Cycles in Load_unload Prior to Shift	683
Example 3 — Drc_pattern Reporting for Pulse Generators	685
Example 4 — Single Post Shift	686
Example 5 — Single Post Shift with Cycles Between Main and Post Shift	687
Example 6 — Cycles After Apply Shift Statement in Load_unload	689
Example 7 — No Statements in Load_unload Prior to Apply Shift	691
Example 8 — Basic with Specified Capture Clock	691
Example 9 — Setting Stability Check to Off and All_shift	692
Example 10 — Pin Constraints, Test_setup, and State Stability	693
Example 11 — Single Pre Shift	694
Example 12 — Basic with Enhanced Stability Check for NCPs	696

Display of State Stability Data

You have several options for displaying state stability data.

Using the [set_gate_report](#) command with the Drc_pattern option, you can display simulation data for different procedures using the [report_gates](#) command or DFTVisualizer. The following examples show how to use the set_gate_report command:

```
set_gate_report drc_pattern test_setup
set_gate_report drc_pattern load_unload
set_gate_report drc_pattern shift
```

For “drc_pattern load_unload” and “drc_pattern shift”, the superimposed values for all applications of the procedure are reported. This means a pin that is 1 for only the first application of shift, but 0 or X for the second application of shift shows up as X.

When you set the Drc_pattern option of the set_gate_report command to State_stability as shown in the following example:

```
set_gate_report drc_pattern state_stability
```

the state stability report also includes the load_unload and shift data for the first application of these procedures.

When you debug state stability, you normally compare the state stability values (the values during the first application of the procedures) with the superimposed (stable) values from “drc_pattern load_unload” and “drc_pattern shift.”

State Stability Data Format

State stability data is only available after DRC and only if there is a test_setup procedure.

The format of the state stability data displayed for a pin is shown in the following example:

(ts)	(ld)	(shift)	(ld)	(shift)	(cell_con)	(cap)	(stbl)
// CLK I	(0)	(0)	(010~0)	(00)	(010)	(0)	(0X0) (0)

The first row lists the column labels in parentheses. The second row displays the pin name and five or more groups of data in parentheses. Each group has one or more bits of data corresponding to the number of events in each group.

Following is a description of the data columns in the state stability report:

- **ts** — Last event in the test_setup procedure. This is always one bit.
- **ld** — Any event in the load_unload procedure that is either before and/or after the apply shift statement. When load_unload is simulated, all primary input pins that are not explicitly forced in the load_unload procedure or constrained with an add_input_constraints command are set to X. Constrained pins are forced to their constrained values at the end of test_setup prior to load_unload. This is because even if your test_setup procedure gives those pins different values coming out of test_setup, in

the final patterns saved, an event is added to test_setup to force those pins to their constrained values. (You can see that event in the test_setup procedure if you issue write_procfile after completing DRC.) Consequently, for the first pattern, a pin has its constrained value going into load_unload. For all subsequent patterns, load_unload follows the capture cycle, during which the tool would have enforced the constrained value. Therefore, for all patterns, a constrained pin is forced to its constrained value going into load_unload and hence this value can be used for stability analysis. This is also true for the state stability analysis where the first application of load_unload is simulated.

The number of bits in this group depends on the number of events in the procedure prior to the apply shift statement. If there are no events prior to the apply shift statement, this group still exists (with one bit).

- **shift** — Apply shift statement (main shift or independent shift). If the load_unload procedure has multiple apply shift statements, then multiple (shift) groups are displayed. If this is also the main shift application (that is, the number of shifts applied is greater than one), the group includes a tilde (~). The values before the tilde correspond to the very first application of the shift procedure, with the number of bits corresponding to the number of primary input events in the shift procedure. The group without a tilde (~) is the independent shift.

The last bit (after the tilde) corresponds to the stable state after application of the last shift in the main shift statement. By default, the precise number of shifts is not simulated. The stable state shown corresponds to an infinite number of shift cycles; that is, in some cases, when for instance the depth of non-scan circuitry is deeper than the scan chain length, sequential circuitry that should be 1 or 0 after the first load_unload may show up as X. For more information, see the description of the “set_stability_check all_shift” command in “[Example 4 — Single Post Shift](#)” on page 686.

After the third group, there could be additional groups if there are the following:

- Multiple apply shift statements
- Events in the load_unload procedure after the apply shift statement
- **shdw_con** — Shadow_control procedure. This column is not shown in the examples. Look at the test procedure file to determine the meaning of this group.
- **cell_con** — Cell constraints. If there are cell constraints, an extra simulation event is added to show the value. This is always one bit.
- **cap** — Capture procedure. This is the simulation of the first capture cycle. Notice that this is not the simulation of pattern 0 or any specific pattern. Therefore, normally the capture clock going to 0X0 (or 1X1 for active low clocks) is displayed, indicating that the clock may or may not pulse for any given pattern. If you issue the “set_capture_clock -Atpg” command to force the tool to use a specific capture clock exactly once for every pattern, the capture cycle simulation is less pessimistic and “010” or “101” is reported.

- **stbl** — Final stable values after several iterations of simulating load_unload and capture procedures. If the value of a gate is always the same at the end of the test_setup and capture procedures, then its stable value is the same; otherwise, the stable value is X.

Note

 The skew_load, master_observe, or shadow_observe procedures are not simulated as part of state stability analysis, so their simulation data is not displayed in a state stability report. To view the simulation data for any of these procedures, use the “[set_gate_report Drc_pattern](#)” command with the specific *procedure_name* of interest.

State Stability Examples

This section contains examples of state stability reporting that show different behaviors in state stability analysis.

The examples are based on the design shown in [Figure B-1](#). The design has the following three clocks:

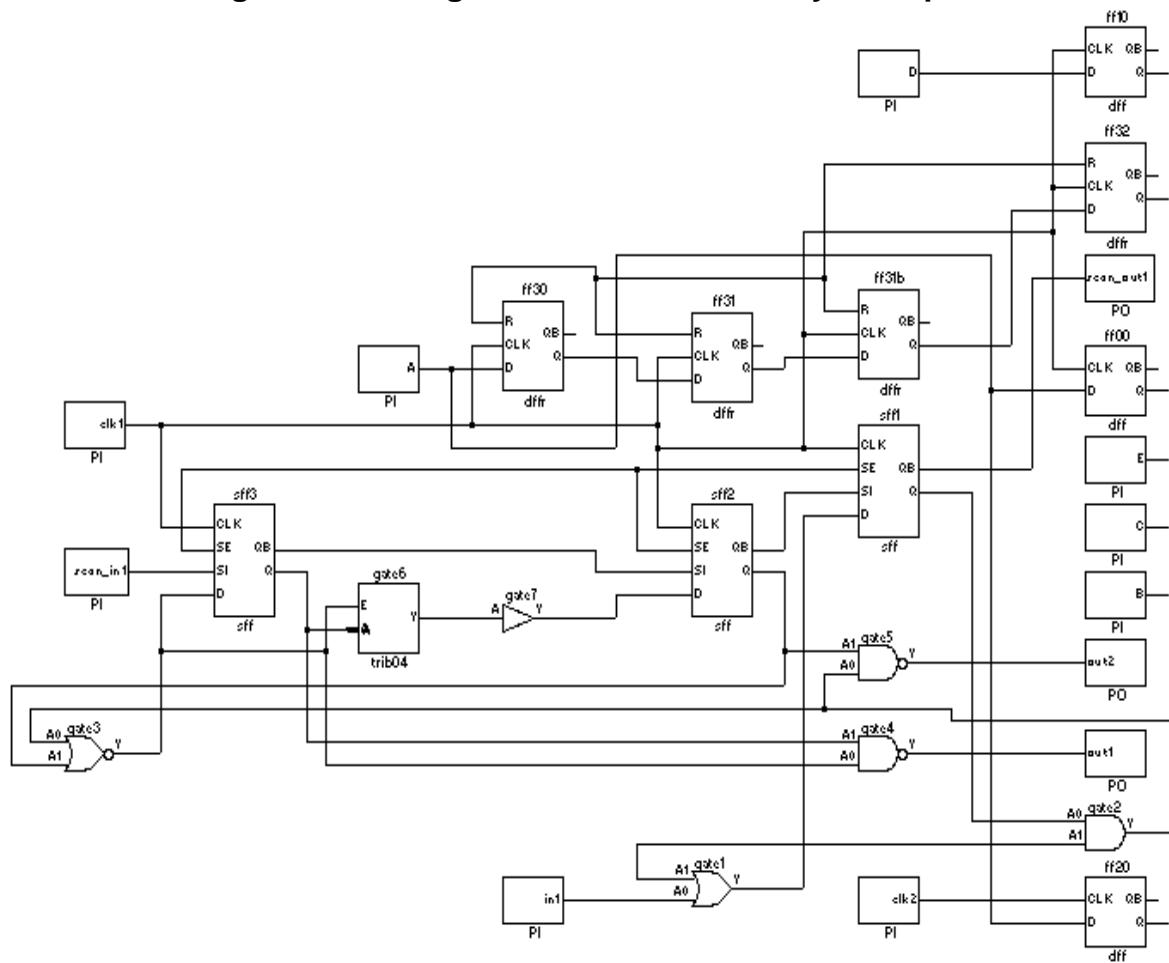
- **clk1** — The only scan clock
- **clk2** — Clocks a particular non-scan flip-flop
- **reset** — Resets four non-scan flip-flops connected as a register

The design has five pins (A, B, C, D, and E) that are exercised in the procedures to specifically show state stability analysis. The circuit has the following characteristics:

- Pin D has a C0 pin constraint. There are no other constrained pins.
- Non-scan flip-flop ff00 is clocked by clk1 and driven by pin A. The flip-flop is initialized in test_setup, but loses state when it is pulsed after test_setup.
- Non-scan flip-flop ff10 is clocked by clk1 and driven by pin D. Notice that this one gets initialized and “sticks” (is converted to TIE0) due to the pin constraint.
- Non-scan flip-flop ff20 is clocked by clk2 and driven by pin A.
- Non-scan flip-flop ff32 is the fourth flip-flop in a serial shift register. The first flip-flop in the register is ff30 and driven by pin A. The flip-flops are all clocked by clk1 and reset by the reset signal. ff32 is reset in test_setup, maintains state through the first application of shift, but then loses stability.
- There is one scan chain with three scan cells: sff1, sff2, and sff3.

The timeplate and procedures that were used for the first (basic) example are shown following the design.

Figure B-1. Design Used in State Stability Examples



```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 10;
    pulse clk1 20 10;
    pulse clk2 20 10;
    pulse reset 20 10;
    period 40;
end;

procedure capture =
    timeplate gen_tp1;
    cycle =
        force_pi;
        measure_po;
        pulse_capture_clock;
    end;
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse clk1;
        force C 0;
    end;
end;

procedure test_setup =
    scan_group grp1;
    timeplate gen_tp1;
    // First cycle, one PI event (force)
    cycle =
        force clk1 0;
        force clk2 0;
        force reset 0;
        force A 0;
        force B 0;
        force C 0;
        force D 0;
        force E 0;
    end;
    // Second cycle, two PI events (pulse on, pulse off)
    cycle =
        pulse reset;
        pulse clk2;
    end;
    // Third cycle, three PI events (force, pulse on, and pulse off)
    cycle =
        force A 1;
        pulse clk1;
    end;
end;

procedure load_unload =
    scan_group grp1;
```

```

timeplate gen_tp1;
// First cycle, one PI event (force)
cycle =
    force clk1 0;
    force clk2 0;
    force reset 0;
    force scan_en 1;
    force B 1;
    force C 1;
end;
apply shift 3;
end;

```

With the exception of Example 3, the only difference between these examples is the test procedure file. The procedures used in subsequent examples differ from the basic example as follows:

Example 1 — Basic Example	680
Example 2 — Multiple Cycles in Load_unload Prior to Shift.....	683
Example 3 — Drc_pattern Reporting for Pulse Generators	685
Example 4 — Single Post Shift	686
Example 5 — Single Post Shift with Cycles Between Main and Post Shift	687
Example 6 — Cycles After Apply Shift Statement in Load_unload	689
Example 7 — No Statements in Load_unload Prior to Apply Shift	691
Example 8 — Basic with Specified Capture Clock.....	691
Example 9 — Setting Stability Check to Off and All_shift	692
Example 10 — Pin Constraints, Test_setup, and State Stability.....	693
Example 11 — Single Pre Shift.	694
Example 12 — Basic with Enhanced Stability Check for NCPs	696

Example 1 — Basic Example

In the basic example, you invoke the tool on the design and issue the following commands.

```

add_clocks 0 clk1 clk2 reset
add_scan_groups grp1 scan1.testproc
add_scan_chains c0 grp1 scan_in1 scan_out1
add_input_constraints D -c0
set_gate_report drc_pattern state_stability
set_system_mode analysis
report_gates A B C D E

```

```

...
//          (ts) (ld) (shift) (cap) (stbl)
//      A   O    ( 1) ( 1)(111~1) (XXX) ( X )
//      B   O    ( 0) ( 1)(111~1) (XXX) ( X )
//      C   O    ( 0) ( 1)(000~0) (XXX) ( X )
//      D   O    ( 0) ( 0)(000~0) (000) ( 0 )
//      E   O    ( 0) ( 0)(000~0) (XXX) ( X )

```

The results of this operation are:

- Pins A through E are explicitly forced in test_setup.
- Pins A, B, C, and E are forced in load_unload.
- Pin D is constrained (but not forced in load_unload) and is the only one of the pins with an explicit value during capture.

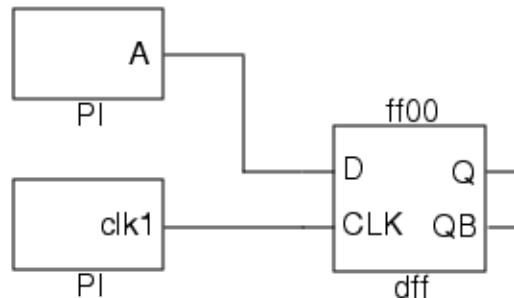
A typical initialization problem is illustrated in ff00. Figure B-2 shows the relevant circuitry and statements in the test_setup procedure that seemingly initialize this flip-flop.

Figure B-2. Typical Initialization Problem

```

procedure test_setup =
  . . .
  // Third cycle...
  cycle =
    force A 1;
    pulse clk1;
  end;
  . . .

```



In this case, you might expect the output to always be 1 after initialization because the third cycle of the test_setup procedure forced a 1 on input A and pulsed clk1. For comparison purposes, following is the reported state_stability data together with the data reported for load_unload and shift. Notice especially the Q output:

```

//  /ff00  dff
//          (ts) (ld) (shift) (cap) (stbl)      (ld) (shift)
//      CLK   I    ( 0) ( 0)(010~0) (0X0) ( 0 )      ( 0) ( 010) (010)
//      D     I    ( 1) ( X)(XXX~X) (XXX) ( X )      ( X) ( XXX) (XXX)
//      Q     O    ( 1) ( 1)(1XX~X) (XXX) ( X )      ( 1) ( XXX) (XXX)
//      QB    O    ( 0) ( 0)(0XX~X) (XXX) ( X )      ( 0) ( XXX) (XXX)

```

You can see from the state stability display that, after test_setup, the output of Q is set to 1. In the first application of load_unload it is still 1, but it goes to X during the first shift. Compare this to what is shown for “drc_pattern load_unload” and “drc_pattern shift”.

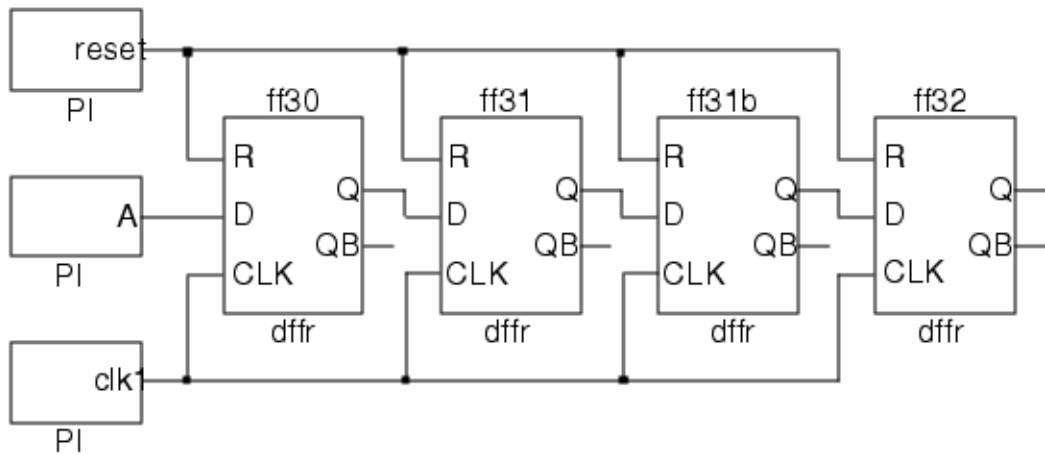
Example 1 — Basic Example

A stable initialization value can be better achieved by doing for ff00 something similar to what occurs for ff10, where the D input is connected to the constrained D pin:

```
//  /ff00  dff
//          (ts) (ld) (shift) (cap) (stbl)      (ld) (shift)
//    CLK   I    ( 0) ( 0)(010~0)(0X0) ( 0 )    ( 0) ( 010)  (010)
//    D     I    ( 1) ( X)(000~0)(000) ( X )    ( X) ( 000)  (000)
//    Q     O    ( 1) ( 1)(000~0)(000) ( X )    ( 0) ( 000)  (000)
//    QB    O    ( 0) ( 0)(111~1)(111) ( X )    ( 1) ( 111)  (111)
```

Another interesting observation can be made for ff32. This flip-flop is at the end of a 4-bit shift register where all the flip-flops are reset during test_setup as shown in [Figure B-3](#).

Figure B-3. Three-bit Shift Register (Excerpted from Figure D-1)



```
procedure test_setup =
    . . .
    // Second cycle...
    cycle =
        pulse reset;
    . . .
end;
```

Notice how Q in this case is stable for the first application of load_unload and shift, but the stable state after the last shift (after ~) is X. This is due to an optimization the tool does by default for the state_stability check. (Compare this output to example [Example 9 — Setting Stability Check to Off and All_shift.](#))

```
//  /ff32  dfffr
//          (ts) (ld) (shift) (cap) (stbl)      (ld) (shift)
//    R     I    ( 0) ( 0)(000~0)(0X0) ( 0 )    ( 0) ( 000)  (000)
//    CLK  I    ( 0) ( 0)(010~0)(0X0) ( 0 )    ( 0) ( 010)  (010)
//    D     I    ( 0) ( 0)(000~0)(XXX) ( X )    ( 0) ( 000)  (XXX)
//    Q     O    ( 0) ( 0)(000~1)(XXX) ( X )    ( 0) ( 000)  (XXX)
//    QB    O    ( 1) ( 1)(111~1)(XXX) ( X )    ( 1) ( 111)  (XXX)
```

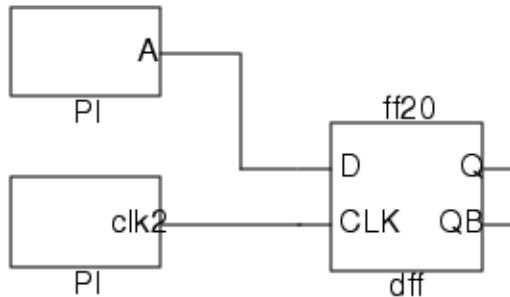
Non-scan flip-flop ff20 is clocked by clk2, which is not a shift clock. This flip-flop is also initialized during test_setup as shown in [Figure B-4](#).

Figure B-4. Initialization with a Non-Shift Clock

```

procedure test_setup =
    // First cycle...
    cycle =
        ...
        force A 0;
        ...
    end;
    // Second cycle...
    cycle =
        ...
        pulse clk2;
    end;
    ...

```



The Q output is disturbed during capture, not during shift, because this element is not exercised during shift:

```

//  /ff20  dff
//          (ts) (ld) (shift) (cap) (stbl)      (ld) (shift)
//      CLK   I      ( 0) ( 0)(000~0)(0X0) ( 0 )      ( 0) ( 000)      (000)
//      D     I      ( 1) ( X)(XXX~X)(XXX) ( X )      ( X) ( XXX)      (XXX)
//      Q     O      ( 0) ( 0)(000~0)(0XX) ( X )      ( 0) ( 000)      (XXX)
//      QB    O      ( 1) ( 1)(111~1)(1XX) ( X )      ( 1) ( 111)      (XXX)

```

Notice that the load_unload and shift data for ff32 and ff20 is almost identical (except for the clock data), but that the state_stability data enables you to see that they become unstable in very different ways.

Example 2 — Multiple Cycles in Load_unload Prior to Shift

The main difference between this example and the basic example is that in the load_unload procedure there are multiple cycles prior to the apply shift statement (statements that are new in this example are highlighted in bold).

Example 2 — Multiple Cycles in Load_unload Prior to Shift

```

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tpl ;
    // First cycle, one PI event (force)
    cycle =
        force clk1 0 ;
        force clk2 0 ;
        force reset 0 ;
        force scan_en 1 ;
        force B 1;
        force C 1;
        force E 1;
    end ;
    // Second cycle, three PI events
    (force, pulse on, pulse off)
    cycle =
        force E 0;
        pulse clk2;
    end ;
    // Third cycle, two PI events
    (pulse on, pulse off)
    cycle =
        pulse clk2;
    end ;
    apply shift 3;
end;

```

As a result, multiple events are displayed in the second group of state stability data. Notice there are now three cycles. The following gate report excerpts show six bits of data (in bold), corresponding to the total number of events. The first bit is from the first cycle (one event), the next three bits are from the second cycle (three events), and the last two bits are from the third cycle, which has two events.

```

// /E primary_input
//           (ts)( ld )(shift)(cap)(stbl)
//   E      O  ( 0)(100000) (000~0) (XXX) (   X)
// /A primary_input
//           (ts)( ld )(shift)(cap)(stbl)
//   A      O  ( 1)(xxxxxxxx) (XXX~X) (XXX) (   X)
// /clk2 primary_input
//           (ts)( ld )(shift)(cap)(stbl)
//   clk2   O  ( 0)(001010) (000~0) (0X0) (   0)

// /ff20  dff
//           (ts)( ld )(shift)(cap)(stbl)
//   CLK    I  ( 0)(001010) (000~0) (0X0) (   0)
//   D      I  ( 1)(xxxxxxxx) (XXX~X) (XXX) (   X)
//   Q      O  ( 0)(00xxxx) (XXX~X) (XXX) (   X)
//   QB     O  ( 1)(11xxxx) (XXX~X) (XXX) (   X)

```

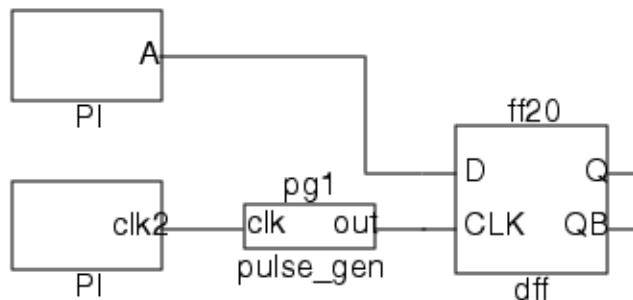
Notice how A goes to X for the load_unload simulation. This is because it is not explicitly forced in the load_unload procedure (or constrained with an [add_input_constraints](#) command).

Example 3 — Drc_pattern Reporting for Pulse Generators

If a circuit contains a pulse generator (PG) with user-defined timing, the tool performs additional simulation steps for the PG's output changes. When a rising edge event occurs at its input, a PG outputs a 1 after a certain delay, which the tool simulates as an additional event. After another delay, the PG's output signal returns to 0, which is also simulated as a separate event. Both output events are added to reports within a pair of brackets ([]) following the input event.

Suppose the non-scan flip-flop ff20 of the preceding example is clocked by a PG as shown in Figure B-5. Excerpts below the figure show how the PG events would appear in gate reports.

Figure B-5. Clocking ff20 with a Pulse Generator



```

// /clk2 primary_input
//           (ts) (      ld      ) (shift) (  cap ) (stbl)
//     clk2  O  ( 0 ) (001[11]01[11]0) (000~0) (0X[X]0) (   0)  /pg1/clk

// /pg1  pulse_gen
//           (ts) (      ld      ) (shift) (  cap ) (stbl)
//     clk   I  ( 0 ) (001[11]01[11]0) (000~0) (0X[X]0) (   0)  /clk2
//     out   O  ( 0 ) (000[10]00[10]0) (000~0) (00[X]0) (   0)  /ff20/CLK

// /ff20  dff
//           (ts) (      ld      ) (shift) (  cap ) (stbl)
//     CLK   I  ( 0 ) (000[10]00[10]0) (000~0) (00[X]0) (   0)  /pg1/out
//     D     I  ( 1 ) (XXX[XX]XX[XX]X) (XXX~X) (XX[X]X) (   X)  /A
//     Q     O  ( 0 ) (000[XX]XX[XX]X) (XXX~X) (XX[X]X) (   X)
//     QB    O  ( 1 ) (111[XX]XX[XX]X) (XXX~X) (XX[X]X) (   X)

```

The rising edge events on clk2 initiate pg1's two output pulses (highlighted in bold). Notice the pulses are not shown simultaneous with the input changes that caused them. This is an exception to the typical display of output changes simultaneous with such input changes, as shown for ff20. Notice also how the active clock edge at ff20's CLK input is one event later than clk2's active edge and is seen to be a PG signal due to the brackets.

For an introduction to pulse generators, see “[Pulse Generators](#)” on page 132. For detailed information about the tool’s pulse generator primitive, see “[Pulse Generators with User Defined Timing](#)” in the *Tessent Cell Library Manual*.

Example 4 — Single Post Shift

This example has multiple cycles in the load_unload procedure. What is new is the single post shift (highlighted in bold).

```

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // First cycle, one PI event (force)
    cycle =
        force clk1 0 ;
        force clk2 0 ;
        force reset 0 ;
        force scan_en 1 ;
        force B 1;
        force C 1;
    end ;
    // Second cycle, three PI events (force, pulse on, pulse off)
    cycle =
        force E 0;
        pulse clk2;
    end;
    apply shift 2;
    apply shift 1;
end;

```

In this case, the state stability data has an additional group (shown in bold) between the main shift and the capture cycle. This corresponds to the first application of the post shift:

```

// /ff32 dffr
//          (ts)( ld )(shift)(shift)(cap)(stbl)
//      R      I ( 0)(0000)(000~0)( 000)(0X0)( 0)  /reset
//      CLK    I ( 0)(0000)(010~0)(010 )(0X0)( 0)  /clk1
//      D      I ( 0)(0000)(000~X)( XXX)(XXX)(  X)  /ff31b/Q
//      Q      O ( 0)(0000)(000~X)( XXX)(XXX)(  X)
//      QB     O ( 1)(1111)(111~X)( XXX)(XXX)(  X)

```

You can see that ff32 is really stable during the first application of shift. If you use the [set_stability_check All_shift](#) command in this case, the output is slightly different:

```

set_stability_check all_shift
set_system_mode setup
set_system_mode analysis
report_gates ff32

```

```
// /ff32 dffr
//           (ts) ( 1d ) (shift) (shift) (cap) (stbl)
//   R      I  ( 0) (0000) (000~0) ( 000) (0X0) (    0)  /reset
//   CLK    I  ( 0) (0000) (010~0) (010) (0X0) (    0)  /clk1
//   D      I  ( 0) (0000) (000~1) ( 1XX) (XXX) (    X)  /ff31b/Q
//   Q      O  ( 0) (0000) (000~0) ( 011) (1XX) (    X)
//   QB     O  ( 1) (1111) (111~1) ( 100) (0XX) (    X)
```

Notice how ff32 is now 0 throughout the main shift application, but is set to 1 during the post shift. This is due to how A is set to 1 in test_setup and this pulse is clocked through.

Example 5 — Single Post Shift with Cycles Between Main and Post Shift

In this example, a post shift exists but there is an additional cycle (shown in bold font) between the main shift and the post shift. This causes yet another group of data to be displayed when you report the state stability data.

```

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // First cycle, one PI event (force)
    cycle =
        force clk1 0 ;
        force clk2 0 ;
        force reset 0 ;
        force scan_en 1 ;
        force B 1;
        force C 1;
    end ;
    // Second cycle, three PI events (force, pulse on, pulse off)
    cycle =
        force A 1;
        pulse clk2;
    end ;
    apply shift 3;
    // Third cycle, three PI events (force, pulse on, pulse off)
    cycle =
        force C 1;
        force A 0;
        pulse clk2;
    end ;
    apply shift 1;
end;

procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    cycle =
        force_sci ;
        measure_sco ;
        pulse clk1 ;
        force C 0;
    end;
end;

```

The fourth data group (highlighted in bold) represents the cycle between the two applications of shift (for the first application of the load_unload procedure). The fifth data group represents the application of the post shift, and the last (sixth) group represents capture. Notice how pins A and C vary state due to the values forced on them in the load_unload and shift procedures.

```

// /A primary_input
//          (ts)( ld )(shift)( 1d )(shift)(cap)(stbl)
//          A      ( 1)(X111)(111~1)(000)( 000 )(XXX)(   X)
// /B primary_input
//          (ts)( ld )(shift)( 1d )(shift)(cap)(stbl)
//          B      O  ( 0)(1111)(111~1)(111)( 111 )(XXX)(   X)
// /C primary_input
//          (ts)( ld )(shift)( 1d )(shift)(cap)(stbl)
//          C      O  ( 0)(1111)(000~0)(111)( 000 )(XXX)(   X)

```

Example 6 — Cycles After Apply Shift Statement in Load_unload

Notice also that clk2 in this case is pulsed during load_unload only, not in test_setup. Here is the modified test_setup procedure (with the clk2 pulse removed from the second cycle):

```

procedure test_setup =
  scan_group grp1 ;
  timeplate gen_tp1 ;
  // First cycle, one event (force)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0;
    force A 0;
    force B 0;
    force C 0;
    force D 0;
    force E 0;
  end ;
  // Second cycle, two events (pulse on, pulse off)
  cycle =
    pulse reset ;
  end;
  // Third cycle, three events (force, pulse on, pulse off)
  cycle =
    force A 1;
    pulse clk1 ;
  end;
end;

```

This results in the following behavior for ff20, which is clocked by clk2:

```

//  /ff20  dff
//          (ts)( ld )(shift)( ld )(shift)(cap)(stbl)
//      CLK  I  ( 0)(0010)(000~0)(010)( 000 )(0X0)( 0)  /clk2
//      D    I  ( 1)(X111)(111~1)(000)( 000 )(XXX)(  X)  /A
//      Q    O  ( X)(XX11)(111~1)(100)( 000 )(0XX)(  X)
//      QB   O  ( X)(XX00)(000~0)(011)( 111 )(1XX)(  X)

```

Notice how the state of this flip-flop is disturbed during capture.

Example 6 — Cycles After Apply Shift Statement in Load_unload

This example reuses the load_unload procedure of the basic example (with only one apply shift statement), but adds three new cycles (shown in bold font) after the application of shift.

Example 6 — Cycles After Apply Shift Statement in Load_unload

```

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    // First cycle, one PI event (force)
    cycle =
        force clk1 0 ;
        force clk2 0 ;
        force reset 0 ;
        force scan_en 1 ;
        force B 1;
        force C 1;
    end ;
    apply shift 3;
    // Second cycle, one PI event (force)
    cycle =
        force C 1;
    end ;
    // Third cycle, three PI events (force, pulse on, pulse off)
    cycle =
        force C 0;
        pulse clk2;
    end ;
    // Fourth cycle, one PI event (force)
    cycle =
        force C 1;
    end ;
end;
procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    cycle =
        force_sci ;
        measure_sco ;
        pulse clk1 ;
        force C 0;
    end;
end;

```

In this case, the second data group in the state stability report represents the one event in the cycle before the apply shift statement. The fourth data group represents the events in the three cycles after the apply shift statement (but still for the first application of load_unload). The first bit is the one event in the second cycle, the next three bits represent the third cycle, and the last bit represents the fourth cycle:

```

// /C primary_input
//           (ts)(ld)(shift)( ld )(cap)(stbl)
//           C   O  ( 0 )( 1 )(000~0)(10001)(XXX)(   X)
// /clk2  primary_input
//           (ts)(ld)(shift)( ld )(cap)(stbl)
//           clk2 O  ( 0 )( 0 )(000~0)(00100)(0X0) (    0)  /ff20/CLK

```

Example 7 — No Statements in Load_unload Prior to Apply Shift

In this example, there are no statements in the load_unload procedure prior to the apply shift statement and scan_en is forced in the shift procedure instead of in a separate cycle in load_unload. Other procedures are as in the basic example.

Here are the modified load_unload and shift procedures:

```
procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tpl ;
    apply shift 3;
end;

procedure shift =
    scan_group grp1 ;
    timeplate gen_tpl ;
    cycle =
        force_scan_en 1 ;
        force_sci ;
        measure_sco ;
        pulse clk1 ;
        force C 0;
    end;
end;
```

In this case, the report still shows an event for load_unload in the second data group. This makes it easier to see differences between the end of test_setup and the entry into the first load_unload. Such differences can occur because during load_unload the tool sets to X any primary input pins that are not constrained with an [add_input_constraints](#) command or explicitly forced in the load_unload procedure. This is the case for pin A in this example:

```
//  /A primary_input
//          (ts) (ld) (shift) (cap) (stbl)
//      A      O  ( 1) ( X)(XXX~X)(XXX) (  X)  /ff30/D  /ff20/D  /ff00/D
//  /ff20  dff
//          (ts) (ld) (shift) (cap) (stbl)
//      CLK    I  ( 0) ( 0)(000~0)(0X0) (  0)  /clk2
//      D      I  ( 1) ( X)(XXX~X)(XXX) (  X)  /A
//      Q      O  ( 0) ( 0)(000~0)(0XX) (  X)
//      QB     O  ( 1) ( 1)(111~1)(1XX) (  X)
```

Example 8 — Basic with Specified Capture Clock

If you specify a capture clock using the set_capture_clock command with the -Atpg switch, the format looks slightly different: the specified clock shows up as “010” (or “101”) during capture instead of “0X0” (or “1X1”).

Example 9 — Setting Stability Check to Off and All_shift

If the “[set_capture_clock](#) clk1 -Atpg” command is used, the state stability display for a scan cell clocked by clk1 in the example design looks like the following:

```
//  /sff1  sff
//          (ts) (1d) (shift) (cap) (stbl)
//    SE    I  ( X) ( 1) (111~1) (XXX) (   X)  /scan_en
//    D     I  ( X) ( X) (XXX~X) (XXX) (   X)  /gate1/Y
//    SI    I  ( X) ( X) (XXX~X) (XXX) (   X)  /sff2/QB
//    CLK   I  ( 0) ( 0) (010~0) (010) (   0)  /clk1
//    Q     O  ( X) ( X) (XXX~X) (XXX) (   X)  /gate2/A0
//    QB    O  ( X) ( X) (XXX~X) (XXX) (   X)  /scan_out1
```

Example 9 — Setting Stability Check to Off and All_shift

The preceding examples, except as noted in example 3, used the default On setting of the `set_stability_check` command. This command has two other settings: All_shift and Off.

For All_shift, the tool simulates the exact number of shifts. This means for situations where particular events take place during shift, the tool will simulate these exactly rather than simulating the stable state after an “infinite” number of shifts. The following state_stability displays show the difference between using On and using All_shift for ff32 and the procedures of the basic example (values of interest are highlighted in bold):

`set_stability_check On:`

```
//  /ff32  dffr
//          (ts) (1d) (shift) (cap) (stbl)
//    R     I  ( 0) ( 0) (000~0) (0X0) (   0)  /reset
//    CLK  I  ( 0) ( 0) (010~0) (0X0) (   0)  /clk1
//    D     I  ( 0) ( 0) (000~X) (XXX) (   X)  /ff31b/Q
//    Q     O  ( 0) ( 0) (000~X) (XXX) (   X)
//    QB    O  ( 1) ( 1) (111~X) (XXX) (   X)
```

`set_stability_check All_shift:`

```
//  /ff32  dffr
//          (ts) (1d) (shift) (cap) (stbl)
//    R     I  ( 0) ( 0) (000~0) (0X0) (   0)  /reset
//    CLK  I  ( 0) ( 0) (010~0) (0X0) (   0)  /clk1
//    D     I  ( 0) ( 0) (000~X) (XXX) (   X)  /ff31b/Q
//    Q     O  ( 0) ( 0) (000~1) (1XX) (   X)
//    QB    O  ( 1) ( 1) (111~0) (0XX) (   X)
```

In the All_shift case, notice how the stable state after shift differs from the On case. After exactly three applications of the shift procedure, the state is 1, but after “infinite” applications of the shift procedure, it is X.

When stability checking is set to off, the tool reports only dashes:

```
// /ff32 dffr
//   R      I  (-)  /reset
//   CLK    I  (-)  /clk1
//   D      I  (-)  /ff31/Q
//   Q      O  (-)
//   QB     O  (-)
```

Example 10 — Pin Constraints, Test_setup, and State Stability

In this example, the test_setup procedure is modified so that pin D, which is constrained to C0 by an add_input_constraints command, is not forced in the first cycle, but *is* forced to 1 in the second cycle. It is never forced to its constrained value 0.

It is a good practice to always force the constrained pins to their constrained state at the end of the test_setup procedure. If you do not do that, the tool will add an additional cycle to the test_setup procedure when you write out the patterns. This recommended practice is not followed in this example:

```
procedure test_setup =
  scan_group grp1
  timeplate gen_tp1 ;
  // First cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force clk1 0 ;
    force clk2 0 ;
    force reset 0 ;
    force A 0 ;
    force B 0 ;
    force C 0 ;
    force E 0 ;
    pulse clk1 ;
  end ;
  // Second cycle, three PI events (force, pulse on, pulse off)
  cycle =
    force D 1 ;
    pulse clk1 ;
  end ;
end;
```

Notice what happens during test_setup:

```
set_gate_report drc_pattern test_setup
report_gates D ff10
```

Example 11 — Single Pre Shift

```

//  /D      primary_input
//      D      O      /ff10/D
//
//      Cycle:  0    1
//      ----- --- ---
//                  23 467
//      Time:   000 000
//      ----- --- ---
//      D      XXX 111
//
//  /ff10  dff
//      CLK  I      /clk1
//      D    I      /D
//      Q    O
//      QB   O
//
//      Cycle:  0    1
//      ----- --- ---
//                  23 467
//      Time:   000 000
//      ----- --- ---
//      CLK    010 010
//      D     XXX 111
//      Q     XXX X11
//      QB    XXX X00

```

Then, for state stability, notice how D changes from 1 to 0 between test_setup and the first application of load_unload. This is because of the pin constraint to 0 on this pin:

```

set_gate_report drc_pattern state_stability
report_gates D ff10

//  /D      primary_input
//          (ts) (ld) (shift) (cap) (stbl)
//      D      O      ( 1) ( 0) (000~0) (000) (    X)  /ff10/D
//  /ff10  dff
//          (ts) (ld) (shift) (cap) (stbl)
//      CLK  I      ( 0) ( 0) (010~0) (0X0) (    0)  /clk1
//      D    I      ( 1) ( 0) (000~0) (000) (    X)  /D
//      Q    O      ( 1) ( 1) (100~X) (XXX) (    X)
//      QB   O      ( 0) ( 0) (011~X) (XXX) (    X)

```

Example 11 — Single Pre Shift

In the load_unload procedure for this example, a single pre shift is followed by one cycle and then the main shift. Also, pins C and E vary in the different cycles of the load_unload and shift operations. The differences from the basic versions of these procedures are highlighted in bold.

```

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tpl ;
    // First cycle, one event (force)
    cycle =
        force clk1 0 ;
        force clk2 0 ;
        force reset 0 ;
        force scan_en 1 ;
        force B 1;
        force C 1;
        force E 1;
    end ;
    apply shift 1;      // Pre shift
    // Second cycle, three PI events (force, pulse on, pulse off)
    cycle =
        force C 0;
        force E 0;
        pulse clk2;
    end;
    apply shift 2;      // Main shift
end;

procedure shift =
    scan_group grp1 ;
    timeplate gen_tpl ;
    cycle =
        force_sci ;
        measure_sco ;
        pulse clk1 ;
        force C 0;
    end;
end;

```

The third group in the state_stability display (highlighted in bold) is the pre-shift. The fourth group is the cycle between the shift applications, and the fifth group is the main shift.

```

set_gate_report drc_pattern state_stability

report_gates C E clk1 clk2

//   /C      primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)
//           C      O  ( 0)( 1)( 000 )(000)(000~0)(XXX)(   X)
//           /E      primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)
//           E      O  ( 0)( 1)( 111 )(000)(000~0)(XXX)(   X)
//           /clk1   primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)(stbl)
//           clk1   O  ( 0)( 0)( 010 )(000)(010~0)(0X0)(   0)/ff32/CLK
// /ff31b/CLK...
//           /clk2   primary_input
//           (ts)(ld)(shift)( ld)(shift)(cap)
//           clk2   O  ( 0)( 0)( 000 )(010)(000~0)(0X0) /ff20/CLK

```

Example 12 — Basic with Enhanced Stability Check for NCPs

If you enable the enhanced stability check for NCPs using “set_stability_check -sim_capture_procedures”, the report_gates output format looks slightly different. Notice that for the “(cap)” column of the stability report below, the tool presents two simulation values separated with a tilde character (~). The first value is the value at the beginning of capture, and the second value is the value at the end of capture. The tool determines these values by taking the common value among all NCP simulation values. Since different NCPs can be of different length in sequence, the tool uses the tilde separator to skip the events in between in order to highlight only the first and last value.

There is only one exception to the display of the tilde separator, which is when the simulation is a constant 1 or 0 among all NCP cycles. Then instead of showing the tilde separator, the middle value is the same constant value (1 or 0). The first and last value is usually of the most interest when debugging stability analysis.

In the following alteration to the basic example on [page 680](#), the tool is already invoked on the design, and you enter the following commands:

```
add_clocks 0 clk1 clk2 reset
add_scan_groups grp1 scan1.testproc
add_scan_chains c0 grp1 scan_in1 scan_out1
add_input_constraints D -c0
set_gate_report drc_pattern state_stability
set_stability_check -sim_capture_procedures on // enables more detailed analysis
set_system_mode analysis
report_gates A B C D E

...
//          (ts) (ld) (shift) (cap) (stbl)
//      A    O    ( 1) ( 1)(111~1) (X-X) (   X  )
//      B    O    ( 0) ( 1)(111~1) (X-X) (   X  )
//      C    O    ( 0) ( 1)(000~0) (X-X) (   X  )
//      D    O    ( 0) ( 0)(000~0) (0~0) (   0  )
//      E    O    ( 0) ( 0)(000~0) (X-X) (   X  )
```

Appendix C

Tessent Shell Batch Jobs

You can use Tessent Shell in either an interactive or non-interactive manner. You conduct a tool session interactively by entering the commands manually, or the session can be completely scripted and driven using a dofile. This non-interactive mode of operation enables the entire session to be conducted without user interaction. This method of using Tessent Shell can be further expanded to enable the session to be scheduled and run as a true batch or cron job. This appendix focuses on the features of Tessent Shell that support its use in a batch environment.

Commands and Variables for the dofile	697
Command Line Options	700
Scheduling a Batch Job for Execution Later	700

Commands and Variables for the dofile

The following shell script invokes Tessent Shell and then runs a dofile that sets the context to “patterns -scan,” reads a design, and reads a cell library.

The dofile also specifies an option to exit from the dofile upon encountering an error:

```
set_dofile_abort exit
```

The exit option sets the exit code to a non-zero value if an error occurs during execution of the dofile. This enables a shell script that launches a Tessent Shell session to control process flow based on the success or failure of a tool operation. Note the line check for the exit status following the line that invokes Tessent Shell.

```

#!/bin/csh -b
##
## Add the pathname of the <Tessent_Tree_Path>/bin directory to the PATH
## environment variable so you can invoke the tool without typing the full
## pathname
##
setenv PATH <Tessent_Tree_Path>/bin:${PATH}
##
setenv DESIGN `pwd`
##
## 
tesson -shell -dofile ${DESIGN}/tshell.do \
    -license_wait 30 -log ${DESIGN}/`date +log_file_%m_%d_%y_%H:%M:%S` 
setenv proc_status $status
if ("$proc_status" == 0 ) then
echo "Session was successful"
echo " The exit code is: " $proc_status
else echo "Session failed"
echo " The exit code is: " $proc_status
endif
echo $proc_status " is the exit code value."

```

You can use environment variables in a Tessent Shell dofile. For example, the shell script sets the DESIGN environment variable to the current working directory. When a batch job is created, the process may not inherit the same environment that existed in the shell environment. To assure that the process has access to the files referenced in the dofile, the DESIGN environment variable is used. A segment of a Tessent Shell dofile displaying the use of an environment variable follows:

```

# The shell script that launches this dofile sets the DESIGN environment
# variable to the current working directory.
add_scan_groups g1 ${DESIGN}/procfile
#
add_scan_chains c1 g1 scan_in CO
...
#
write_faults ${DESIGN}/fault_list -all -replace

```

You can also use a startup file to alias common commands. To set up the predefined alias commands, use the file *.tesson_startup* (located by default in your home directory). For example:

```
alias save_my_pat write_patterns $1/pats.v -$2 -replace
```

The following dofile segment displays the use of the alias defined in the *.tesson_startup* file:

```

# The following alias is defined in the .tesson_startup file
#
save_my_pat $DESIGN verilog

```

Another important consideration is to exit in a graceful manner from the dofile. This is required to assure that Tessent Shell exits instead of waiting for additional command line input.

```
# The following command terminates the Tessent Shell dofile.  
#  
exit -force
```

Command Line Options

Several Tessent Shell command line options are useful when running Tessent Shell as a batch job. One of these options is the -LICense_wait option, which sets a limit for retrying license acquisition after you set a context. The default is no time limit for a license.

If Tessent Shell is unable to obtain a license after the specified number of retries, the tool exits. An example of the Tessent Shell invocation line with this option follows:

```
% tessent -shell-dofile tshell.do -license_wait 30 \
-log ${DESIGN}/`date +log_file_%m_%d_%y_%H:%M:%S`
```

Another item of interest is the logfile name created using the Linux “date” command for each Tessent Shell run. The logfile is based on the month, day, year, hour, minute, and second that the batch job was launched. An example of the logfile name that would be created follows:

log_file_05_30_12_08:42:37

Scheduling a Batch Job for Execution Later..... 700

Scheduling a Batch Job for Execution Later

You can schedule a batch job using the Linux “at” or “cron” command. For more information about using either of these commands, use the Linux “man” command.

Appendix D

Net Pair Identification with Calibre for Bridge Fault Test Patterns

You can use the ATPG tool to identify a list of net pairs that should be targeted for the bridging fault model and generated scan patterns. For complete information, refer to the “Bridge Fault Test Pattern Generation Flow” section of the *Calibre Solutions for Physical Verification* manual in your Calibre software tree or Support Center.

These sections cover the specifics of the bridge fault model.

The Static Bridge Fault Model	701
Four-Way Dominant Fault Model	702
Top-level Bridging ATPG	702
Incremental Multi-Fault ATPG	703
The Bridge Parameters File	705

The Static Bridge Fault Model

Certain pairs of nets in a design have specific characteristics that make them vulnerable to bridging. The *static bridge* fault model is used by the ATPG tool to test against potential bridge sites (net pairs) extracted from the design. You can load the bridge sites from a bridge definition file or from the Calibre query server output file.

For more information, see “[Net Pair Identification with Calibre for Bridge Fault Test Patterns](#)” on page 701.

This model uses a 4-Way Dominant fault model that works by driving one net (dominant) to a logic value and ensuring that the other net (follower) can be driven to the opposite value.

Let `sig_A` and `sig_B` be two nets in the design. If `sig_A` and `sig_B` are bridged together, the following faulty relationships exist:

- `sig_A` is dominant with a value of 0 (`sig_A=0; sig_B=1/0`)
- `sig_A` is dominant with a value of 1 (`sig_A=1; sig_B=0/1`)
- `sig_B` is dominant with a value of 0 (`sig_B=0; sig_A=1/0`)
- `sig_B` is dominant with a value of 1 (`sig_B=1; sig_A=0/1`)

By default, the tool creates test patterns that test each net pair against all four faulty relationships.

Four-Way Dominant Fault Model

The ATPG tool uses the 4-Way Dominant fault model to target net pairs for bridging. The 4-Way Dominant fault model works by driving a net to a dominant value (0 or 1) and ensuring that the follower can be driven to the opposite value.

A simplified bridging fault model is adopted by defining the target net pairs as a set of stuck-at faults. Each of the net pairs targeted by the bridge fault model is classified as dominant or follower. A dominant net will force the follower net to take the same value as the dominant net in the faulty circuit when the follower has an opposite logical value than the dominant net.

Let A and B be two gates with output signals sig_A and sig_B. When sig_A and sig_B are bridged together, four faulty relationships can be defined:

- sig_A has dominant value of 0 (sig_A=0; sig_B s@0)
- sig_A has dominant value of 1 (sig_A=1; sig_B s@1)
- sig_B has dominant value of 0 (sig_B=0; sig_A s@0)
- sig_B has dominant value of 1 (sig_B=1; sig_A s@1)

By default, the ATPG tool targets each net pair using these four faulty relationships by generating patterns which drive one net to the dominant value and observing the behavior of the other (follower) net.

Top-level Bridging ATPG

This flow generates patterns for a list of net pairs generated by another tool such as Calibre.

The following input files are used in this flow:

- Gate-level Netlist
- ATPG Library
- Bridge Definition File or Calibre Server output file, here: from_calibre.sites

The following commands are specific to bridging fault ATPG and should be issued once in analysis mode:

```
set_fault_type bridge
// FAULT TYPE SET TO 4WAY_DOM
read_fault_sites from_calibre.sites
// ALL BRIDGE NET PAIRS ARE LOADED
add_faults all
// GENERATE A LIST OF FAULTS BASED ON THE LOADED BRIDGE NET PAIRS
create_patterns
// GENERATES PATTERNS
write_patterns bridge_patterns.ascii
exit
```

The `set_fault_type` command with the bridge argument is the only specification needed for generation of patterns that target the 4-Way Dominant fault model. The `read_fault_sites` command is used to load the list of net pairs that should be targeted for the bridging fault model.

Incremental Multi-Fault ATPG

The following flow generates patterns for a list of net pairs generated by another tool such as Calibre. After the initial ATPG for bridging, you can generate patterns for other fault models such as stuck-at. The flow fault simulates each pattern set for other fault models and generates additional patterns to target the remaining faults.

The following input files are used in this flow:

- Gate-level Netlist
- ATPG Library
- Bridge Definition File or Calibre Server output file, here: *from_calibre.sites*

The following example shows generation of patterns for bridging faults followed by stuck-at faults. The following commands should be issued once in analysis mode:

```
set_fault_type bridge
// fault type set to 4way_dom
read_fault_sites from_calibre.sites
// all bridge net pairs are loaded
create_patterns
//generate a list of faults based on the loaded bridge net pairs,
// then generate patterns
write_patterns bridge_patterns.ascii
set_fault_type stuck
add_faults all
// adds all stuck-at faults
read_patterns bridge_patterns.ascii
// load external patterns and add to internal patterns
simulate_patterns
// simulate bridge patterns for stuck-at faults
report_statistics
set_fault_protection on
// protect stuck-at faults that were detected by bridge patterns
reset_state
// remove bridge patterns that were effective in detecting stuck-at faults
create_patterns
// generate new patterns for remaining faults
write_patterns stuck_patterns.ascii
exit
```

The next example shows generation of patterns for stuck-at faults followed by bridging faults. The following commands should be issued once in analysis mode:

```
set_fault_type stuck
create_patterns
// adds all stuck-at faults and generates patterns
write_patterns stuck_patterns.ascii
set_fault_type bridge
// fault type set to 4way_dom
read_fault_sites from_calibre.sites
// all bridge net pairs are loaded
read_patterns stuck_patterns.ascii
simulate_patterns
// simulate stuck-at patterns for bridge faults
write_faults bridge_faults.detected -class DT
// save detected bridge faults to file
create_patterns
reset_state
// remove stuck-at patterns that were effective in detecting bridge faults
read_faults bridge_faults.detected -retain
// load detected bridge faults and retain detection status
create_patterns
// generate new patterns for remaining undetected bridge faults
write_patterns patterns_bridge.ascii
exit
```

The Bridge Parameters File

The bridge definition is a text file which you automatically generate using the Extraction Package.

This section provides information for interpreting the entries in the bridge parameters file.

Format

Normally, you do not modify the generated bridge parameters file. If you do modify this file, then you must adhere to the following syntax:

- Precede each line of comment text with a pair of slashes (//).
- Do not modify keywords. They can be in upper-or lowercase.
- Use an equal sign to define a value for a keyword.
- Enclose all string values in double quotation marks (" ").
- Bridge declarations must be enclosed in braces ({}).
- A semicolon (;) must separate each entry within the bridge declaration.

Parameters

Use the keywords described in the following table to create a bridge definition file. Keywords cannot be modified and can be in upper- or lowercase.

Table D-1. Bridge Definition File Keywords

Keyword (s)	Usage Rules
VERSION	Required. Used to specify the version of the bridge definition file and must be declared before any bridge entries. Must be a real number or integer written in non-scientific format starting with 1.1.
FAULT_TYPE	Optional. Used to indicate what type of fault is declared by the FAULTS keyword. Must be a string value enclosed in quotation marks (" ").
BRIDGE	Required. Used to start a bridge entry.
NET1 NET2	Required. Identifies the net/pin pair for the bridge entry. Specifies the regular or hierarchical net/pin pathname to gate(s). Use the following guidelines when using these keywords: <ul style="list-style-type: none">• Declare either net or pin value pairs.• Net/Pin pairs are the first two items declared in the bridge entry.• Net/Pin pathnames are string values and should be enclosed in quotation marks (" ").• A net can be used in multiple bridge entries.• Nets can be defined in any order.

Table D-1. Bridge Definition File Keywords (cont.)

Keyword (s)	Usage Rules
FAULTS	<p>Optional. Provides a fault classification for each of the four components of the 4-way bridge fault model. Four classifications must be specified in a comma-separated list enclosed in braces ({}). You can use any of the following 2-digit codes for each of the four components:</p> <ul style="list-style-type: none"> • UC (uncontrolled) • UO (unobserved) • DS (det_simulation) • PU (posdet_untestable) • PT (posdet_testable) • TI (tied) • BL (blocked) • AU (ATPG_untestable) • NF (directs the tool to nofault this component of the bridge) <p>Using fault classifications, you can filter and display the desired fault types. For more information on fault classes and codes, see “Fault Classes” on page 72.</p>
NAME	Optional. A unique string, enclosed in quotation marks (" "), that specifies a name for the bridge.
DISTANCE ¹	Optional. Real number that specifies the distance attribute in microns (um) for the bridge entry.
PARALLEL_RUN ₁	Optional. Real number that specifies the parallel_run attribute in microns (um) for the bridge entry.
LAYER ^{1,2}	Optional. String that specifies the layer attribute for the bridge entry. Must be enclosed in quotation marks (" ").
WEIGHT ^{1,2}	Optional. Real number that specifies the weight attribute for the bridge entry.
TYPE ^{1,2}	<p>Optional. 3-, 4-, or 5-character code that specifies a type identification for the bridge entry. Must be enclosed in quotation marks (" "). Use one of the following codes:</p> <ul style="list-style-type: none"> • S2S — Side-to-side • SW2S — Side-Wide-to-Side: same as S2S but at least one of the two signal lines is a wide metal line • S2SOW — Side-to-Side-Over-Wide: same as S2S, but the bridge is located over a wide piece of metal in a layer below • C2C — Corner-to-Corner • V2V — Via-to-Via: an S2S for vias • VC2VC — Via-Corner-to-Via-Corner • EOL — End-of-Line: the end head of a line faces another metal line

1. This keyword is not used by the software to define any specific value. However, you can use it to specify a value which can be filtered on by the `read_fault_sites` command.
2. You can display a histogram of the number of equivalent classes of bridges based on either their layer, weight, or type attribute using the `report_fault_sites` or the `report_fault` command.

Examples

The following example shows the format of the bridge definition file. The keywords used in this file are case insensitive.

```
// Any string after // is a comment and will be ignored by the tool.
VERSION 1.1// An optional item that declares the version of
           // the bridge definition file. VERSION must be
           // defined before any bridge entry. The version
           // number starts from 1.0.
FAULT_TYPE = BRIDGE_STATIC_4WAY_DOM
           // An optional header that indicates the fault type
// to be declared by keywords FAULTS in
           // bridge body. Currently, the only valid fault type
           // is BRIDGE_STATIC_4WAY_DOM
BRIDGE { // Define the bridge entry body and can be repeated
           // as many times as necessary.
NET1 = NET_NAME; // Defines the first net name.
NET2 = NET_NAME; // Defines the second net name.
// NET1 and NET2 must be defined in each bridge
// entry and they must be declared before any other
// items defined in the bridge entry body.
FAULTS = {FAULT_CATEGORY_1, FAULT_CATEGORY_2, FAULT_CATEGORY_3,
FAULT_CATEGORY_4};
           // An optional item that defines the fault
           // classes for each of the four faulty
           // relationships. All four fault categories must be
           // declared and must be in the order shown in the
           // previous section. Examples of fault classes are
           // UC, UO, DS, DI, PU, // PT, TI, BL and AU. A new
           // fault class NF (No Fault) can be used to exclude
           // any specific faulty relationship for ATPG.
NAME = STRING; // An optional string specifying the name
               // of the bridge.
PARALLEL_RUN = floating number; // An optional item specifying the
               // parallel run length of the nets of the bridge.
DISTANCE = floating number; // An optional item specifying the distance
               // of the nets of the bridge.
WEIGHT = floating number; // An optional item specifying a weight assigned
               // to the bridge.
LAYER = LAYER_NAME; // An optional item specifying the name of the
               // layer the bridge is in.
} // End of bridge body
```

The items DISTANCE, PARALLEL_RUN, WEIGHT, and LAYER are collectively referred to as attributes of the bridge. For ATPG purposes, these attributes are ignored, and by default not reported. You have to specify the `-attribute` option to according commands to see them. The unit of length used for DISTANCE and PARALLEL_RUN is um (10^{-6}) meters.

Appendix E

Getting Help

There are several ways to get help when setting up and using Tesson software tools. Depending on your need, help is available from documentation, online command help, and Mentor Graphics Support.

The Tesson Documentation System	709
Mentor Support Services.....	710

The Tesson 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 “[Documentation Options](#)” in the *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 Tesson tool with the -manual invocation switch.
- **File System** — Access the Tesson InfoHub or PDF bookcase directly from your file system, without invoking a Tesson tool. For example:

HTML:

```
firefox $MGC_DFT/docs/infohubs/index.html
```

PDF

```
acroread $MGC_DFT/docs/pdfdocs/_bk_tesson.pdf
```

- **Application Online Help** — You can get contextual online help within most Tesson 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.

Mentor Support Services

Mentor provides a range of industry-leading support services that keep design teams productive and up-to-date with Mentor products.

A Mentor support contract includes the following:

- **Software Updates** — Get the latest releases and product enhancements to keep your environment current.
- **Mentor Graphics Support Center** — Access our online knowledge base, personalized to your Mentor products.
- **Support Forums** — Learn, share, and connect with other Mentor users.
- **Technical Support** — Collaborate with Mentor support engineers to solve complex design challenges.
- **Regular Communications** — Receive the latest knowledge base articles and announcements for your Mentor products.
- **Mentor Ideas** — Share ideas and vote for your favorites to shape future products.

More information is available here:

<https://support.mentor.com>

If your site is under a current support contract, but you do not have a Support Center login, register today:

<https://support.mentor.com/register>

Index

— A —

Abort limit, 317
Aborted faults, 317
 changing the limits, 317
 reporting, 317
Ambiguity
 edge, 356
 path, 355
ASCII WGL format, 552
ASIC Vector Interfaces, 29
ATPG
 applications, 41
 basic operations, 285
 constraints, 307
 default run, 314
 defined, 40
 for IDDQ, 322
 for path delay, 349
 for transition fault, 326
 full scan, 41
 function, 308
 inputs and outputs, 277
 MacroTest, using, 434
 non-scan cell handling, 127
 pattern types, 280
 process, 306
 setting up faults, 299
 test cycles, 280
 timing model, 280
 tool flow, 275
ATPG commands
 add_cell_constraints, 297
 add_clocks, 296
 add_false_paths, 343
 add_faults, 299, 304
 add_input_constraints, 286, 288
 add_lists, 301
 add_nofaults, 298
 add_output_masks, 288

add_primary_inputs, 286
add_scan_chains, 297
add_scan_groups, 296
add_slow_pad, 291
add_tied_signals, 290
analyze_atpg_constraints, 309
analyze_bus, 292
analyze_fault, 352
analyze_graybox, 621
analyze_restrictions, 309
create_flat_model, 98
create_patterns, 387
delete_atpg_constraints, 309
delete_atpg_functions, 309
delete_false_paths, 343
delete_fault_sites, 352
delete_faults, 304
delete_input_constraints, 286, 291
delete_multicycle_paths, 343, 349
delete_primary_inputs, 287
delete_primary_outputs, 287
delete_scan_chains, 297
delete_scan_groups, 296
delete_tied_signals, 290
read_fault_sites, 352
read_patterns, 300
report_aborted_faults, 317, 318
report_atpg_constraints, 309
report_atpg_functions, 310
report_bus_data, 292
report_cell_constraints, 298
report_clocks, 296
report_false_paths, 343
report_fault_sites, 352
report_faults, 300, 304, 317
report_gates, 293
report_input_constraints, 286, 292
report_nofaults, 298
report_nonscan_cells, 131
report_paths, 352

report_primary_inputs, 287
report_primary_outputs, 287
report_scan_chains, 297
report_scan_groups, 297
report_statistics, 300
report_tied_signals, 290
reset_state, 302
run, 315
set_abort_limit, 317
set_atpg_timing, 390, 393
set_bus_handling, 292
set_capture_clock, 300
set_checkpointing_options, 313
set_clock_restriction, 297
set_contention_check, 292
set_driver_restriction, 292
set_fault_mode, 305
set_fault_type, 299, 303, 352
set_learn_report, 292
set_list_file, 301
set_net_dominance, 293
set_net_resolution, 293
set_pattern_buffer, 292
set_pattern_type, 131, 282, 283, 331
set_possible_credit, 306
set_random_atpg, 318
set_random_clocks, 300
set_random_patterns, 300
set_system_mode, 286, 298
set_tied_signals, 290
set_z_handling, 293
simulate_patterns, 300
write_fault_sites, 353
write_faults, 305
write_patterns, 318
ATPG tool, definition of, 27
At-speed test, 325

— B —

Batch jobs, 697
Bidirectional pin
 as primary input, 287
 as primary output, 287
Boundary scan, defined, 32
Bridge fault testing
 creating the test set, 341

Bus
 float, 127
Bus contention, 127
 checking during ATPG, 292
 fault effects, 293

— C —

Capture handling, 294
Capture point, 53
Capture procedure
 seeNamed capture procedure
Chain test, 550
Checkpointing example, 313
Checkpoints, setting, 313
Clock
 capture, 296, 427
 list, 296
 off-state, 296
 scan, 296
Clock PO patterns, 280
Clock procedure, 280
Clock sequential patterns, 280
Clocked sequential test generation, 130
Combinational loop, 118, 119, 120, 121, 122, 123
 cutting, 119
Constant value loops, 119
Constraints
 ATPG, 307
 pin, 290
 scan cell, 297
Contention, bus, 108
Copy cell or element, 92
Copy, scan cell element, 92
Coupling loops, 122
CPF commands, 584
create_patterns, 387
Creating a delay test set, 325
Creating patterns
 default run, 314
Cycle count, 551
Cycle test, 550

— D —

Data capture simulation, 294
Debugging simulation mismatches, 465

Defect, 43
Delay test coverage, 384, 385
Design Compiler, handling pre-inserted scan cells, 164
Design flattening, 98
Design flow, delay test set, 325
Design flow, EDT
 tasks and products, 226
Design rules checking
 blocked values, 112
 bus keeper analysis, 111
 bus mutual-exclusivity, 108
 clock rules, 111
 constrained values, 112
 data rules, 110
 extra rules, 112
 forbidden values, 112
 general rules, 107
 introduction, 107
 procedure rules, 107
 RAM rules, 111
 scan chain tracing, 109
 scannability rules, 112
 shadow latch identification, 109
 transparent latch identification, 110
Design-for-Test, defined, 28
Deterministic test generation, 40
DFF modeling, 311
DFTVisualizer
 see "DFTVisualizer" in the Tesson Shell User's Manual
Differential scan input pins, 549
Distributed ATPG, troubleshooting SSH, 487

— E —
Edge ambiguity, 356
EMPTY, 29, 35, 37, 84, 104, 107, 131, 139, 140, 285, 319, 331, 367, 373
Existing scan cells
 handling, 164
External logic location flow
 tasks and products, 226
External pattern generation, 41
Extra, scan cell element, 93

— F —
Fault
 aborted, 317
 classes, 72
 collapsing, 48
 detection, 69
 no fault setting, 298
 representative, 48, 84
 simulation, 299
 small delay, 383
 undetected, 317
Fault grading
 in multiple fault model flow, 380
 pattern generation and, 380
Fault models
 path delay, 53
Fault sampling, 313
Feedback loops, 118
Flattening, design, 98
Frame, simulation, 311
Full scan, 34, 149

— G —
Gate duplication, 121
Good simulation, 301

— H —
Hierarchical instance, definition, 98
Hold_pi, 393

— I —
IDDQ testing, 322
 creating the test set, 322
 methodologies, 44
 test pattern formats, 548
 vector types, 45
Incomplete designs, 140
Instance, definition, 98
Internal scan, 31, 32

— L —
Latches
 handling as non-scan cells, 127
 modeling, 311
 scannability checking of, 117
Launch point, 53
Learning analysis, 104

dominance relationships, 106
equivalence relationships, 104
forbidden relationships, 106
implied relationships, 105
logic behavior, 104
Line holds, 73, 74
Loop count, 551
Loop cutting, 119
 by constant value, 119
 by gate duplication, 121
 for coupling loops, 122
 single multiple fanout, 120
Loop handling, 118

— M —

Macros, 45
MacroTest, 434
 basic flow, 434, 435, 436
 capabilities, summary, 434
 examples, 449
 basic 1-Cycle Patterns, 449
 leading&trailing edge observation, 449
 synchronous memories, 449
 macro boundary
 defining, 441
 with instance name, 441
 with trailing edge inputs, 443
 without instance name, 442
 reporting&specifying observation sites, 443
 overview, 434
 qualifying macros, 437
 recommendations for using, 447
 test values, 445
 when to use, 438
Manufacturing defect, 43
Mapping scan cells, 159
Mask_po, 393
Masking primary outputs, 290
Master cell or element, 90
Master, scan cell element, 90
Memories, testing, 434
Module, definition, 99
MTFI, 609
Multiple load patterns, 280

— N —

Named capture procedure, 359
 internal and external mode of, 360
 on-chip clocks and, 360
No fault setting, 298
Non-scan cell handling, 127
 ATPG, 127
 clocked sequential, 130
 sequential transparent, 129
Non-scan sequential instances
 reporting, 178

— O —

Off-state, 96, 163, 296
One-shot circuit, 393
Output mask, 393

— P —

Parallel scan chain loading, 546
Partition scan, 35, 149
Path ambiguity, 355
Path definition file, 352
Path delay testing, 53, 349
 basic procedure, 357
 limitations, 358
 multiple fault models and
 flow, 380
 path ambiguity, 355
 path definition checking, 355
 path definition file, 352
 patterns, 349
 robust detection, 351
 transition detection, 351
Path sensitization, 70
Pattern formats
 binary, 552
 text, 549
 Verilog, 552
 WGL (ASCII), 552
Pattern generation
 deterministic, 40
 external source, 41
 multiple fault models and
 flow, 380
Pattern generation phase
 test procedure waveforms, example, 668

Pattern types
 basic scan, 280
 clock PO, 280
 clock sequential, 280
 multiple load, 280
 RAM sequential, 280
 sequential transparent, 280
Performing ATPG, 306
Pin constraints, 290
plusargs, Verilog, 459
Possible-detect credit, 75, 305
Possible-detected faults, 75
power-aware ATPG procedure, 586
power-aware overview, 582
Pre-inserted scan cells
 handling, 164
Primary inputs
 bidirectional pins as, 287
 constraining, 290
 constraints, 290
Primary outputs
 bidirectional pins as, 287
 masking, 290
Primitives, simulation, 100
Pulse generators
 gate reporting for, 685
 introduction, 132

— R —

RAM
 ATPG support, 135
 comm on read and clock lines, 137
 common write and clock lines, 137
 related commands, 138
 rules checking, 139
 sequential mode, 136
 sequential patterns, 280
 testing, 134
report_graybox_statistics, 621
ROM
 ATPG support, 135
 related commands, 138
 rules checking, 139
 testing, 134

— S —

Sampling, fault, 313
Saving patterns, serial versus parallel, 546
Scan
 basic operation, 33
 clock, 33
Scan and/or wrapper test structures, 149
Scan cell
 concepts, 89
 constraints, 297
 existing, 164
 mapping, 159
 pre-inserted, 164
Scan cell elements
 copy, 92
 extra, 93
 master, 90
 shadow, 91
 slave, 90
Scan chains
 definition, 93
 parallel loading, 546
 serial loading, 548
 serial versus parallel loading, 546
 specifying, 297
Scan clocks, 95, 163
 specifying, 163, 296
Scan design
 simple example, 33
Scan design, defined, 31
Scan groups, 94, 296
Scan output mapping, 159
Scan patterns, 280
Scan related events, 533
Scan sub-chain, 546
Scan test, 550
Scannability checks, 117
SDF file, 390
Sequential element modeling, 311
Sequential loop, 118, 124
Sequential transparent latch handling, 129
Sequential transparent patterns, 280
Serial scan chain loading, 548
set_atpg_timing, 390, 393
Shadow, 91

-
- Shadow cell or element, 91
 Simulating captured data, 294
 Simulation data formats, 549
 Simulation formats, 544
 Simulation frame, 311
 Simulation mismatches, debugging, 465
 Simulation primitives, 100
 - AND, 101
 - BUF, 100
 - BUS, 102
 - DFF, 101
 - INV, 101
 - LA, 101
 - MUX, 101
 - NAND, 101
 - NMOS, 102
 - NOR, 101
 - OR, 101
 - OUT, 104
 - PBUS, 102
 - RAM, 103
 - ROM, 103
 - STFF, 101
 - STLA, 101
 - SW, 102
 - SWBUS, 102
 - TIE gates, 101
 - TLA, 101
 - TSD, 102
 - TSH, 102
 - WIRE, 102
 - XNOR, 101
 - XOR, 101
 - ZHOLD, 102
 Single multiple fanout loops, 120
 Sink gates, 295
 Slack, 383
 Slave, 90
 Slave cell or element, 91
 Small delay fault model, 383
 Source gates, 295
 SSH protocol
 - troubleshooting, 487
 Static-inactive path, 393
 Structural loop, 118
- combinational, 118
 sequential, 118
 Structured DFT, 28
 System-class
 - non-scan instances, 176
 - scan instances, 176
- T —**
- Tessent Core Description File (TCD), 258, 503
 - top-level, 508
 Tessent FastScan, available in Tessent Shell, 27
 Tessent Scan
 - features, 37
 - inputs and outputs, 146
 - invocation, 148
 - supported test structures, 149
 Tessent Scan commands
 - add_clocks, 163
 - add_nonscan_instances, 177
 - delete_cell_models, 163
 - delete_clocks, 164
 - delete_nonscan_instances, 178
 - delete_scan_instances, 178
 - report_cell_models, 163
 - report_clocks, 164
 - report_control_signals, 179
 - report_nonscan_models, 178
 - report_primary_inputs, 164
 - report_statistics, 179
 - report_test_logic, 163
 - set_system_mode, 167
 Tessent Scan, available in Tessent Shell, 27
 Tessent Shell, providing ATPG and Scan functionality, 27
 Tessent TestKompress, available in Tessent Shell, 27
 Test bench, Verilog, 458
 Test clock, 132, 161
 Test logic, 117, 132, 160
 Test Pattern File Format
 - Functional Chain Test, 571
 - Header_Data, 568
 - Scan Cell, 575
 - Scan Test, 572
 Test patterns, 40

- chain test block, [550](#)
- scan test block, [550](#)
- Test points
 - definition of, [150](#)
- Test procedure file
 - in Tesson Scan, [147](#)
- Test structures
 - full scan, [34](#), [149](#)
 - partition scan, [35](#), [149](#)
 - scan and/or wrapper, [149](#)
 - supported by Tesson Scan, [149](#)
 - test points, [150](#)
- Test types
 - at-speed, [46](#)
 - IDDQ, [44](#)
- Test vectors, [40](#)
- Testability, [28](#)
- Testing
 - memories, [434](#)
- TI TDL 91 format, [554](#)
- TIE0, scannable, [117](#)
- TIE1, scannable, [117](#)
- Timing-aware ATPG
 - detection paths, [393](#)
 - example, [388](#)
 - limitations, [385](#)
 - reducing run time, [390](#)
 - setting up, [387](#)
 - troubleshooting, [390](#)
- Tools used in EDT flow, [226](#)
- Toshiba TSTL2 format, [555](#)
- Transition ATPG, [385](#)
- Transition testing
 - basic procedures, [330](#)
- Transparent slave handling, [129](#)
- Troubleshooting
 - sdf file, [390](#)
 - timing-aware ATPG, [390](#)

— V —

- Verilog, [552](#)
- Verilog plusargs, [459](#)
- Verilog test bench, [458](#)

- Undetected faults, [317](#)
- UPF commands, [584](#)
- User-class
 - non-scan instances, [176](#)
 - scan instances, [177](#)

Third-Party Information

Provides information on open source and third-party software that may be included in Tessent products.

For third-party information, refer to the [*Third-Party Software for Tessent Products*](#) document. Additional open source and third-party software information may be found in <your_Mentor_Graphics_documentation_directory>/legal

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. BETA CODE.

- 3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively “Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer’s use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer’s evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer’s feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

4. RESTRICTIONS ON USE.

- 4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer’s product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer’s employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively “Files”), including without limitation files containing Standard Verification Rule Format (“SVRF”) and Tcl Verification Format (“TVF”) which are Mentor Graphics’ trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer’s employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
- 4.3. Customer agrees that it will not subject any Product to any open source software (“OSS”) license that conflicts with this Agreement or that does not otherwise apply to such Product.
- 4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise (“Attempted Transfer”), without Mentor Graphics’ prior written consent and payment of Mentor Graphics’ then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics’ prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics’ option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer’s permitted successors in interest and assigns.
- 4.5. The provisions of this Section 4 shall survive the termination of this Agreement.

5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics’ then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.
6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations (“Third Party Terms”) may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

7. LIMITED WARRANTY.

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
 - 7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. THIRD PARTY CLAIMS.

- 9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.
- 9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.
- 9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. INFRINGEMENT.

- 10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
- 10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE,

SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

11. TERMINATION AND EFFECT OF TERMINATION.

- 11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.
16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.