



# DFT Compiler 1 Workshop

## Student Guide

30-I-011-SSG-012

2007.12

**Synopsys Customer Education Services**  
700 East Middlefield Road  
Mountain View, California 94043

Workshop Registration: **1-800-793-3448**

[www.synopsys.com](http://www.synopsys.com)

# Copyright Notice and Proprietary Information

Copyright © 2008 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Destination Control Statement

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

## Disclaimer

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

## Registered Trademarks (®)

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

## Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSI, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

## Service Marks (SM)

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

SystemC is a trademark of the Open SystemC Initiative and is used under license. ARM and AMBA are registered trademarks of ARM Limited. Saber is a registered trademark of SabreMark Limited Partnership and is used under license. All other product or company names may be trademarks of their respective owners.

Document Order Number: 30-I-011-SSG-012  
DFT Compiler 1 Student Guide

# Table of Contents

---

## Unit 1: Understanding Scan Testing

Unit Objectives .....	1-2
Introduction to Scan Testing: Agenda .....	1-3
What is Manufacturing Test?.....	1-4
What Is a Physical Defect? .....	1-5
Physical Defects in CMOS .....	1-6
Stuck-At Fault Model .....	1-7
Rules for Detecting a Stuck-At Fault.....	1-8
Introduction to Scan Testing: Agenda .....	1-9
Combinational Logic: Stuck-At Fault Testing.....	1-10
Activate the SA0 Fault (1/4) .....	1-11
Propagate Fault Effect (2/4) .....	1-12
Anatomy of a Test Pattern (3/4) .....	1-13
Record the Test Pattern (4/4) .....	1-14
Assessment of D-algorithm.....	1-15
Exercise: Detect SA1 Fault.....	1-16
Solution: Detect SA1 Fault .....	1-17
Is High Coverage Needed? .....	1-18
High Coverage Is Low DPPM .....	1-19
Single Stuck-At Fault Assumption (SSAF) .....	1-20
Introduction to Scan Testing: Agenda .....	1-21
Testing Sequential Designs.....	1-22
Scannable Equivalent Flip-Flop.....	1-23
The Full Scan Strategy.....	1-24
Scan Testing Protocol: Example 1 .....	1-25
Scan Testing Protocol: Example 2 .....	1-26
Test Patterns Overlap .....	1-27
Scan Strategies Summarized.....	1-28
Full Scan Summary .....	1-29
Unit Summary.....	1-30

---

## Unit 2: DFTC User Interfaces

Unit Objectives .....	2-2
DFT Compiler Flows: Agenda.....	2-3
What is Design-For-Testability?.....	2-4
Structured DFT Defined .....	2-5
Ad Hoc DFT Defined.....	2-6
DFT Compiler.....	2-7
BSD Compiler.....	2-8
DFT Compiler Flows: Agenda.....	2-9
DFT Compiler Flows Supported.....	2-10
Top-Down Vs. Bottom-Up Flow .....	2-11

# Table of Contents

Unmapped Flow Vs. Mapped Flow .....	2-12
DFT Compiler Test-Ready or Unmapped Flow .....	2-13
DFT Compiler Mapped Flow.....	2-14
DFT Compiler Existing Scan Flow.....	2-15
DFT Compiler Flows: Agenda.....	2-16
Typical Scan Insertion Flow .....	2-17
Read Design.....	2-18
Create Test Protocol.....	2-19
DFT DRC.....	2-20
Specify Scan Architecture.....	2-21
Preview .....	2-22
Insert Scan Paths .....	2-23
DFT DRC Coverage .....	2-24
Handoff Design.....	2-25
Unit Summary.....	2-26
Command Summary (Lecture, Lab) .....	2-27

---

## Unit 3: DFTC User Interfaces

Unit Objectives .....	3-2
DFT Compiler Setup: Agenda .....	3-3
DFTC: Multiple Tool Environments .....	3-4
DFT Compiler Setup: Agenda .....	3-5
One Startup File Name – Three File Locations .....	3-6
Default .../admin/setup/.synopsys_dc.setup .....	3-7
Project-specific (CWD) .synopsys_dc.setup.....	3-8
DFT Compiler Setup: Agenda .....	3-9
Tool Command Language (TCL) .....	3-10
Helpful UNIX-like dc_shell Commands .....	3-11
More Helpful dc_shell Commands .....	3-12
DFT Compiler Setup: Agenda .....	3-13
Typical Scan Insertion Flow .....	3-14
Synthesis Transformations.....	3-15
Technology Library .....	3-16
How is the Target Library Used?.....	3-17
Resolving ‘References’ with link_library .....	3-18
Use link to Resolve Design References .....	3-19
Set the search_path Variable.....	3-20
DC-Topographical Setup .....	3-21
Unit Summary.....	3-22
Command Summary (Lecture, Lab) .....	3-23

---

## Unit 4: DFTC User Interfaces

Unit Objectives .....	4-2
Test Protocol: Agenda.....	4-3

# Table of Contents

Typical Scan Insertion Flow .....	4-4
What is a Test Protocol? .....	4-5
Steps for Creating a Test Protocol .....	4-6
DFT Compiler UI command conventions.....	4-7
Test Protocol: Agenda.....	4-8
The <code>set_dft_signal</code> command .....	4-9
Identify Ports in RTL to Use for Scan .....	4-10
DFT Compiler “Views” .....	4-11
How to Specify a Clock .....	4-12
Using <code>set_dft_signal -timing</code> .....	4-13
How to Specify a Reset.....	4-14
Asynchronous Resets (and Sets).....	4-15
How to Specify a Constant .....	4-16
How to Specify a Scan In.....	4-17
How to Specify a Scan Out.....	4-18
How to Specify a Scan Enable.....	4-19
How to Specify a Differential Clock?.....	4-20
When to use <code>-view existing_dft</code> .....	4-21
When to use <code>-view spec</code> .....	4-22
Example: Declaring DFT Signals .....	4-23
Managing DFT Signals .....	4-24
Test Protocol: Agenda.....	4-25
Tester Timing Configuration .....	4-26
Specifying Tester Timing .....	4-27
Pre-Clock Measure vs. End-of-Cycle Measure .....	4-28
Pre-Clock Measure.....	4-29
End-of-Cycle Measure .....	4-30
Synopsys Test Tip.....	4-31
Test Protocol: Agenda.....	4-32
The <code>create_test_protocol</code> command .....	4-33
Test Protocol Generation Example .....	4-34
Use Protocol Inference Only When No Choice.....	4-35
Generic Capture Procedures .....	4-36
Creating a Generic Capture Procedures SPF .....	4-37
Multiple Clock Capture Procedure .....	4-38
Allclock Capture Procedures .....	4-39
Generic Capture Procedures Example .....	4-40
Already Have a Test Protocol? .....	4-41
Using <code>read_test_protocol</code> for Initialization .....	4-42
Customizing STIL Protocol Files .....	4-43
Default Initialization Macro is Very Basic .....	4-44
Initialization Protocol Flow Detail.....	4-45
Custom Initialization Example .....	4-46
Example STIL Initialization Macro .....	4-47
AutoFix for Fixing Internal Tristate Buses.....	4-48
Override Global Internal Bus Fixing .....	4-49

# Table of Contents

DRC Fixing: Agenda ..... 4-50

---

## Unit 5: Creating Test Protocols

Unit Objectives .....	5-2
DFT Design Rule Check: Agenda .....	5-3
Typical Scan Insertion Flow .....	5-4
Test Design Rule Checks (DFT DRC).....	5-5
Running dft_drc .....	5-6
DFT Design Rule Check: Agenda .....	5-7
Running DFT DRC on RTL Code.....	5-8
Example: Ripple-Counter Violation in RTL.....	5-9
Ripple-Counter RTL DFT Solution.....	5-10
DFT DRC on RTL? Use Text Reporting.....	5-11
Example: RTL DFT DRC .....	5-12
DFT Design Rule Check: Agenda .....	5-13
Scan Style.....	5-14
Test-Ready Compile .....	5-15
Design Compiler Ultra.....	5-16
Automatic Shift Register Identification .....	5-17
Shift Register Identification Control.....	5-18
Using Dedicated Scan Pins .....	5-19
Test-Ready Compile .....	5-20
DFT Compiler Scan State .....	5-21
Manually Setting the Scan State .....	5-22
DFT Design Rule Check: Agenda .....	5-23
Designing for Risk-Free Scan.....	5-24
Potential Scan-Shift Issues.....	5-25
Gated-Clock DRC Violation.....	5-26
What DFTC Reports Pre-DFT .....	5-27
Enhanced DFT DRC Reporting.....	5-28
Enabling Enhanced DFT DRC Reporting.....	5-29
Example: Enhanced DFT DRC Report (1/2) .....	5-30
Example: Enhanced DFT DRC Report (2/2) .....	5-31
Gated-Clock DRC Solution .....	5-32
Clock-Divider Violation .....	5-33
Clock-Divider Solution.....	5-34
DFT DRC: What are D rules?.....	5-35
DFT DRC: List of D rules .....	5-36
Why Check for DRC Violations? .....	5-37
Limitations of DFT DRC Checking.....	5-38
Libraries and DFT DRC.....	5-39
User Defined Simulation Libraries .....	5-40
Controlling Internal Nodes .....	5-41
Using Custom Initialization Sequence.....	5-42

# Table of Contents

Example Initialization Protocol Flow .....	5-43
Using Internal Pins to Bypass Initialization.....	5-44
Using set_test_assume to Bypass Initialization .....	5-45
Protocol File Warning.....	5-46
DFT Design Rule Check: Agenda .....	5-47
What DFTC Reports Post-DFT .....	5-48
What Are Capture Violations?.....	5-49
Enhanced DFT DRC Report – Post-DFT .....	5-50
Risk-Free Capture Example.....	5-51
What Causes Risky Capture?.....	5-52
Capture Problem Due to Skew .....	5-53
Capture-Problem Timing Details.....	5-54
Managing Clock-Skew Problems .....	5-55
Clock-as-Data Violation .....	5-56
Clock-as-Data Timing Details .....	5-57
Clock-as-Data Solution.....	5-58
Top DRC Rules at a Glance.....	5-59
Unit Summary.....	5-60
Lab 5: DFT Design Rule Checks.....	5-61
Command Summary (Lecture, Lab) .....	5-62
Appendix A.....	5-63
What are Internal Pins?.....	5-64
Enabling Internal Pins Support .....	5-65
Internal Pins Example.....	5-66
Supported Data Types.....	5-67
Valid Internal Pins Hookup Locations.....	5-68
Specification Examples (1/2).....	5-69
Specification Examples (2/2).....	5-70
Internal Pins in Reports.....	5-71
Appendix B .....	5-72
Scan Chain Extraction.....	5-73
“existing_dft” View .....	5-74
Scan State After Extraction.....	5-75
Example: Scan Chain Extraction .....	5-76

---

## Unit 6: Creating Test Protocols

Unit Objectives .....	6-2
DFT DRC GUI Debug: Agenda .....	6-3
GUI Debug of dft_drc Violations .....	6-4
GUI Debug Usage Models.....	6-5
Design Vision: Top Level.....	6-6
GUI Components .....	6-7
Violation Browser: After Test □ Browse Violations .....	6-8
Violation Inspector: D1 Analysis.....	6-9

# Table of Contents

Violation Inspector: D2 Analysis.....	6-10
Violation Inspector: Schematic Viewer: Simulation Values .....	6-11
DFT DRC GUI Debug: Agenda .....	6-12
What is Pindata? .....	6-13
Pindata in Design Vision .....	6-14
Pindata Types.....	6-15
Pindata: Clock On .....	6-16
Pindata: Clock Off .....	6-17
Pindata: Constraint Data .....	6-18
Pindata: Load .....	6-19
Pindata: Shift.....	6-20
Pindata: Master Observe .....	6-21
Pindata: Shadow Observe .....	6-22
Pindata: Stability Patterns .....	6-23
Pindata: Tie Data.....	6-24
Pindata: Test Setup .....	6-25
DFT DRC GUI Debug: Agenda .....	6-26
Violation Inspector: Waveform Viewer: test_setup analysis.....	6-27
DFT DRC GUI Debug: Agenda .....	6-28
Helpful Commands .....	6-29
Helpful Commands .....	6-30
Helpful Commands .....	6-31
Unit Summary.....	6-32
Lab 6: Introduction to Design Vision .....	6-33
Command Summary (Lecture, Lab) .....	6-34
Appendix A.....	6-35
GUI Debug with TetraMAX .....	6-36
DFT DRC on Gates? Use TetraMAX GUI.....	6-37
GUI Debug with dft_drc_interactive .....	6-38
Appendix B .....	6-39
Usage: gui_inspect_violations .....	6-40
Examples: gui_inspect_violations .....	6-41
Usage: gui_wave_add_signal.....	6-42
Examples: gui_wave_add_signal.....	6-43
Usage: guiViolation_schematic_add_objects .....	6-44
Examples: guiViolation_schematic_add_objects .....	6-45

---

## Unit 7: DFT for Clocks and Resets

Unit Objectives .....	7-2
DRC Fixing: Agenda .....	7-3
How to Fix DRC Violations .....	7-4
Using a Test-Mode signal for DRC fixing.....	7-5
How to handle multiple internal clocks for Test?.....	7-6
Bypass or On-Chip Clocking (OCC) .....	7-7

# Table of Contents

Single Test Clock Issue.....	7-8
Option 1: Ideal Solution.....	7-9
Option 2: Multiple OCC Controllers .....	7-10
Related Clock Violations .....	7-11
Power-On Reset Solution.....	7-12
Internal-Reset Violation.....	7-13
Internal-Reset Solution .....	7-14
Alternate Internal-Reset Solution .....	7-15
Another Internal-Reset Solution .....	7-16
DRC Fixing: Agenda .....	7-17
Typical AutoFix Flow.....	7-18
When to Use AutoFix?.....	7-19
AutoFix for Uncontrolled Internal Clocks.....	7-20
AutoFix for Uncontrolled Reset/Set .....	7-21
Local Control of AutoFix.....	7-22
Logic Added by AutoFix .....	7-23
Examples of AutoFix Test Points .....	7-24
Simple AutoFix Script (1/2).....	7-25
Simple AutoFix Script (2/2).....	7-26
Preview of Test Points .....	7-27
Local Exclusion with AutoFix .....	7-28
General Test Point Insertion .....	7-29
DRC Fixing: Agenda .....	7-30
Identifying Tristate DFT Issues .....	7-31
Some Tristate Faults are Testable .....	7-32
An Undetectable Tristate Enable Fault.....	7-33
Adding a Pull-Up Resistor .....	7-34
Adding a Bus Keeper .....	7-35
ATPG with a Bus Keeper .....	7-36
DFT Improves Tristate Coverage .....	7-37
Another Low-Coverage Case.....	7-38
Identifying Tristate DFT Issues .....	7-39
Contention in Scan Shift (1/3) .....	7-40
Contention-Free Scan (2/3).....	7-41
Contention-Free Scan (3/3).....	7-42
Controlling Bidi Direction for Scan Shift.....	7-43
Identifying Tristate DFT Issues .....	7-44
Contention During Capture.....	7-45
Can Your ATE Observe Zs? .....	7-46
DRC Fixing: Agenda .....	7-47
AutoFix for Fixing Internal Tristate Buses .....	7-48
Override Global Internal Bus Fixing .....	7-49
DRC Fixing: Agenda .....	7-50
AutoFix for Bidirectionals .....	7-51
Customizing Bidirectional Control DFT Logic .....	7-52
Enabling Output During Scan .....	7-53

# Table of Contents

Unit Summary.....	7-54
Lab 7: DFT for Clocks and Resets.....	7-55
Command Summary (Lecture, Lab) .....	7-56
Appendix.....	7-57
User Defined Test Points .....	7-58
Types of User Defined Test Points .....	7-59
Specifying a Test Point Element.....	7-60
Test Point Types .....	7-61
UDTP Types .....	7-62
How to specify Control and Clock signals .....	7-63
Enabling Control or Observe Registers .....	7-64
Specifying a Control Source or Observe Sink .....	7-65
Enabling Scan Register Test Point Enables .....	7-66
Specifying a Test Point Enable .....	7-67
Test Point Register Sharing .....	7-68
Test Point Enable Register Sharing .....	7-69
Saving Power .....	7-70
Example UDTP Flow Script (1/2) .....	7-71
Example UDTP Flow Script (2/2) .....	7-72
Recommendation for At-speed Tests.....	7-73

---

## Unit 8: Top-Down Scan Insertion

Unit Objectives .....	8-2
Typical Scan Insertion Flow .....	8-3
Top-Down Scan Insertion: Agenda .....	8-4
Specifying Scan Architecture .....	8-5
Using set_scan_configuration (1/5) .....	8-6
Using set_scan_configuration (2/5) .....	8-7
Using set_scan_configuration (3/5) .....	8-8
Using set_scan_configuration (4/5) .....	8-9
Using set_scan_configuration (5/5) .....	8-10
Using Lock-up Elements.....	8-11
Reporting & Resetting Scan Configurations.....	8-12
Top-Down Scan Insertion: Agenda .....	8-13
A Fast Iteration Loop .....	8-14
What Does Scan Preview Do? .....	8-15
Typical Scan Preview Log .....	8-16
Using preview_dft -show all .....	8-17
Top-Down Scan Insertion: Agenda .....	8-18
Specifying DFT Insertion Parameters.....	8-19
Using set_dft_insertion_configuration.....	8-20
Rapid Scan Synthesis (RSS) .....	8-21
Avoiding Default “Test Uniquification” .....	8-22
Preserving Design Names .....	8-23

# Table of Contents

Synopsys Test Tip.....	8-24
Note: Inserted DFT Logic Not Optimized.....	8-25
What Does Scan Insertion Do?.....	8-26
Gate-Level Remapping .....	8-27
Design Compiler Topographical Mode (DCT).....	8-28
DFT Flow in DCT.....	8-29
DCT Details .....	8-30
Top-Down Scan Insertion: Agenda .....	8-31
Why Balanced Scan Chains? .....	8-32
Wasted ATE Pattern Memory (1/3).....	8-33
Wasted ATE Pattern Memory (2/3).....	8-34
Wasted ATE Pattern Memory (3/3).....	8-35
Scan Access Pins Vs. Tester Time.....	8-36
Using Functional Output as Scan-Out Port.....	8-37
Using Functional Input Pin as Scan-In Port.....	8-38
Specify Chain Count .....	8-39
How DFTC Balances Scan Chains .....	8-40
What Does DFTC Consider a Clock Domain? .....	8-41
How DFTC Considers Dual-Edge Clocking.....	8-42
An Internal Clock Domain.....	8-43
Specifying Internal Clocks.....	8-44
Internal Clocks and Clock Gating.....	8-45
Mixing Clocks: Conservative .....	8-46
Scan Ordering by NICE Rule .....	8-47
Mixing Clocks: Need Lock-ups .....	8-48
Rules for Inserting Lock-up Latches.....	8-49
Rules for Inserting Lock-up Latches.....	8-50
Clock Equivalence .....	8-51
Test - How Many Scan Chains?.....	8-52
Top-Down Scan Insertion: Agenda .....	8-53
Scan Path Exclusion.....	8-54
How to Control Unscanning .....	8-55
Unscan/Exclude Behavior.....	8-56
What Defines a Scan Chain?.....	8-57
The set_scan_path command .....	8-58
Scan Chain Access Ports.....	8-59
Scan Chain Elements .....	8-60
How to Specify a Scan Path Using Lists .....	8-61
Example: -ordered_elements.....	8-62
Explicit Chain Length and Clock Control .....	8-63
Multiple Scan Enables .....	8-64
Example Multiple Scan Enable Preview.....	8-65
Managing Scan Paths .....	8-66
Unit Summary.....	8-67
Lab 8: Top-Down Scan Insertion.....	8-68
Command Summary (Lecture, Lab) .....	8-69

# Table of Contents

Appendix.....	8-70
Scan Groups.....	8-71
Scan Group Commands .....	8-72
Using set_scan_group .....	8-73
Using set_scan_group .....	8-74
What is Supported for Members of a Group?.....	8-75

---

## Unit 9: Exporting Design Files

Unit Objectives .....	9-2
Exporting Design Files: Agenda.....	9-3
ATE: Final Destination of Handoff .....	9-4
Protocol Path from DFTC to ATE.....	9-5
Inside the ATE .....	9-6
How a Fault is Detected.....	9-7
Individual Pin Modes.....	9-8
ATE “Executes” STIL Test Program .....	9-9
Pin Timing from Protocol in STIL Format.....	9-10
Exporting Design Files: Agenda.....	9-11
DFTC-to-TetraMAX Flow.....	9-12
DFTC Handoff Script .....	9-13
What Affects the Test Protocol? .....	9-14
STIL Protocol File Structure.....	9-15
TetraMAX Baseline Flow.....	9-16
Exporting Design Files: Agenda.....	9-17
Physical DFT .....	9-18
Example: DFT Optimization.....	9-19
DC / IC Compiler DFT Flow .....	9-20
DDC Based SCANDEF Data.....	9-21
DDC Based SCANDEF Benefits.....	9-22
What is SCANDEF? .....	9-23
Partitioning with SCANDEF .....	9-24
Example: SCANDEF File.....	9-25
Alpha Numeric Ordering .....	9-26
Reordering Within Scan-Chain.....	9-27
Reordering Across Scan-Chains .....	9-28
Clock Tree Based Reordering .....	9-29
Example Script.....	9-30
SCANDEF Notes .....	9-31
Exporting Design Files: Agenda.....	9-32
Formality Support .....	9-33
What DFT Data is Passed to Formality?.....	9-34
Formality Sample Script .....	9-35
Limitations for SVF .....	9-36
Unit Summary.....	9-37

# Table of Contents

Lab 9: Export Design Files .....	9-38
Command Summary (Lecture, Lab) .....	9-39

---

## Unit 10: High Capacity DFT Flows

Unit Objectives .....	10-2
High Capacity DFT Flow: Agenda .....	10-3
Hierarchical Scan Synthesis (HSS).....	10-4
Test Models.....	10-5
Test Model Contents .....	10-6
How to Create Test Models .....	10-7
How to Write/Read Test Models .....	10-8
How to Enable/Disable Test Models .....	10-9
How to Report Test Models.....	10-10
Test Model Usage Reported During dft_drc.....	10-11
Linking a Test Model to a Library Cell .....	10-12
How to Verify a Test Model? .....	10-13
High Capacity DFT Flow: Agenda .....	10-14
Bottom-Up Capacity Issues .....	10-15
Recall: Top-Down Scan Insertion Flow .....	10-16
Bottom-Up Scan Insertion .....	10-17
Solving Bottom-Up Bottleneck with HSS .....	10-18
Saving Test Models.....	10-19
Reading Test Models for Top-Level Stitching .....	10-20
Test for Understanding .....	10-21
Test Models Versus Complete Designs .....	10-22
Getting the Most Benefit from Test Models .....	10-23
Solving Bottom-Up Bottleneck with ILMs.....	10-24
Creating an ILM.....	10-25
High Capacity DFT Flow: Agenda .....	10-26
Bottom-Up Balancing Issues .....	10-27
Avoid Very Long Subdesign Chains .....	10-28
Create Short Block Level Chains.....	10-29
How to Control Block Chain Length .....	10-30
Bottom-Up Flow for Better Top-Level Balance.....	10-31
High Capacity DFT Flow: Agenda .....	10-32
Bottom-Up Clock Domain Issues .....	10-33
Multiple Clocks Limit Balancing.....	10-34
No Block Level Lock-up Latches .....	10-35
End of Chain Lock-up Latches .....	10-36
Block Level Tip: Avoid Clock Mixing.....	10-37
Top-Level Tip: Allow Clock Mixing.....	10-38
Test for Understanding .....	10-39
Script for Block Level HSS .....	10-40
Script for Top-Level HSS .....	10-41

# Table of Contents

Enhanced DFT DRC Reporting With HSS.....	10-42
High Capacity DFT Flow: Agenda .....	10-43
Example Script (DC Block-level).....	10-44
Example Script (DC Top-level).....	10-45
SCANDEF Example with Test Models .....	10-46
Unit Summary.....	10-47
Lab 9: High Capacity DFT Flows.....	10-48
Command Summary (Lecture, Lab) .....	10-49
Appendix A.....	10-50
CTL Syntax: Scan Chain .....	10-51
CTL Syntax: Procedures .....	10-52
CTL Syntax: Scan Signals (1/2) .....	10-53
CTL Syntax: Scan Signals (2/2) .....	10-54
CTL Syntax: Terminal Lock-ups.....	10-55
CTL Syntax: Feedthroughs .....	10-56
Appendix B .....	10-57
How to Extract a Test Model? .....	10-58
Flow for Extraction .....	10-59
Appendix C .....	10-60
Script to Generate CTL Test Models .....	10-61

---

## Unit 11: Advanced Features

Unit Objectives .....	11-2
Multi-Mode DFT: Agenda.....	11-3
Why Use Multi-Mode? .....	11-4
Multiple Test Configurations.....	11-5
Multiple Package Configurations .....	11-6
Embedded Core Applications .....	11-7
Multi-Mode DFT: Agenda.....	11-8
Using Multi-Mode .....	11-9
Specifying Mode Control Ports .....	11-10
Using define_test_mode.....	11-11
Encoding Example .....	11-12
Encoding Example: preview_dft Report.....	11-13
Multi-Mode DFT: Agenda.....	11-14
Performing Mode-Specific Specifications.....	11-15
Multi-Mode Scan Specification .....	11-16
preview_dft report in Multi-Mode scan .....	11-17
Example: preview_dft report .....	11-18
Example: Multi-mode Multiplexers.....	11-19
Example: Internal_scan Mode .....	11-20
Example: burnin Mode .....	11-21
Multi-Mode DFT: Agenda.....	11-22
Running DRC in Multiple Modes.....	11-23

# Table of Contents

Writing Test Protocol.....	11-24
Multi-Mode: Example Script .....	11-25
Unit Summary.....	11-26
Command Summary (Lecture, Lab) .....	11-27

---

## Unit 12: DFT MAX

Unit Objectives .....	12-2
DFT MAX: Agenda .....	12-3
Why Use Compression?.....	12-4
How Compression Works .....	12-5
Test Compression Concepts.....	12-6
Test Cycle Savings.....	12-7
Adaptive Scan Compression .....	12-8
Test Modes Created During Insertion.....	12-9
Shared Scan-Ins Can Reduce Coverage.....	12-10
Adaptive Scan Reduces Dependency.....	12-11
Load Decompressor Principle	
Multiple Shared Scan-in Configurations .....	12-12
Built-In (Default) X-Tolerance .....	12-13
Some Chains Still Good, Some Bad .....	12-14
Scan Output Redundancy.....	12-15
This Row Is All Good.....	12-16
Some Still Bad, But Others Good.....	12-17
With Many X's, All Paths May Be Blocked.....	12-18
High X-Tolerance .....	12-19
DFT MAX: Agenda .....	12-20
DFT Compiler: Regular Scan Flow .....	12-21
DFT Compiler: DFT MAX Flow.....	12-22
DFT MAX Commands (1/2).....	12-23
DFT MAX Commands (2/2).....	12-24
Incremental Improvements in TATR.....	12-25
DFT MAX & Test Modes.....	12-26
TestMode Signal .....	12-27
DFT MAX: Agenda .....	12-28
Bottom Up HSS Flow .....	12-29
Hierarchical Adaptive Scan Synthesis (HASS) .....	12-30
Typical HASS Flow .....	12-31
HASS Example: Multiple Compressed Cores .....	12-32
HASS Example: Adaptive & Regular Scan Cores .....	12-33
HASS Example: Hybrid Flow .....	12-34
HASS Details .....	12-35
HASS ATPG vs. Regular Scan ATPG .....	12-36
Example Script: Top Level Integration.....	12-37
DFT MAX: Agenda .....	12-38

# Table of Contents

Multi-Mode with DFT MAX .....	12-39
Setting the Base Mode .....	12-40
Controlling Clock Mixing by Mode .....	12-41
Multi-Mode Adaptive Scan Example .....	12-42
How to Generate Output for TetraMAX.....	12-43
Unit Summary.....	12-44
Lab 12: DFT MAX .....	12-45
Command Summary (Lecture, Lab) .....	12-46

---

## Unit 13: Conclusion

Workshop Goal .....	13-2
Curriculum Flow .....	13-3
Advanced DFT Compiler Topics.....	13-4
What is TetraMAX?.....	13-5
Want More Training?.....	13-6
Helpful SolvNet Articles.....	13-7
How to Download Lab Files (1/3) .....	13-8
How to Download Lab Files (2/3) .....	13-9
How to Download Lab Files (3/3) .....	13-10
Course Evaluation.....	13-11
Thank You! .....	13-12

---

## Unit CS: Customer Support

Synopsys Support Resources .....	CS-2
SolvNet Online Support Offers.....	CS-3
SolvNet Registration is Easy .....	CS-4
Support Center: AE-based Support.....	CS-5
Other Technical Sources .....	CS-6
Summary: Getting Support .....	CS-7



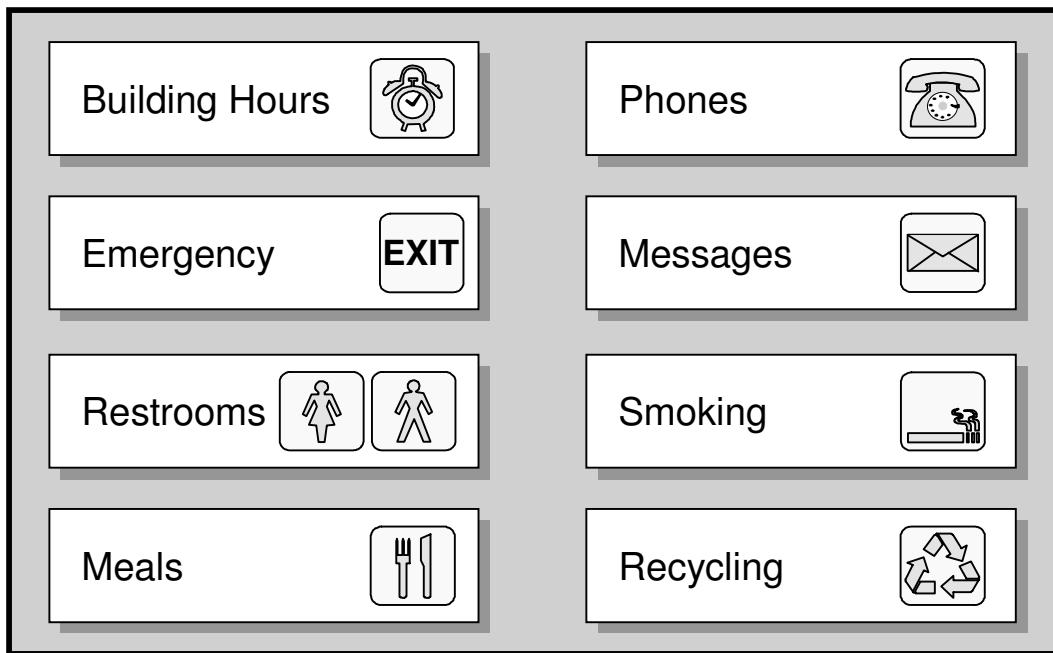
# DFT Compiler 1

2007.12

**Synopsys Customer Education Services**  
© 2008 Synopsys, Inc. All Rights Reserved

Synopsys 30-I-011-SSG-012

# Facilities



Please turn off cell phones and pagers

i-2

# Workshop Prerequisites

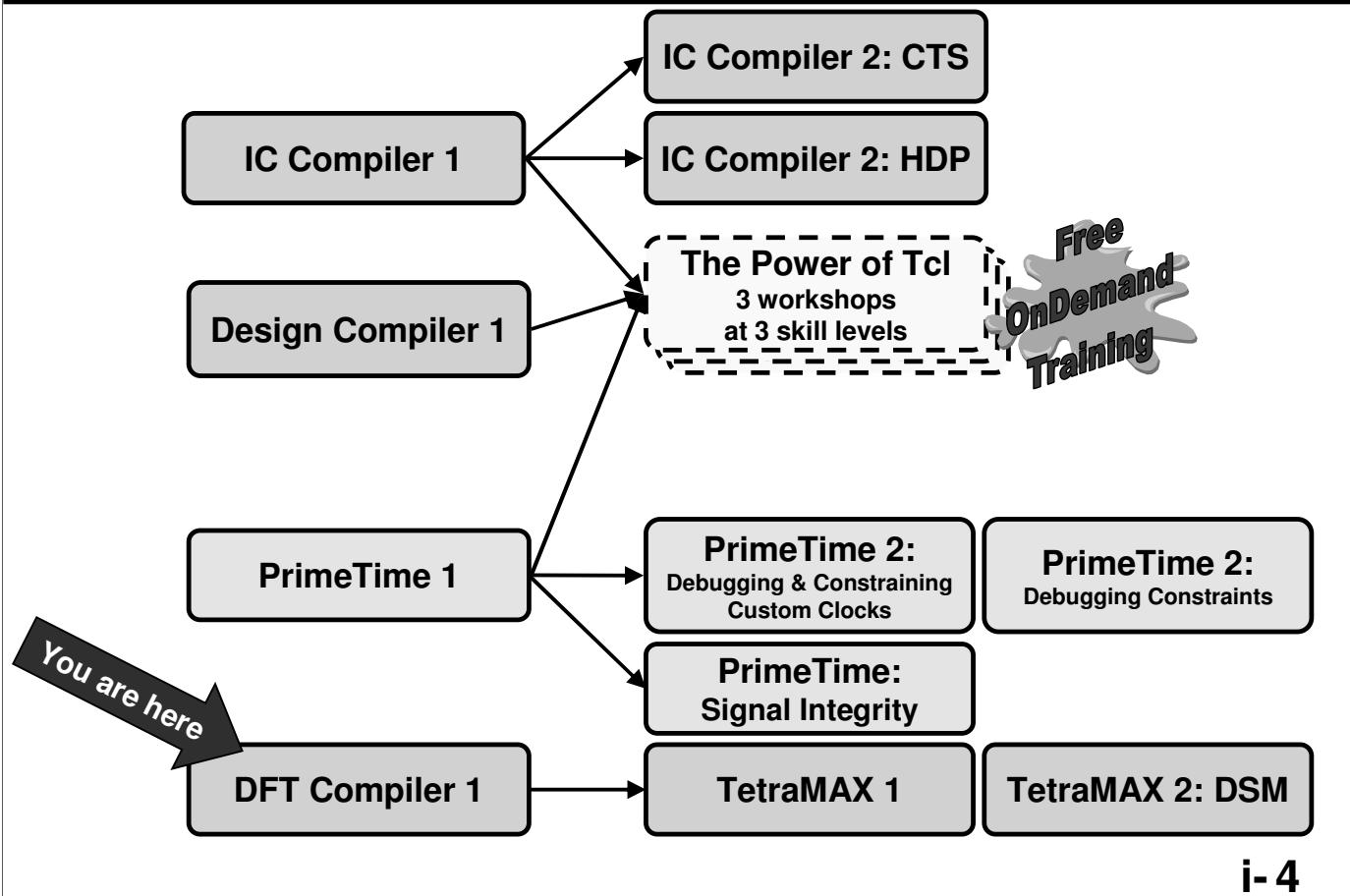
---

**You should have experience in the following areas:**

- Digital IC design
- Verilog or VHDL
- UNIX and X-Windows
- A Unix based text editor

**i-3**

# Curriculum Flow



The entire Synopsys Customer Education Services course offering can be found at:

<http://training.synopsys.com>

Synopsys Customer Education Services offers workshops in two formats: The “classic” workshops, delivered at one of our centers, and the virtual classes, that are offered conveniently over the web. Both flavors are delivered *live* by expert Synopsys instructors.

In addition, a number of workshops are also offered as OnDemand playback training for FREE! Visit the following link to view the available workshops:

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

(see under “Tool and Methodology Training”)

# Target Audience

**Design and Test engineers who need to identify and fix DFT violations in their RTL or gate-level designs, insert scan into multi-million gate SoCs, and export design files to ATPG and P&R tools**



i-5



SoC	System On a Chip
DFT	Design for Test
RTL	Register Transfer Level, for example, VHDL and Verilog
ATPG	Automatic Test Pattern Generation
P&R	Place and Route, also known as layout

# Introductions

---

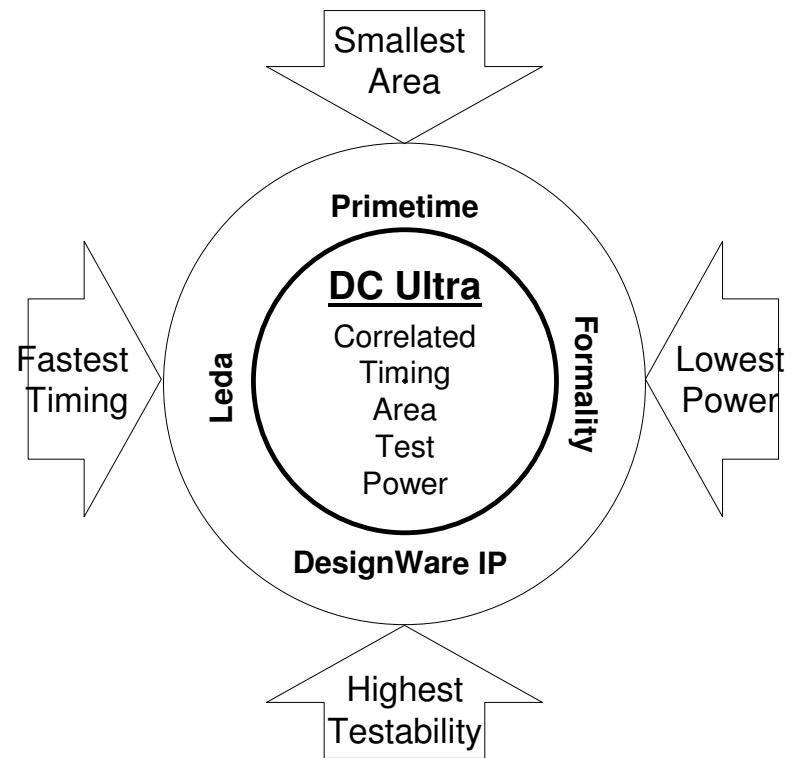
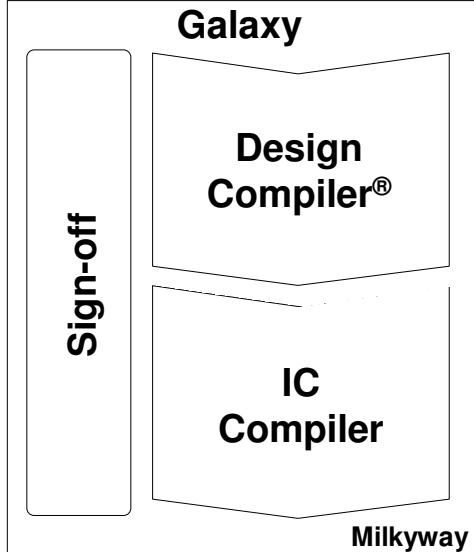
- Name
- Company
- Job Responsibilities
- EDA Experience
- Main Goal(s) and Expectations for this Course

i-6



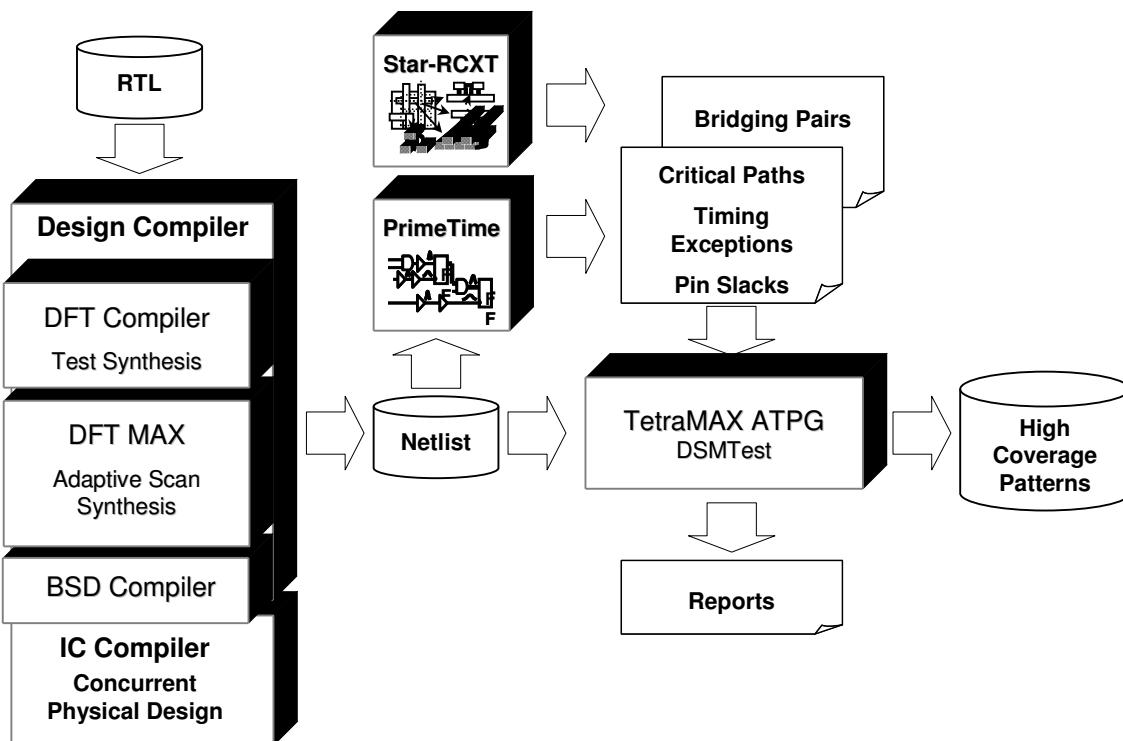
EDA      Electronic Design Automation

# Galaxy™ Design Platform



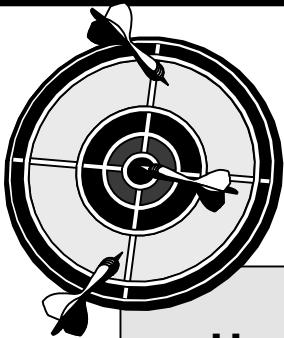
i- 7

# Galaxy™ Design Platform for Test



i-8

# Workshop Goal



**Use DFT Compiler to check RTL and mapped designs for DFT violations, insert scan chains into very large multi-million gate designs, and export all the required files for downstream tools**

i-9

# Agenda

**DAY  
1**

**1** Introduction to Scan Testing

**2** DFT Compiler Flows

**3** DFT Compiler Setup

**4** Test Protocol

**5** DFT Design Rule Checks

i- 10

# Agenda

**DAY  
2**

**6 DFT DRC GUI Debug**



**7 DRC Fixing**



**8 Top-Down Scan Insertion**



**i-11**

# Agenda

**DAY  
3**

- 9 Export** 
- 10 High Capacity DFT Flow** 
- 11 Multi-Mode DFT** 
- 12 DFT MAX** 
- 13 Conclusion** 

i-12

# Icons Used in this Workshop



**Lab Exercise**



**Caution**



**Recommendation**



**Definition of Acronyms**



**For Further Reference**



**Question**



**“Under the Hood” Information**



**Group Exercise**

**i- 13**

# Test Automation Docs are on SolvNet!

[https://solvnet.synopsys.com/dow\\_retrieve/A-2007.12/ni/test.html](https://solvnet.synopsys.com/dow_retrieve/A-2007.12/ni/test.html)



SOLVNET HOME | SYNOPSYS.COM | FEEDBACK | SITE MAP | HELP | SIGN OUT

Documentation Support Center Download Center Training Center My Profile

SolvNet®

Search

DOW Home | Search | Browse Articles | Archives | Help

Test Automation

## DFT Compiler/DFT MAX

- DFT Compiler User Guide: Scan, version A-2007.12
- DFT MAX User Guide: Adaptive Scan, version A-2007.12
- DFT Overview User Guide, version A-2007.12

## TetraMAX

- Test Pattern Validation User Guide, version A-2007.12
- TetraMAX ATPG User Guide, version A-2007.12

## BSD Compiler

- BSD Compiler Reference Manual, version Z-2007.03
- BSD Compiler User Guide, version A-2007.12

## Quick Reference

- Test Automation Quick Reference, version A-2007.12
- TetraMAX ATPG Quick Reference, version A-2007.12

## Synthesis Man Pages, version A-2007.12

- Tool Invocation Commands (man1)
- Synthesis Commands (man2)
- Variables and Attributes (man3)
- Error Messages (mann)

## Documentation on the Web:

[https://solvnet.synopsys.com/dow\\_search](https://solvnet.synopsys.com/dow_search)

i-14

# Agenda

**DAY  
1**

**1 Introduction to Scan Testing**

**2 DFT Compiler Flows**

**3 DFT Compiler Setup**

**4 Test Protocol**

**5 DFT Design Rule Checks**

# Unit Objectives



**After completing this unit, you should be able to:**

- **Explain how to use the D-algorithm to generate a pattern that detects a given Stuck-At fault in a combinational design**
- **Do the same in a full scan sequential design**
- **Explain why scan-chains are necessary to support ATPG**

**1-2**

# Introduction to Scan Testing: Agenda

**Design For Manufacturing Test**



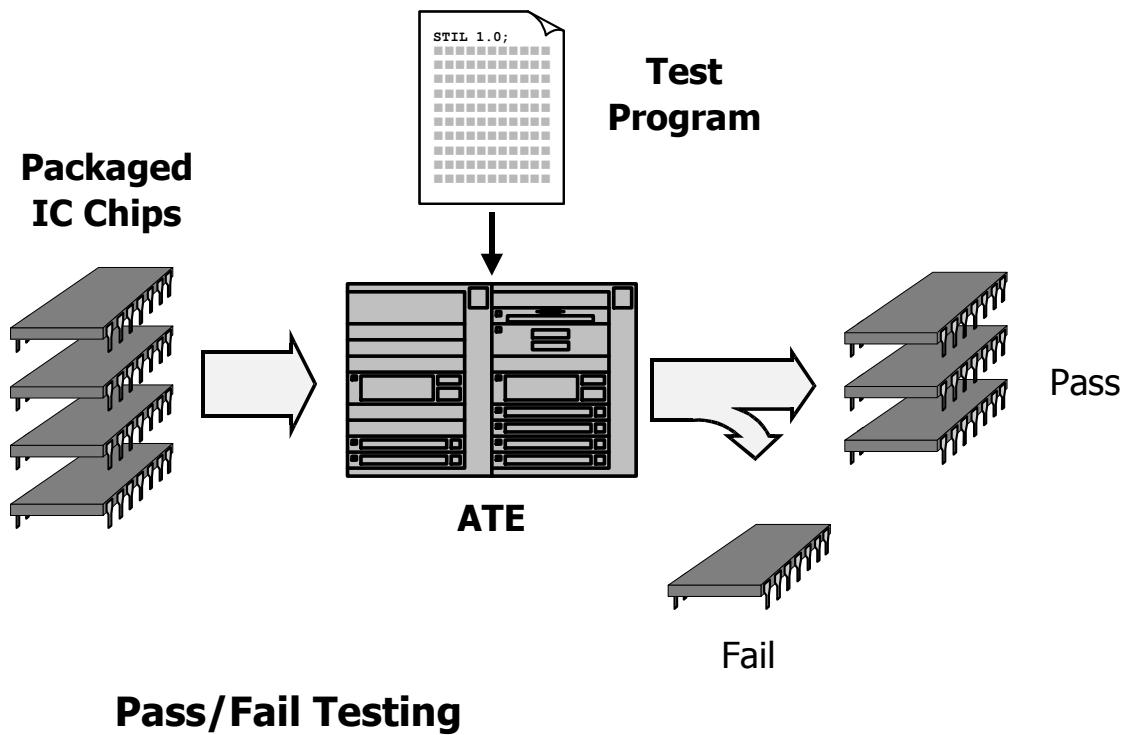
**D-algorithm as applied to purely combinational logic**



**D-algorithm as applied to sequential logic (Full Scan)**

1-3

# What is Manufacturing Test?



1-4

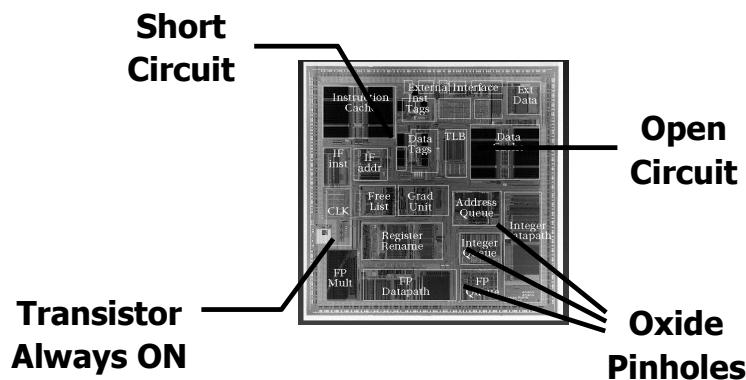
- High-volume production testing is a **pass/fail** proposition.
- Packaged IC chips are fixtured and tested on **automated test equipment** (ATE).
- The test equipment simply **separates** functional units from those with **physical defects**.
- The ATE steps through a **test program**—a lengthy series of tests for potential defects. The program shown here is written in IEEE STIL (Standard Test Interface Language).
- If a unit **fails any test** in the program, it is discarded—or sent to the failure-diagnosis lab. Only those units that **pass every test** in the program are ever shipped to the end user.
- Today, automatic test-pattern generation (ATPG) tools generate most test programs; thus, the **logic designer** can be increasingly involved in test-program development.

# What Is a Physical Defect?

## Physical Defect:

A on-chip **flaw** introduced during fabrication or packaging of an individual ASIC that makes the device **malfunction**

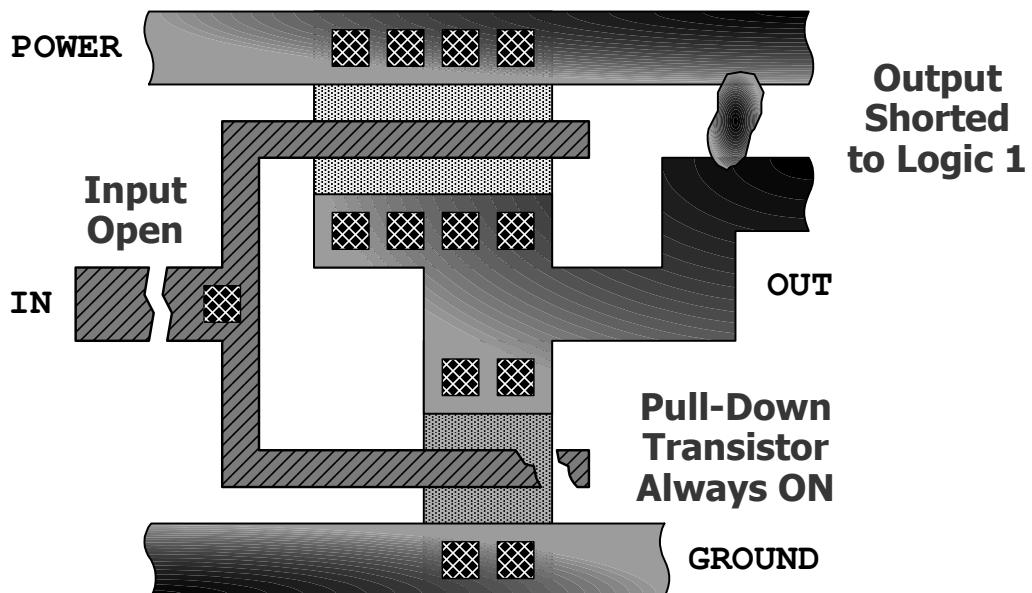
### Common Physical Defects



1-5

- The test techniques that will be covered in this workshop do **not** look for **design errors**.
- They test for **manufacturing defects** introduced during semiconductor processing.
- These defects “land” on an IC in a wide variety of ways:
  - **Open** and **short circuits**.
  - **Bridging** between metal lines.
  - Conductive **pinholes** through insulating oxides.
  - etc.
- In this workshop, only **permanent** defects that alter the logic **function** are being considered.
- Sufficient time for all logic outputs to **settle** before checking responses is assumed.  
This is known as **static (low-speed)** testing—**at-speed** testing will not be covered.
- Briefly mentioned are defects that affect **propagation delay**, but not correct function.

# Physical Defects in CMOS



This physical view of a CMOS inverter has several defects!

1-6

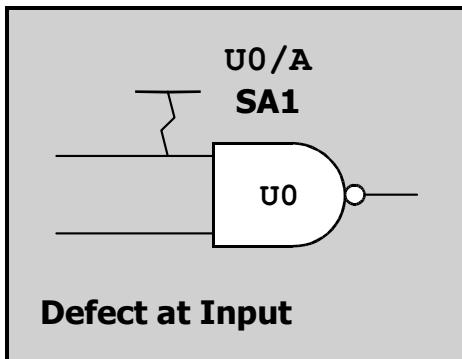
This is a physical layout (polygon level) view of a simple CMOS inverter:

- It consists of an *n*-type **pull-down transistor** and a *p*-type **pull-up transistor**.
- A MOS transistor is the **area of overlap** between a polysilicon and diffusion line.
- Polysilicon lines are shown in pink, and *n*- or *p*-type diffusion lines in green or yellow.
- A one-micron **dust particle** landing on 90 nm geometry can easily cause an open.
- Excess **un-etched metal** can cause bridging, or direct shorts to the power or ground rails.
- A defective pull-down transistor that is **always on** can act like a direct short to ground.

## Conclusion:

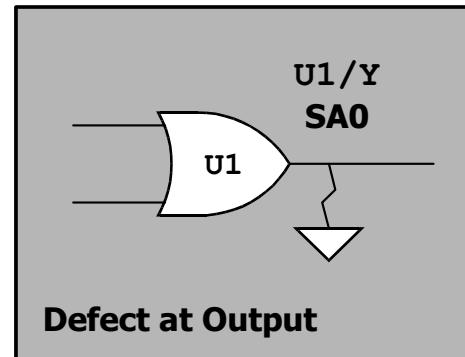
- Many—though not all—defects behave just like permanent shorts to power or ground.
- These kind of defects cause input or output pins to appear **stuck at logic 0** or **logic 1**.
- The input and output stages of most CMOS gates are patterned after the basic inverter; thus the simple model of an input or output pin stuck at **0** or **1** has **wide applicability**.

# Stuck-At Fault Model



## SA1 Fault:

Due to a defect, input pin A of U0 acts as if stuck high, independent of input signal



## SA0 Fault:

Due to defect, output pin Y of U1 acts as if stuck low, independent of the inputs

## Fault Model:

A logical model representing the effects of a physical defect

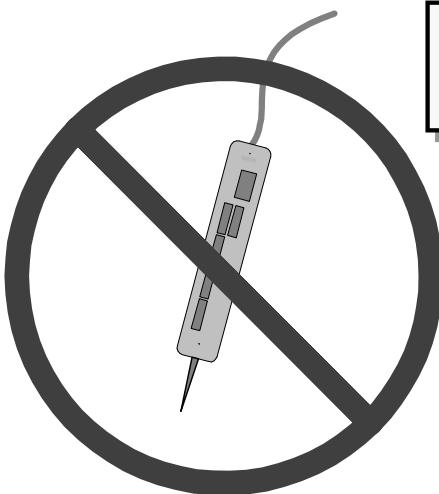
1-7

- The semiconductor community does not attempt to model physical defects **directly**—relying on simpler, technology independent **logical models** instead.
- The **Stuck-At Fault** (SAF) model is still the most prevalent fault model in use today.
- This slide shows **common examples** of stuck-at faults on gate input or output pins.
- The **fault symbols** on the slide are not industry standard, but are useful for illustration.
- Defects land at **random locations** on a wafer in a semiconductor processing plant. You can therefore assume that stuck-at faults can occur **anywhere** in the logic on a chip. **Any pin** on a CMOS gate or flip-flop can be the site of a potential SA0 or SA1 fault.
- Chip complexity prevents us from considering stuck-at faults at the **transistor** level.
- **Other** fault models exist, but a key advantage of the stuck-at model is its **simplicity**. It significantly reduces the complexity of CPU-intensive ATPG algorithms.

## Technical Reference:

Timoc, Buehler, et al., *Logical Models of Physical Failures* [Proc. ITC (Oct.1983)], p.546

# Rules for Detecting a Stuck-At Fault



**Internal Probing  
of IC Not Practical!**

## **Rules of the Game:**

Tester access to the device-under-test (DUT)  
is **only** allowed through its **primary** I/O ports

**1-8**

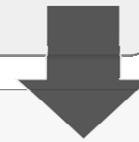
- Historically, production testing was largely done on PC boards, using bed-of-nails fixtures to probe for stuck-at faults located within **system-on-a-board** hardware.
- Today, most of the logic is inside a **system-on-a-chip**—making nodes **inaccessible**.
- Production chip testers can access internal logic **only** through the **primary** I/O ports.
- **Primary port** means an I/O port that is directly accessible to external ATE hardware.
- This corresponds to a **package pin** on a chip that has already gone through packaging; thus the ATE can:

Individually drive—or control—the DUT’s primary inputs (**PIs**).

Individually strobe—or observe—the DUT’s primary outputs (**POs**).

# Introduction to Scan Testing: Agenda

**Design For Manufacturing  
Test**



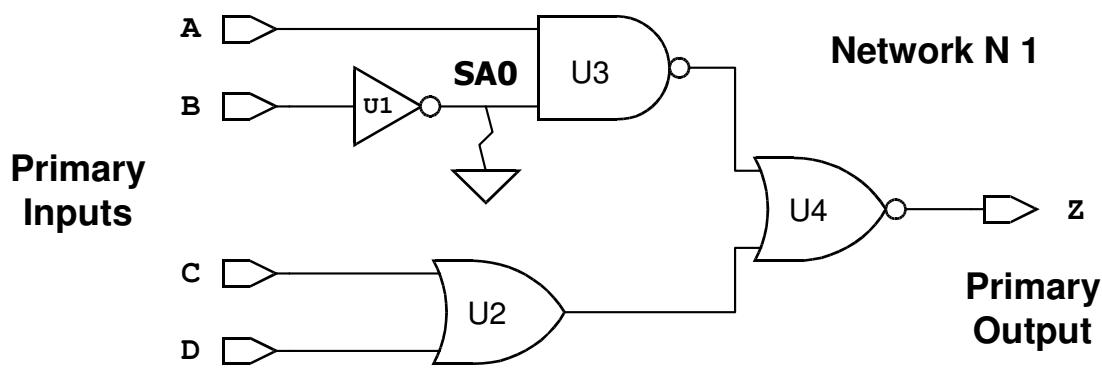
**D-algorithm as applied to  
purely combinational logic**



**D-algorithm as applied to  
sequential logic (Full Scan)**

1-9

# Combinational Logic: Stuck-At Fault Testing



If this SA0 fault is **not** present,  
then node can be driven to **1**

Else SA0 fault **is** present,  
and U1/Y remains at **0**

This if/else behavior can be exploited to detect the fault!

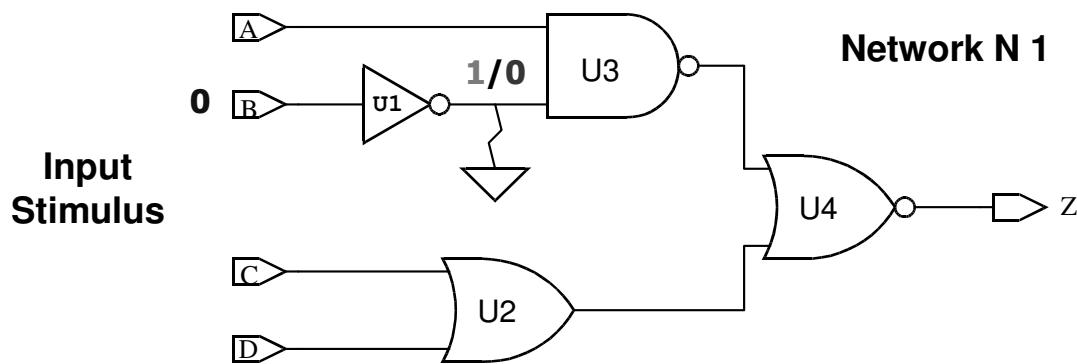
1-10

- Presented now the **classic algorithm** for detecting almost any stuck-at fault.
- This automated procedure, from IBM in the Sixties, is known as the **D-algorithm**. It detects a Stuck-At fault anywhere in combinational logic—yet requires no internal probing.
- Since the original 1966 paper, many spin-offs of the **D-algorithm** have been published. Its fundamental concepts are still essential to understanding most ATPG tools.

## Technical Reference:

J. P. Roth, *Diagnosis of Automata Failures*,  
IBM Journal Res. & Dev., Vol. 10, No. 4,  
p. 278, July 1966

# Activate the SA0 Fault (1/4)



## D-Algorithm:

1. Target a specific stuck-at fault
2. Drive fault site to **opposite** value  
*(continued)*

**Fault-Free Value**  
**Faulty Value**  
**Legend**

**1/0**

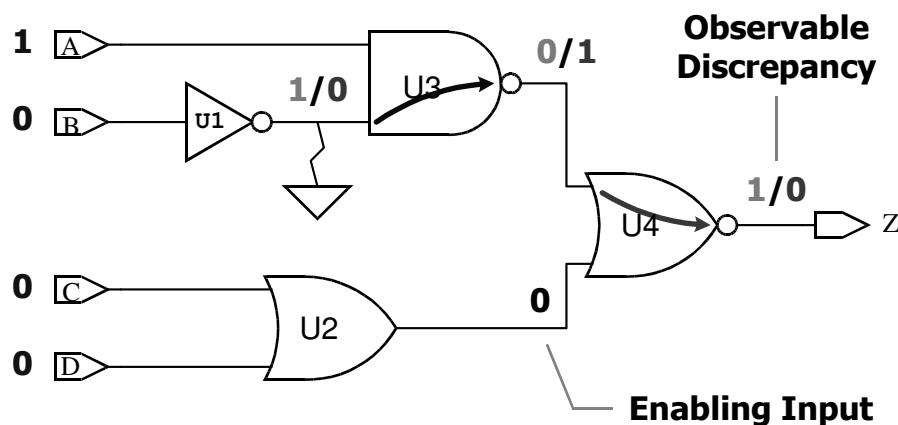
1-11

- The D-algorithm detects **one stuck-at fault** at a time, making it very CPU-intensive.
- You will apply the algorithm step by step to logic network N 1, a very simple chip.
- The first step is to **target** a particular fault—e.g., the SA0 fault on the output of **U1**.
- The next step is to **activate** the target fault by driving the node to the **opposite** value.  
That is easy to do in network N 1—just tell the ATE to apply a 0 to primary input **B**.
- This creates a **fault effect**—a measurable logic difference—at the fault site **U1/Y**.
- If the target fault is **not** present on a manufactured copy of the chip, then **U1/Y** is 1. This possibility is indicated by the **fault-free** value shown in green.
- If this fault **is** present on a manufactured chip, then **U1/Y** remains stuck at 0. This possibility is indicated by the **faulty** value in red.

## Conclusion:

- The D-algorithm creates a **discrepancy** between the **fault-free** and the **faulty** values.
- The composite logic value **1 / 0** is often symbolized by **D**.
- The **D** (for **discrepancy**) gives the algorithm its name.

## Propagate Fault Effect (2/4)



### D-Algorithm:

1. Target a specific stuck-at fault
2. Drive fault site to **opposite** value
3. **Propagate** error to primary output  
*(continued)*

The **1/0** is **inverted**,  
but the discrepancy is  
still easy to **measure**

1-12

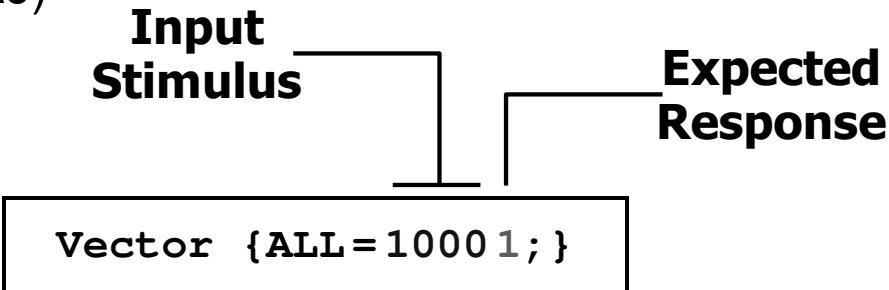
- The third step is to **propagate** the fault effect to a PO, where it is accessible to the ATE.
- It may be **inverted** by logic along the path—but the measurable **discrepancy** remains.
- The ATE then **strokes** this primary output port at a specific time in the test program.  
In **static** testing, sufficient time must elapse for outputs to settle to their **steady state**.
- If the ATE sees the **expected** fault-free response, then the fault is assumed **not** present.
- The ATE continues to execute the rest of the test program, targeting other faults.
- If it sees the **faulty** response, the SA0 fault is assumed present, and a **failure** is logged.

### Justification:

- Activating a fault site, and propagating the fault effect, often require **justification**.
- To justify a node means to force a **0** or **1** onto that node, from the primary inputs.
- In N 1, the OR gate output was justified to **0** by applying **0s** to the ports **C** and **D**.

# Anatomy of a Test Pattern (3/4)

## Test Pattern (U1/Y SA0)



### Test Pattern:

A sequence of one or more vectors that applies a **stimulus** and checks for an expected **response** to detect a target fault

1-13

- This slide shows the test pattern generated by the **D-algorithm** to detect the SA0 fault.
- Every test pattern has an **input stimulus** and **output response** for detecting a fault.
- The blue bit string **1000** is the input stimulus; the green bit **1** is the expected output.
- The pattern is shown in **STIL syntax** as a single vector statement, requiring one cycle.
- The alias **ALL** represents an ordered list of DUT ports, including all the PIs and POs.
- This **test pattern** only has **one** vector; in general, **many** vectors may be required.
- In this workshop, the term **test pattern** is used just as it is defined in the STIL spec.  
A test pattern is a series of **one or more vectors** comprising a test for a target SAF.
- In this workshop, a **vector** will always correspond to exactly one tester **clock cycle**.
- Notice that timing details—like cycle length and strobe time—do not appear here.
- These ATE-oriented details have been decoupled from the actual test-pattern data.
- As you will see in lab, they reside instead in the **test protocol** data for the DUT.

# Record the Test Pattern (4/4)

This pattern detects the fault U1/Y SA0.

```
STIL 1.0;
.
.
Pattern "N1_Burst" {
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
    Vector {ALL=1000 1;}
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
    Vector {■■■■■■■■}
```

Test Program for N 1

## D-Algorithm:

1. Target a specific stuck-at fault
2. Drive fault site to **opposite** value
3. **Propagate** error to primary output
4. **Record** pattern; drop detected fault  
*(continued)*

1-14

- The fourth step in the **D**-algorithm is just bookkeeping. The successful test pattern is **recorded** in memory, and the detected fault is **dropped** from the list of target faults.
- The algorithm is repeated for all the remaining faults, resulting in a **test program**.
- A test program is a series of test patterns that detects all possible faults in the DUT.
- In lab, you will investigate a short but complete test program written out in STIL.

# Assessment of D-algorithm

## Advantages:

- Deterministic step-by-step method of detecting Stuck-At faults
- Exhaustive - succeeds unless a fault is undetectable

## Limitations:

- Generates a test for one stuck-at fault at a time
- Involves decision making at almost every step
- May backtrack excessively for hard-to-test faults

1-15

For a more precise textbook definition, a fault is **detectable** under the following criteria:

- Assume a **fault-free** combinational network **N** with Boolean output function  $Y(x)$ . Since **N** is combinational, any input pattern  $i$  yields a corresponding output  $Y(i)$ .
- In the presence of a single stuck-at fault  $f$ , the output function  $Y(x)$  is **altered**. Designate the output function altered by the target fault  $f$  as  $Y_f(x)$ .
- Apply the **same** input pattern  $i$  to the **faulty** and the **fault-free** networks,  $\mathbf{N}_f$  and **N**.
- The pattern  $i$  is said to **detect** fault  $f$  only if output  $Y_f(i)$  **differs from** output  $Y(i)$ .
- You will see later how the ATE uses **XOR hardware** to flag this binary output difference.
- Not **all** stuck-at faults, however, in a **general** combinational-logic network are detectable.

## Technical Reference:

This definition was adapted from the classic text:

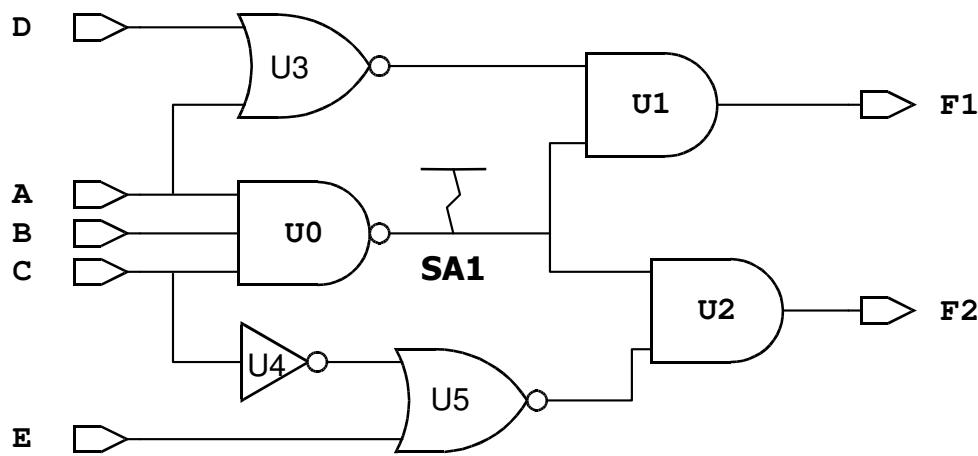
*Digital Systems Testing & Testable Design*

Abramovici, Breuer & Friedman,

[Computer Science Press, 1990] p.95.

This page was intentionally left blank.

# Exercise: Detect SA1 Fault



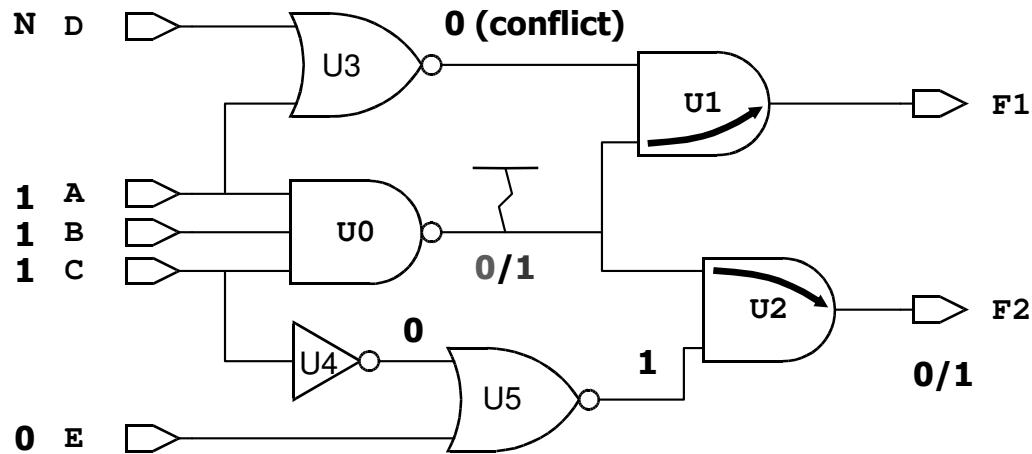
## What To Do:

- Use the **D-algorithm** to generate a test pattern to detect the **SA1** fault
- Record your test pattern (both *stimulus* and *response*) at right

Vector {ALL= \_\_\_\_\_; }

1-17

# Solution: Detect SA1 Fault



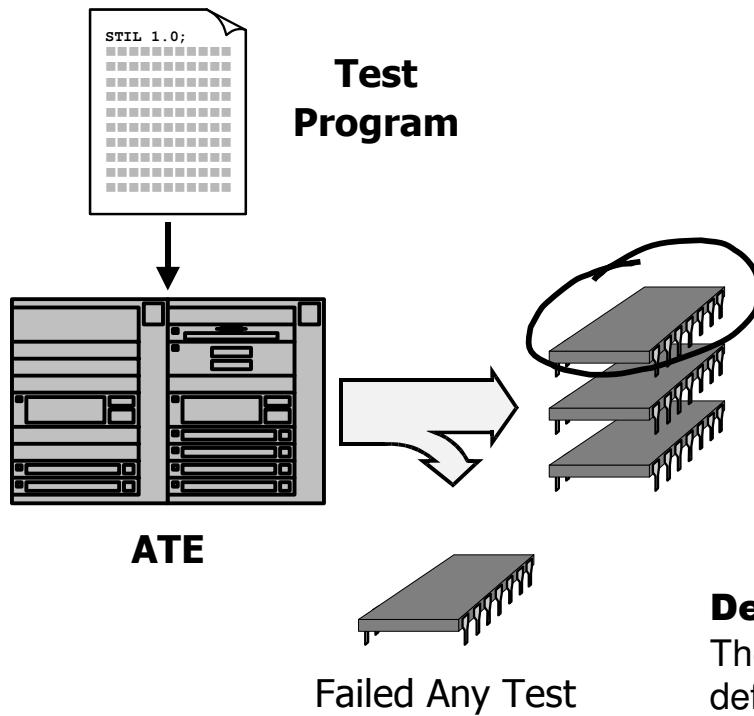
## Legend:

- Input **N** is a don't-care (0 or 1)
- Output **X** is don't-strobe (mask)

Vector {ALL = 111N0 X0; }

1-18

# Is High Coverage Needed?



**Test Escapes:**  
Parts that **pass** every test,  
but still have **undetected**  
defects that reach users!

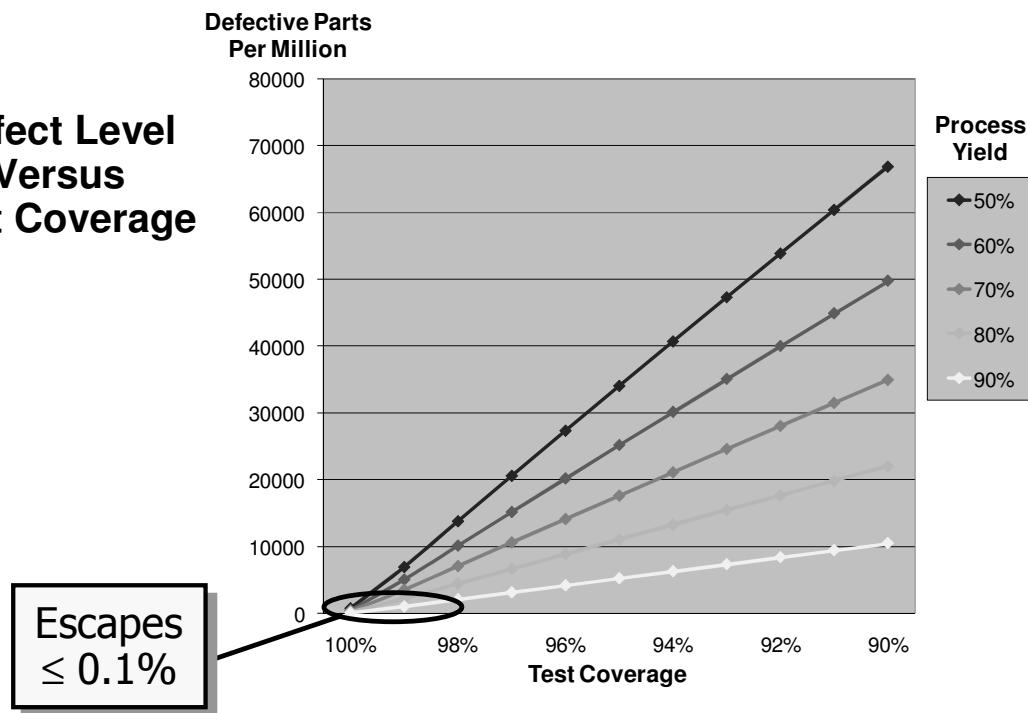
**Defect Level:**  
The **fraction** of test escapes, in  
defective parts per million (**DPPM**)

1-19

- Too many **test escapes** undermine the confidence of the end user in your products.
- In this example, three out of four of the manufactured parts are **passing** all tests.
- Of these three, how many still have defects **undetected by** the test program?
- Defective units can go undetected if the test program coverage is less than 100%.
- The percentage of parts **shipped with defects** to users is termed by the **defect level**.

# High Coverage Is Low DPPM

## Defect Level Versus Test Coverage



You need high test coverage to keep defect levels low!

1-20

- Here the **defect level** is defined as the probability (that is, the fraction) of test escapes.
- This slide shows that **defect level** is a function of **test coverage** and **process yield**.
- **Yield** is the probability that a new, untested semiconductor chip is **free of flaws**. The **lower** the yield, the more **likely** it is that defects exist on the fabricated chip.
- **Test coverage**, is the fraction of **tested** chips that pass all the tests; thus, low yield requires **higher** test coverage to screen out excessive test escapes.

### Conclusion:

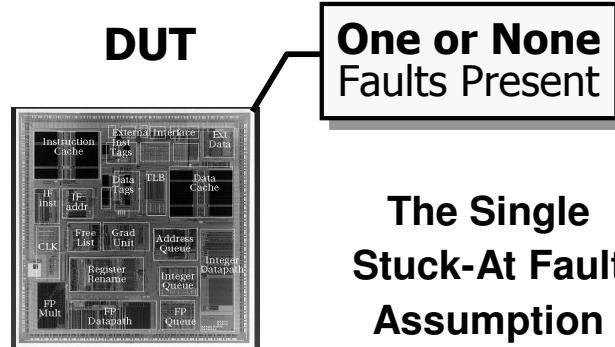
- The curves are only **approximations**, since the Stuck-At Fault model does **not** cover **all** defects.
- The point remains that these parameters are related, not linearly, but **exponentially**.
- For 0.1% defect levels (red area), ensure that test coverage is as **high** as you can get.

### Technical Reference:

- The classic relation between these parameters is:  $DL = 1 - Y^{(1 - FC)}$   
Williams and Brown, *Defect Level as a Function of Fault Coverage*  
[IEEE Trans. Computers, C-30(12)], p.987

# Single Stuck-At Fault Assumption (SSAF)

- There is a possibility that multiple faults in the ASIC can mask faults tested using the SSAF assumption
- The effect of simultaneous Stuck-At faults isn't considered
- Testing for multiple Stuck-At faults is too time consuming
- The SSAF assumption is one possible reason for test escapes



1-21

- A **multiple** fault occurs when **two or more** Stuck-A faults are present simultaneously. Multiple faults are increasingly common on submicron CMOS chips, but the total number of multiple faults is **too high** for reasonable run times; thus, most ATPG tools assume only **one or none** Stuck-At faults are present on a DUT.
- This **Single Stuck-At Fault** (SSAF) assumption simplifies test generation dramatically.
- One reason that this assumption holds up in practice is that every multiple SAF is **a set** of single Stuck-At faults—and at least **one** of them is likely to be detected anyway!

## Fault Masking:

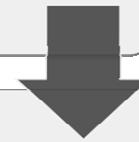
- Multiple faults *can* be an issue if the target fault is **masked** by a second fault.
- This is possible, but unlikely, and is only an issue in ultra-high-reliability ICs.

## Technical Reference:

- For an example of a target fault masked by an undetectable fault, see:  
*Digital Systems Testing & Testable Design*  
Abramovici, Breuer & Friedman,  
[Computer Science Press, 1990], **Figure 4.5**.

# Introduction to Scan Testing: Agenda

**Design For Manufacturing Test**



**D-algorithm as applied to purely combinational logic**

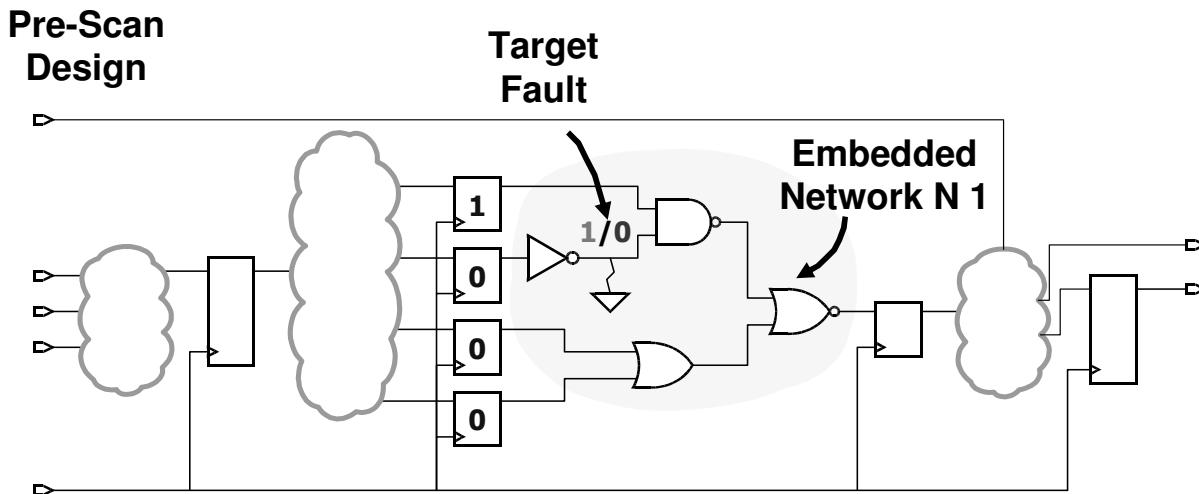


**D-algorithm as applied to sequential logic (Full Scan)**

1-22

# Testing Sequential Designs

## Can the D-algorithm handle flip-flops?



- Still need to activate the fault and propagate its effect
- Replace each flop with a testable flip-flop
- This replacement allows serial loading/unloading of bits

1-23

- This slide introduces the challenges of detecting faults when **flip-flops** are present.
- When a normal flip-flop is replaced with a testable flip-flop that is called **scan replacement**.

The testable flip-flop has two modes:

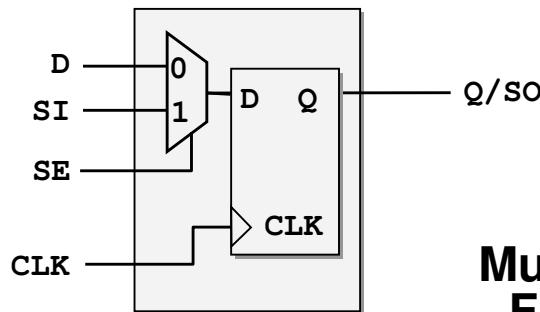
Normal mode—where the ASIC is operating as it was functionally designed to operate.

Test mode—where the ASIC is undergoing manufacturing test.

- When the ASIC is undergoing manufacturing test the flip-flops will be connected together just like a shift register—this configuration is called a scan-chain.
- The scan-chain will be used to serially shift in test patterns to parts of the design which are not directly accessible to the primary input ports—such as the example shown above.

# Scannable Equivalent Flip-Flop

**Scannable  
Equivalent  
for Ordinary  
D Flip-Flop**



**Multiplexed  
Flip-Flop  
Scan Style**

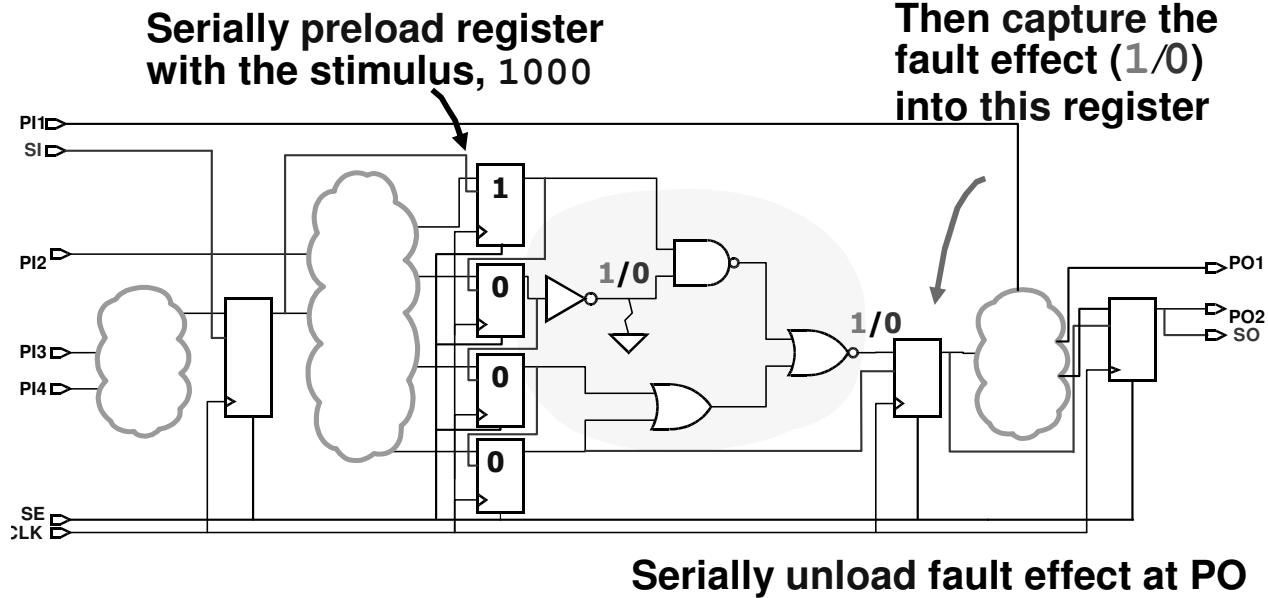
- The scan equivalent has a serial path from pin SI to SO
- This path is enabled only during testing, by asserting SE

1-24

This slide shows the most common scan-replacement style, DFTC supports other styles:

- The **compile -scan** command enables a “test-ready” compile where all flip-flops are synthesized into their scannable equivalents.
- The vast majority of ASIC’s which require test use a multiplexed flip-flop as the scan-replacement for the normal flip-flops.
- **Clocked scan** style uses a dual-port flip-flop with a **dedicated** scan clock. Its area penalty is higher than for multiplexed flip-flop, but its timing impact is lower.
- **Level-sensitive scan design (LSSD)** is a latch-based replacement for single latches, master-slave latches, or flip-flops. It requires three separate clock lines to be routed.
- For more information on these styles, see the DFT Compiler Documentation section on **Performing Scan Replacement**.

# The Full Scan Strategy

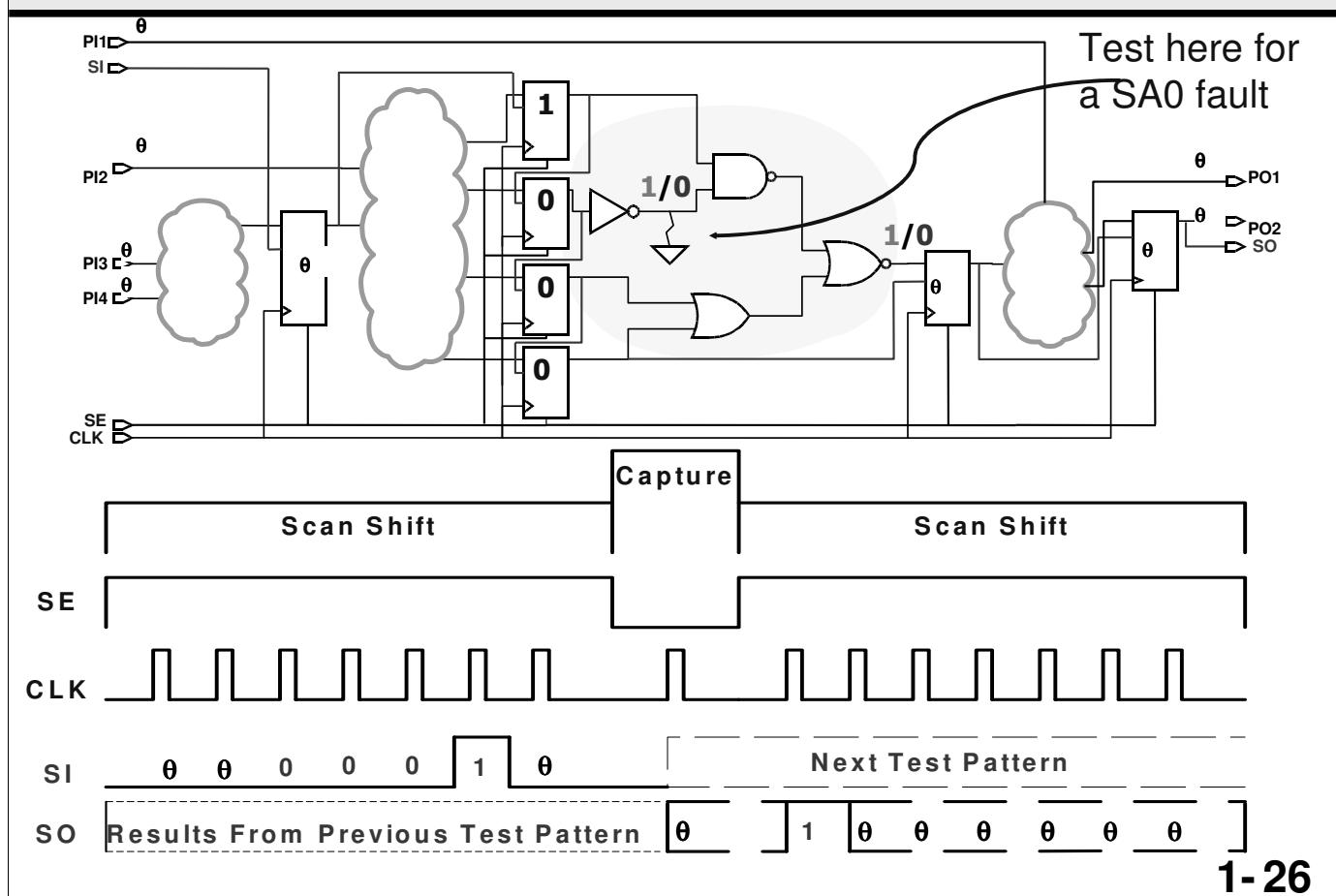


**Stitch together a serial path through all the scan flops, enabling ATE to preload registers and capture responses!**

1-25

- A **Full Scan** design is one in which all of the flip-flops in the design have been scan replaced and will be included in the scan-chain.  
Note: No memories, latches.
- This slide shows the results of stitching a scan-chain (see the red path, starting at the SI port and ending at the SO port).
- Notice there are three additional ports now: SI (Scan In), SO (Scan Out), and SE (Scan Enable).
- The command to stitch up the scan-chains is: **insert\_dft**.

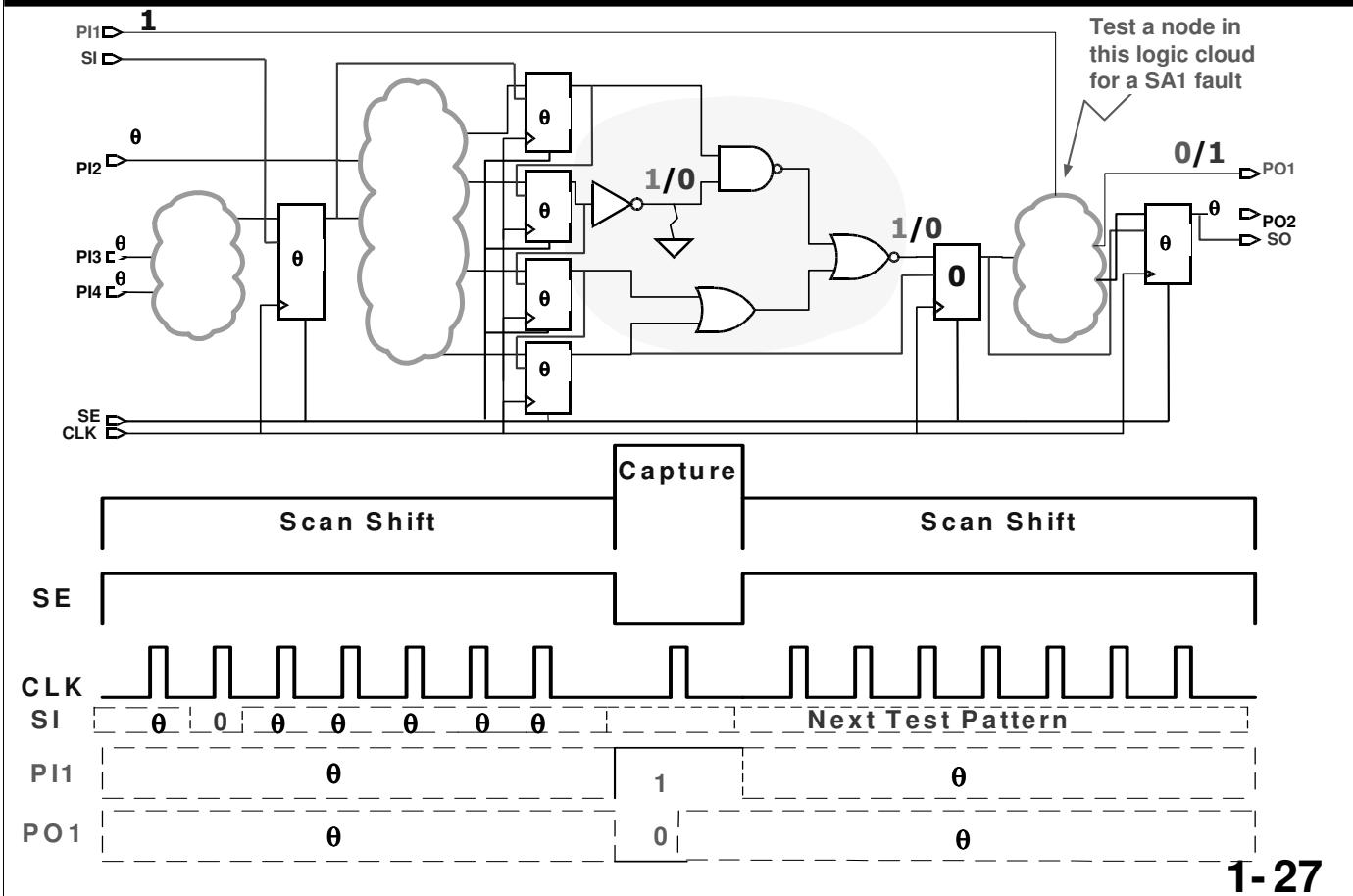
# Scan Testing Protocol: Example 1



1-26

Input / Output =  $\theta$  is a don't-care ( $0$  or  $1$ ).

# Scan Testing Protocol: Example 2



To perform a SAF test on a port or pin which has a path through a combinational logic cloud to a primary output port (PO1 or PO2 in this design), the node must be initialized during the Capture phase of test. In the case above there are two “inputs” which initialize the node. PI1 must be set to one, and FF6 must be set to zero.

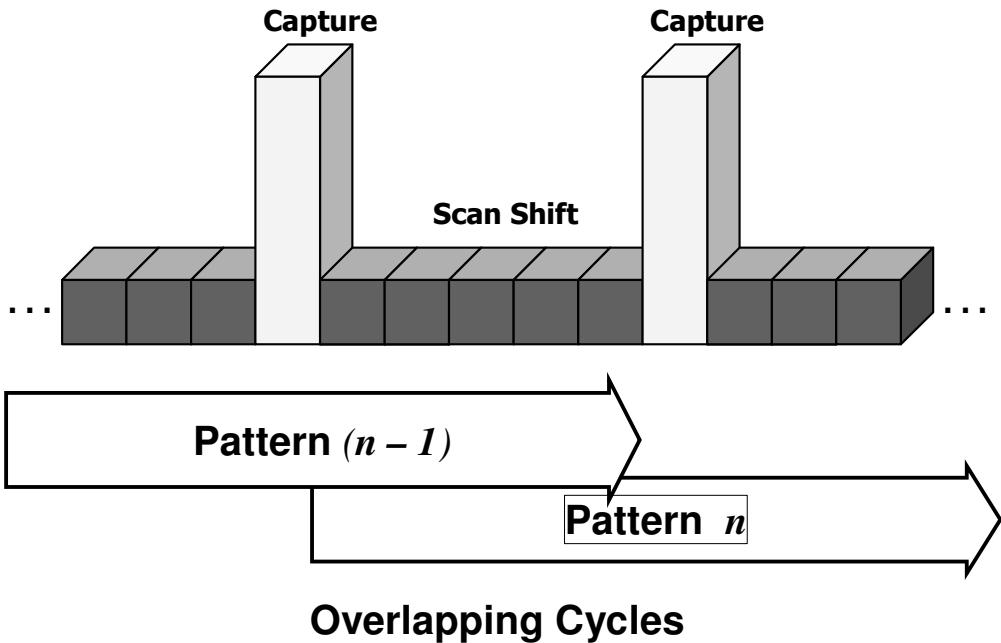
The initialization of all flip-flops in the scan-chain is performed during the scan shift phase of test. In this case it takes 7 clock cycles to initialize the scan-chain contents.

Next, during the capture phase, the primary input ports are initialized (usually at the beginning of the capture cycle). In this case as soon as SE goes low, PI1 will be set to a 1.

Since the logic cloud connects directly to a primary output port, the test can be performed IMMEDIATELY (with a pause for the data to pass through the logic cloud). The ATE will **strobe** (measure) PO1 just before the clock (port CLK) goes active.

For example, if the capture cycle is 100 ns long, and the clock goes active at 45 ns, PI1 will be applied by the ATE at or near 0 ns, and the ATE will strobe port PO1 at or near 40 ns.

# Test Patterns Overlap



- Scanning out of previous pattern overlaps scanning in of next—for all but first and last patterns in the test program

1-28

The five scan-out cycles of one pattern actually **overlap** the scan-in cycles of the next. The overlapping cycles are referred to as **scan-shift** cycles, since they scan in **and** out; thus, a program of **100** test patterns requires, not 1100 test vectors, but  $600 + 5$ , or **605**. With this overlap, scan unloading—except for the very last—costs no extra tester time. Since tester time is expensive in SOC production testing this is a significant savings.

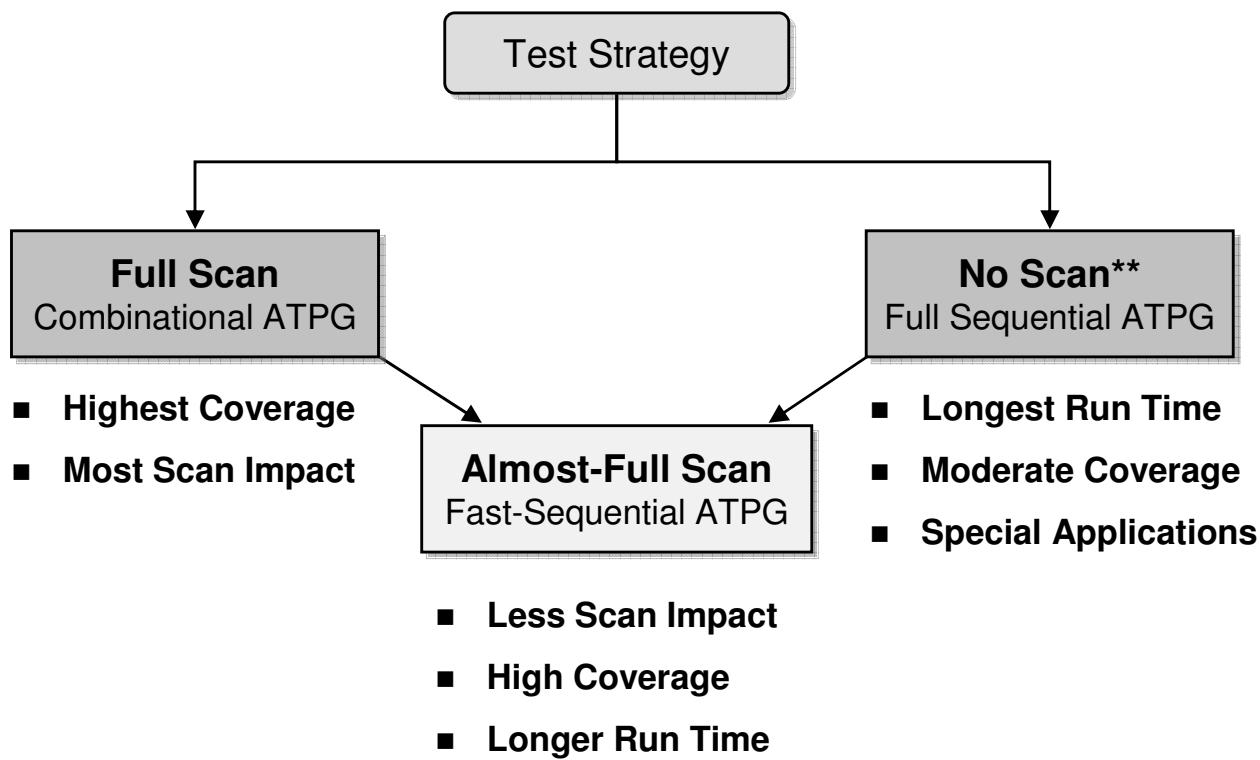
## Tester Time:

Total tester time is roughly **proportional to** the length of the **longest DUT scan-chain**.

Synopsys recommends reducing tester time by **partitioning** one long scan-chain into several.

You will see exactly how to partition scan-chains and balance their lengths in **a later unit**.

# Scan Strategies Summarized



1-29

This slide summarizes testing strategies for user-defined logic (UDL) blocks on a chip.

Full scan—highest coverage fastest and easiest to implement:

DFTC can implement

Uses D-algorithm

No memories

No latches

All FF's included on the scan-chain

If the design is NOT full scan and the coverage estimate in DFTC is not adequate that may be O.K. because TetraMAX may still be able to increase the coverage in the following ways.

Almost-Full Scan—very high coverage slightly slower to implement:

Requires TetraMAX to implement

Uses an extension of the D-algorithm

Most or all FF's included on the scan-chain

May have memories

May have latches

No scan—lower coverage, very slow:

Requires TetraMAX to implement

Requires functional patterns

Requires fault Simulation and Fault Grading

Uses full-sequential algorithm

Use this approach when test coverage is not high enough and have tried first Full Scan and second Almost-Full Scan Approach

If you cannot meet timing on a critical path, try excluding only the destination flop.

You will see in a later unit how to exclude specific flip-flops from inclusion in scan-chains.

## Caveats:

The non-scan approach, using sequential ATPG, is only practical for special purpose ICs, which cannot afford the added area cross-section of scan.

\*\*No Scan or Very Low Scan

# Full Scan Summary

## Full Scan Advantages:

- Needs only combinational ATPG (D-algorithm) for testing all Stuck-At faults
- Combinational ATPG gives shorter ATPG run times
- Predictable and applies across most architectures
- Gives highest test coverage of all the algorithms
- Easiest to implement

## Full Scan Limitations:

- Adds nonfunctional pins to the package
- Timing and density impact of scan-equivalent flops
- Embedded memory, latches, and non-scan flip-flops, require fast sequential ATPG (“almost-full scan”)

1-30

# Unit Summary

---

**Having completed this unit, you should now be able to:**

- **Explain how to use the D-algorithm to generate a pattern that detects a given Stuck-At fault in a combinational design**
- **Do the same in a full scan sequential design**
- **Explain why scan-chains are necessary to support ATPG**

**1-31**

This page was intentionally left blank.

# Agenda

**DAY  
1**

**1** Introduction to Scan Testing

**2** DFT Compiler Flows

**3** DFT Compiler Setup

**4** Test Protocol

**5** DFT Design Rule Checks

# Unit Objectives



**After completing this unit, you should be able to:**

- Tell the difference between Ad-hoc and Structured DFT techniques
- Name the different scan insertion flows that DFTC supports
- Recognize the steps used in a typical scan insertion flow

**2-2**

# DFT Compiler Flows: Agenda

DFT Compiler Introduction

DFT Compiler Scan  
Insertion Flows Supported

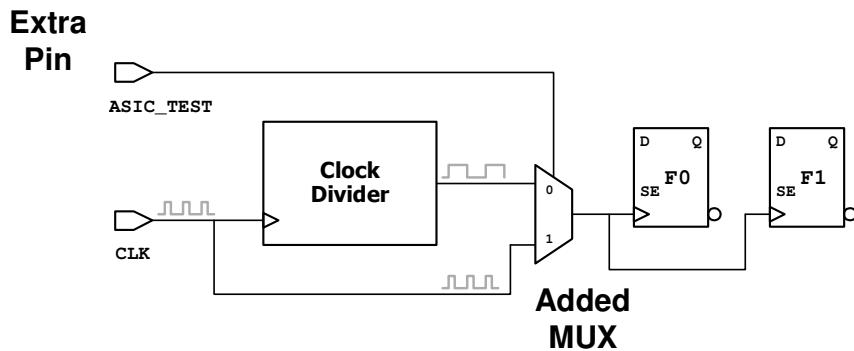
Basic Steps In a Typical  
Scan Insertion Flow

2-3

# What is Design-For-Testability?

## Design For Testability (DFT):

- Extra design effort invested in making an IC testable
- Involves inserting or modifying logic, and adding pins
- You can apply structured or ad hoc DFT techniques



Typical DFT Hardware

2-4

Before the era of million-gate ASICs, testability was often considered a **back-end** issue. Large designs were “thrown over the wall” to a dedicated Production Engineering group. Frequently, this “over the wall” approach resulted in **excessive rework** and wasted time. Today’s design complexities and time-to-volume pressures require a proactive approach. Testability must be **designed into** the chip architecture right from the RTL coding phase. This proactive ASIC design strategy has come to be called **design for testability** (DFT). The requirement for DFT means IC designers must be able to **think like** Test Engineers.

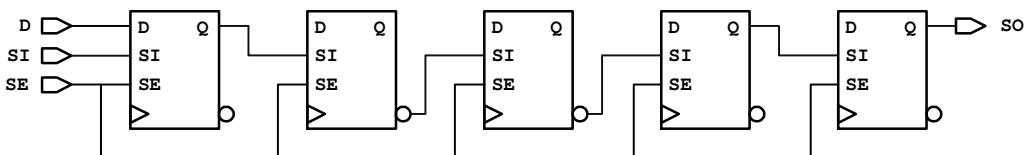
In this unit, the focus will lean towards the **testability implications** of various design practices. You will also use the power of synthesis tools to **synthesize DFT** as well as functional logic.

# Structured DFT Defined

## Structured DFT:

- Highly pushbutton—little intervention by designer
- Each design is checked against a set of DRC rules
- Violations are corrected by adding or modifying logic

Serial path **automatically** routed—**independent** of design details



**Scan-path insertion is a common example of automated DFT**

**2-5**

The complexity of SoC designs is driving the need for **structured** DFT methodologies. A structured methodology is one that **globally resolves** many on-chip testability issues. Such methodologies are **widely applicable** to most designs without much customization. They are **automated**. The tools do most of the DFT work with minimal intervention. **Scan-based testing** is a typical structured methodology that will be emphasized in this course. Scan insertion, for example, is relatively automated and independent of design function.

# Ad Hoc DFT Defined

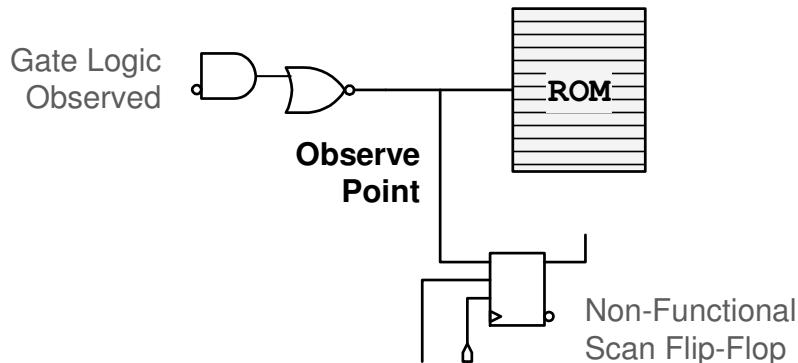
## Ad Hoc DFT:

- Adding testability logic at designer's discretion

### Example: control points or observe points

Other examples: bus keepers or pull-up resistors

- Location and type left up to designer's DFT savvy



2-6

As SoC designs adopt more kinds of on-chip logic, however, there is a competing trend. Embedded memories, IP cores, and on-chip analog blocks all present **diverse** test needs. Structured methodologies like traditional scan-based testing cannot meet *all* these needs. Making these different kinds of circuits testable often requires **ad hoc** DFT techniques.

Test-point logic is **nonfunctional**—that is, not a requisite part of on-chip application logic.

Adding it to the chip represents **overhead**, but can provide large gains in testability.

### Conclusion:

You will need to utilize both **structured** and **ad hoc** DFT to meet SOC test requirements.

# DFT Compiler

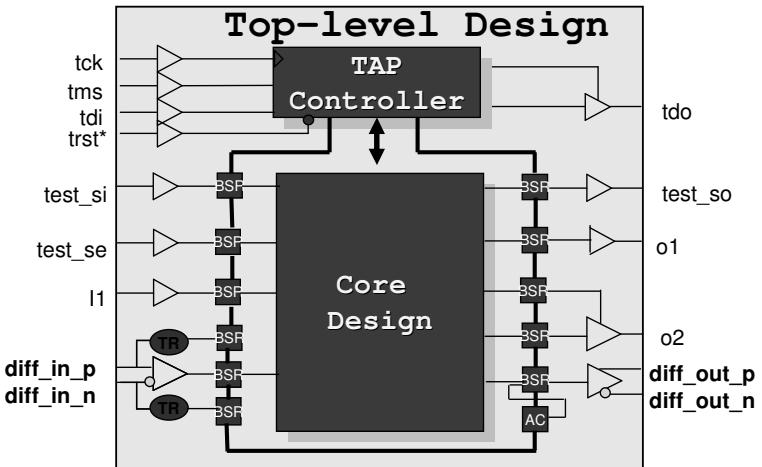
- **DFT Compiler is Synopsys' advanced Test synthesis solution**
- **Some key benefits of DFT Compiler**
  - Offers transparent DFT implementation within the synthesis flow
  - Accounts for testability early in the design cycle at RTL
  - Removes unpredictability from the back end of the design process
  - Achieves predictable timing, power, and signal integrity closure concurrent with test

2-7

# BSD Compiler

## IEEE 1149.1 & 1149.6 Boundary-Scan DFT

- Synthesis of IEEE 1149.6 “AC-JTAG” components
- BSDL generation in the presence of 1149.1 and 1149.6 components
- Support for BSRs embedded in PADs
- Pattern generation for 1149.1 and 1149.6



2-8

Note: BSD Compiler is outside the scope of the DFT Compiler 1 class. For more information on BSC Compiler, consult the BSC Compiler User Guide documentation on SolvNet:

[https://solvnet.synopsys.com/dow\\_retrieve/A-2008.03/bsdxg/bsdxg.html](https://solvnet.synopsys.com/dow_retrieve/A-2008.03/bsdxg/bsdxg.html)

# DFT Compiler Flows: Agenda

DFT Compiler Introduction

DFT Compiler Scan  
Insertion Flows Supported

Basic Steps In a Typical  
Scan Insertion Flow

2-9

# DFT Compiler Flows Supported

## ■ Top-Down

- Scan insertion is performed at the top-level for the entire design in one step

## ■ Bottom-Up

- Scan insertion is performed at the block-level and results are later combined at the top-level

## ■ Unmapped Flow

- Scan insertion begins with an unmapped (RTL) design

## ■ Mapped Flow

- Scan insertion begins with a mapped (gate-level) design

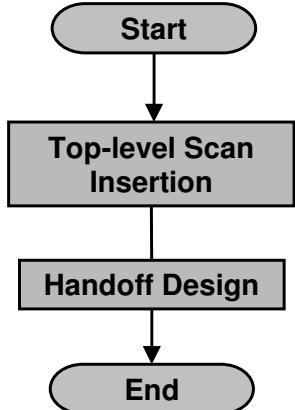
## ■ Extraction (Inference)

- Scan insertion begins with a design that has existing scan chains already inserted

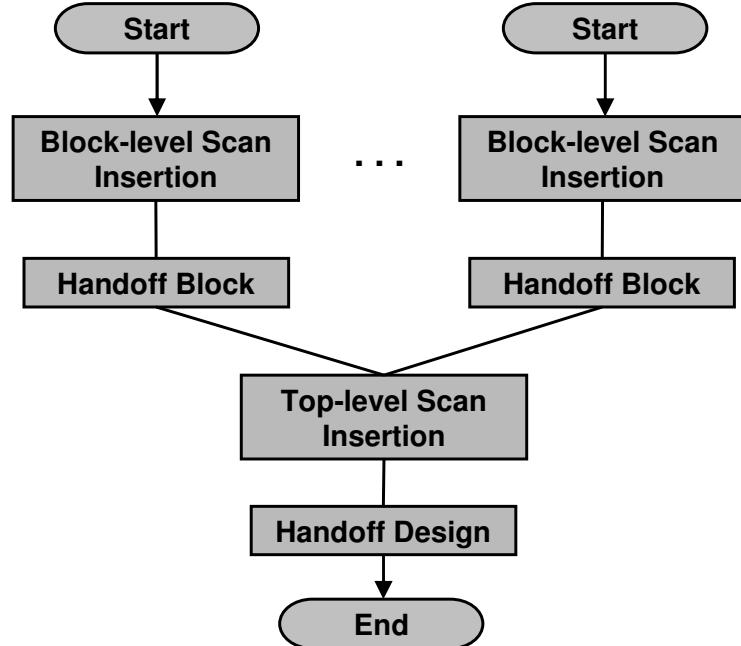
**2-10**

# Top-Down Vs. Bottom-Up Flow

Top-Down Flow



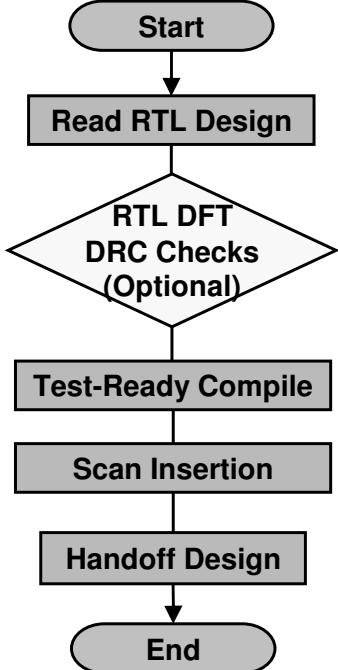
Bottom-Up Flow



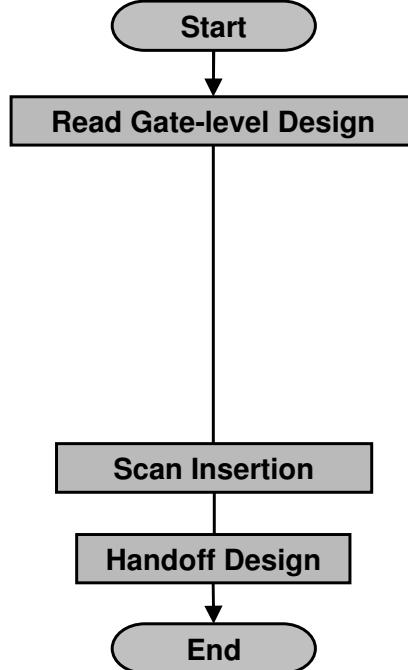
2-11

# Unmapped Flow Vs. Mapped Flow

Unmapped Flow

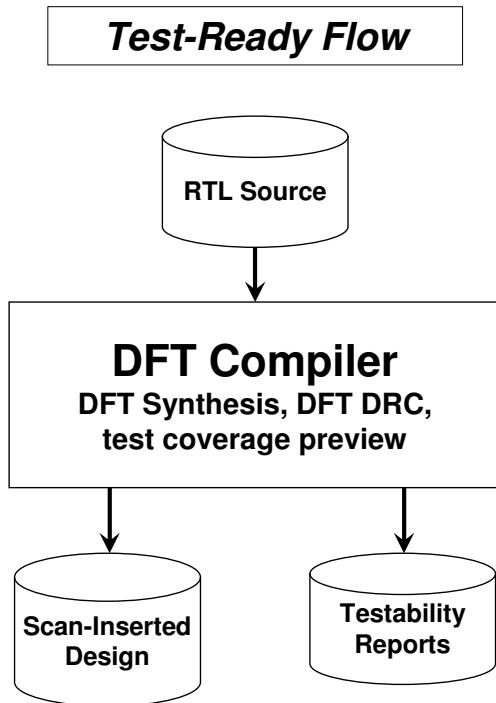


Mapped Flow



2-12

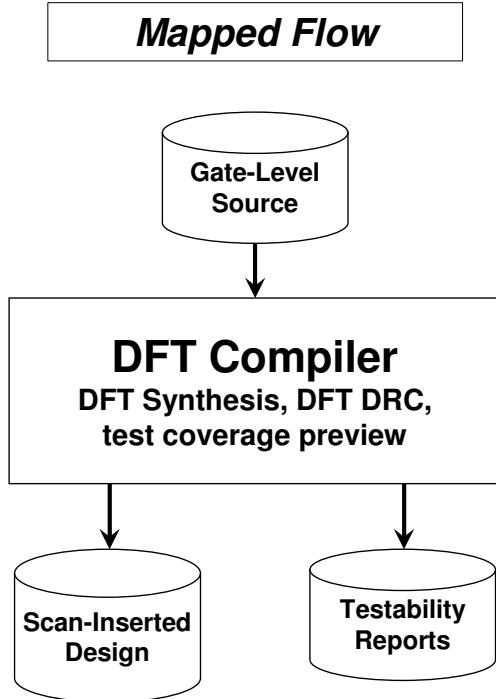
# DFT Compiler Test-Ready or Unmapped Flow



- Starting point is RTL (unmapped) design
- IDEAL starting point
- Scan synthesis achieved by taking RTL directly to a scan synthesized design

2-13

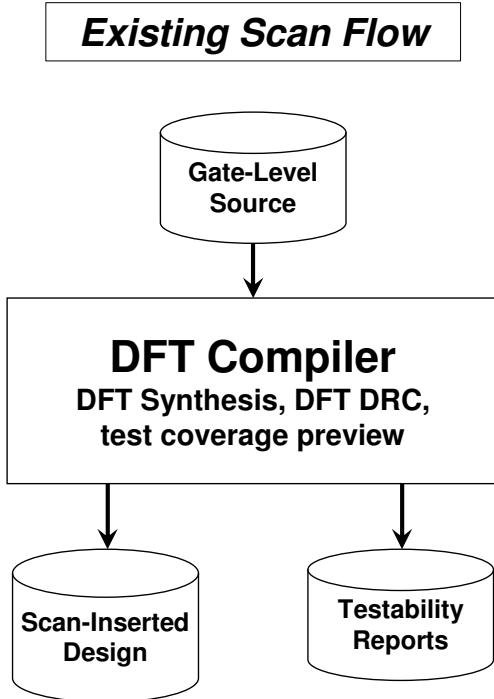
# DFT Compiler Mapped Flow



- Starting point is a gate-level (mapped) design with no scan circuitry yet
- DFT Compiler performs scan cell replacement and scan chain synthesis

2-14

# DFT Compiler Existing Scan Flow



- Starting point is a gate-level design that already includes scan cells and chains
- DFT Compiler performs scan chain extraction & DFT DRCs in preparation for TetraMAX ATPG

**2-15**

If the existing scan design was created by DFT Compiler, saved in .ddc format and no test attributes removed (3 important conditions), DFT Compiler automatically recognized the chains it inserted earlier. If you were to bring in a Verilog or VHDL netlist with scan chains in it, however, there is nothing in the netlist to describe all the test attributes...so DFT Compiler has to use your test protocol and defined test attributes to "extract" the scan chain from the netlist.

# DFT Compiler Flows: Agenda

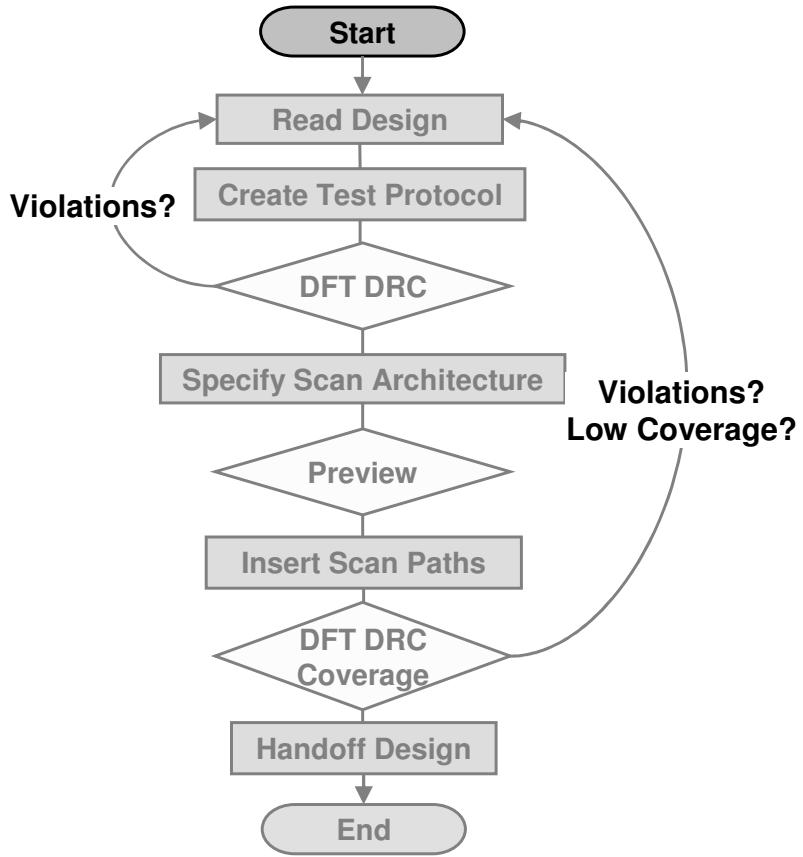
DFT Compiler Introduction

DFT Compiler Scan  
Insertion Flows Supported

Basic Steps In a Typical  
Scan Insertion Flow

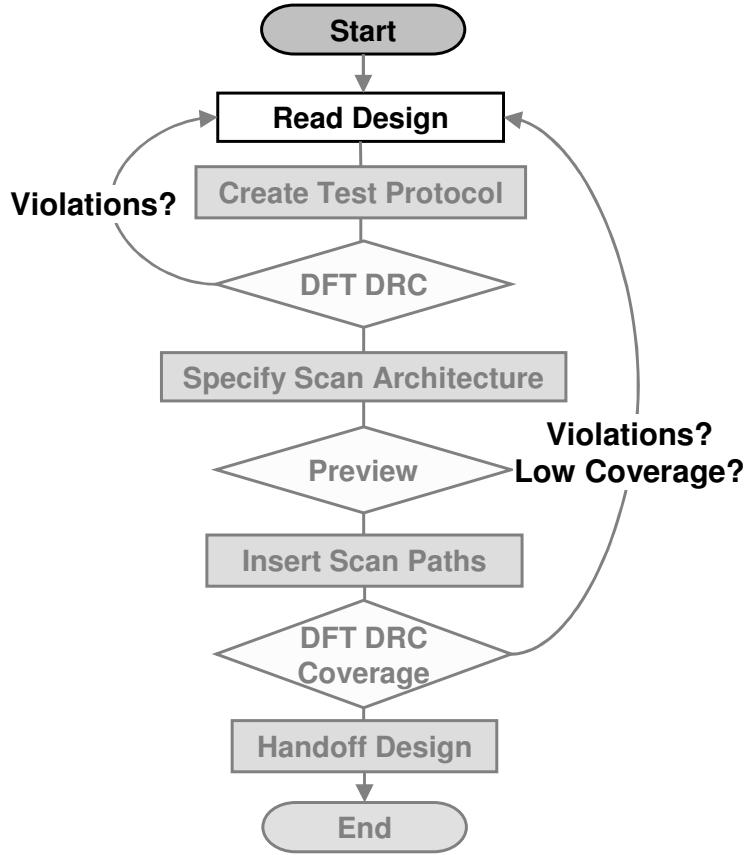
2-16

# Typical Scan Insertion Flow



2-17

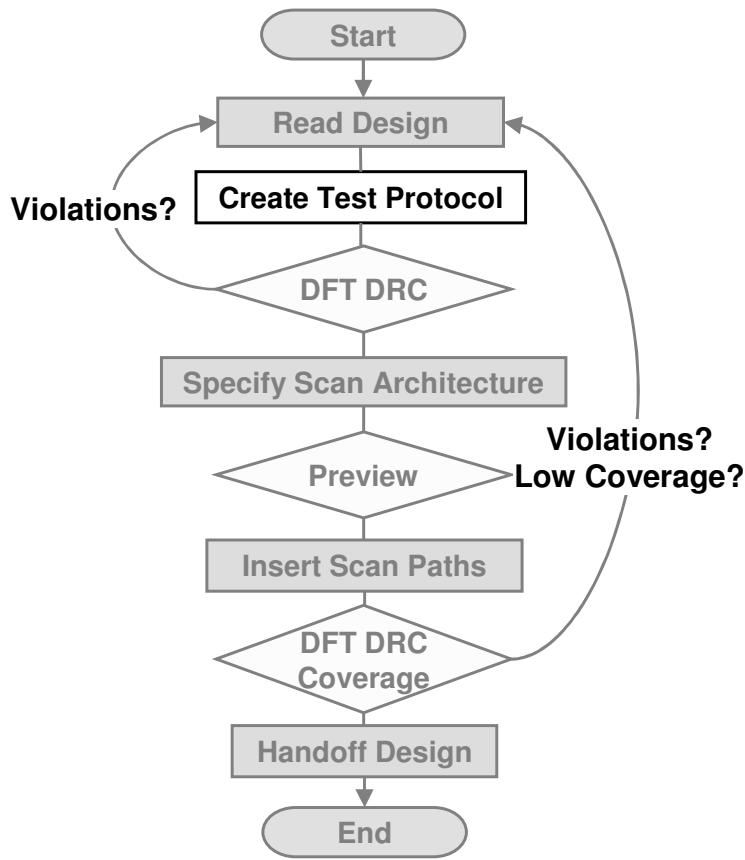
# Read Design



- Read a mapped design (`read_ddc`, `read_verilog`)
- Scan insertion is performed on mapped designs

2-18

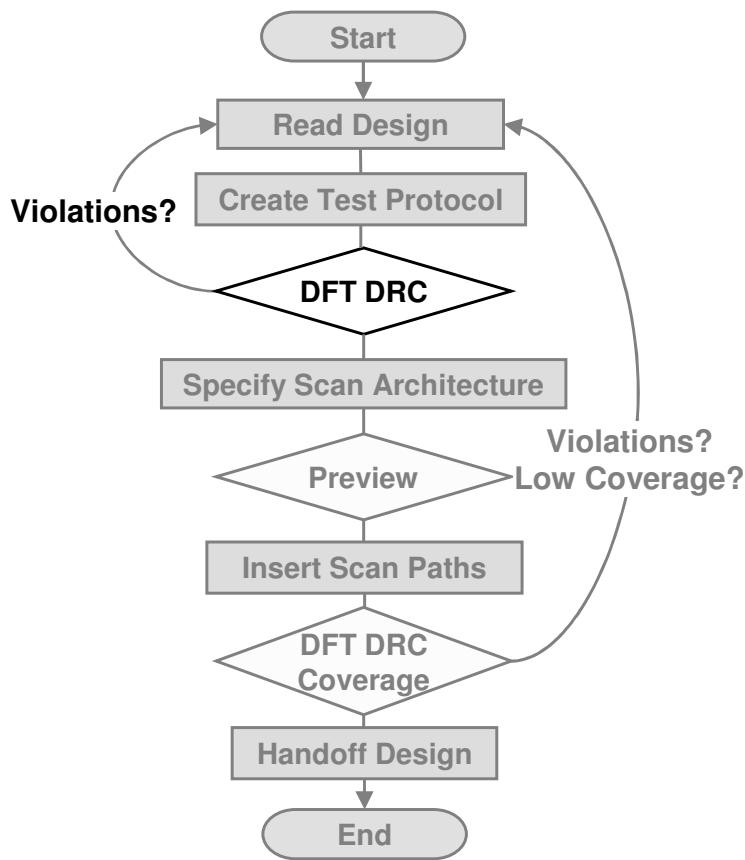
# Create Test Protocol



- The Test Protocol describes how the design operates in scan mode
- Signals involved in the protocol are declared with the `set_dft_signal` command
- The protocol is created by the `create_test_protocol` command

2-19

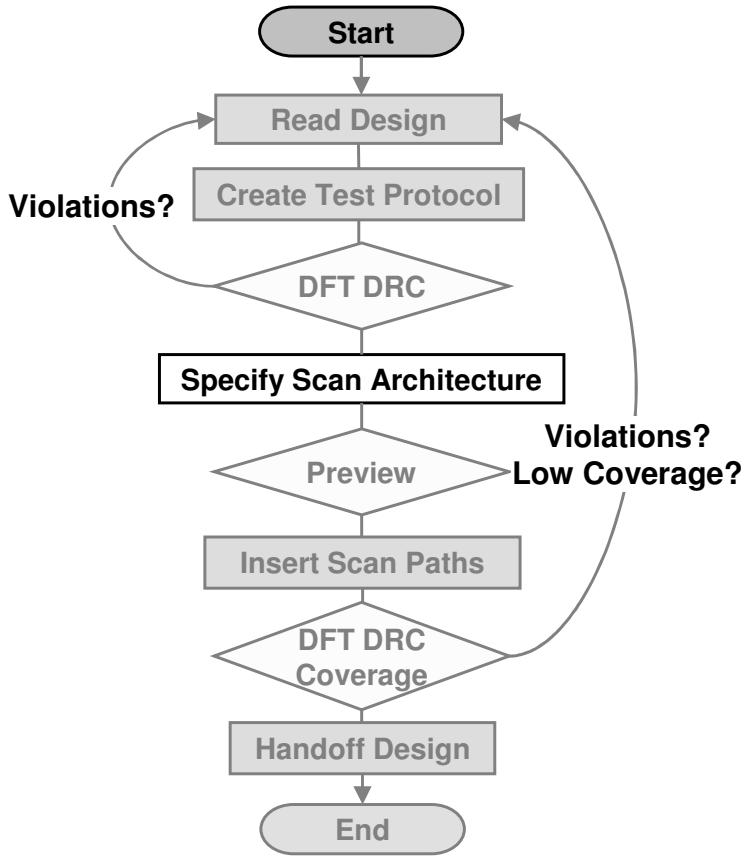
# DFT DRC



- The `dft_drc` command performs DRC checks prior to scan insertion
- DRC violations can be debugged graphically with DesignVision or fixed by DFT Compiler with Autofix

2-20

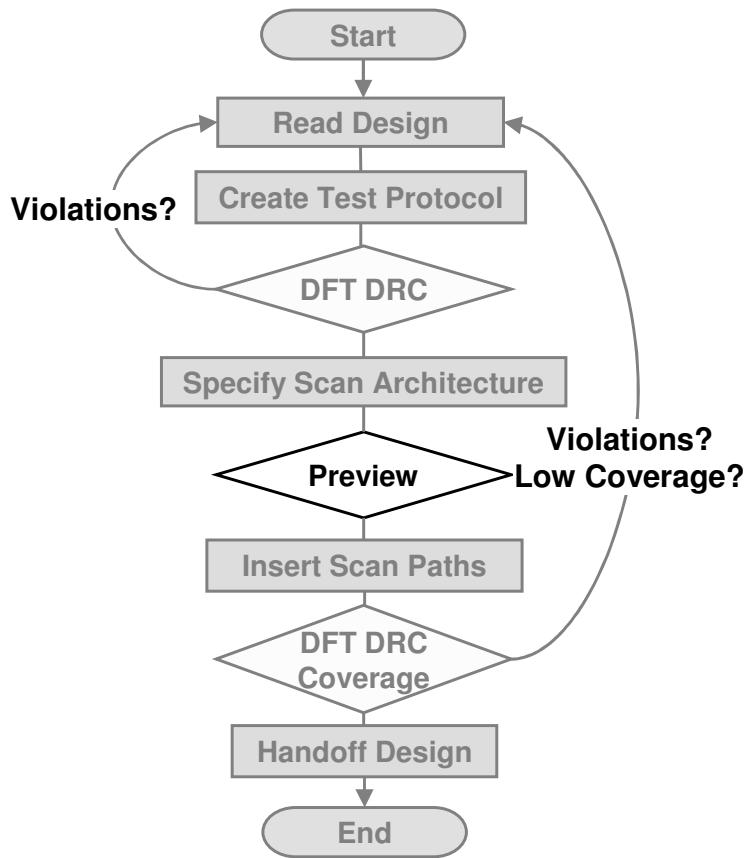
# Specify Scan Architecture



- Various commands control the scan architecture (number of scan chains, how clock domain are handled, etc.)
- The commands primarily used to control the scan architecture are the `set_scan_configuration` and `set_scan_path` commands

2-21

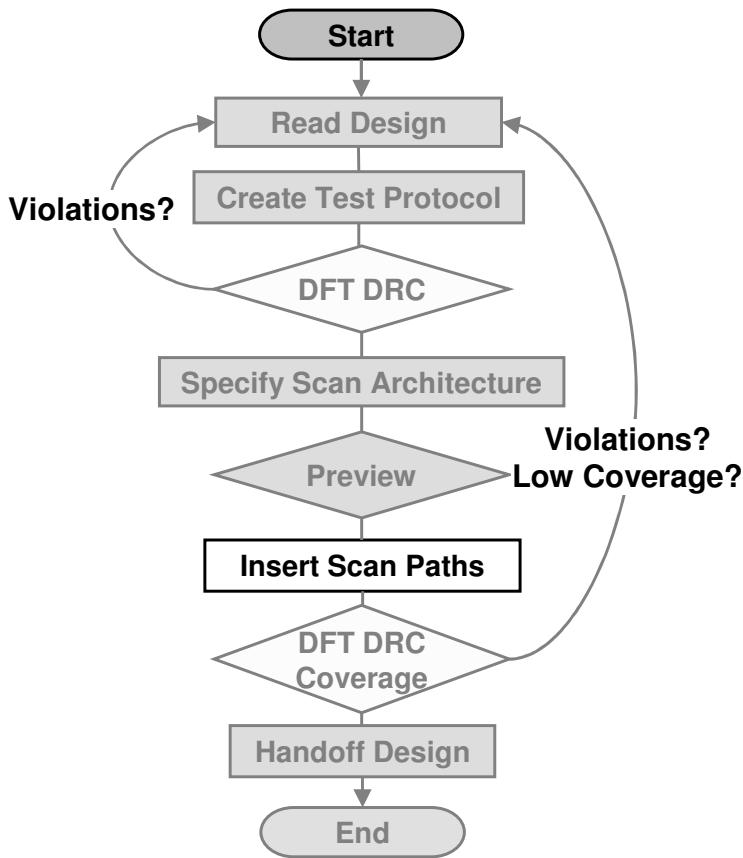
# Preview



- The `preview_dft` command is used to get a preview of the scan architecture before it is actually implemented in the design
- The preview step allows for a quicker iteration cycles when changes need to be made to the scan architecture

2-22

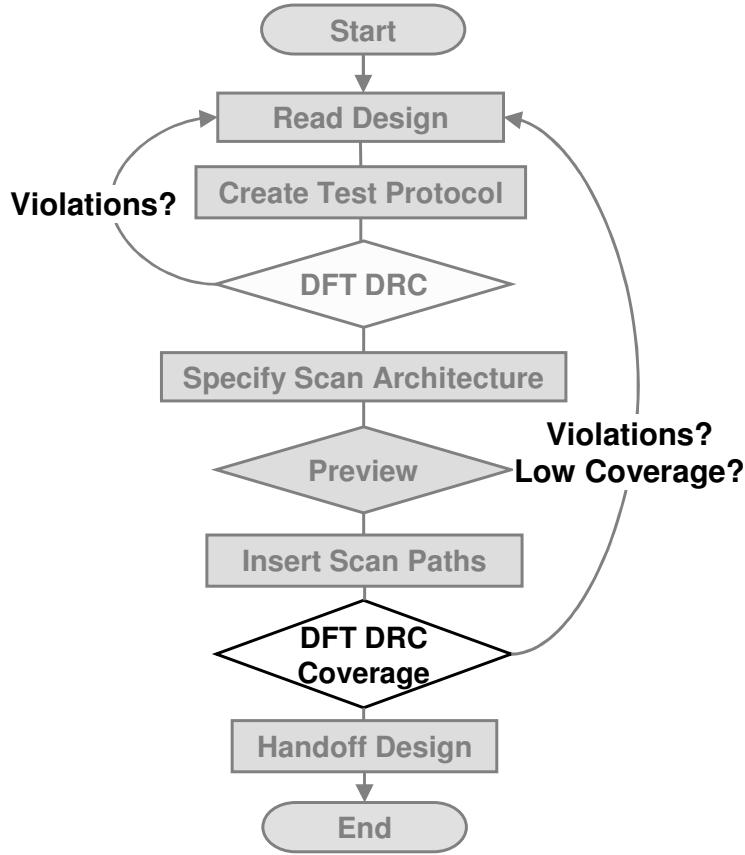
# Insert Scan Paths



- The scan architecture is inserted into the design by the `insert_dft` command

2-23

# DFT DRC Coverage

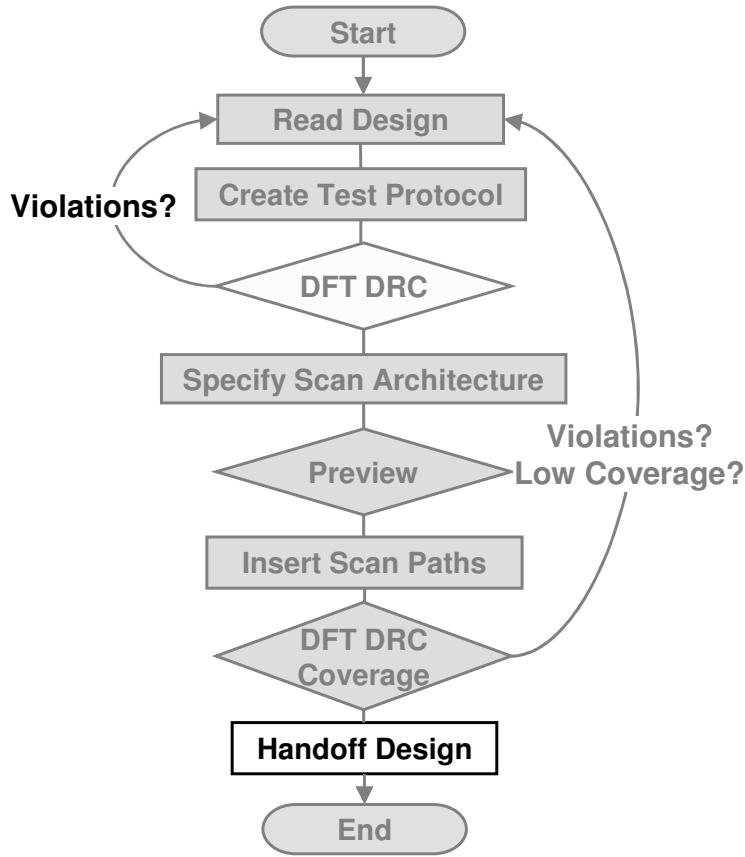


- DFT DRC checks can also be run after scan insertion to validate that the scan chains trace properly
- DFT DRC can also be used to get an ATPG coverage estimate for the scan inserted design

2-24

Note: the coverage provided by DFT DRC on a post-scan design is only an **estimate**. TetraMAX ATPG should be used to get a full and complete analysis of the test coverage.

# Handoff Design



- The design handoff is where files are written to disk that will be needed later on in the design process
- Examples: design DDC, verilog netlist, protocol file (for TetraMAX), Test Model (for Bottom-Up flows), SCANDEF (for backend scan chain reordering), etc.

2-25

# **Unit Summary**

---

**Having completed this unit, you should now be able to:**

- **Tell the difference between Ad-hoc and Structured DFT techniques**
- **Name the different scan insertion flows that DFTC supports**
- **Recognize the steps used in a typical scan insertion flow**

**2-26**

# Command Summary (Lecture, Lab)

<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>create_test_protocol</code>	Creates a test protocol based on user specification
<code>dft_drc</code>	Checks the current design against test design rules
<code>set_scan_configuration</code>	Specifies the scan chain design
<code>set_scan_path</code>	Specifies a scan chain for the current design
<code>preview_dft</code>	Previews, but doesn't implement, the scan architecture
<code>insert_dft</code>	Adds scan circuitry to the current design

**2-27**

This page was intentionally left blank.

# Agenda

**DAY  
1**

**1** Introduction to Scan Testing



**2** DFT Compiler Flows



**3** DFT Compiler Setup



**4** Test Protocol



**5** DFT Design Rule Checks



# Unit Objectives

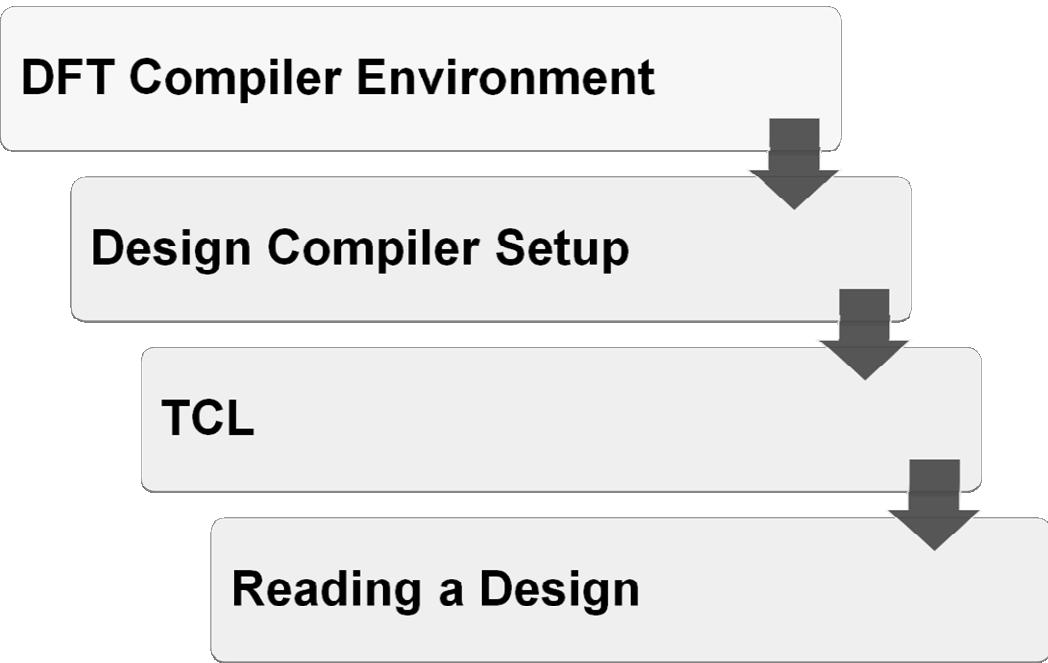


**After completing this unit, you should be able to:**

- Create the setup file for DFT Compiler (DFTC)
- Specify the target library and link library
- Read an RTL design into DFTC

**3-2**

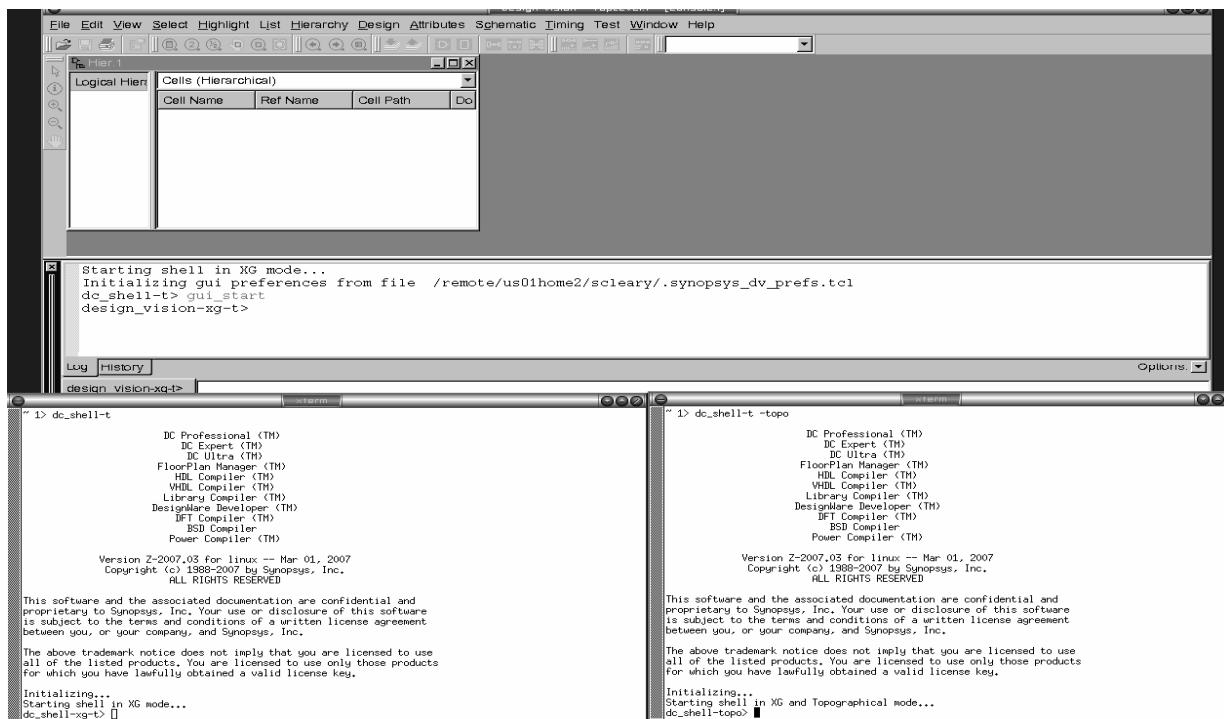
# DFT Compiler Setup: Agenda



3-3

# DFTC: Multiple Tool Environments

## DC Shell, DC Topographical (DCT), Design Vision (GUI)



3-4

DC shell also other environments – Multi-voltage mode, UPF mode, and Topographical mode.

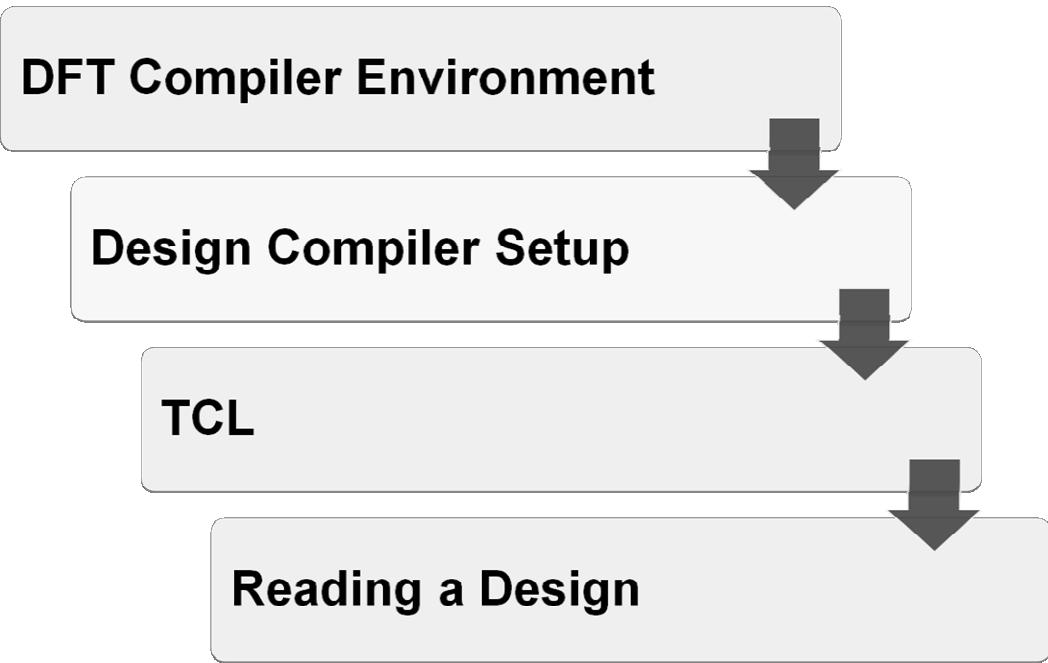
DC shell can be run interactively or in “batch mode”.

DCT has a GUI mode called “DC Graphical”.

Design Vision (**design\_vision**) must be used for Graphical Debug of DFT DRC violations.

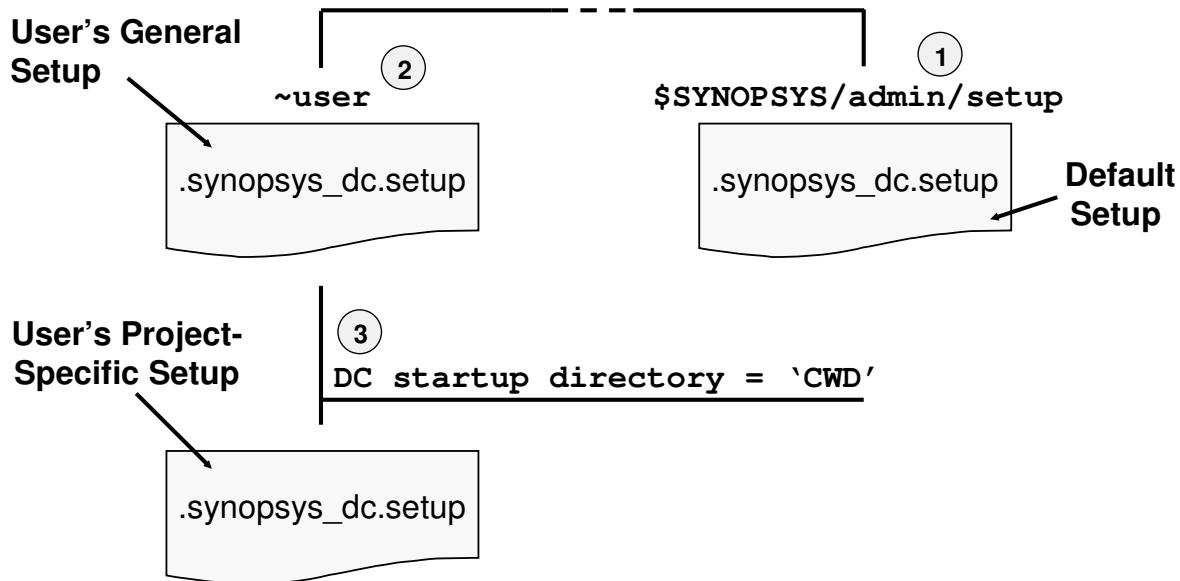
IC compiler (ICC) can be used for scan chain reordering in the backend. There isn’t a “scan insertion” flow in ICC.

# DFT Compiler Setup: Agenda



3-5

# One Startup File Name – Three File Locations



**These files are automatically executed,  
in the order shown, upon startup of DC**

**3-6**

CWD stands for ‘current working directory’. We will use this acronym throughout the workshop to represent the UNIX directory in which your DC session was invoked.

## **Default .../admin/setup/.synopsys\_dc.setup**

```
# .synopsys_dc.setup file in $SYNOPSYS/admin/setup
.
.
.

set target_library your_library.db
set link_library {* your_library.db}
set symbol_library your_library.sdb
set search_path ". <Install_dir>/libraries/syn . . ."
.
```

**This file is automatically executed  
first upon tool startup**

**3-7**

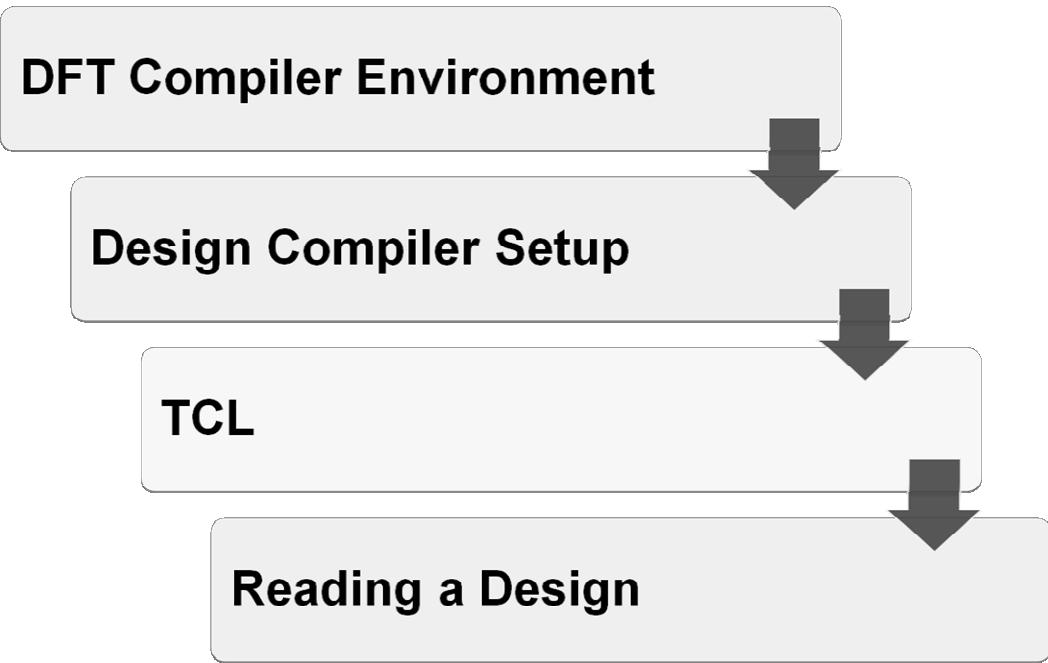
# Project-specific (CWD) *.synopsys\_dc.setup*

```
#* .synopsys_dc.setup file in project's 'CWD'  
set search_path "$search_path mapped rtl libs cons"  
set target_library 65nm.db  
set link_library "* $target_library"  
set symbol_library 65nm.sdb; # Contains symbols for  
# Design Vision GUI  
  
history keep 200  
alias h history  
alias rc "report_constraint -all_violators"
```

If present, this file is executed last upon tool startup and overrides previously set variables

3-8

# DFT Compiler Setup: Agenda



3-9

# Tool Command Language (TCL)

- **TCL is the command language that is used in the Design Compiler command shell and many other Synopsys tools**
- **Additional Synopsys specific commands are added to the TCL commands in the command shell**
- **Documentation on “Using TCL with Synopsys Tools” is available on SolvNet:**
  - [https://solvnet.synopsys.com/dow\\_retrieve/A-2008.03/tclug/tclug.html](https://solvnet.synopsys.com/dow_retrieve/A-2008.03/tclug/tclug.html)
- **Flash based TCL training modules (including labs) can be viewed on SolvNet:**
  - <https://solvnet.synopsys.com/retrieve/020353.html>

**3-10**

# Helpful *UNIX-like* dc\_shell Commands

**Find the location and/or names of files<sup>1</sup>**

```
dc_shell> pwd; cd; ls
```

**Show the history of commands entered:**

```
dc_shell> history
```

**Repeat last command:**

```
dc_shell> !!
```

**Execute command no. 7 from the history list:**

```
dc_shell> !7
```

**Execute the last report\_\* command:**

```
dc_shell> !rep
```

**Execute any UNIX command:**

```
dc_shell> sh <UNIX_command>
```

**Get any UNIX variable value:**

```
dc_shell> get_unix_variable <UNIX_env_variable>2
```

**3-11**

<sup>1</sup> Use cd very carefully, because you will change the relative starting point ( . ) for your directory search\_path. By default “.”, the current working directory, is set to the directory in which you invoked Design Compiler (shell or GUI).

<sup>2</sup> For example, use the following to determine if you are in a Sun or Linux environment:  
dc\_shell> get\_unix\_variable ARCH → may return “linux” or “sparcOS5”

# More Helpful dc\_shell Commands

**Man page for command:**

```
dc_shell> man "command_name"
```

**Commands matching current pattern:**

```
dc_shell> "command_pattern" + TAB
```

**To get help on a particular command:**

```
dc_shell> help "command_name"
```

**To get detailed help on a command, use the -verbose (-v) switch:**

```
dc_shell> help -v "command_name"
```

**To see the current value of a TCL variable (can use wildcards):**

```
dc_shell> printvar "variable_name"
```

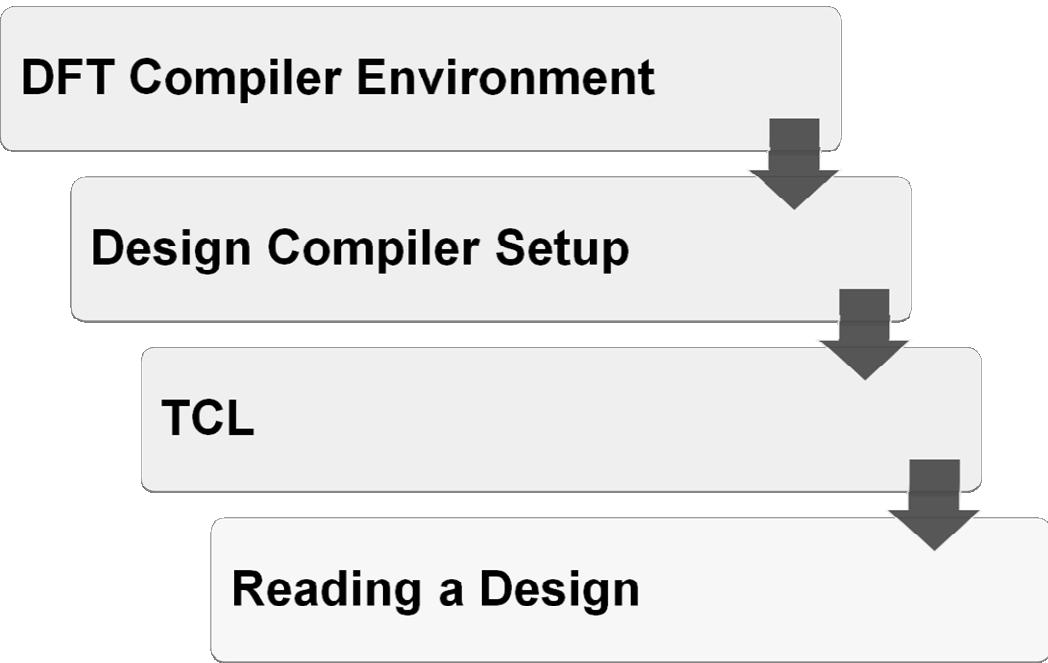
```
dc_shell> printvar test_default*
```

**To recall a previous command use the up/down arrows**

**Command line editing is enabled by default and is controlled by the  
sh\_enable\_line\_editing and sh\_line\_editing\_mode variables**

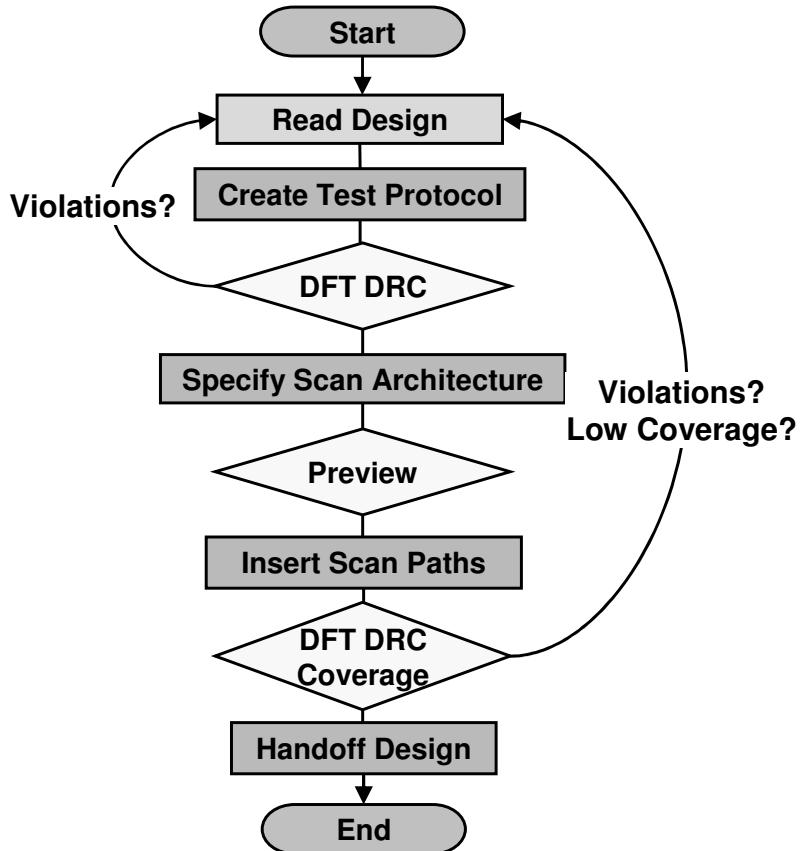
**3-12**

# DFT Compiler Setup: Agenda



3-13

# Typical Scan Insertion Flow



3-14

# Synthesis Transformations

Synthesis = Translation + Logic Optimization + Gate Mapping

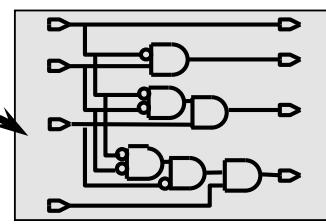
```
residue = 16'h0000;    RTL Source
if (high_bits == 2'b10)
    residue = state_table[index];
else
    state_table[index] = 16'h0000;
```

1 Translate (read\_verilog  
read\_vhdl )

Constraints

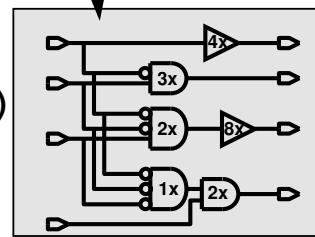
```
set_max_area ...
create_clock ...
set_input_delay ...
```

2 Constrain (source)



Generic Boolean Gates  
(GTECH or unmapped ddc format)

3 Optimize + Map  
(compile\_ultra)



Technology-specific Gates

4 Save (write -f ddc) (mapped ddc format)

3- 15

GTECH components have no timing or load characteristics, and are only used by Design Compiler.

ddc is, by default, an internal DC format, which can also explicitly be written out.

# Technology Library

- 3 steps involved in synthesis:

- Translation
- Logic Optimization
- Mapping



**When DC maps a circuit, how will it know  
which cell library you are using?  
How will it know the timing of your cells?**

Your ASIC vendor must provide a DC-compatible  
technology library for synthesis!

**3-16**

# How is the Target Library Used?

- The *target library* is used during compile to create a technology-specific gate-level netlist
- DC optimization selects the smallest gates that meet the required timing and logic functionality
- Default setting: (`printvar target_library`)  

```
target_library = your_library.db
```

Non-existent default library name
- The user must specify the actual synthesis library file provided by the silicon vendor or library group

```
set target_library libs/65nm.db
```

TCL: Variable definition      Reserved DC variable

3-17

# Resolving ‘References’ with link\_library

- Default: `link_library = "* your_library.db"`

"\*"  
represents  
DC Memory

- To “resolve” the reference DC:

- First looks in DC memory for a matching design name
- Next looks in the technology library(ies) listed in the *link\_library* variable for a matching library cell name

- The user must replace the default link library with the name of the vendor-provided technology library before *link*

```
set link_library "* $target_library"
```

TCL: “Soft” quotes define a ‘list’ while allowing variable substitution (\$var)

3-18

A ‘reference’ is any gate, block or sub-design that is *instantiated* in your design.

*Designs* and *Library cells* are key DC database “objects”, to be discussed later in the Unit.

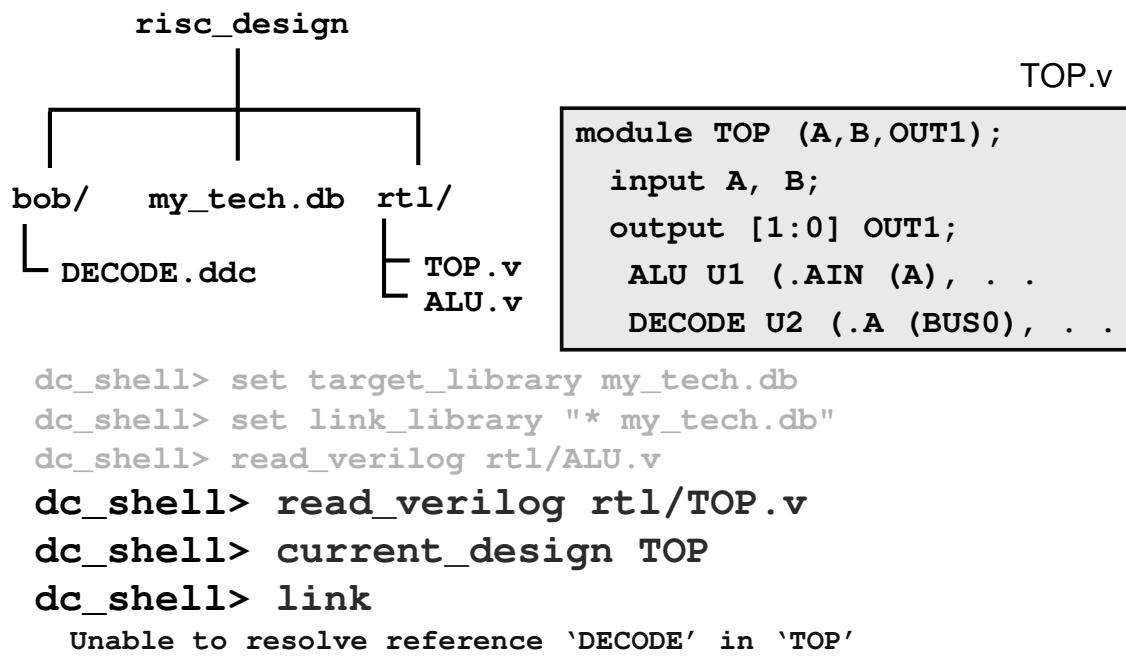
To display the value of the *link\_library* variable: `echo $link_library`

TCL: Curly braces, {...}, are treated as “hard quotes”: No variable substitutions or expression calculations are performed. The use of curly braces in the slide example would not work since the value of the *target\_library* variable would not be substituted - *link\_library* will be literally set to the character string: \* \$target\_library instead of \* libs/65nm.db

TCL: Variable substitution syntax: `$varName`. Variable name can be letters, digits, underscores: a-zA-Z0-9\_. Variables do not need to be declared: All of type “string” of arbitrary length. Substitution may occur anywhere within a word:

Sample command	Result
<code>set b 66</code>	66
<code>set a b</code>	b
<code>set a \$b</code>	66
<code>set a \$b+\$b+\$b</code>	66+66+66
<code>set a \$b.3</code>	66.3 (non-variable character “.” delineates the variable)
<code>set a \$b4</code>	“no such variable”
<code>set a \${b} 4</code>	664

# Use link to Resolve Design References



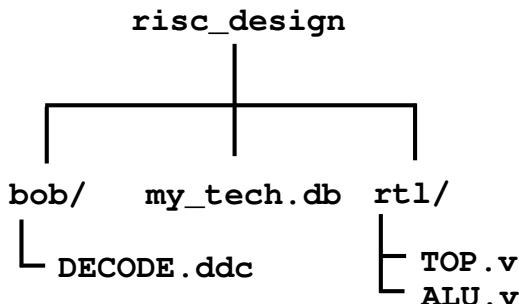
How do you tell DC to find DECODE.ddc in bob?

**3-19**

The purpose of the link command is to locate all of the designs and library components referenced in the current design and connect (link) them to the current design.

For a complete description and more details refer to the man page for link.

# Set the search\_path Variable



```
module TOP (A,B,OUT1);
  input A, B;
  output [1:0] OUT1;
  ALU U1 (.AIN (A), . .
  DECODE U2 (.A (BUS0), . .
```

```
dc_shell> set target_library my_tech.db
dc_shell> set link_library "* my_tech.db"
dc_shell> lappend search_path bob
dc_shell> read_verilog rtl/ALU.v
dc_shell> read_verilog rtl/TOP.v
dc_shell> current_design TOP
dc_shell> link
Loading ddc file bob/DECODE.ddc
```



**link only automatically loads ddc files, not Verilog or VHDL files**

**3-20**

Auto-loading is **NOT** recommended.

In general it is better to explicitly read in any design files that are needed. This reduces the risk of DC inadvertently auto-loading an un-intended design, or missing a design because the file name did not exactly match the auto-loading naming convention.

<sup>1</sup> Case-sensitive! If the file was called *DECODE.ddc* instead of *decode.ddc* then the *link* command would resolve this design as well.

Alternate commands to add a directory to the search\_path:

```
set search_path "$search_path bob"
set search_path [concat $search_path bob]
```

# DC-Topographical Setup

- When running DC-Topographical, physical libraries must be setup
- Example setup for Milkyway libraries

```
set mw_reference_library "./milkyway/max_lib"  
set mw_design_library my_design.mw  
create_mw_lib -technology $TECH_FILE \  
    -mw_reference_library $mw_reference_library \  
    $mw_design_library
```

3-21

# **Unit Summary**

---

**Having completed this unit, you should now be able to:**

- Create the setup file for DFT Compiler (DFTC)**
- Specify the target library and link library**
- Read an RTL design into DFTC**

**3-22**

# Command Summary (Lecture, Lab)

<code>dc_shell</code>	Invokes the Design Compiler command shell
<code>dc_shell -topo</code>	Invokes Design Compiler in topographical mode
<code>design_vision</code>	Runs Design Vision visualization GUI
<code>read_verilog</code>	Read one or more verilog files
<code>read_vhdl</code>	Read one or more vhdl files
<code>compile</code>	Performs logic-level and gate-level synthesis and optimization on the current design
<code>source</code>	Read a file and execute it as a script
<code>write -f ddc</code>	Writes a design netlist from memory to a file
<code>set target_library</code>	List of technology libraries to be used during compile
<code>set link_library</code>	List of design files and libraries used during linking
<code>current_design</code>	Sets the working design
<code>link</code>	Resolves design references
<code>mw_reference_library</code>	Contains the milkyway reference libraries
<code>mw_design_library</code>	Contains the milkyway design library
<code>create_mw_lib</code>	Creates a Milkyway library

3-23

This page was intentionally left blank.

# Agenda

**DAY  
1**

**1** Introduction to Scan Testing



**2** DFT Compiler Flows



**3** DFT Compiler Setup



**4** Test Protocol



**5** DFT Design Rule Checks



# Unit Objectives



**After completing this unit, you should be able to:**

- **Describe the elements that are included in a Test Protocol file**
- **Declare constants, clocks, resets, scan ins, scan outs, and scan enables to DFTC**
- **Name at least two DFTC user interface commands**
- **Create the test protocol and modify in necessary to include a custom initialization sequence**

**4- 2**

# Test Protocol: Agenda

**Test Protocol**

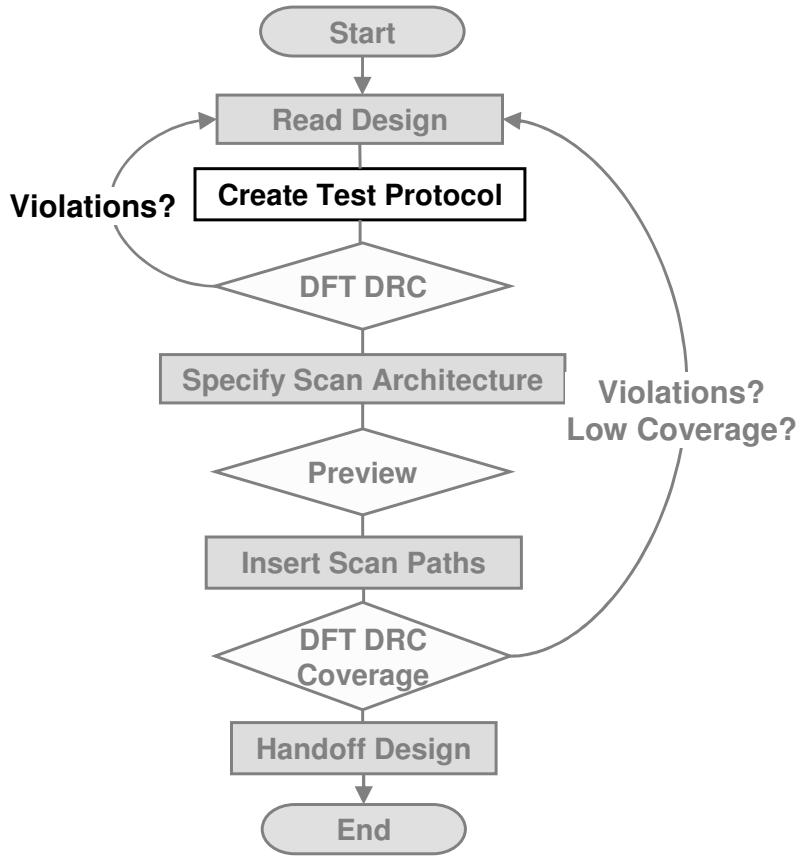
**DFT Signals**

**Protocol Timing**

**Creating a Protocol**

4-3

# Typical Scan Insertion Flow



4-4

# What is a Test Protocol?

- The test protocol explains how to control the design in Test mode (clocking, scan shifting, disabling asynchronous Resets ...)
- A test protocol needs to be created prior to running DFT operations such as DRC checks, previewing the scan architecture, and inserting the scan architecture

4-5

# Steps for Creating a Test Protocol

1. Define scan signals, clocks, resets, constants, etc.

`set_dft_signal ....`

2. Create / Read the test protocol using

`create_test_protocol`

or

`read_test_protocol`

4-6

# DFT Compiler UI command conventions

- DFT specification commands (`set_*`) have a corresponding `report_*` and `reset_*/remove_*` commands

- ◆ `set_*`      Creates the specification
- ◆ `report_*`    Reports the specification
- ◆ `reset_*`    Restores settings back to default
- ◆ `remove_*`   Removes the specification

- Example:

```
set_dft_signal » report_dft_signal » remove_dft_signal
```

4-7

# Test Protocol: Agenda

**Test Protocol**

**DFT Signals**

**Protocol Timing**

**Creating a Protocol**

4-8

# The set\_dft\_signal command

- The `set_dft_signal` command is used to define signals that are required for the test protocol

```
set_dft_signal -view <existing_dft | spec>  
    -type <signal_type>  
    -port <port_list>  
    -active_state <0 / 1>  
    -hookup_pin <pin_path>  
    -timing <clock/reset timing>
```

4-9

# Identify Ports in RTL to Use for Scan

- You can define dedicated scan-access ports (ScanEnable, ScanIn, ScanOut) in your top-level RTL code

```
entity SINE_GEN
port(
  CLOCK:in BIT;
  SE:in    BIT;
  SI:in    BIT;
  SO:out   BIT;
  SINE:out ...);

```

- This avoids editing your RTL testbench to match the port list after scan insertion
- Identify dedicated ports and reuse functional ports as scan access ports using the `set_dft_signal` command

4-10

# DFT Compiler “Views”

- DFT Compiler uses views with the `set_dft_signal` command to control how DFT signal are used by `dft_drc` and `insert_dft`
  - `-view spec` : DFT signals that DFT Compiler should use during `insert_dft`
  - `-view existing_dft` : DFT signals which must be understood for the design to pass `dft_drc`
- The default view is `-view spec`
- The reporting command, `report_dft_signal`, displays both views by default
- Rule of thumb:
  - `-view spec` is used ONLY for `insert_dft`
  - `-view existing_dft` is used ONLY for `dft_drc`

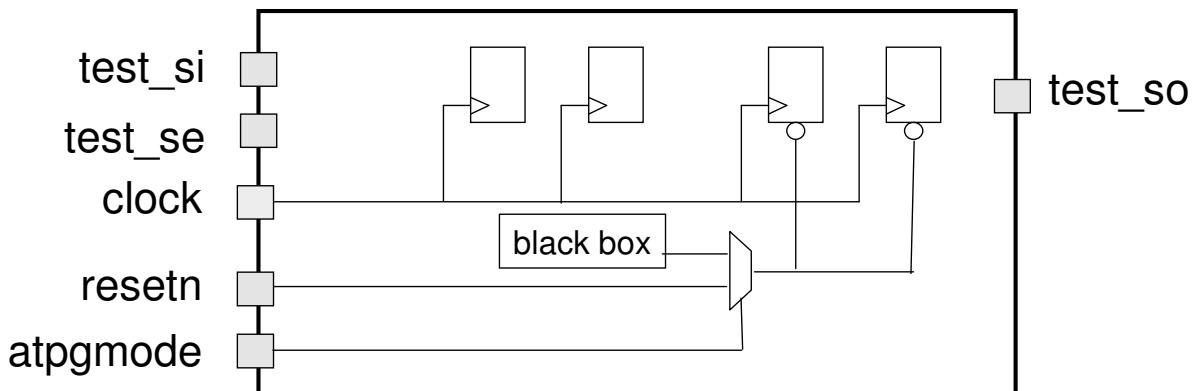
4-11

The `-view` options can be abbreviated. For example:

“`-view spec`” -> “`-view s`” -> “`-v s`”  
“`-view existing_dft`” -> “`-view exist`” -> “`-v e`”

# How to Specify a Clock

```
set test_default_period 100
set_dft_signal -view existing_dft \
    -type ScanClock \
    -port clock -timing [list 45 55]
```



4-12

For Multiplexed Flip-flop design styles, the signal “type” of **ScanClock** is shorthand for specifying a **ScanMasterClock** (scan shift clock) and a **MasterClock** (capture clock). For a “Mux-D” scan designs, a clock is typically used for both shift and capture. When a **ScanClock** is defined and the reported with “report\_dft\_signal”, you will see both **ScanMasterClock** and **MasterClock** attributes. For example:

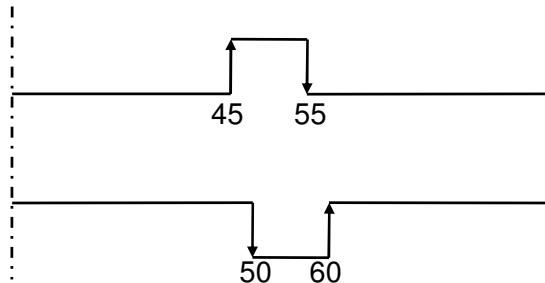
```
> report_dft_signals -view existing_dft
```

```
=====
TEST MODE: all_dft_user
VIEW      : Existing DFT
=====
Port        SignalType       Active   Hookup   Timing
-----  -----  -----  -----
ext_clk     ScanMasterClock  1        -        P 100.0 R 50.0 F 80.0
ext_clk     MasterClock      1        -        P 100.0 R 50.0 F 80.0
rst_b       Reset            0        -        P 100.0 R 55.0 F 45.0
```

# Using set\_dft\_signal -timing

- The `set_dft_signal -timing` option is used to define timing and waveform for a clock or reset signal
- The `-timing` option accepts a list argument with two values corresponding to the Rising Edge (RE) and Falling Edge (FE) of the clock, in that order

Return To Zero (RTZ) Waveform  
`-timing {45 55}`

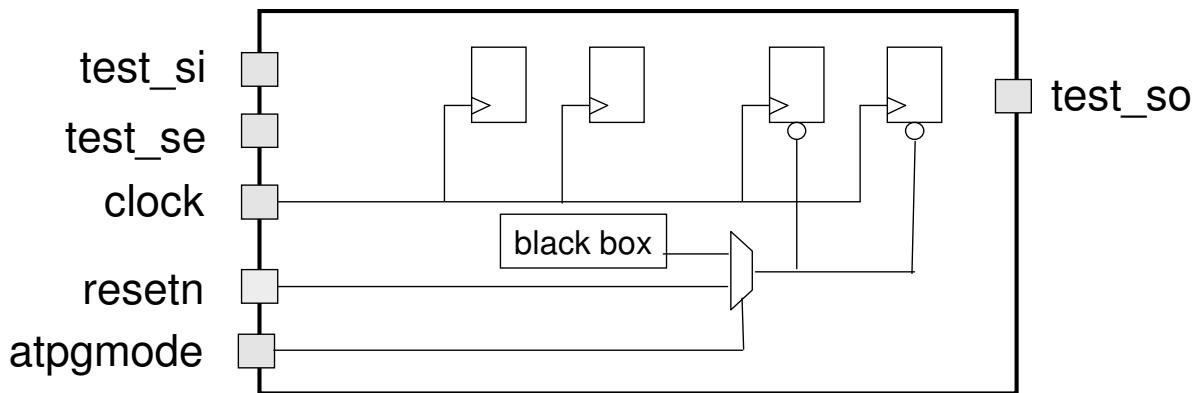


Return To One (RTO) Waveform  
`-timing [list 60 50]`

4-13

# How to Specify a Reset

```
set_dft_signal -view existing_dft \
    -type Reset \
    -port resetn -active_state 0
```



4-14

The `-timing` option is optional with “`-type Reset`”.

If the `-timing` option is omitted, DFT Compiler will apply the default timing of {45 55} for “`-active_state 1`”, or {55 45} for “`-active_state 0`”. The default active state is “1”.

# Asynchronous Resets (and Sets)

- DFTC treats both clocks (-type ScanClock) and asynchronous resets (-type Reset) as “clocks” in the protocol
- The default timing for Reset is active high with rising edge at 45 and falling edge at 55

```
set_dft_signal -view exist -type Reset -port S_RESET_N
Port          SignalType    Active      Timing
-----        -----        -----      -----
S_RESET_N     Reset         1           P 100.0 R 45.0 F 55.0
```

- The Reset timing can be more explicitly specified by using the -timing option

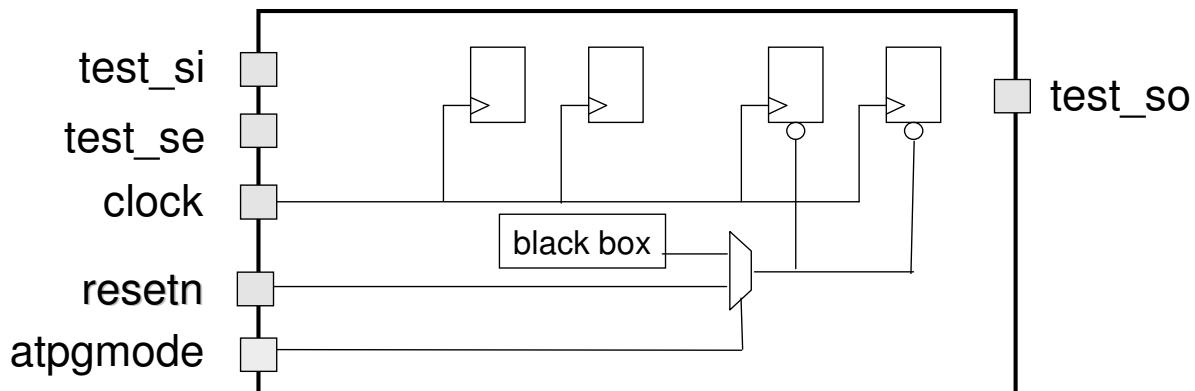
4-15

Recall that the type of waveform is controlled by the “-timing” option and how waveform timing is specified in a Rising Edge (RE) then Falling Edge (FE) order.

The Timing column in the above reports behave the same way. It reports the Period: **P 100.0** , Rising Edge: **R 55.0**, and Falling Edge: **F 45.0**

# How to Specify a Constant

```
set_dft_signal -view existing_dft \
    -type Constant \
    -port atpgmode -active_state 1
```

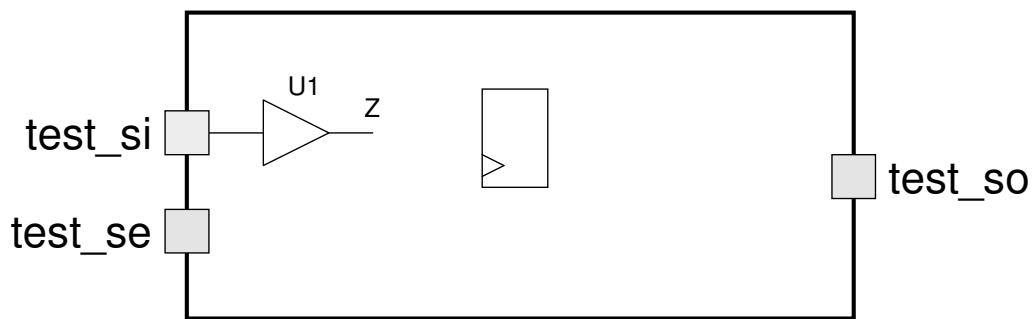


4-16

Since the connection is made to bypass reset using **atpgmode**, we define this port as “**-view existing\_dft -type constant**” NOT “**-type TestMode**”.

# How to Specify a Scan In

```
set_dft_signal \
    -view spec           « Specify the view (default is spec)
    -type ScanDataIn \ « The DFT signal type
    -port test_si \     « Top level port to use
    -hookup_pin U1/z   « Hookup pin: omit if you want
                        to connect directly to port
```



4-17

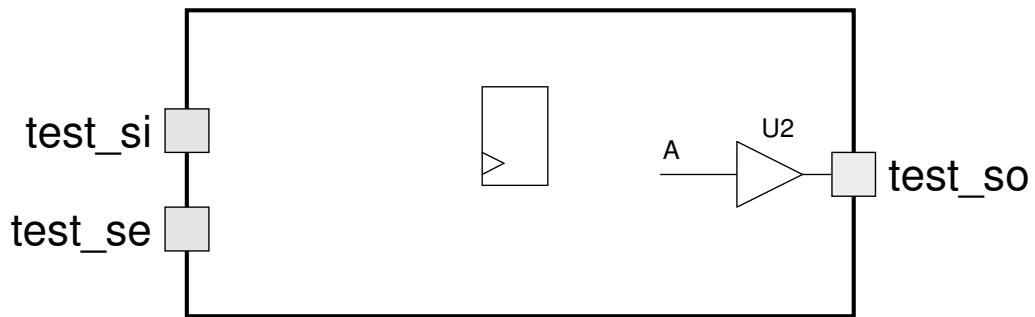
Signal DataTypes in `set_dft_signal` can be specified either fixed keywords or lowercase

Example:

```
set_dft_signal -type ScanDataIn
set_dft_signal -type scandatain
```

# How to Specify a Scan Out

```
set_dft_signal \
    -view spec           « Specify the view
    -type ScanDataOut \
    -port test_so \
    -hookup_pin U2/A    « Connection point
```



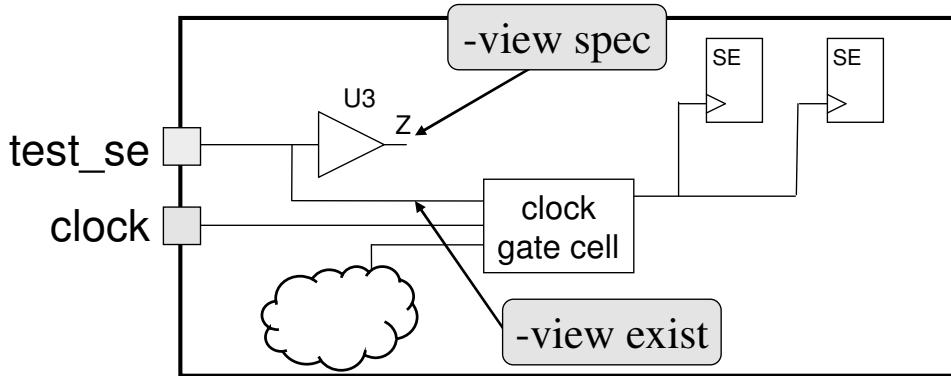
4-18

# How to Specify a Scan Enable

- Scan enable used for both `dft_drc` and `insert_dft`

```
set_dft_signal -view exist -active 1 \
    -type ScanEnable -port test_se \
```

```
set_dft_signal -view spec -active 1 \
    -type ScanEnable -port test_se \
    -hookup_pin U3/Z
```



4-19

Since the scan enable is already routed to the clock gating cells and we need `dft_drc` to understand that it will be set to a 1 during shift, we need to declare the `test_se` port as `-view exist`.

If It was only declared as `-view spec`, then `dft_drc` will flag all the flops that are driven by these clock gating cells as violated. You will end up with uncontrollable clock violations D1/D9.

# How to Specify a Differential Clock?

- **Used on IO pads**

- Provides additional noise margin over single ended clock pads
- Typically used in high speed interfaces to cancel board layout noise
- The two clock inputs need to be at OPPOSITE values in order to generate output clock pulses

- **Example specification of differential clock ports clk\_p and clk\_n**

```
set_dft_signal -view existing_dft -type ScanClock \
    -timing {45 55} -port clk_p
set_dft_signal -view existing_dft -type ScanClock \
    -port clk_n -differential {clk_p}
```

- **Only the reference clock (clk\_p) contains timing information**

- Must be specified first

- **The differential clock (clk\_n) will automatically use the complementary waveform of the reference clock**

- Any timing information that is specified will be ignored

**4-20**

Differential clocks must be top level port (internal clocks not supported)

Supported in mux-scan and LSSD styles

# When to use `-view existing_dft`

- “Rule of Thumb” – signals that need to be identified for `dft_drc`

Purpose	View and Type
Mux-D scan clock for DRC	<code>-view exist -type ScanClock</code>
Asynchronous reset (or set) for DRC	<code>-view exist -type Reset</code>
Constant signal for DRC	<code>-view exist -type Constant</code>
ScanEnable as Test pin for clock gates	<code>-view exist -type ScanEnable</code>
Port for LSSD A clock	<code>-view exist -type ScanMasterClock</code>
Port for <code>clocked_scan</code> clock	
Port for LSSD B clock	<code>-view exist -type ScanSlaveClock</code>

4-21

This shows “rule of thumb” usage of `-view exist` for the `set_dft_signal` command. It only covers signal types required for a typical scan chain insertion flow. There are additional signal types used for other DFT flows (i.e. On-Chip Clocking – OCC).

# When to use `-view spec`

- “Rule of Thumb” – signals that are connected by `insert_dft`

Purpose	View and Type
Scan input for stitching	<code>-view spec -type ScanDataIn</code>
Scan output for stitching	<code>-view spec -type ScanDataOut</code>
Scan enable for stitching	<code>-view spec -type ScanEnable</code>
Mode port for DFTMAX or Autofix	<code>-view spec -type TestMode</code>
Data signal for Autofix	<code>-view spec -type TestData</code>
‘A’ port for LSSD scan stitching	<code>-view spec -type ScanMasterClock</code>
Clk port for <code>clocked_scan</code> stitching	
‘B’ port for LSSD scan stitching	<code>-view spec -type ScanSlaveClock</code>

4-22

This shows “rule of thumb” usage of `-view spec` for the `set_dft_signal` command. It only covers signal types required for a typical scan chain insertion flow. There are additional signal types used for other DFT flows (i.e. On-Chip Clocking – OCC).

# Example: Declaring DFT Signals

```
read_file -f verilog rtl.v
current_design top
link
set_scan_configuration -style multiplexed_flip_flop
compile -scan
set_dft_signal -view existing_dft -type ScanClock \
    -port clk -timing [list 45 55]
set_dft_signal -view existing_dft -type Reset \
    -port resetn -active_state 0
create_test_protocol -capture_procedure multi_clock
dft_drc
set_dft_signal -type ScanDataIn -port test_si
set_dft_signal -type ScanDataOut -port test_so
set_dft_signal -type ScanEnable -port test_se
preview_dft
insert_dft
change_names -rules verilog -hier
write_scan_def -o scanned.scandef
write -f ddc -hier -o scanned.ddc
write -f verilog -hier -o scanned.v
write_test_protocol -o scanned.spf
dft_drc -coverage_estimate
report_scan_path -view existing_dft -chain all
```

4-23

Recall: The default for `-view` is “**spec**”

All clocks, resets, constants, etc. need to be declared with **set\_dft\_signal** prior to generating a protocol with **create\_test\_protocol**.

# Managing DFT Signals

- You can report specified DFT signals using  
`report_dft_signal`
- You can remove selected signals via  
`remove_dft_signal -port portname`

4- 24

# Test Protocol: Agenda

**Test Protocol**

**DFT Signals**

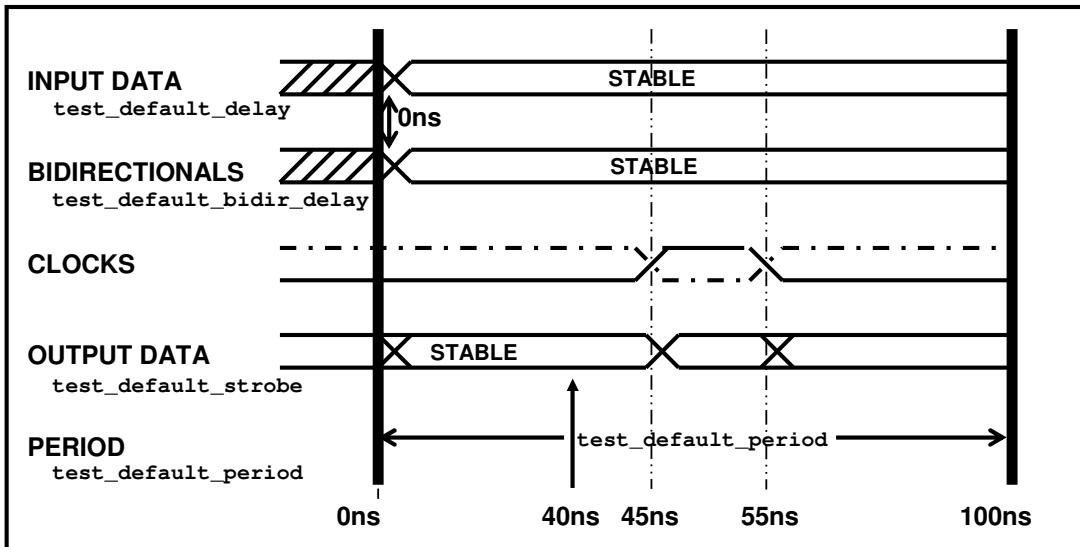
**Protocol Timing**

**Creating a Protocol**

**4-25**

# Tester Timing Configuration

- The test protocol also includes timing and event order to be applied on the tester



4-26

# Specifying Tester Timing

## ■ Variables that affect the timing in the Test Protocol

### *Defaults*

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

- Defaults will setup “Pre-Clock Measure” timing (Recommended)

```
set_dft_signal -view exist -type ScanClock \
               -port clk_RTZero -timing [list 45 55 ]
set_dft_signal -view exist -type ScanClock \
               -port clk_RTOne -timing [list 55 45 ]
```

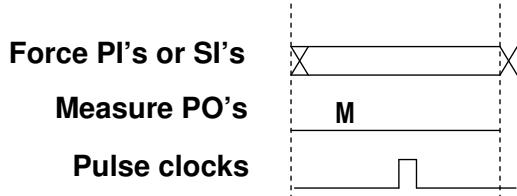
4-27

Prior to version 2006.06-SP3, the defaults were setup for “End-of-Cycle Measure”:

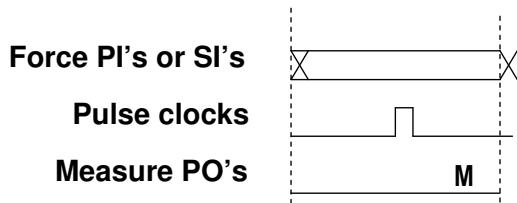
```
set test_default_period      100
set test_default_delay       5
set test_default_bidir_delay 55
set test_default_strobe      95
```

# Pre-Clock Measure vs. End-of-Cycle Measure

- Two types of timing configurations used on testers
- Pre-clock Measure
  - Default timing used by TetraMAX and DFTC (recommended)



- End-of-cycle Measure
  - Common timing used by older testers

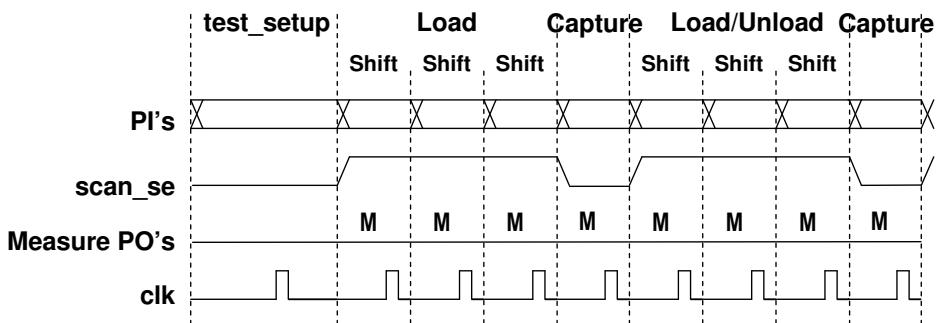


4-28

# Pre-Clock Measure

- When the default pre-clock timing is used, the capture procedure can use a single vector (required for At-Speed ATPG)

```
"capture_clk" {  
    W "_default_WFT_";  
    V { "_pi"=\r61 # ; "_po"=\r90 # ; "clk"=P; }  
}
```



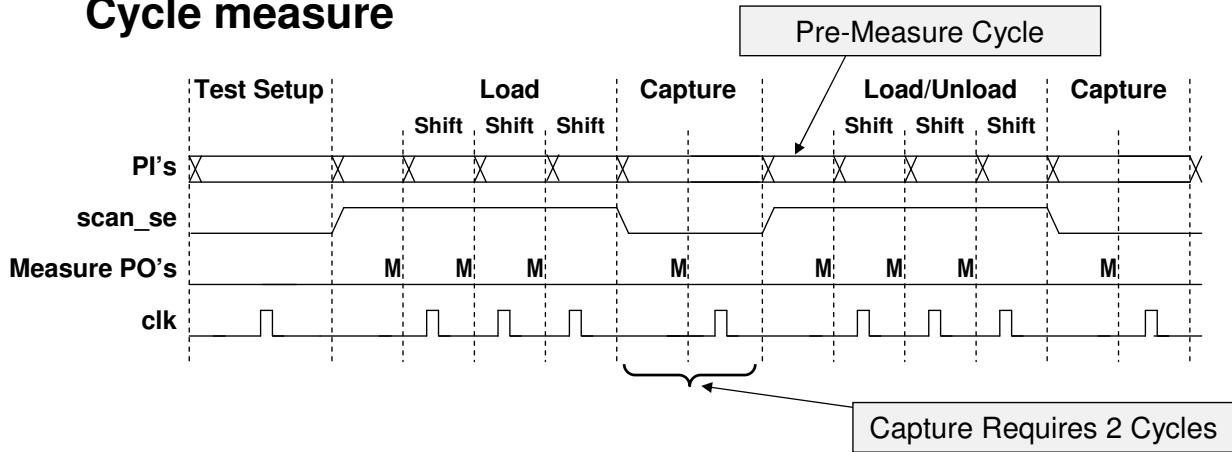
4-29

Example **load\_unload** and **Shift** procedures for Pre-Clock measure timing:

```
Procedures {  
    "load_unload" {  
        W "_default_WFT_"; // use default timing  
        V {CLK=0; SCLK=0; RSTB=1; OE=0;SE=1; bidi_pins = ZZZZ;}  
        Shift {  
            V { "_si"= # ; "_so"= # ; CLK=P; }  
        } // end shift  
    } //end load_unload  
} // end procedures
```

# End-of-Cycle Measure

- At least 2 additional tester cycles required for End-of-Cycle measure



- Capture Cycle 1:** Force PI's/Measure PO (end of cycle)  
**Capture Cycle 2:** Pulse clocks/Mask PO's

## 900679 NON-END-OF-CYCLE-Measure vs. END-OF-CYCLE-Measure

<https://solvnet.synopsys.com/retrieve/print/900679.html>

4-30

Note: For End-of-Cycle measure, at least one additional tester cycle will be needed for every ATPG pattern. This additional cycle is placed in the **load\_unload** procedure and performs a scan chain pre-measure prior to the **Shift** procedure.

Example **load\_unload** and **Shift** procedures with End-of-Cycle measure timing:

```
Procedures {
    "load_unload" {
        W "_default_WFT_"; // use default timing
        V {CLK=0; SCLK=0; RSTB=1; OE=0;SE=1; bidi_pins = ZZZZ;}
        V {"_so" = # ;} // pre-measure the first scan chain output
        Shift {
            V { "_si"= # ;"_so"= # ; CLK=P; }
        } // end shift
    } //end load_unload
} // end procedures
```

# Synopsys Test Tip

## 3.1 Package Timing Defaults

Test-clock definitions and timing defaults always work together. Think of them as a timing package. This avoids inconsistencies like an active test-clock edge occurring *before* a pre-clock strobe.

One way to package these specifications together is to put them all into a setup script like the following:

```
# Company XYZ Setup Script
set test_default_period 100
set test_default_strobe 40
set test_default_delay 0
. . .
```

**4-31**

# Test Protocol: Agenda

**Test Protocol**

**DFT Signals**

**Protocol Timing**

**Creating a Protocol**

**4-32**

# The `create_test_protocol` command

- The `create_test_protocol` is used to create the test protocol based on the DFT signals and timings currently defined

```
create_test_protocol  
[-infer_asynch] [-infer_clock]  
[-capture_procedure single_clock | multi_clock]
```

- Once a test protocol has been created, it can be written out to disk with the `write_test_protocol` command

```
write_test_protocol -output <filename>
```

4-33

**Note:** There are some commands that can “invalidate” the current test protocol and will require the protocol to be regenerated with `create_test_protocol`. Among the commands that can invalidate a protocol are:

```
create_port, remove_port,  
read_test_model, remove_test_model,  
read_test_protocol, remove_test_protocol,  
set_dft_signal, remove_dft_signal,  
set_dft_equivalent_signals, remove_dft_equivalent_signals,  
set_dft_clock_controller,  
set_test_assume,  
set_dft_drc_configuration -internal_pins enable,  
define_test_mode,  
define_dft_design, remove_dft_design,  
set_scan_group -serial_routed true
```

# Test Protocol Generation Example

```
# Enable RTL source line tracking
set hdlin_enable_rtldrc_info true

# Read RTL Design
read_verilog rtl/ORCA.v
current_design ORCA ; link

# Specify test clocks and other attributes
set_dft_signal -view exist -type ScanClock \
    -timing {45 55} -port pclk
set_dft_signal -view exist -type Reset \
    -active_state 0 -port prst_n
set_dft_signal -view exist -type Constant \
    -active_state 1 -port test_mode

# Step 2: Create the test protocol
create_test_protocol -capture_procedure multi_clock

# Run test design rule checking
dft_drc

# Write out test protocol for later use
write_test_protocol -o unmapped/ORCA.spf
```

4-34

# Use Protocol Inference Only When No Choice

```
# Unfamiliar design!!
# Specify the clocks and resets that you do know...

set_dft_signal -view exist -type ScanClock \
                -timing {45 55} -port clk1
set_dft_signal -view exist -type Reset \
                -active_state 0 -port rst1_n
set_dft_signal -view exist -type Constant \
                -active_state 1 -port test_model

# ...and infer the remaining clocks and resets at the
# expense of additional runtime and perhaps accuracy

create_test_protocol -infer_clock -infer_asynch

# run test design rule checking
dft_drc

# Iterate, if needed, in this discovery process
# After a clock/reset is discovered, specify it
# explicitly in the next run
```

4-35

Caution when inferring clocks and resets there are situations where you could end up with wrong info:

- Designs with clock gating
- Designs with both positive and negative edge flops

You could also end up with incorrect timing defined on the clocks.

# Generic Capture Procedures

- **Special clock procedures (generic capture procedures) which replace the individual capture clock procedures (i.e. “capture\_<clk>”)**
- **Allows greater flexibility during the ATPG process**
  - Same procedure called for any number of clocks
- **The resulting ATPG patterns are easier to understand when debugging with generic capture procedures**
- **The same protocol file can now be used for both Stuck-At and Transition ATPG**
  - Users still need to modify the WFT for At-Speed timing
  - Not supported for Path Delay fault model

4-36

# Creating a Generic Capture Procedures SPF

- A Test Protocol with generic capture procedures is created using the following command

```
create_test_protocol -capture_procedure multi_clock
```

- Default is “single” (i.e. no generic capture procedures)

- All procedures inherit the timing from \_default\_WFT\_

- User needs to edit the SPF from DFT Compiler to meet design timing requirements for delay testing in ATPG

4-37

# Multiple Clock Capture Procedure

- The Multiple Clock Capture procedure is one that can be used for any capture operation
  - Zero (0), one (1) or multiple
- Clocks are part of the signalGroup \_pi
- WFT timing must follow TetraMAX event ordering:
  - Force PI, measure PO, pulse clock
- Defined by `multiclock_capture{ }` in the SPF

4-38

# Allclock Capture Procedures

- `allclock_capture{ } : applied to tagged capture operations in launch/capture contexts only`
- `allclock_launch{ } : applied to tagged launch operations in launch/capture contexts only`
- `allclock_launch_capture{ } : applied to tagged launch-capture operations only`
- Tagged – A label for a procedure that will be called by TetraMAX ATPG
- Only `system_clock` launch is supported for delay test ATPG

4-39

# Generic Capture Procedures Example

```
Procedures {
    "multiclock_capture" {
        W "_multiclock_capture_WFT_";
        C {
            "all_inputs" = 00 \r91 N 111 \r14
            0 \r33 N 1 \r32 N;
            "all_outputs" = \r165 X;
            "all_bidirectionals" = ZZZ;
        }
        F {
            "i_scan_block_sel[0]" = 1;
            "i_scan_block_sel[1]" = 1;
            "i_scan_compress_mode" = 0;
            "i_scan_testmode" = 1;
        }
        V {
            "_po" = \r168 #;
            "_pi" = \r179 #;
        }
    }
}
```

```
"allclock_capture" {
    W "_allclock_capture_WFT_";
    C {
        ...
    }
    F {
        ...
    }
    V {
        "_po" = \r168 #;
        "_pi" = \r179 #;
    }
}
"allclock_launch" {
    W "_allclock_launch_WFT_";
    C {
        ...
    }
    F {
        ...
    }
    V {
        "_po" = \r168 #;
        "_pi" = \r179 #;
    }
}
"allclock_launch_capture" {
    W "_allclock_launch_capture_WFT_";
    C {
        ...
    }
    F {
        ...
    }
    V {
        "_po" = \r168 #;
        "_pi" = \r179 #;
    }
}
```

4-40

# Already Have a Test Protocol?

```
# Read mapped design and test protocol
read_ddc mapped/ORCA.ddc
current_design ORCA
link

# Read protocol with Clocks and Constants defined
read_test_protocol unmapped/ORCA.spf

# Run test design rule checking at the gate-level
dft_drc

# Continue with rest of the flow
# ...
```

4-41

# Using `read_test_protocol` for Initialization

- `read_test_protocol -section test_setup` can be used to load a sequence of vectors which place the design in a desired state *prior* to starting DFT rule checks

```
read_test_protocol -section test_setup <filename>
create_test_protocol -capture_procedure multi
dft_drc
```

- Design Vision can be used to view and debug custom initialization sequences

4-42

# Customizing STIL Protocol Files

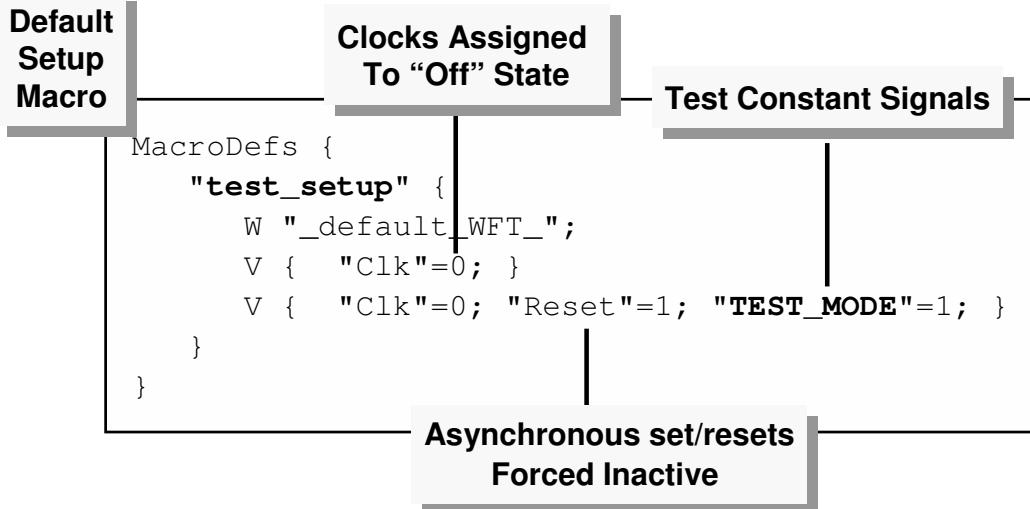
```
STIL 1.0;  
Header {  
}  
  
Signals {  
}  
  
ScanStructures {  
}  
  
Timing {  
}  
  
Procedures {  
}  
  
MacroDefs {  
    "test_setup" {  
    }  
}
```

Customizing  
STIL Protocol Files  
(.spf)

Custom  
Initialization

4- 43

# Default Initialization Macro is Very Basic

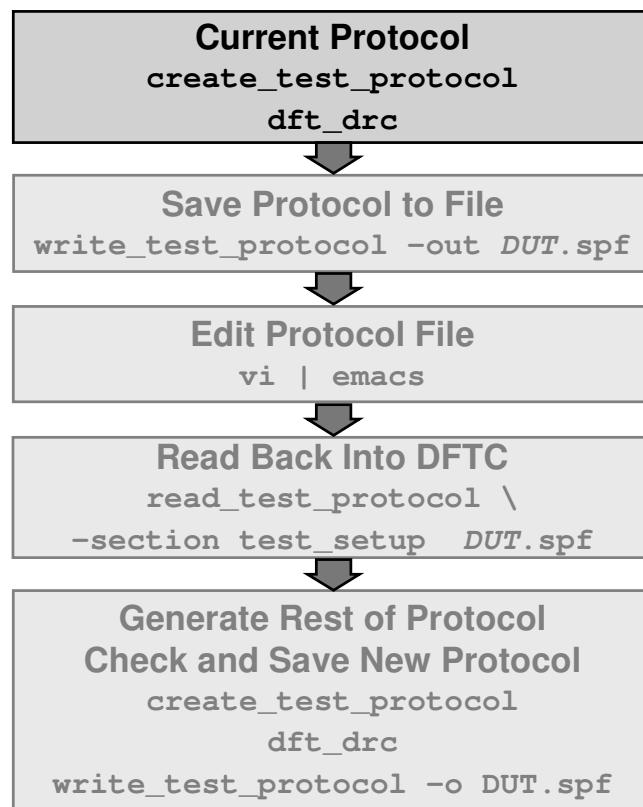


- Created automatically by DFTC for basic initialization
- For a more complex initialization sequence, the user must define the `test_setup` macro

4- 44

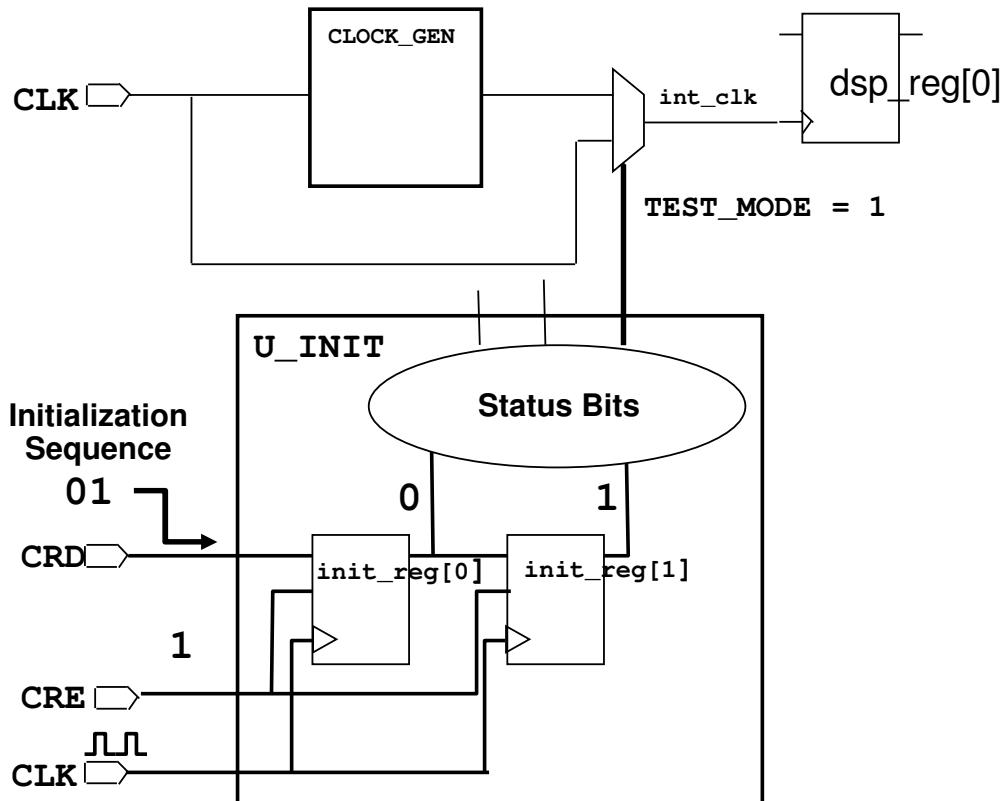
# Initialization Protocol Flow Detail

## Defining Custom Protocols



4-45

# Custom Initialization Example



4-46

In this design, **TEST\_MODE** is **not** a primary input; you cannot use **set\_dft\_signal -type Constant**. This is the case for pin-limited designs, where you can not afford a **TEST\_MODE** port. Existing **configuration logic** is used here to generate an on-chip **TEST\_MODE** signal. Bits serially clocked into flip-flops **init\_reg[0]** and **init\_reg[1]** are decoded to generate status signals. You can assume one serial bit pattern is **unused** in normal operation; it will be used for test. The penalty is that you have to initialize the configuration register **before** scan shift; therefore, you must add a **custom initialization sequence** to the test protocol.

# Example STIL Initialization Macro

```
Custom  
Initialization  
Macro
```

```
MacroDefs {  
    "test_setup" {  
        . . .  
        V {"CRD" = 1; "CRE" = 1; "CLK" = P;}  
        V {"CRD" = 0; } // CLK=P again, CRE still 1  
        V {"CRE" = 0; "CLK" = 0; }  
        . . .  
    }  
}
```

**STIL Signal Names Are Case-Sensitive**

**STIL state assignments are persistent**

**CLK off at end**

4-47

In STIL, state assignments are persistent. Only need to specify the signals that change from vector to vector. A signal will maintain its value on subsequent vectors unless explicitly changed.

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- **Describe the elements that are included in a Test Protocol file**
- **Declare constants, clocks, resets, scan ins, scan outs, and scan enables to DFTC**
- **Name at least two DFTC user interface commands**
- **Create the test protocol and modify in necessary to include a custom initialization sequence**

**4- 48**

# Lab 4: Creating Test Protocols



60 minutes

**After completing this lab, you should be able to:**

- **Write a script to create and verify a test protocol**
- **Explore a `dc_shell` log file**

**4- 49**

# Command Summary (Lecture, Lab)

<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>report_dft_signal</code>	Displays options set by <code>set_dft_signal</code> command
<code>remove_dft_signal</code>	Removes attributes that identify a port as a DFT signal
<code>create_test_protocol</code>	Creates a test protocol based on user specification
<code>create_test_protocol -capture_procedure multi_clock</code>	Specifies the capture procedure type. Use <code>multi_clock</code> to create a protocol with generic capture procedures
<code>read_test_protocol</code>	Reads a test protocol file into memory
<code>read_test_protocol \ -section test_setup</code>	Reads only the specified section and ignores others. The only valid parameter is <code>test_setup</code>
<code>write_test_protocol</code>	Writes a test protocol file
<code>set test_default_period</code>	Defines the default length of a test vector cycle
<code>set test_default_delay</code>	Defines the default time to apply values to input ports
<code>set test_default_bidir_delay</code>	Defines the default switching time of bidirectional ports
<code>set test_default_strobe</code>	Defines the default strobe time for output ports
<code>dft_drc</code>	Checks the current design against test design rules
<code>insert_dft</code>	Adds scan circuitry to the current design

4-50

# Agenda

**DAY  
1**

**1** Introduction to Scan Testing



**2** DFT Compiler Flows



**3** DFT Compiler Setup



**4** Test Protocol



**5** DFT Design Rule Checks



# Unit Objectives

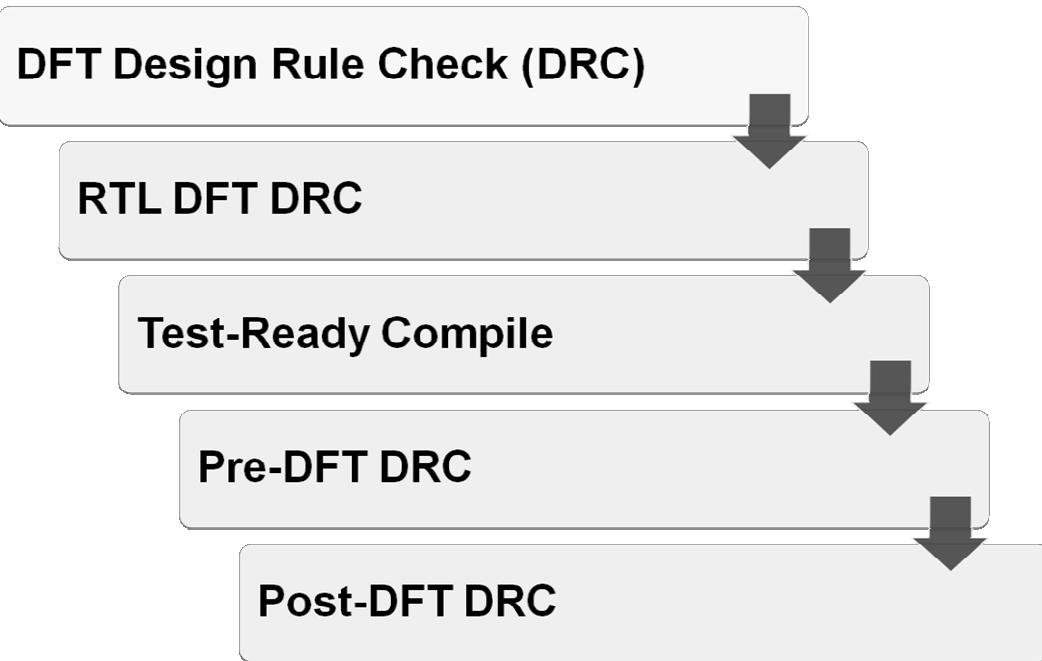


After completing this unit, you should be able to:

- Name the three type of DRC checks DFTC can perform on a design
- Perform a Test-Ready compile on a design
- Control internal nodes for the purposes of passing DRC

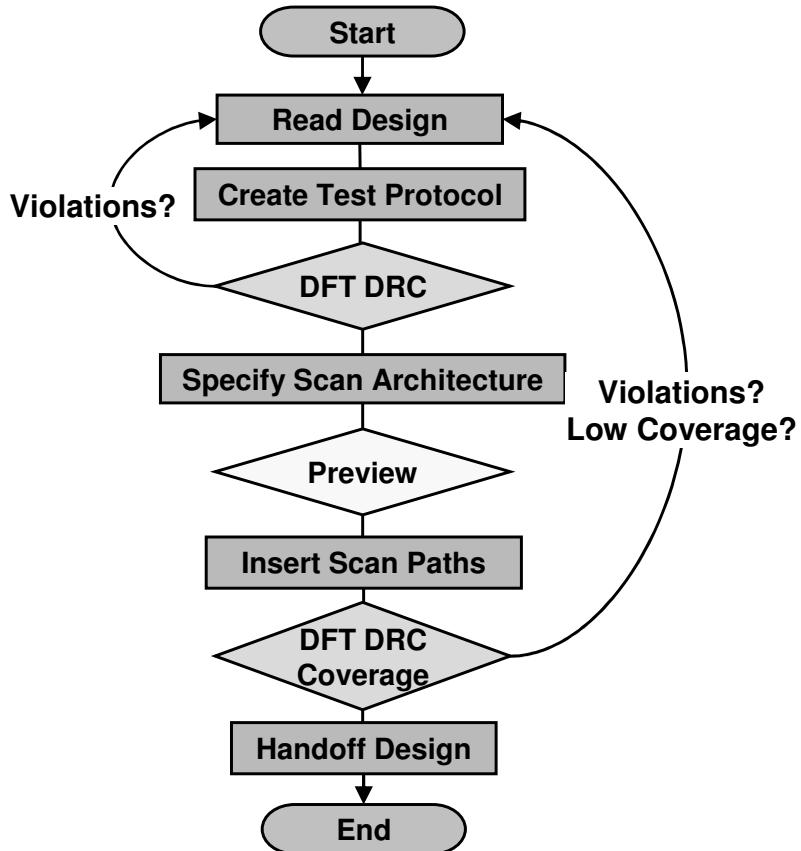
5-2

# DFT Design Rule Check: Agenda



5-3

# Typical Scan Insertion Flow



5-4

# Test Design Rule Checks (DFT DRC)

- DFT DRC uses a shared, unified DRC engine common to TetraMAX ATPG and DFT Compiler (`dft_drc` command)
- Benefits:
  - Same Design Rule Checker from **RTL through gates**
  - Checks for the same design rule violations **between DFT and ATPG tools**
  - Same design rule violation **messages** between DFT and ATPG tools
  - Enhanced debugging through Design Vision GUI

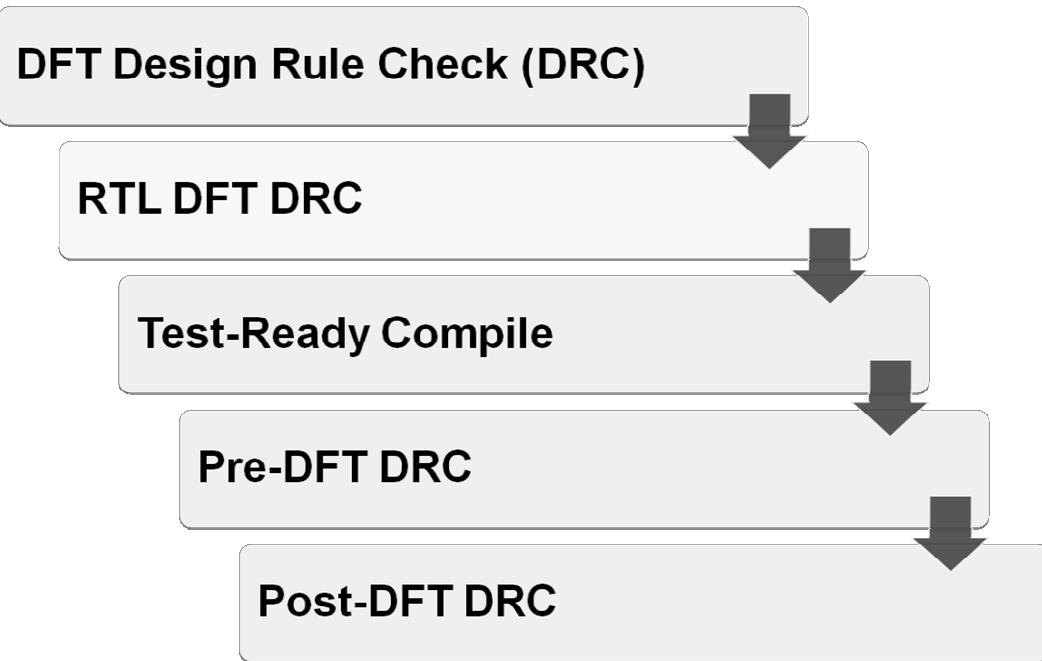
5-5

# Running dft\_drc

- Once the test protocol is available, DFT DRC checks can be run using: `dft_drc`
- Options:
  - `-verbose`: Verbose reporting
    - ◆ Lists every violation in design (same level of detail)
    - ◆ Lists all sequential cells in design
    - ◆ Does not facilitate better debug, use DV for debug
  - `-coverage`: Estimate test coverage (gate post-DFT only)
  - `-sample`: percentage of faults sampled (post-DFT only)
  - `-pre_dft`: Force pre-DFT DRC checks to performed

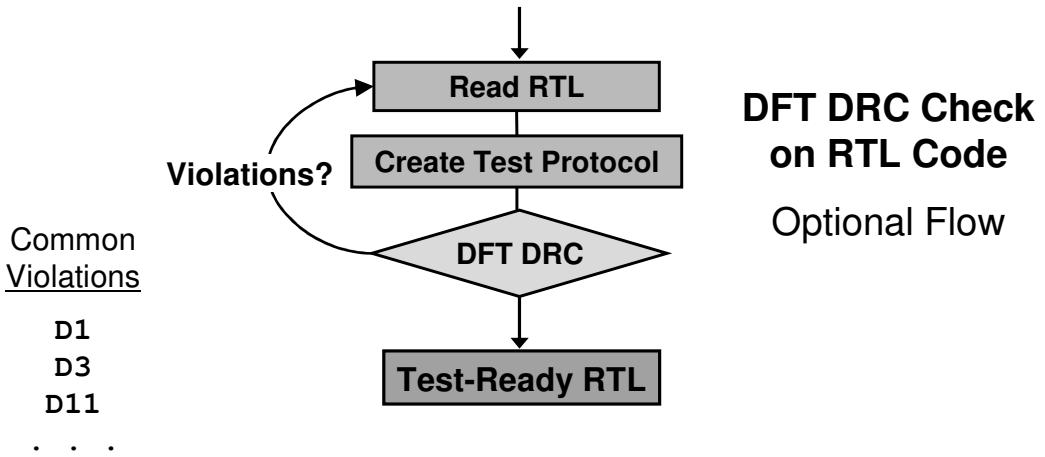
5-6

# DFT Design Rule Check: Agenda



5-7

# Running DFT DRC on RTL Code



5-8

This slide shows the optional **dft\_drc** flow, done **prior** to Test-Ready Compile; beforehand, you must define the **test protocol** with details such as tester-clock period. It is strictly an **optional** part of the DFTC flow—it helps you write DFT-smart code. It can **correlate** violations with specific source code file names and **line numbers**.

## Technical Reference:

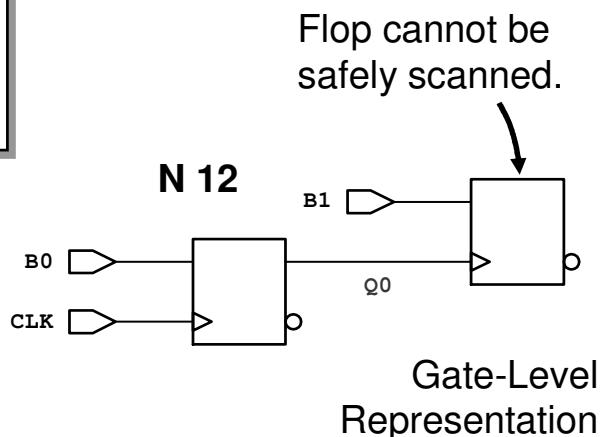
For full details on violation codes such as **D11**, see the corresponding man pages.

The **D11** violation warns of clocks which affect both flip-flop clock **and** data pins.

# Example: Ripple-Counter Violation in RTL

```
// Violating RTL Fragment
always @(posedge CLK)
    Q0 <= B0;
always @(posedge Q0)
    Q1 <= B1;
```

## D1: Uncontrollable Clock Violation



- This kind of violation is caught as early as the coding phase
- Enables DFT closure and reinforces your DFT guidelines

5-9

This slide shows one typical violation that **dft\_drc** is also proficient at detecting.

Scan or Capture Violations:

- Uncontrollable clock lines and asynchronous set/reset lines.
- Master/slave latches clocked on the same phase.
- Clocks used as data, or affecting asynchronous set/reset pins.
- Glitch prone latched clock-gating circuits.

Modeling and Topology Problems:

- Black box cells or timing shells.
- Unclocked feedback loops.

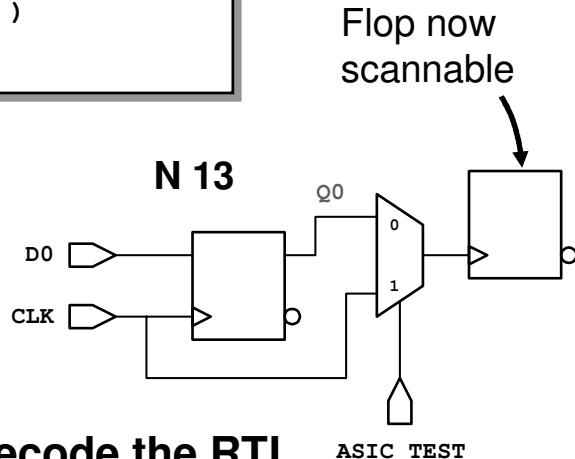
Tristate Contention:

- Potential contention between tristate drivers.

# Ripple-Counter RTL DFT Solution

```
// Corrected RTL Fragment
always @(posedge CLK)
    Q0 <= D0;
assign CLK_Q0 = ASIC_TEST ? CLK : Q0;
always @(posedge CLK_Q0)
    Q1 <= D1;
```

**Violation Corrected  
in RTL Code**



**Another solution is to recode the RTL  
to describe a synchronous counter**

**5-10**

This tool does *not* advise you on how to recode the RTL—it only *flags* the violating code. Most violation codes, such as **D1**, have man pages suggesting what to do next. In this case, one intuitive solution is to **bypass** the **Q0** clocking signal during testing. An even better alternative is to redesign the counter RTL to be fully **synchronous**.

# DFT DRC on RTL? Use Text Reporting

```
dft_drc
  Loading test protocol
  Loading design 'RISC_CORE'
  Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
  ...checking vector rules...
  ...checking pre-dft rules...

-----
Begin Pre-DFT violations...
  Warning: Clock input clocked_on of DFF
I_ALU/Carry_Flag_reg(/RISC_CORE/ALU.vhd 40) was not controlled. (D1-1)
Information: There are 309 other cells with the same violation. (TEST-171)
  Warning: Reset input clear of DFF
I_ALU/Carry_Flag_reg(/rtl/RISC_CORE/ALU.vhd 40) was not controlled. (D3-1)
Information: There are 89 other cells with the same violation. (TEST-171)
Source Code
File and Line
Pre-DFT violations completed...

-----
400 PRE-DFT VIOLATIONS
  310 Uncontrollable clock input of flip-flop violations (D1)
  90 DFF set/reset line not controlled violations (D3)
```

5-11

**dft\_drc** does *not* operate directly on the *text* of your HDL source code! It checks the elaborated **GTECH** representation of your design after read into **dc\_shell**. In effect, the tool uses HDL Compiler as a preprocessor to translate text to topology. This pre-synthesis focus allows **rapid iterations** and **large capacity** for SoC designs. You *can* run **dft\_drc** on **mapped** logic, or a mix of mapped and unmapped logic. **dft\_drc** is also done at a gate-level where it performs **gate-level** DRC checks.

# Example: RTL DFT DRC

```
# Enable HDL source file & line-number info.  
set hdlin_enable_rtldrc_info true ← Enable Filename and Line  
Number Tracking  
  
# Read in RTL Source Code:  
# Must be register-transfer-level HDL code  
# Could use analyze/elaborate or read_vhdl/read_verilog  
set acs_hdl_source "../rtl/vhdl"  
acs_read_hdl -f vhdl RISC_CORE ← Read RTL Source Code  
  
# DEFINE TEST-PROTOCOL CLOCKS & CONTROLS:  
# Specify existing inputs that control scan.  
set_dft_signal -view exist -type ScanClock -timing {45 55} -port clk  
set_dft_signal -view exist -type Reset -active_state 0 -port rst_n  
  
# DEFINE TEST-PROTOCOL HOLDS:  
# Specify ASIC_TEST input held at constant 1.  
set_dft_signal -view exist -type Constant -active_state 1 -port TEST_MODE  
  
# Create the Test Protocol:  
create_test_protocol -capture_procedure multi_clock ← Create Test Protocol  
  
# RUN RTL DFT DRC:  
dft_drc ← Invoke RTL DFT DRC
```

5-12

Code in blue indicates commands, variables, and options specific to **dft\_drc**. Other syntax, such as design-specific names and user choices, are shown in black.

For full details on commands, see the man pages.

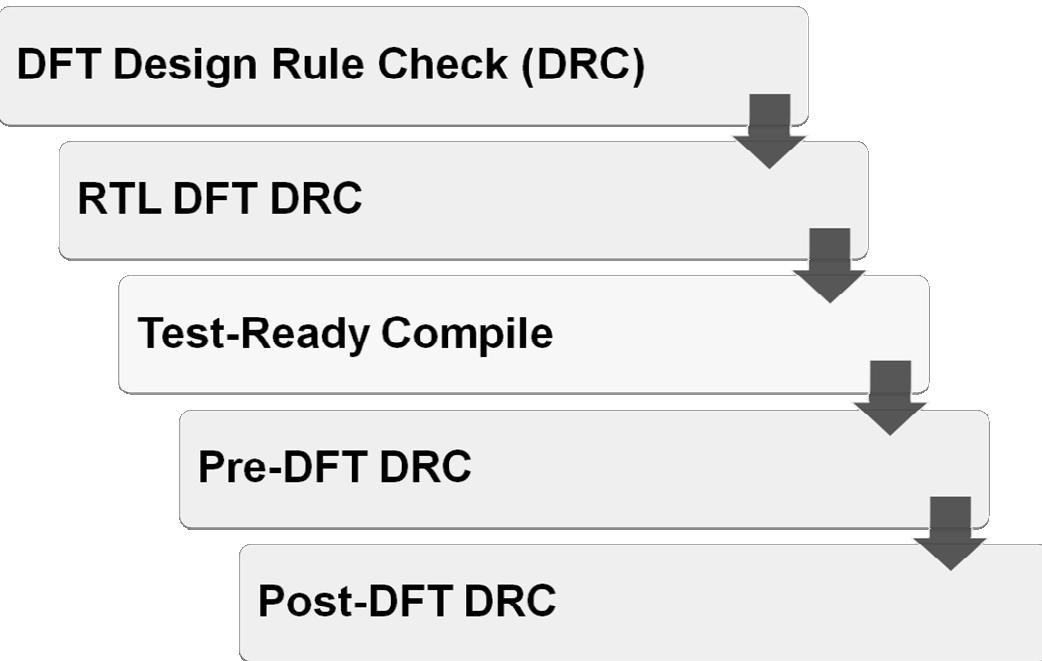
## A few key insights:

**hdlin\_enable\_rtldrc\_info:**

A **true** value tells (V)HDL Compiler to track HDL code **filenames** and **line numbers**.

This is a convenience in correlating DRC violations to specific lines of source code.

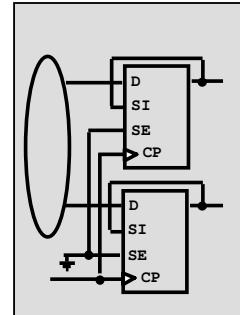
# DFT Design Rule Check: Agenda



5-13

# Scan Style

- The scan style tells DFTC what type of scan-equivalent flip-flop to use during test-ready compile
- Your choice of style must be supported by vendor library



## Chip-Wide Default Scan Style (.synopsys\_dc.setup):

```
set test_default_scan_style multiplexed_flip_flop
```

5-14

A consistent way to set the scan style is to use the `test_default_scan_style` variable shown above. Include this line in `.synopsys_dc.setup` to set the style as a design-group standard.

Another effective way to set scan style is: `set_scan_configuration -style`. This overrides test-default variable settings—but applies only to the current design.

Other common scan-style choices include: `clocked_scan`, `lssd`, and `combinational`.

Be sure to check with your ASIC vendor in advance to verify which styles are supported.

Specify one global scan style for the entire chip—interfacing multiple styles is difficult.

The scan style must be established prior to running a “Test-Ready Compile” (`compile -scan`)

# Test-Ready Compile

## What Is It?

This DFTC feature synthesizes all registers out of **scan flip-flops** instead of regular flip-flops

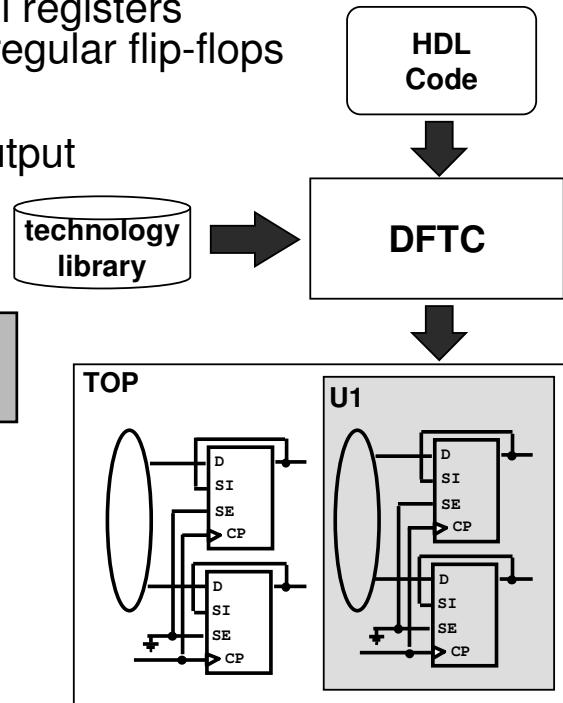
## Why Do It?

Scan flip-flop area, timing, and output loading are **taken into account** during synthesis

```
dc_shell> compile -scan  
dc_shell> compile_ultra -scan
```

All **unused** pins are **degenerated**:

- **SE** input pin is **grounded**
- **SI** input pin **driven** by **Q**
- **Q** output pin **loaded** by **SI**



5-15

Scan-ready synthesis virtually eliminates the need for post-insertion **reoptimization**.

Compile option **-scan** tells DFTC to use **scan-equivalent** flip-flops from the outset; thus each flip-flop has the area, timing, and power characteristics of a scan flip-flop. This eliminates the usual flip-flop **replacement** step during **traditional** scan insertion. If scan insertion does little or no replacing, there is **no impact** on area, timing, and power. This is called test-ready compile.

After test-ready compile, all flip-flops are degenerated scan-equivalent flip-flops. **Degenerated** means that unused test pins, such as **SE**, are temporarily tied off.

# Design Compiler Ultra

- DC Ultra (`compile_ultra`) is an add-on to DC Expert (`compile`) that enables several additional features required for high-performance ASIC or IC design
- DC Ultra is required for running DC in topographical mode
- One of the additional features in DC Ultra is Automatic Shift Register Identification

5-16

# Automatic Shift Register Identification

- Automatic shift register Identification is only performed by DC Ultra (`compile_ultra`)
- It will recognize existing functional shift registers in the design
- When running Test-Ready compile (`compile_ultra -scan`), only the first register in the shift register will be replaced with a scan equivalent register
- This will improve timing and area QOR

5-17

# Shift Register Identification Control

- To verify that this feature is enabled, look in the log file for the following informational message

Information: Automatic shift-register identification is enabled for scan. Not all registers will be scan-replaced (OPT-467)

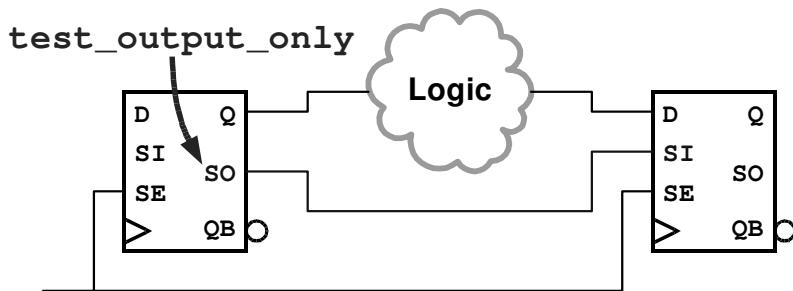
- To disable this optimization

```
set compile_seqmap_identify_shift_registers false
```

5-18

# Using Dedicated Scan Pins

- Vendors may offer scan flops with dedicated SO pins
- This supports capacitive load reduction on the Q pin
- Can be designed slower, to minimize hold violations
- Target-library cell description must have pin attribute:  
`test_output_only: true`
- Attribute is recognized by `compile -scan`, etc



5-19

Some ASIC vendors offer scan flip-flops with an output pin **dedicated** for scan-out. This **SO** pin off-loads the **Q** pin, and its timing minimizes problems along scan paths. Only the vendor can declare a dedicated scan-out pin—it requires cell-library support. The cell must have a **test\_output\_only** attribute set in the vendor's target library. By default, various DFTC commands recognize this attribute and use it accordingly; thus, **compile -scan** reserves these pins for scan—not functional—connections.

Ensure that **compile\_dont\_use\_dedicated\_scanout** is set to 1 for default behavior.

# Test-Ready Compile

- Libraries that have only scannable flip-flops and no normal flip-flops are called “Scan-Only” libraries
- For “Scan-Only” libraries, it is still required to use `compile/compile_ultra -scan` to perform a Test-Ready compile
- After initial scan replacement during Test-Ready compile, the `-scan` option must be included on all subsequent incremental compiles
  - Example

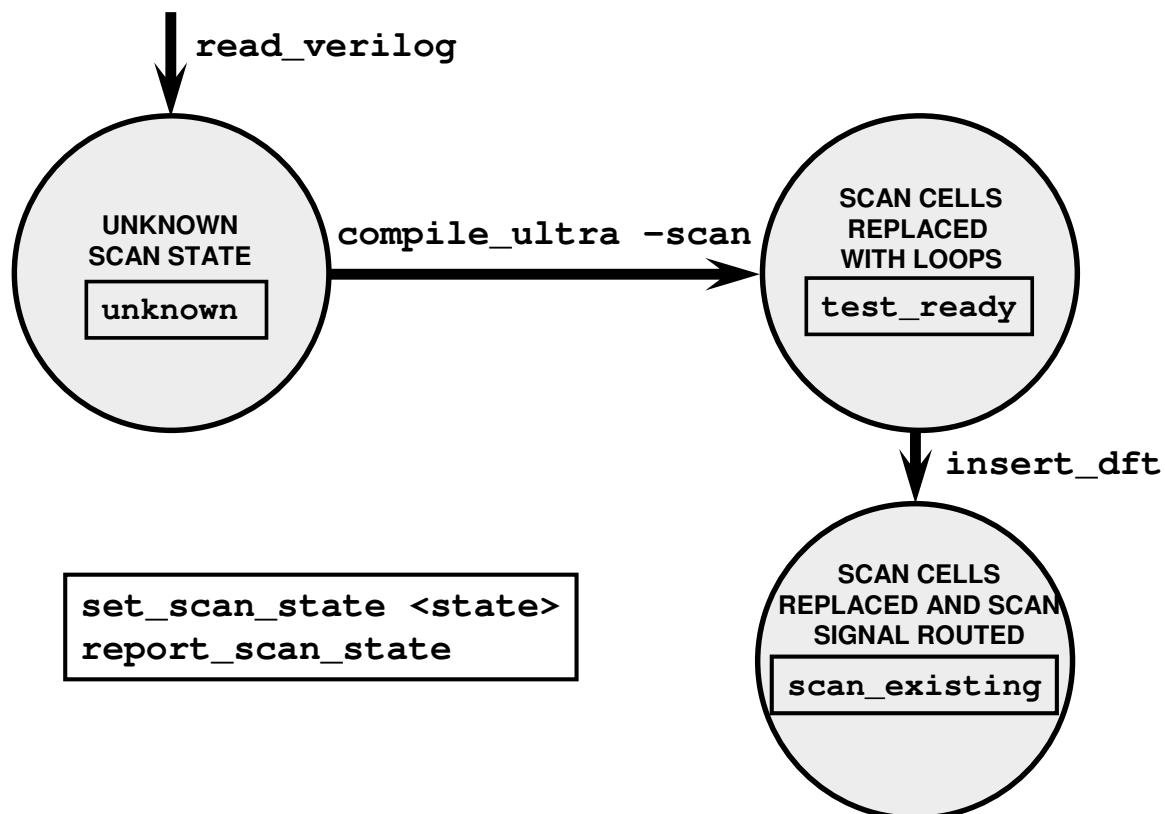
```
compile -incremental -scan
compile_ultra -incremental -scan
```

5-20

Scan replacement can be manually guided with the `set_scan_replacement` command. In the vast majority of cases, `compile -scan` is sufficient and no guidance is required.

Scan replacement can also be done by `insert_dft` if the netlist read into DFT Compiler is not “Test-Ready”. Scan replacement by `insert_dft` is not done if the design was already scan replaced by “`compile -scan`” and the design is in a “test\_ready” state. Scan replacement in `insert_dft` can be disabled with “`set_scan_configuration -replace false`”.

# DFT Compiler Scan State



5-21

# Manually Setting the Scan State

- **set\_scan\_state <state>**
  - Manually sets the scan state for the *ENTIRE* design
  - Directly affects whether pre-scan or post-scan rules are used with **dft\_drc**
  - Use **set\_scan\_configuration -replace false** to disable scan replacement
- **unknown : all flops are assumed to be non-scan**

**dft\_drc** will run **Pre-DFT** rule checks  
**insert\_dft** will re-insert scan on the design (double scan insertion can occur)  
This is the state of design after **read\_verilog** or **read\_vhdl**
- **test\_ready : all flops are assumed to be scan replaced**

**dft\_drc** will run **Pre-DFT** rule checks  
**insert\_dft** will build scan chains.
- **scan\_existing : entire design is assumed to be scan stitched**

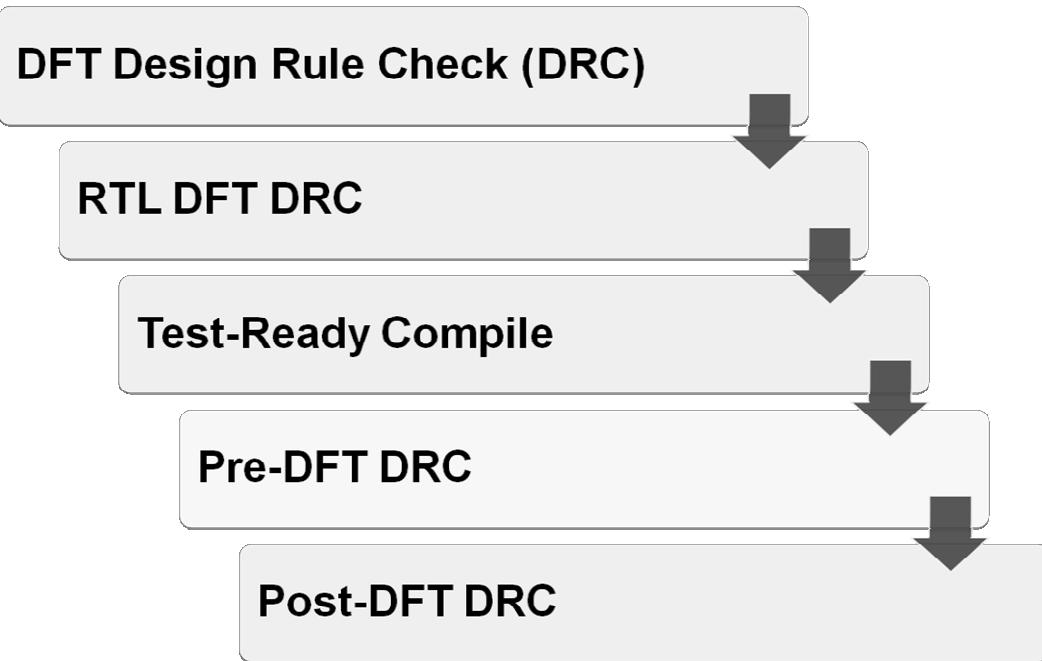
**dft\_drc** will run **Post-DFT** rule checks  
**insert\_dft** will rebuild scan chains. No scan replacement occurs

**Use with  
caution**

**5-22**

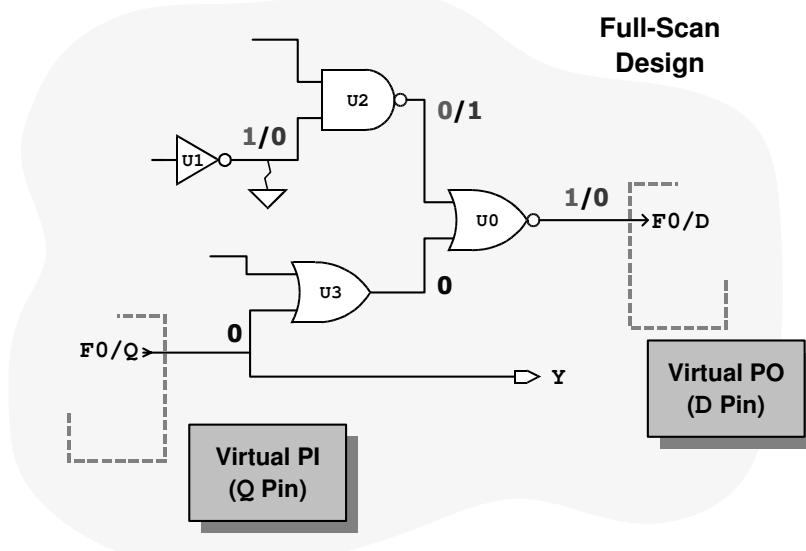
Setting the scan state is sometimes necessary when reading in a “test-ready” gate level netlist (**read\_verilog**). All attributes that were previously applied during **compile -scan** are lost when reading a verilog netlist (as opposed to a **ddc**). In this case, the scan state is manually set as “test ready” with the **set\_scan\_state test\_ready** command prior to running **dft\_drc**.

# DFT Design Rule Check: Agenda



5-23

# Designing for Risk-Free Scan



- Combinational ATPG effectively allow scan flops to be considered as virtual primary inputs and outputs
  - This simplification requires that scan-shift be risk-free!

5- 24

Why is risk-free scan during the loading and unloading of scan chains so important?

For each test pattern, the ATPG tool determines the **serial data** to load into each chain. ATPG tools **do not simulate** scan-shift cycles to ensure that correct data is loaded. They simply *assume* that correct scan data has been loaded after the last scan-in cycle. Timing and structural problems that **invalidate** this assumption lead to failed patterns.

## Conclusion:

- Run a DRC rules check on synthesized logic both **before** and **after** scan-path insertion.
  - Pre-insertion DRC flags any flip-flops that are **noncompliant** with risk-free scan rules.
  - Post-insertion DRC warns you of **inconsistencies** with the scan paths as synthesized.

# Potential Scan-Shift Issues

Design scenarios affecting risk-free scan include:

- No clock pulse at scan flop due to gating of clock
- No clock pulse due to dividing down of the clock
- Unexpected asynchronous reset asserted at flop
- Hold time problem due to short net or clock skew



## DFT Rule of Thumb:

The ATE must **fully** control the clock and asynchronous set and reset lines reaching **all** the scan-path flip-flops

5-25

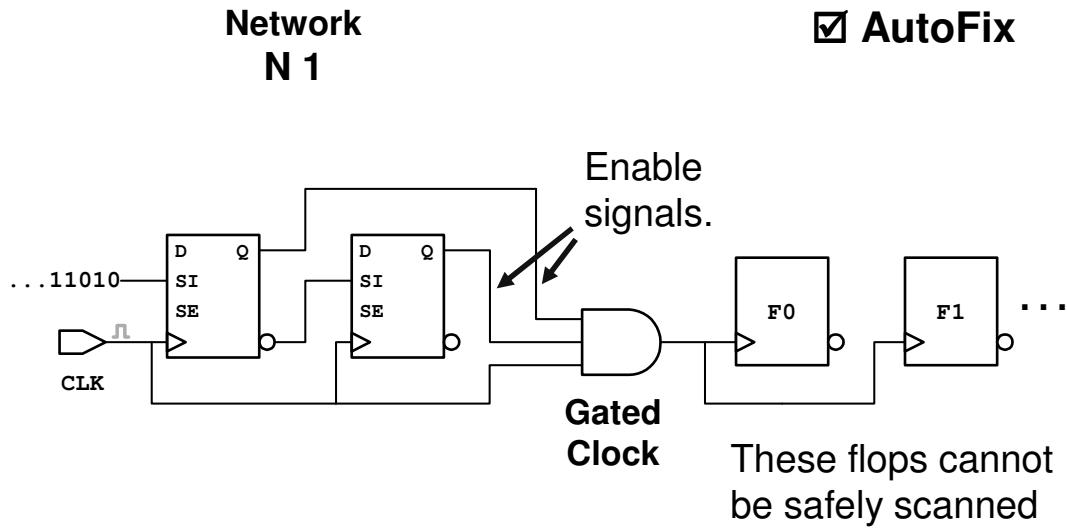
Risk-free scan requires that:

- Every scan flip-flop must receive the synchronizing clock pulses generated by the ATE.
- Clock pulses should not be **blocked** from reaching a scan flop due to **gating** of the clock.
- Clock edges must arrive regularly at scan flops, without being **divided** or **multiplied**.
- No scan flops are unexpectedly **set** or **reset** asynchronously during the scan-shift process.
- No **hold violations** occur at scan flops, due to clock skew or short **Q-to-SI** interconnects.

Conclusion:

- Run the recommended DRC checks **early and often** to identify and fix most problems.
- Exception: scan-path **hold time** problems *cannot* be detected by DRC rules checking.

# Gated-Clock DRC Violation



- Clock pulses can be inhibited from reaching F0
- This occurs when any of the enable signals is 0
- This will exclude F0, F1, ... from any scan path

5-26

Risk-free scan-shifting requires that **every** scan flop be **clocked** during **each** scan cycle. Network N 1 is a classic DRC violation caused by **gating the clock** to flops **F0, F1**, etc. Clock gating may be essential for low-power design, but it impacts design testability. The AND gate can **inhibit arrival** of a clock pulse at **F0**, unless all enable lines are high. Note that N 1 is only in violation for high-going (RTZ) clock pulses, not low-going (RT1). Unless this violation is fixed, the affected flip-flops will be **excluded** from any scan path. If *many* flops are excluded, the impact of this DRC violation is a **major** loss of coverage. The DFTC warning for this class of violation is **D1**, “clock pin not controlled.”

## AutoFix DFT:

The **checkbox** on this slide indicates that this kind of violation is corrected by **AutoFix**. This violation has several solutions, each a tradeoff between DFT effort and coverage.

# What DFTC Reports Pre-DFT

This gate-level DRC check is done in DFTC

```
dc_shell> dft_drc
Loading test protocol
Loading target library 'cb13fs120_tsmc_max'
Loading design 'RISC_CORE'
Pre-DFT DRC enabled
...
-----
Begin Pre-DFT violations...
Warning: Clock input CP of DFF I_ALU/Neg_Flag_reg was not controlled. (D1-1)
Information: There are 309 other cells with the same violation. (TEST-171)
Warning: Set input CDN of DFF I_ALU/Neg_Flag_reg was not controlled. (D2-1)
Information: There are 82 other cells with the same violation. (TEST-171)
Warning: Reset input CDN of DFF I_CONTROL/UseData_Imm_Or_RegB_reg was not controlled. (D3-1)
Information: There are 6 other cells with the same violation. (TEST-171)
Pre-DFT violations completed...
```

Pre-DFT  
Violation  
Code

Focus is on pre- and post-insertion scan compliance

5-27

- A **scan-compliant** flip-flop is one that functions correctly during scan-shift without risk.
- Checking compliance is important, since ATPG algorithms **assume** working scan chains.
- Noncompliant flip-flops are **excluded** from all scan chains, and may impact coverage.

For example:

A flip-flop whose clock pin is held **constant** during scan is noncompliant.

You will also use **AutoFix** to correct noncompliant logic automatically.

# Enhanced DFT DRC Reporting

- **Why Enhance DFT DRC reporting?**
  - The default report provides a lot of information but it does not separate it into useful categories (i.e. those requiring attention vs. informational)
  - *Enabled by a variable and OFF by default in 2007.12*
- **The new DFT DRC violation summary is divided into 3 main categories:**
  1. Modeling and user constraints that will prevent scan insertion
  2. DRC violations which will prevent scan insertion
  3. DRC Violations which can affect ATPG coverage
- **A summary including the breakdown of all sequential cells is provided at the end**

5-28

# Enabling Enhanced DFT DRC Reporting

## ■ To Enable Enhanced DFT DRC Reporting

- `set test_disable_enhanced_dft_drc_reporting FALSE`
- The variable is set to TRUE by default which means that the enhanced dft\_drc enhanced reporting is not enabled by default
- `dft_drc` cleans the violation information at the end of the command; no persistency
- Use `set trep_persistency TRUE` to keep the violation information in memory

## ■ Usage:

```
set test_disable_enhanced_dft_drc_reporting FALSE
set trep_persistence TRUE
dft_drc
report_dft_drc_violations
```

5-29

# Example: Enhanced DFT DRC Report (1/2)

```
dc_shell> dft_drc
In mode: all_dft...
Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
  Loading test protocol
    ...basic checks...
    ...basic sequential cell checks...
      ...checking for scan equivalents...
      ...checking vector rules...
      ...checking pre-dft rules...

-----
Modeling Violations and User Constraints preventing scan insertion
-----
Warning: Cell I_CLOCK_GEN/I_CLKMUL (CLKMUL) is unknown (black box) because functionality
        for output pin CLK_1X is bad or incomplete. (TEST-451)
Information: There are 12 other cells with the same violation. (TEST-171)

-----
DRC Violations which will prevent scan insertion
-----
Warning: Clock input EN of DLAT I_ORCA_TOP/I_BLENDER/latched_clk_en_reg was not
        controlled. (D4-1)

-----
Other Violations
-----
Warning: Three-state net I_ORCA_TOP/I_PCI_READ_FIFO/data_out_fifo[31] is not properly
        driven. (TEST-115)
Information: There are 431 other nets with the same violation. (TEST-289)
```

5-30

# Example: Enhanced DFT DRC Report (2/2)

...

Summary of all DFT DRC violations

13 MODELING VIOLATIONS AND USER CONSTRAINTS PREVENTING SCAN INSERTION  
13 Cell has unknown model violations (TEST-451)

1 DRC VIOLATIONS WHICH WILL PREVENT SCAN INSERTION  
1 DLAT clock line not controlled violation (D4)

432 OTHER VIOLATIONS  
432 Improperly driven three-state net violations (TEST-115)

446 TOTAL VIOLATIONS

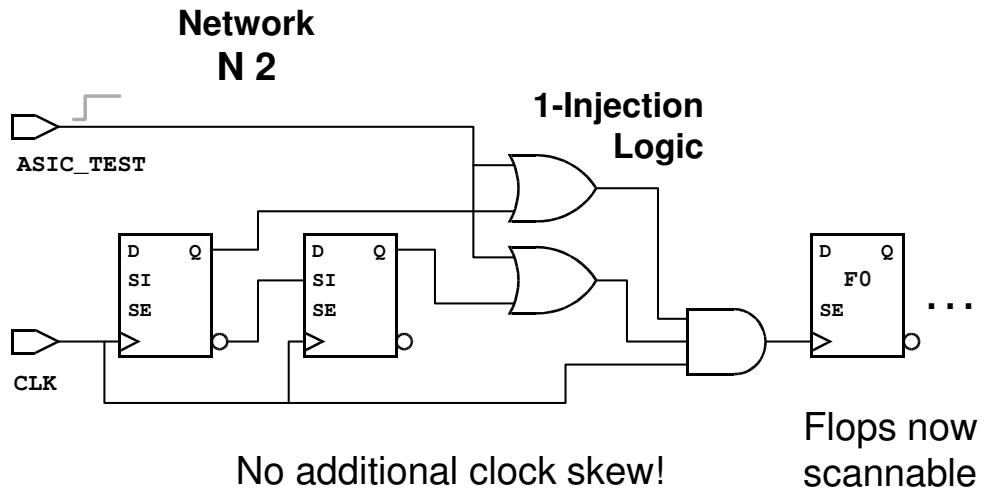
Warning: Violations occurred during test design rule checking. (TEST-124)

Sequential Cell Report:	Sequential Cells	Core Segments	Core Segment Cells
Sequential Elements Detected:	2958	0	0
Clock Gating Cells	: 0		
Synchronizing Cells	: 0		
Non-Scan Elements	: 0		
Excluded Scan Elements	: 0	0	0
Violated Scan Elements	: 33	0	0
(Traced) Scan Elements	: 2925 ( 98.9%)	0	0 ( 0.0%)

Information: Test design rule checking completed. (TEST-123)  
1

5-31

# Gated-Clock DRC Solution



- In normal operation, the added OR logic is transparent
  - During test, scan and capture clocks always reach  $F_0$
  - Flip-flops  $F_0$ , etc., can now be included in a scan path

5-32

This solution adds **1**-injection logic (an **OR** gate) to prevent the inhibiting of clock pulses. The best way to add this injection logic is by editing and resynthesizing your HDL code. In general, it is best to implement testability as **early** in the SOC design flow as possible. Use the early-warning capability of RTL DFT DRC to alert you to clock gating in the code. Then modify the code, describing the injection logic in technology-independent manner.

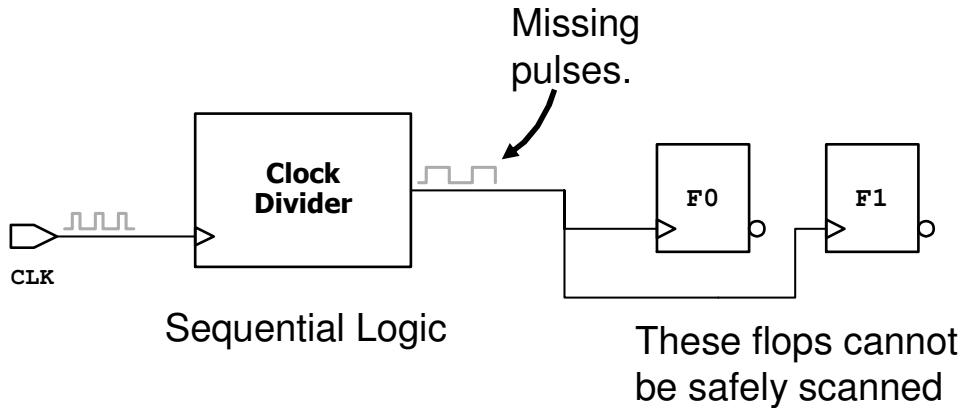
For example:

In Verilog: **assign CLK\_F0 = CLK & (ASIC\_TEST|EN[0]) & (...);**

# Clock-Divider Violation

Network  
N 3

AutoFix



- Not every test-clock pulse will reach **F0**, **F1**, ....
- All such flip-flops excluded from any scan path
- Inserting injection logic will not help in this case

5-33

The clock **divider** (or **multiplier**) is in general an arbitrary sequential circuit or FSM. Because of it, some clock pulses generated by the ATE might not reach flops **F0**, **F1**, ...

Flip-Flops that lack **ATE-controllable** clocking must be excluded from any scan path. If *many* flops are excluded, the impact of this DRC violation is a **major** loss of coverage. One DFTC warning for this kind of violation is **D1** (see the man page for details).

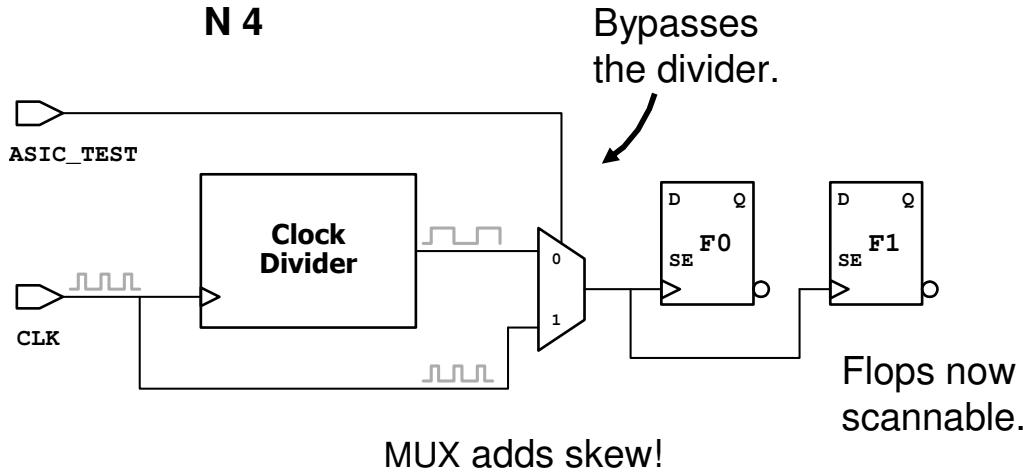
## AutoFix DFT:

The **checkbox** on this slide indicates that this kind of violation is corrected by **AutoFix**. Often, code cannot be edited due to RTL sign-off or lack of knowledge of a legacy design. AutoFix is a **fast workaround** which adds all the correction logic during scan insertion. The AutoFix flow does away with hacking and incrementally recompiling the netlist. Using AutoFix, added logic is **optimized** to meet both synthesis goals and design rules.

# Clock-Divider Solution

Network

N 4



- Added MUX bypasses clock divider during test
- Flops F0, F1, ... are now included in scan chain
- Clock skew due to the MUX introduces risk

5-34

The only solution is to **bypass** the divider with a MUX to get the ATE clock to all flops. The **risk** is that added clock skew may cause **hold time** violations at flops **F0**, **F1**, etc. Clock-tree synthesis from the **MUX** output pin will avoid such timing hazards. Optionally, you can move the **MUX** *into* the clock divider, separate from the clock tree. Use one of the recommended methods below to fix this class of clock DRC violations:

#### HDL Code:

This is the **preferred** method, since it puts design for testability right into the HDL code.

Modify the code, describing the multiplexer in a technology-independent manner.

For example:

In Verilog: `assign CLK_F0 = ASIC_TEST ? CLK : CLK_DIV;`

#### AutoFix DFT:

Use **AutoFix** to insert the MUX, after specifying the name of an ATE-controllable clock. By default, **AutoFix** will correct both **clock-gating** and **uncontrollable-clock** violations.

AutoFix uses a **bypass MUX**, as shown in N 4, to fix **both** of the scenarios N 1 and N 3.

# DFT DRC: What are D rules?

- D rules are a category of rules that are only checked during Pre-DFT (before scan chains are built)

Scan state:	Rules checked:
unknown	Pre-DFT rules (D, L)
test-ready	
scan_existing	Post-DFT rules (C, S, X, Z, L, R, V)

5-35

DFT DRC Rule Types:

D – DFT Rules

L – BIST Rules

S – Scan Chain Rules

C – Clock Rules

X – X-state Rules (Combinational Feedback)

Z – Tristate Rules (Contention)

R – Compression Rules

V – Vector Rules

# DFT DRC: List of D rules

Rule	Post DFT Equivalent	Description	Impact
D1	$\sim C2$	DFF clock input not controlled	Prevents SCAN
D2	$\sim C2$	DFF set input not controlled	Prevents SCAN
D3	$\sim C2$	DFF reset input not controlled	Prevents SCAN
D4	(None)	DLAT clock input not controlled	Prevents SCAN
D5	(None)	DLAT set input not controlled	Prevents SCAN
D6	(None)	DLAT reset input not controlled	Prevents SCAN
D7	$\sim C20$	RAM write input not controlled	ATPG info
D8	C4	{Clock/set/reset} unable to capture	ATPG info
D9	(New)	Clock port not active when clocks set to on	Prevents SCAN
D10	$\sim C26$	Clock feeding data input	ATPG info
D11	$\sim C11, \sim C12, \sim C13$	Clock feeding both clock and data input	Possible race during ATPG
D12	C14	Clock feeding multiple clock/set/reset inputs	Prevents SCAN
D13	C5	Data for LS clock/write input affected by new capture	Requires sequential ATPG
D14	C6	Data for TE clock/write input affected by new capture	Requires sequential ATPG
D15	C8	LS clock/write/set/reset input affected by new capture	Possible race during ATPG
D16	C9	TE clock/write input affected by new capture	Possible race during ATPG
D17	C16	Clock port not capable of capturing data	Prevents SCAN
D20	Z1	Bus gate capable of contention	ATPG info
D21	Z2	Bus capable of holding Z state	ATPG info
D22	Z3	Wire gate capable of contention	ATPG info
D23	X1	Sensitizable feedback path	ATPG info

5-36

# Why Check for DRC Violations?

- Highly-pushbutton structured DFT depends on checking all designs for compliance with a standard set of rules
- DRC violations can tell you where to insert ad hoc DFT
- DRC violations can prevent “scannable” registers from being included on the scan chains
- Many DRC violations hinder test-pattern generation



When is gate-level checking done?

- After DFTC Test-Ready Compile
- Before running TetraMAX ATPG

5-37

# Limitations of DFT DRC Checking

**Do not rely blindly on DFT DRC tools alone!**

- They cannot take gate or net delays into account
- They are unaware of clock-tree delays or skew
- Test tools do not perform static timing analysis



## **What Synopsys ACs Recommend:**

Static timing analysis is a **must** to verify timing in both test and functional modes. Plan on **simulating** all test patterns, written out from TetraMAX in Verilog or VHDL format. Use gate-level timing models to detect hazards.

**5-38**

Synopsys recommends simulating the **entire** test-pattern set on a Verilog or VHDL simulator. It is the only way to spot hold violations and other hazards not detectable during DRC. In TetraMAX, use **write\_patterns** to write the pattern set in a format that can be simulated.



For further reading, refer to the “Test Pattern Validation User Guide” which is part of the Test Automation collection on SolvNet and/or SOLD.

# Libraries and DFT DRC

- When running DFT DRC, DFTC *automatically* translates cell function descriptions from .db libraries into a Verilog library that TetraMAX can use for DFT DRC
- If there is no valid “function” specified in the library .db, DFT DRC will treat that cell as a “black box”

```
cell (NAND2XL) {  
    cell_footprint : nand2;  
    pin(A) {  
        direction : input;  
    }  
    pin(B) {  
        direction : input;  
    }  
    pin(Y) {  
        direction : output;  
        function : "(! (A B))";  
    }  
    ...
```



```
`celldefine  
module NAND2XL (A,B,Z);  
    input A;  
    input B;  
    output Z;  
    or(Z, _3, _4);  
    not(_3, A);  
    not(_4, B);  
endmodule  
`endcelldefine
```

5-39

# User Defined Simulation Libraries

- You can set the `test_simulation_library` variable to point to a user defined simulation library (if available):

```
set test_simulation_library <path_to_Verilog_lib>
```

- User specified libraries are useful for black-boxes such as memories, pads, or anything for which the user wants to specify Verilog structural functionality
- For more details, see the following SolvNet article:  
<https://solvnet.synopsys.com/retrieve/008685.html>

5-40

# Controlling Internal Nodes

- Some designs have internal nodes that must be initialized for a design to pass `dft_drc`
- Can be resolved using
  - Custom initialization sequence in the test protocol (`test_setup` macro)
  - “Internal pin” specifications where a DFT signal is declared at an internal pin with `set_dft_signal`
  - Internal node assumptions declared with `set_test_assume`
- The recommended method is the custom initialization sequence

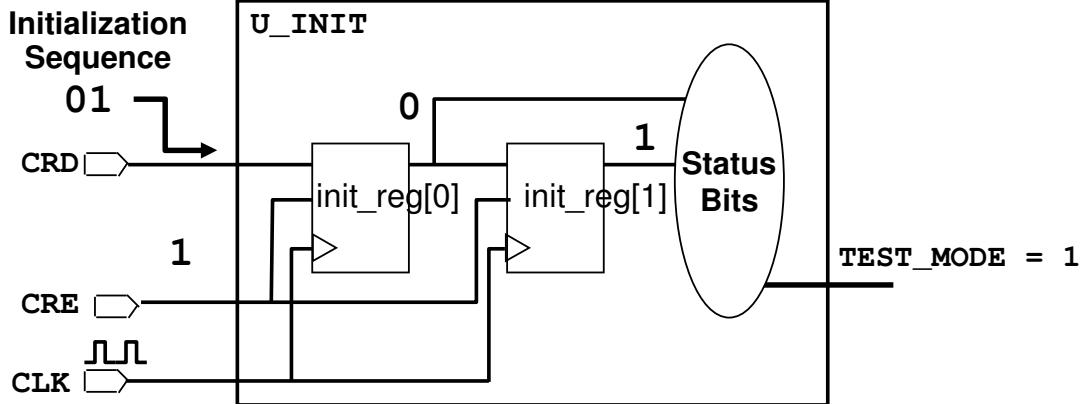
5-41

# Using Custom Initialization Sequence

- The initialization sequence is applied in the `test_setup` macro

```
read_test_protocol \
-section test_setup
  init.spf
create_test_protocol
dft_drc
```

```
MacroDefs {
  "test_setup" {
    V {"CRD"=1; "CRE"=1; "CLK"=P;}
    V {"CRD"=0; }
    V {"CRE"=0; "CLK"=0; }
  }
}
```



5-42

# Example Initialization Protocol Flow

```
dc_shell> dft_drc
  Loading test protocol
  Loading design 'ORCA'
  Pre-DFT DRC enabled
...
4017 PRE-DFT VIOLATIONS
  2832 Uncontrollable clock input of flip-flop violations (D1)
  1133 DFF set/reset line not controlled violations (D3)

-----
Customize initialization; Read protocol back in
-----

dft_drc
  Loading test protocol
  Loading design 'ORCA'
  Pre-DFT DRC enabled
...
Begin Other violations...
Warning: Cell U_INIT/init_reg[0] (/rtl/INIT.vhd 39) has constant 0 value. (TEST-504)
Warning: Cell U_INIT/init_reg[1] (/rtl/INIT.vhd 39) has constant 1 value. (TEST-505)
...
95 PRE-DFT VIOLATIONS
  ...D1 and D3 violations are now gone

2 OTHER VIOLATIONS
  1 Cell is constant 0 violation (TEST-504)
  1 Cell is constant 1 violations (TEST-505)
```

Original  
Test Protocol

Huge # of  
Clock  
and Reset  
Problems

Customized  
Test Protocol

**TEST-504/505  
Violations Consistent  
with “0-1” Init  
Sequence**

**5-43**

Before and after use of a custom initialization sequence.

Note the DRC messages concerning the constant value flops caused by the initialization sequence (**TEST-504/TEST-505**). This is a indication that the initialization sequence was applied properly.

# Using Internal Pins to Bypass Initialization

- Internal pins support must first be enabled

```
set_dft_drc_configuration -internal_pins enable
```

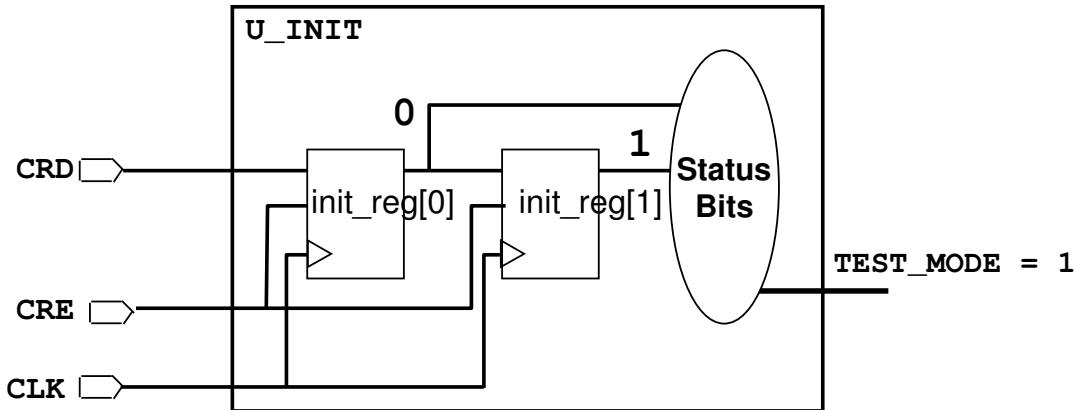
- Internal pins are specified with set\_dft\_signal

```
set_dft_signal -v existing_dft -type Constant \
```

```
    -hookup_pin init_reg[0]/Q -active_state 0
```

```
set_dft_signal -v existing_dft -type Constant \
```

```
    -hookup_pin init_reg[1]/Q -active_state 1
```



**5-44**

Note: Using an “internal pins” specification in this way effectively bypasses the proper initialization sequence.

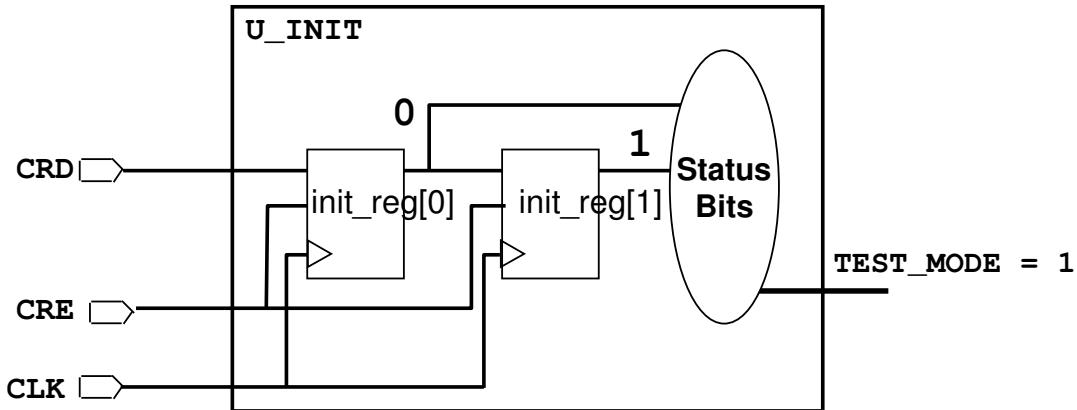
More detail on “internal pins” can be found in the Appendix

# Using set\_test\_assume to Bypass Initialization

- The `set_test_assume` command can be used to set a logic value on an internal node of a design

```
set_test_assume 0 init_reg[0]/Q  
set_test_assume 1 init_reg[1]/Q
```

- Use `report_test_assume` for reporting and `remove_test_assume` removing assumed signals



5-45

Note: Using a `set_test_assume` specification in this way effectively bypasses the proper initialization sequence.

# Protocol File Warning

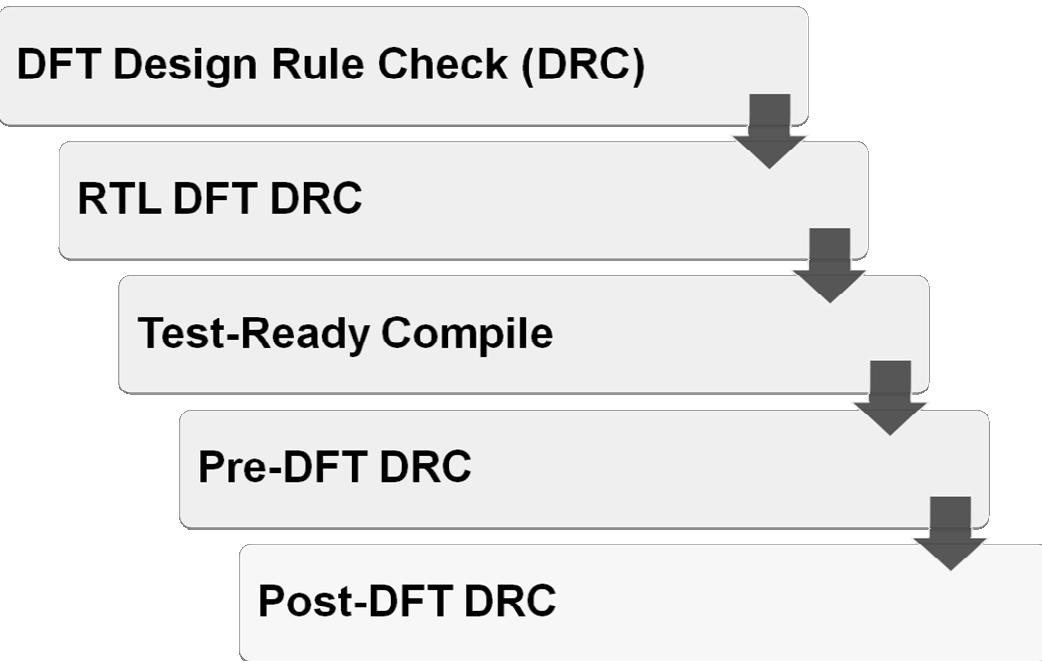
- A test protocol written out when “internal pins” are employed cannot be directly used in TetraMAX
  - User needs to modify the protocol file to specify the relevant top level ports, constraints, or initialization sequence required to enable top level controllability
  - The same warning applies to protocols where “`set_test_assume`” was used for `dft_drc`
  - Warning message will be reported for `insert_dft` and `write_test_protocol` commands when internal pins are enabled



**Warning: Protocol generated after insertion in Internal Pin Flow is not accurate and can not be used. (TESTXG-53)**

**5-46**

# DFT Design Rule Check: Agenda



5-47

# What DFTC Reports Post-DFT

This gate-level DRC check is done in DFTC after scan insertion:

```
dc_shell> dft_drc -coverage_estimate
  Loading test protocol
  Loading design 'RISC_CORE'
  State of design is scan existing
  State of design is scan routed
  Post-DFT DRC enabled
  ...
  Begin Clock violations...
    Warning: Clock PIs off did not force off clock input CDN
            of nonscan DFF I_ALU/Neg_Flag_reg. (C2-1)
  Information: There are 310 other cells with the same violation. (TEST-171)
  ...
  Traces Scan
  Chains
  Begin Scan chain violations...
    Warning: Nonscan DFF I_ALU/Neg_Flag_reg disturbed during
            time 0 of load_unload procedure. (S19-1)
  Information: There are 310 other cells with the same violation. (TEST-171)
```

TetraMAX  
Violation  
Code

Traces Scan  
Chains

5-48

A **scan-compliant** flip-flop is one that functions correctly during scan-shift without risk. Checking compliance is important, since ATPG algorithms **assume** working scan chains. Noncompliant flip-flops are **excluded** from all scan chains, and may impact coverage.

For example:

A flip-flop whose clock pin is held **constant** during scan is noncompliant.  
You can also use **AutoFix** to correct noncompliant logic automatically.

# What Are Capture Violations?

**Post-DFT DRC Report:  
TetraMAX C rules;  
Clock or “Capture”**

DRC Report

95 CLOCK VIOLATIONS

```
36 Trailing edge port captured data affected by new capture violations (C6)
16 Leading edge port captured data affected by clock violations (C12)
16 Trailing edge port captured data affected by clock violations (C13)
1 Clock connected to primary output violation (C17)
16 Clock connected to non-contention-free BUS violations (C19)
10 RAM port unable to capture violations (C21)
```

53 SCAN CHAIN VIOLATIONS

```
42 Nonscan cell disturb violations (S19)
1 Multiply clocked scan chain violation (S22)
10 Unstable RAM during test procedure operation violations (S30)
```

...

**To understand capture warnings, recall that:**

- **Last scan-in cycle is completed, and SE is de-asserted**
- **Entire DUT is now in a known state, with all PIs applied**
- **Target fault effect ready to be clocked in to a scan flop**

**5-49**

Capture violations should *not* be ignored—they can mean **failed patterns** in simulation. Using **dft\_drc**, you can detect these violations as early as the HDL coding phase.

It is important to realize that these DFT violations do *not* occur during scan-shift cycles.

Inserting lock-up latches, or setting **-internal\_clocks**, does *not* address this issue!

# Enhanced DFT DRC Report – Post-DFT

```
dc_shell> dft_drc
. . . Post-DFT DRC enabled . . .

Modeling Violations and User Constraints preventing scan insertion
. . .

DRC Violations which can affect ATPG Coverage
Warning: Clock PIs off failed to allow transparency of nonscan DLAT
  I_ORCA_TOP/I_BLENDER/latched_clk_en_reg. (C3-1)
Warning: Clock sdr_clk connects to LE clock/data inputs CP/D of DFF
  I_ORCA_TOP/I_SDRAM_IF/DQ_in_0_reg_0_. (C12-1)
Information: There are 15 other cells with the same violation. (TEST-171)
. . .

Other Violations
. . .

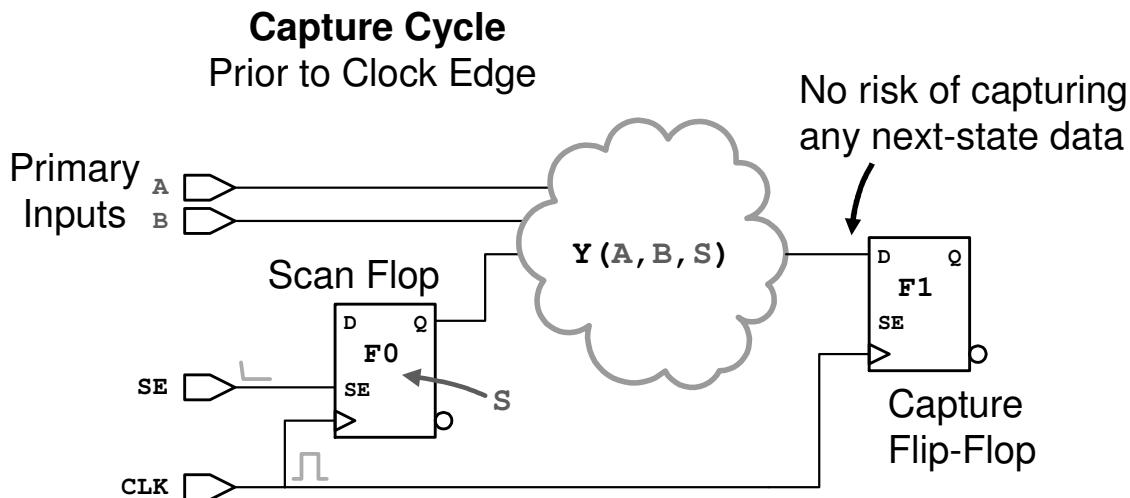
Traced Scan Chains
Chain "chain0" traced through 488 flip-flops
Chain "chain1" traced through 488 flip-flops
Chain "chain2" traced through 488 flip-flops
Chain "chain3" traced through 487 flip-flops
Chain "chain4" traced through 487 flip-flops
Chain "chain5" traced through 487 flip-flops

Summary of all DFT DRC violations
. . .
```

**Traces Scan  
Chains**

**5-50**

# Risk-Free Capture Example



- Just prior to capture, scan flop **F0** loaded with bit **S**
- Intent of capture is to clock in the current state bit **Y**
- Bit **Y** must depend only on scan bit **S** and ports **A, B**

5-51

Why is **risk-free capture** during the exercising of a target fault so important?

Not *all* faults are observed just by strobing a primary output for an expected response.

Many faults are observed by capturing the fault effect in a flop and **scanning it out**.

The fault effect must be **read** by the capture flop **before** scan-flop data is overwritten.

Departing from DRC rules can introduce a risk of capturing **corrupted** (next-state) data.

Using **both edges** of the clock (negedge flops) is an example of introducing risk.

## Caveats:

To avoid clutter, our schematics do **not** explicitly show that **SE** goes to both **F0** and **F1**. **Current state**, refers to the **state of the DUT** after scan loading has been completed. On the other hand, bit **Y** would be called the **next-state** bit in Finite State Machine terminology.

## Conclusion:

Before synthesis, run **dft\_drc** on the **HDL** code—it flags many capture violations.

Prior to **handoff**, run a post-DFT **dft\_drc** to detect any remaining capture issues.

# What Causes Risky Capture?

Risky design practices for capture include:

- Logic that captures on both edges of clock
- Clock also used as input to flop's logic cone
- Asynchronous set or reset used in same way
- Hold time issue due to unexpected clock skew



## DFT Rule of Thumb:

The ASIC must operate correctly in **functional mode**. False, multi-cycle, or out-of-spec paths can cause violations during the capture cycle in test mode.

5-52

Risk-free capture requires that:

The entire chip operates correctly in **functional** mode—that is, whenever **SE** is inactive.

No **hold time violations** occur at capture flip-flops due to clock skew or short data paths.

There are no unoptimized, out-of-spec, or **false paths**, for which hold time was not met.

Be aware that scan data shifted through some designs **can enable** nonfunctional paths!

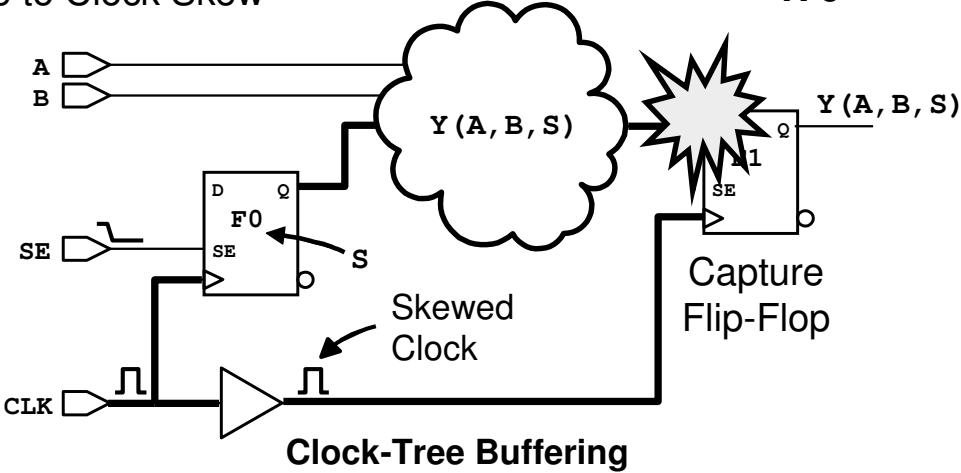
### Conclusion:

Perform DRC rules checking **early and often** in order to identify and fix problems. Remember that **hold time violations** *cannot* be detected by DRC rules checking.

Recommendation: **Simulate** all patterns to spot such hazards before actual testing.

# Capture Problem Due to Skew

Risky Capture  
Due to Clock Skew



Network  
N 9

- Flops **F0** and **F1** are on skewed clock-tree branches
- If the cloud delay is small, **F1** may capture bad data
- This potential hazard is never detected during DRC!

5-53

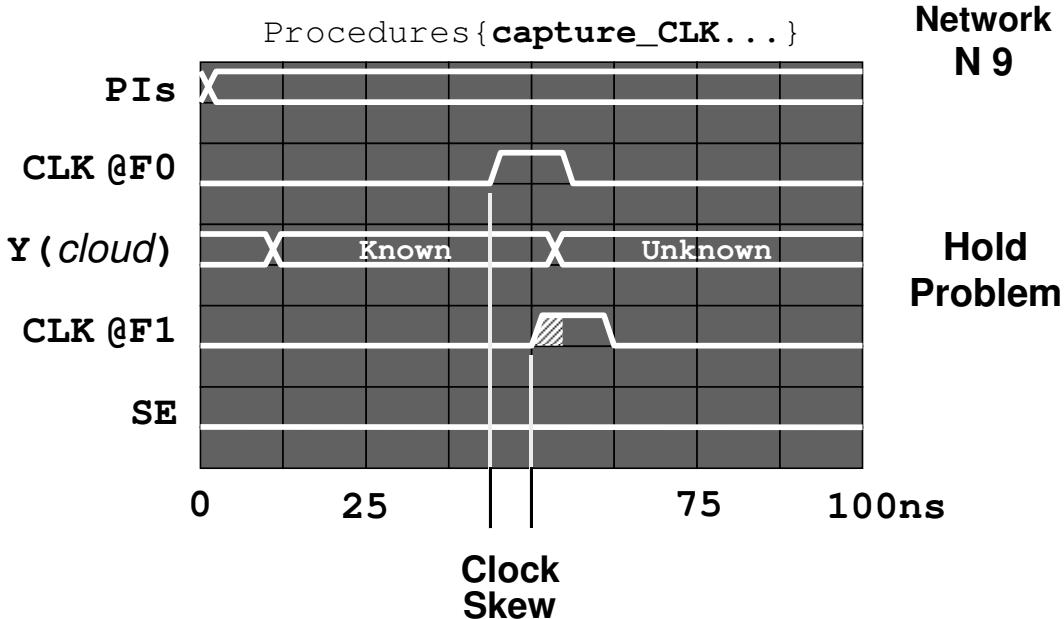
Network N 9 is a possible design scenario that may cause **F1** to capture **corrupt** data. The key point is that DFT checking will **not** warn you of this delay-dependent issue! A related scenario is a design with one clock-tree branch delayed by **test-point** logic. In these scenarios, no issues arise as long as **both** flops are clocked **simultaneously**. Neither is there any clock-capture issue if **no delay path** exists between **F0** and **F1**. Primary inputs **A** and **B**, and scanned-in bit **S**, all reflect the **current state** of the DUT. Under such conditions, **F1** captures the **correct** output  $Y(A, B, S)$  of the logic cloud. Under non-ideal conditions—unexpected clock skew—**next-state** data is captured. This **delay-dependent** potential timing hazard cannot be detected by DFT checking!

## Conclusion:

To avoid failed vectors on the ATE, do **timing simulation** on the **entire set** of patterns or do **static timing analysis** on the chip, with test-specific timing parameters and mode.

Effective cross-chip **clock-skew management** minimizes occurrence of this violation.

# Capture-Problem Timing Details



- For a “thin” logic cloud, output Y may change too soon
- The timing diagram shows how F1 hold time is violated

5-54

The timing diagram shows what can happen in network N 9 if capture occurs **too late**. At the start of this **single-cycle** clock-capture procedure, the primary inputs are applied. Since all flip-flops have just been loaded with scan data, the DUT is in a **known** state. At flip-flop **F0**, there is no problem: at the rising clock edge, known data is captured, but at nearby flip-flop **F1**, the capture-clock edge arrives a little too **late** due to **skew**. Since **F0** was already clocked, new data is potentially propagating through the cloud. The risk is that **Y** may change too soon, **before** the hold time requirement at **F1** is met.

## Conclusion:

For deep-submicron technologies, flip-flop **CLK-to-Q** delay is getting steadily smaller. Skew between branches of clock trees driving thousands of flops can exceed this delay. Effective **clock-skew management** is a key issue not only for design, but for DFT too.

# Managing Clock-Skew Problems

To manage this class of capture problem within DFTC:

1. Use ICC/PrimeTime to identify and fix hold violations, post-layout
2. Avoid false paths that cross between clock domains
3. Allow at most, one active clock per capture procedure
4. Or use dynamic clock grouping in TetraMAX to safely pulse multiple clocks in a capture cycle

These same guidelines apply to asynchronous resets



## Caution:

Hold time violations occur independent of clock period. They can be an issue even for static low-speed testing.

5-55

This slide emphasizes that managing clock skew is a key issue in design-for-test (DFT).

## Hold Methodology:

There is no universal hold time methodology applicable to all SoC designs and flows.

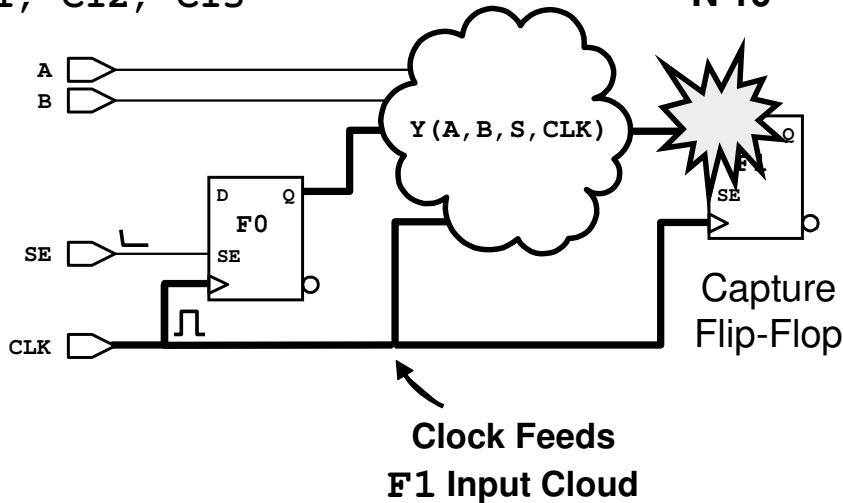
General recommendation is to defer hold time fixing until layout and scan reordering.

In-depth discussion on fixing hold time violations in P&R tools is beyond the scope of this workshop.

# Clock-as-Data Violation

**Risky Capture**  
D11 or C11, C12, C13

**Network**  
N 10



- The clock to F1 also affects its input logic cloud; thus  $Y(A, B, S, CLK)$  is a function of clock level

Does F1 capture  $Y(A, B, S, 0)$  or  $Y(A, B, S, 1)$  ?

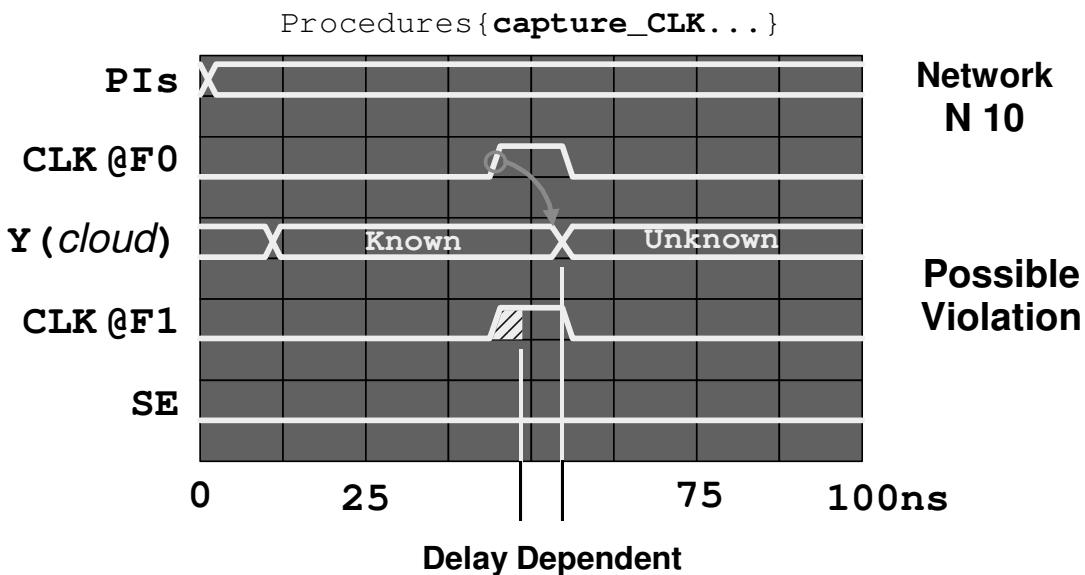
5-56

Network N 10 is a classic DRC violation that may cause F1 to capture corrupt data. Unlike N 9, of which DRC gave no warning, N 10 is flagged as early as RTL `dft_drc`.

The warnings are `D11`, “clock affects both clock and data.”

The diagram shows that this is a **capture-cycle** issue—it occurs when **SE** is **inactive**. This issue does *not* affect scan, and flip-flop F1 can still be **included** in a scan path. As a result, only a few faults in the **neighborhood** of the logic cloud are unobservable. The impact of this DRC violation is **minor**, unless many data pins are in violation.

# Clock-as-Data Timing Details



- Edge of launch clock will affect cloud output **Y**
- After delay-dependent interval (red), **Y** changes
- If interval is too short, **F1** hold time is violated

5-57

The timing diagram shows what can happen in network N 10 as the clock affects **Y**. The waveforms emphasize that a bad capture *may* or *may not* occur on actual silicon! Again, there is no problem at **F0**: at the rising clock edge, known data is captured — but because this clock drives gate-logic, new data can propagate through the cloud.

The risk is that **Y** may change too soon, **before** hold time requirements at **F1** are met. To avoid clutter, the diagram omits a **second** transition of **Y** on the **falling** edge of **CLK**. It depends on gate-level timing—but a DRC warning is issued based on **likelihood**.

The same violation applies to **asynchronous reset** signals affecting the logic cloud. A related violation occurs for clock signals directly affecting a **primary output** port. Since the latter can lower coverage slightly, TetraMAX flags it as a **C17** violation.

## Conclusion:

- Avoid design styles like that of N 10 that depart from pure synchronous clocked logic.
- In particular, avoid on-chip pulse-generator circuits using clock levels to shape pulses.

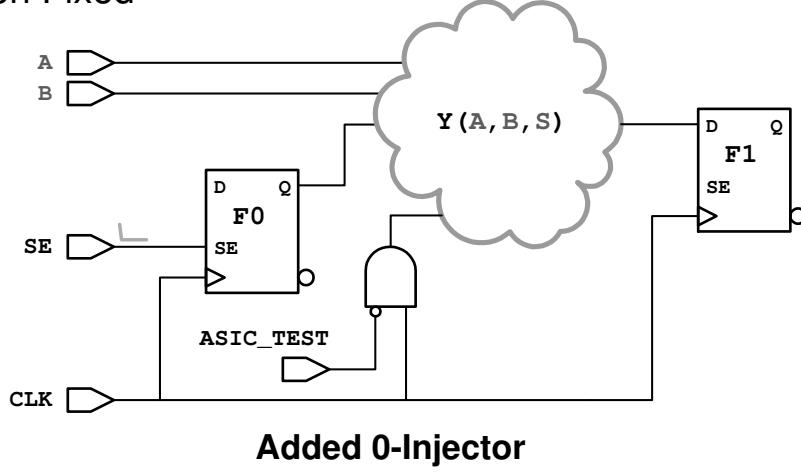
# Clock-as-Data Solution

## Risk-Free Capture

Violation Fixed

## Network

N 11



- Cloud output  $y$  is no longer dependent on clock level
- You could MUX in a one-bit PI instead of the clock
- Or recode, to delete the possibly-redundant clock path

5-58

AutoFix does not address this class of capture violation; it has to be fixed manually. Edit the HDL code, describing the injection logic in a technology-independent way.

For example, in Verilog: `assign CLOUD_IN = CLK & ~ASIC_TEST;`

# Top DRC Rules at a Glance

For optimal DFTC and TetraMAX results, ensure that:

1. State-changing inputs are controllable by ATE from PIs
2. Power-on reset, PLLs, and pulse generators are disabled
3. State-changing input never feeds logic cone into flip-flop



## State-Changing Inputs:

TetraMAX regards both clocks and asynchronous resets or sets as **state-changing** inputs that directly cause flip-flops to change state. They obey **similar** DRC rules.

**5-59**

TetraMAX views all clocks and asynchronous resets or sets as **state-changing inputs**. The reason is that TetraMAX can use any one of these signals in a **capture procedure**. Using **RST** as the “clock” in a capture cycle makes faults on the reset line **observable**. In **STIL**, capture cycles that **pulse a reset** port look like `pulse: Vector{RST = P;}`. This ATPG paradigm often requires you to think of the reset input as a kind of “clock.”; thus, an asynchronous reset or set obeys virtually the **same** DRC rules as do clocks.

There are **exceptions** to this default behavior.

# **Unit Summary**

---

**Having completed this unit, you should now be able to:**

- **Name the three type of DRC checks DFTC can perform on a design**
- **Perform a Test-Ready compile on a design**
- **Control internal nodes for the purposes of passing DRC**

**5-60**

# Lab 5: DFT Design Rule Checks



45 minutes

**After completing this lab, you should be able to:**

- Run DFT Design Rule Checks at varies places in the flow and interpret the results
- Save the test-ready design

**5-61**

# Command Summary (Lecture, Lab)

<code>dft_drc</code>	Checks the current design against test design rules
<code>set hdlin_enable_rtlrdc_info</code>	Controls RTL DRC file name & line number information
<code>set test_default_scan_style</code>	Defines the default scan style for <code>insert_dft</code>
<code>compile -scan</code>	Perform a Test-Ready compile
<code>compile_ultra -scan</code>	Perform a Test-Ready compile with DC Ultra
<code>set_scan_state</code>	Sets the scan state status for the current design
<code>report_scan_state</code>	Displays the scan state of the current design
<code>set test_simulation_library</code>	Indicates the pathname to a Verilog simulation library
<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>read_test_protocol -section test_setup</code>	Reads only the specified section and ignores others. The only valid parameter is <code>test_setup</code>
<code>create_test_protocol</code>	Creates a test protocol based on user specification
<code>set_dft_drc_configuration -internal_pins enable</code>	Enable the internal pins flow for clock, reset, constant, scan-in, scan-out, scan-enable and test_mode signals
<code>set_test_assume</code>	Specify assumed values on pins for test purposes

5-62

# **Appendix A**

## **Internal Pins Specification**

# What are Internal Pins?

- Internal pins specifications are a way to specify internal pins (rather than top-level ports) as scan signals
- More powerful feature than `set_test_assume`
- Common Applications
  - Specify an output of black-box as clock
  - Specify an output of sequential cell (like JTAG register) as `TestMode` pin

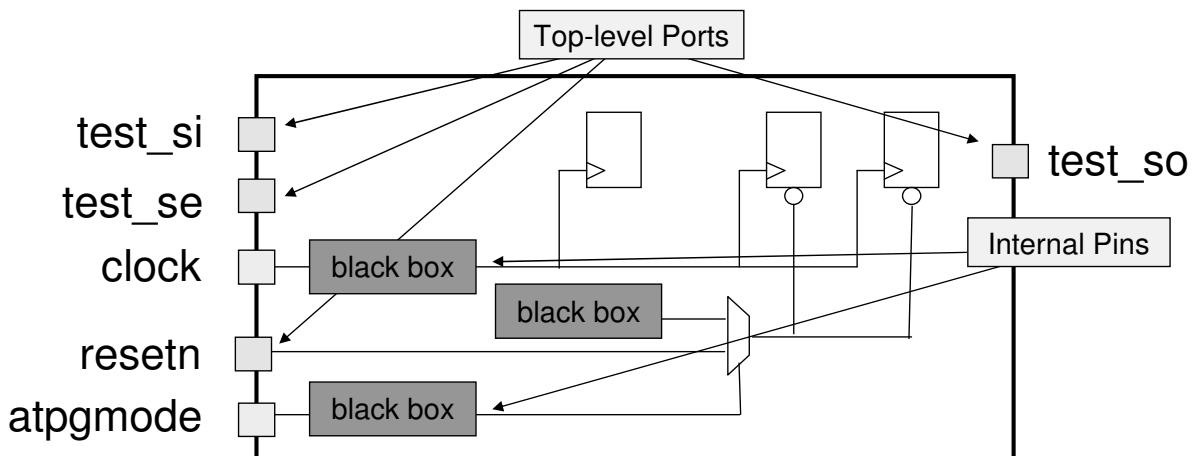
5-64

# Enabling Internal Pins Support

- **To enable running DRC with “internal pins”**
  - `set_dft_drc_configuration -internal_pins enable`
  - Default is “disable”
- **Note that `dft_drc` will assume direct control over internal-pins for DRC purposes**
  - User has the responsibility to ensure that the internal-pins “assumption” is correct
- **Limitations:**
  - Internal pins are supported only for multiplexed-scan style
  - Internal pins are not supported in following flows: core wrapper, boundary scan, and scan extraction

5-65

# Internal Pins Example



- Some signals can be declared as top-level ports while others are internal pins
- In this example, the “clock” and “atpgmode” signals come from “black boxes” and need to be specified as Internal Pins to pass DFT DRC

5-66

# Supported Data Types

- Internal pin signals are specified using the `-hookup_pin` option of the `set_dft_signal` command
- Supported Data Types (`-type <datatype>`):

`ScanDataIn`  
`ScanDataOut`  
`ScanEnable`  
`ScanMasterClock`  
`MasterClock`  
`ScanClock`  
`Reset`  
`Constant`  
`TestMode`

5-67

# Valid Internal Pins Hookup Locations

## ■ Clocks, Resets, Scan-Inputs, Scan-Enable

- Output of combinational gates
- Output of sequential cells
- Output of black-boxes

## ■ Scan-Outputs

- Input of combinational gates
- Input of sequential cells
- Input of black-boxes

5-68

# Specification Examples (1/2)

## ■ Clock:

```
set_dft_signal -view existing_dft -type ScanClock  
-hookup_pin U145/Z -timing {45, 55}
```

## ■ Reset:

```
set_dft_signal -view existing_dft -type Reset  
-hookup_pin U153/Z -active_state 0
```

## ■ Constant:

```
set_dft_signal -view existing_dft -type Constant  
-hookup_pin jtag/data15/Q -active_state 1
```

## ■ TestMode:

```
set_dft_signal -view spec -type TestMode  
-hookup_pin {tm_reg/q} -active_state 1
```

5-69

# Specification Examples (2/2)

## ■ Scan input

```
set_dft_signal -view spec -type ScanDataIn  
-hookup_pin {U3/SI0_int}
```

## ■ Scan output

```
set_dft_signal -view spec -type ScanDataOut  
-hookup_pin {U3/SO0_int}
```

## ■ Scan enable

```
set_dft_signal -view spec -type ScanEnable  
-hookup_pin {U3/SE_int}
```

## ■ Scan path definition for using hookup pins:

```
set_scan_path {ChainName} -scan_data_in {U3/SI_int}  
-scan_data_out {U3/SO_int} -scan_enable {U3/SE_int}
```

5-70

# Internal Pins in Reports

- **Command:**

```
set_dft_signal -view existing_dft -type ScanClock  
-timing {45 55} -hookup_pin u2/oclk2
```

- **preview\_dft report**

Scan chain '2' (SI2 --> OUT2[1]) contains 2 cells:

u1/q_reg[0]	(u2/oclk2, 45.0, rising)
u1/q_reg[1]	

- **report\_dft\_signal report**

Port	SignalType	Active	Hookup	Timing
-----	-----	-----	-----	-----
u2/oclk2	ScanMasterClock	1	-	P 100.0 R 45.0 F 55.0
u2/oclk2	MasterClock	1	-	P 100.0 R 45.0 F 55.0

**5-71**

When internal-pin signals are seen in reports, in this case for a **ScanClock**. Instead of a top-level port, the full hierarchical path of the internal pin is reported.

## **Appendix B**

### **Existing Scan Chain Extraction**

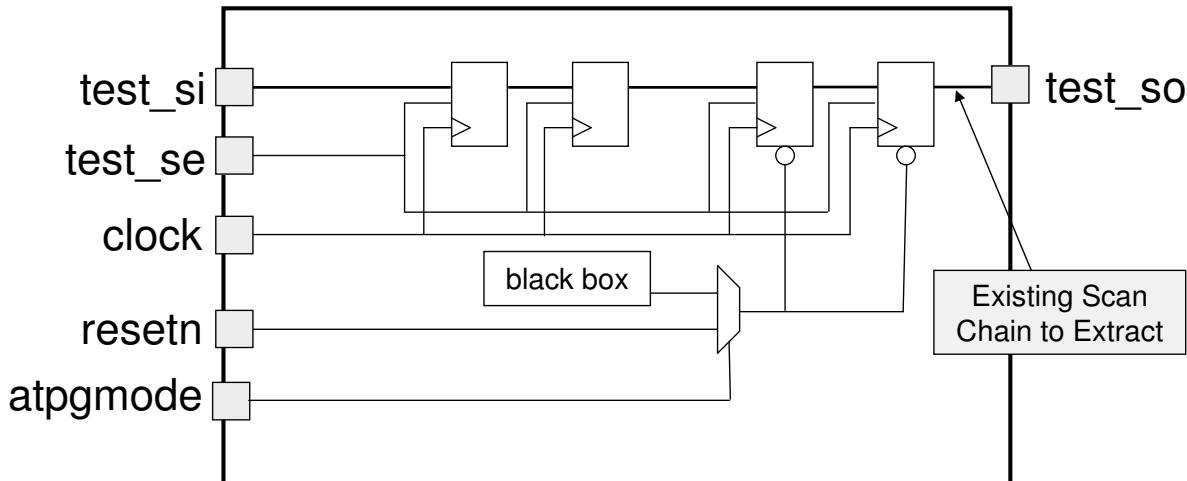
# Scan Chain Extraction

- The scan chain extraction process extracts existing scan chains from a design
- When performing scan extraction, always use `-view existing_dft` since the test structures defined already exist in the design
- The `dft_drc` command is used to perform the actual scan chain extraction

5-73

## “existing\_dft” View

- The “existing\_dft” view describes structures which already exist that must be understood for the design to pass DRC (dft\_drc)
  - Clocks, Resets, Constants or scan in/out/enable

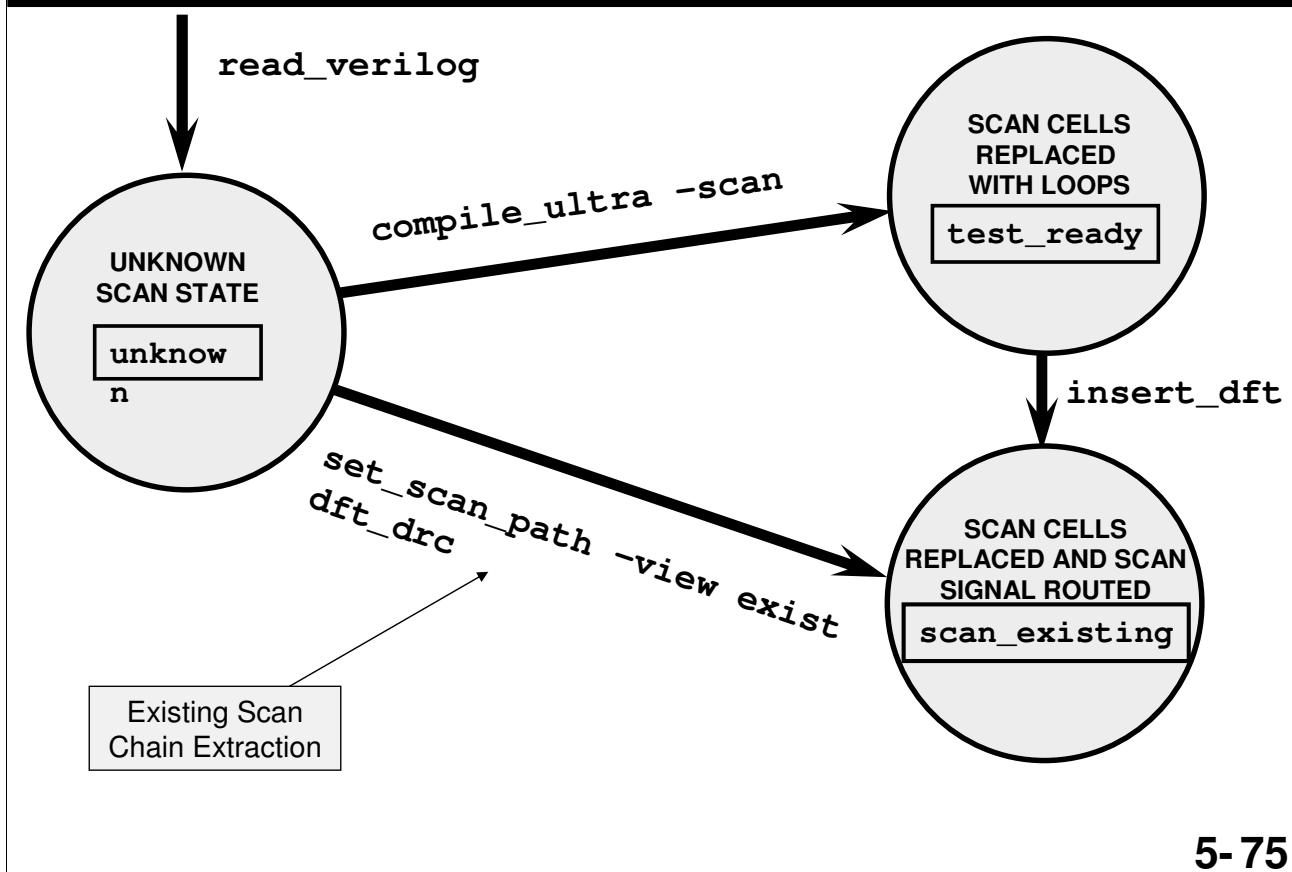


5-74

When using an scan “extraction” flow. The scan chains to be extracted must be declared as  
-view existing\_dft with the **set\_scan\_path** command. For example:

```
set_scan_path my_existing_chain -view -exist -scan_data_in \
test_si -scan_data_out test_so -scan_enable test_se
```

# Scan State After Extraction



5-75

# Example: Scan Chain Extraction

```
# Declare the scan interface signals with set_dft_signal
set_dft_signal -view existing_dft -type ScanDataIn -port test_si
set_dft_signal -view existing_dft -type ScanDataOut -port test_so
set_dft_signal -view existing_dft -type ScanEnable -port test_se

# Declare the scan existing scan chains with set_scan_path
set_scan_path chain1 -view existing_dft -scan_data_in test_si \
-scan_data_out test_so

# Define the test clocks, reset, and test mode signals
set_dft_signal -view existing_dft -type ScanClock -port clock \
-timing [list 45 55]
set_dft_signal -view existing_dft -type Reset -port resetn -active_state 0

# Create the test protocol by using the create_test_protocol command.
create_test_protocol -capture_procedure multi_clock

# Extract the scan chains by using the dft_drc command
dft_drc

# Report the extracted chains
report_scan_path -view existing_dft -chain all
report_scan_path -view existing_dft -cell all
```

5-76

# Agenda

**DAY  
2**

**6 DFT DRC GUI Debug**



**7 DRC Fixing**



**8 Top-Down Scan Insertion**



# Unit Objectives



After completing this unit, you should be able to:

- Use Design Vision to debug DFT DRC violations
- Name several different kinds of Pindata used with the Design Vision Schematic Viewer
- Use the Waveform Viewer to debug initialization sequences in `test_setup`

6-2

# DFT DRC GUI Debug: Agenda

**GUI Debug with Design Vision**

**Schematic Viewer Pindata**

**Waveform Viewer**

**GUI Commands**

**6-3**

# GUI Debug of dft\_drc Violations

- Debug features enabled in Design Vision platform
- Provides GUI based debug environment for:
  - Pre DRC Violations
  - Post DRC Violations
- Multiple TetraMAX Pindata types available
- Does not display test\_simulation\_library models
  - `test_simulation_library` models are still honored by `dft_drc` when running DFT DRC checks
  - `test_simulation_library` models are NOT displayed in the Design Vision GUI
  - Only the models read and linked by Design Vision are displayed in the Design Vision GUI

6-4

# GUI Debug Usage Models

## ■ Run Design Vision in foreground

- Use the `design_vision` command

- ◆ Execute script from the File pulldown menu to run `dc_shell` script
  - ◆ Enter commands from `design_vision` command line

- Browse Violations from the Violation Browser after `dft_drc`

## ■ Launch Design Vision using a TCL script

```
design_vision -f script.tcl | tee script.log
```

- ◆ `dft_drc`
  - ◆ Design Vision top level launches after `dft_drc`, then use **Test → Browse Violations** to launch Violation Browser

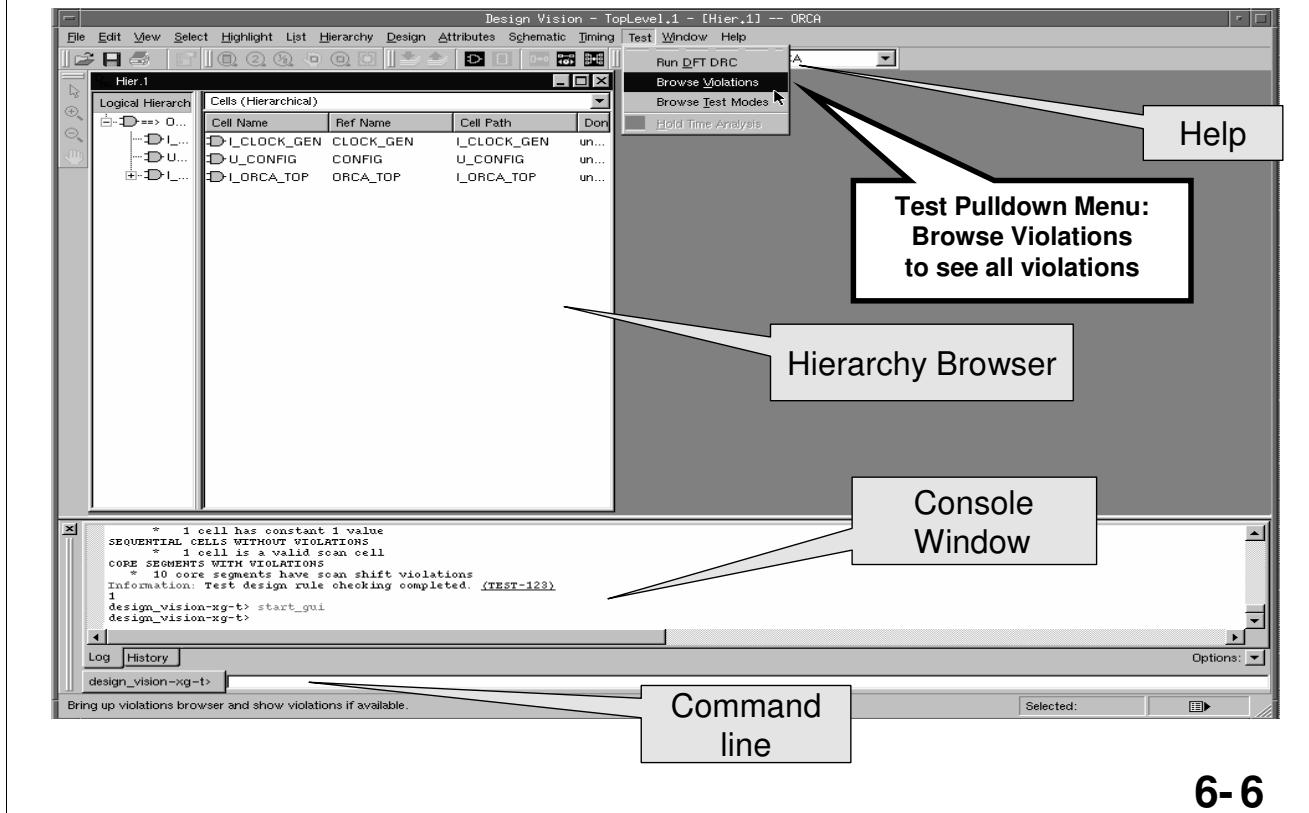
## ■ NOTE: You must run your script in Design Vision to use the GUI debug features

6-5

You can also use `design_vision -no_gui` to start and run `design_vision` in shell mode without a GUI. When ready for debug use

```
dc_shell> gui_start
```

# Design Vision: Top Level



6-6

# GUI Components

## ■ Violation Browser

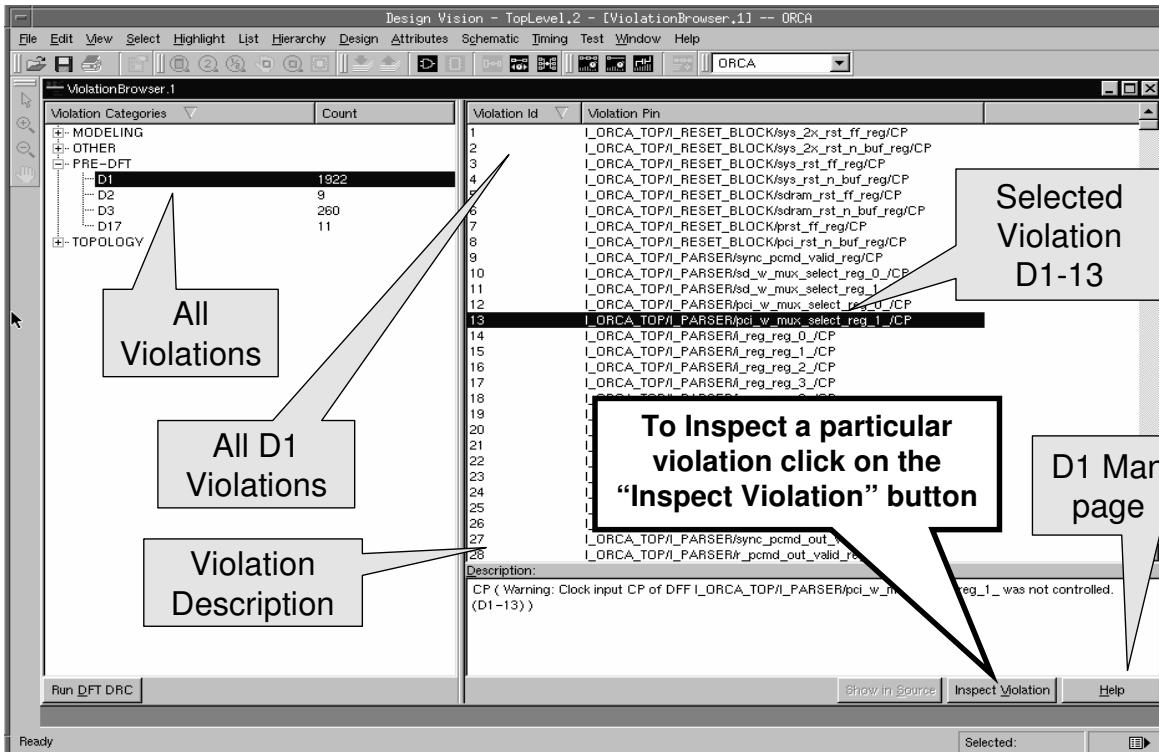
- Launched after `dft_drc` finishes using:
  - ◆ Shell script command
  - ◆ Test pulldown menu → Browse Violations
- Allows selection of multiple violations of one type for analysis

## ■ Violation Inspector

- Schematic Viewer
  - ◆ Path schematic of violation and violation source
  - ◆ Allows selection of Pin Data types
- Waveform Viewer
  - ◆ Debug `test_setup` procedures using simulation waveforms
  - ◆ Launched when Pin Data set to `test_setup`

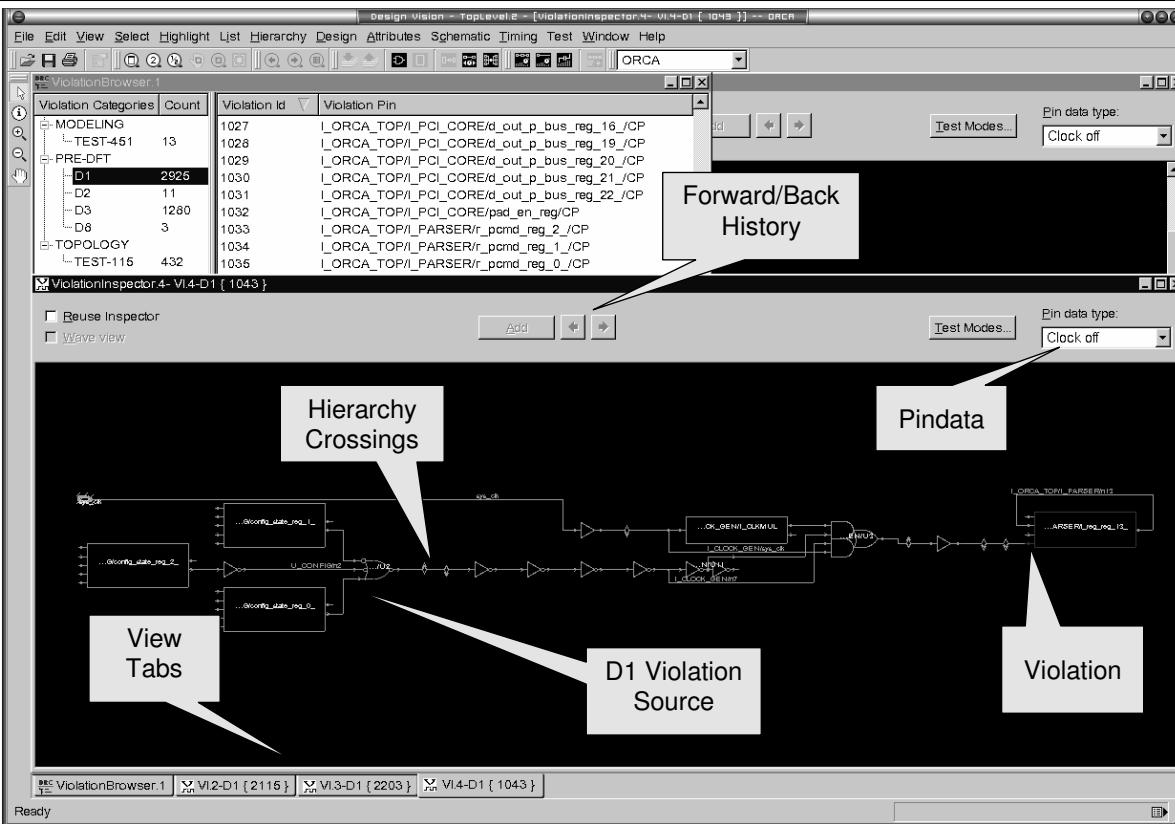
6-7

# Violation Browser: After Test → Browse Violations



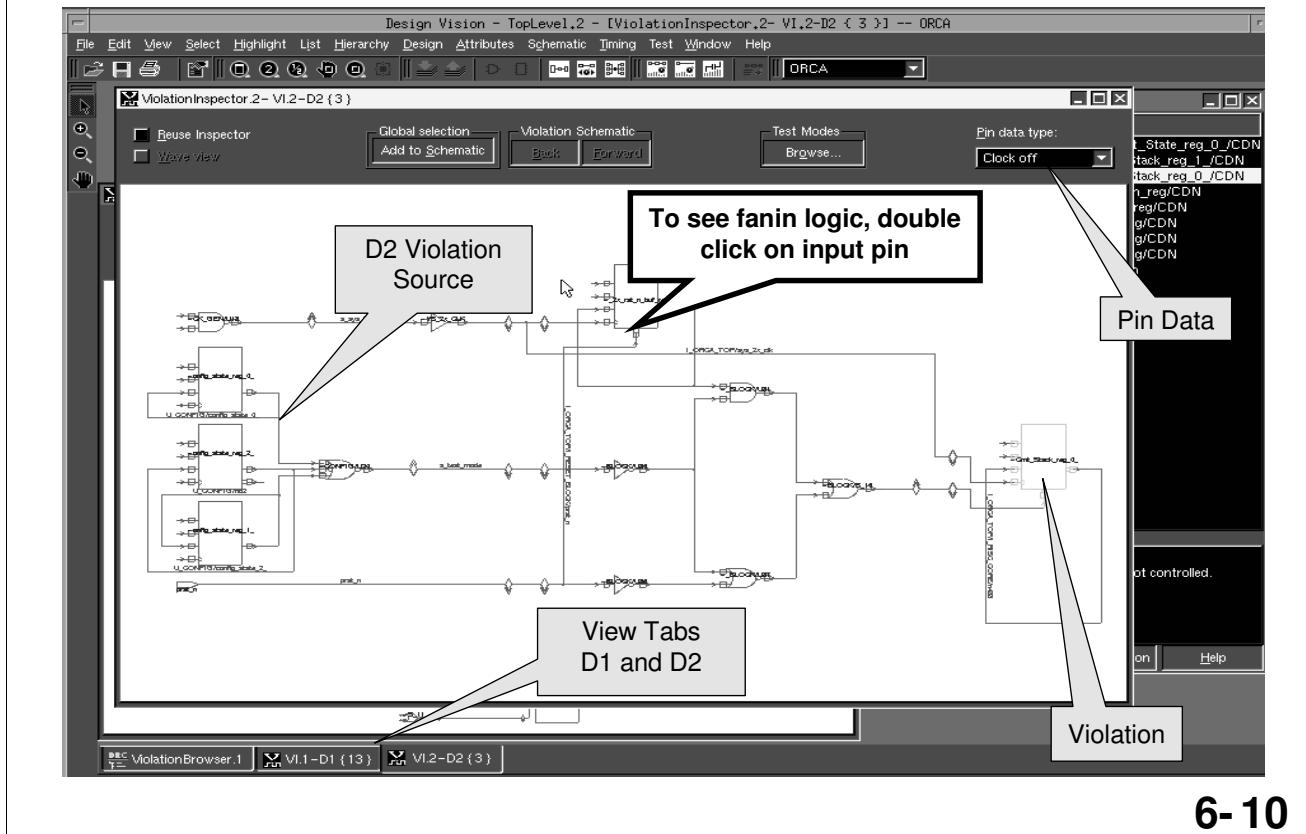
6-8

# Violation Inspector: D1 Analysis



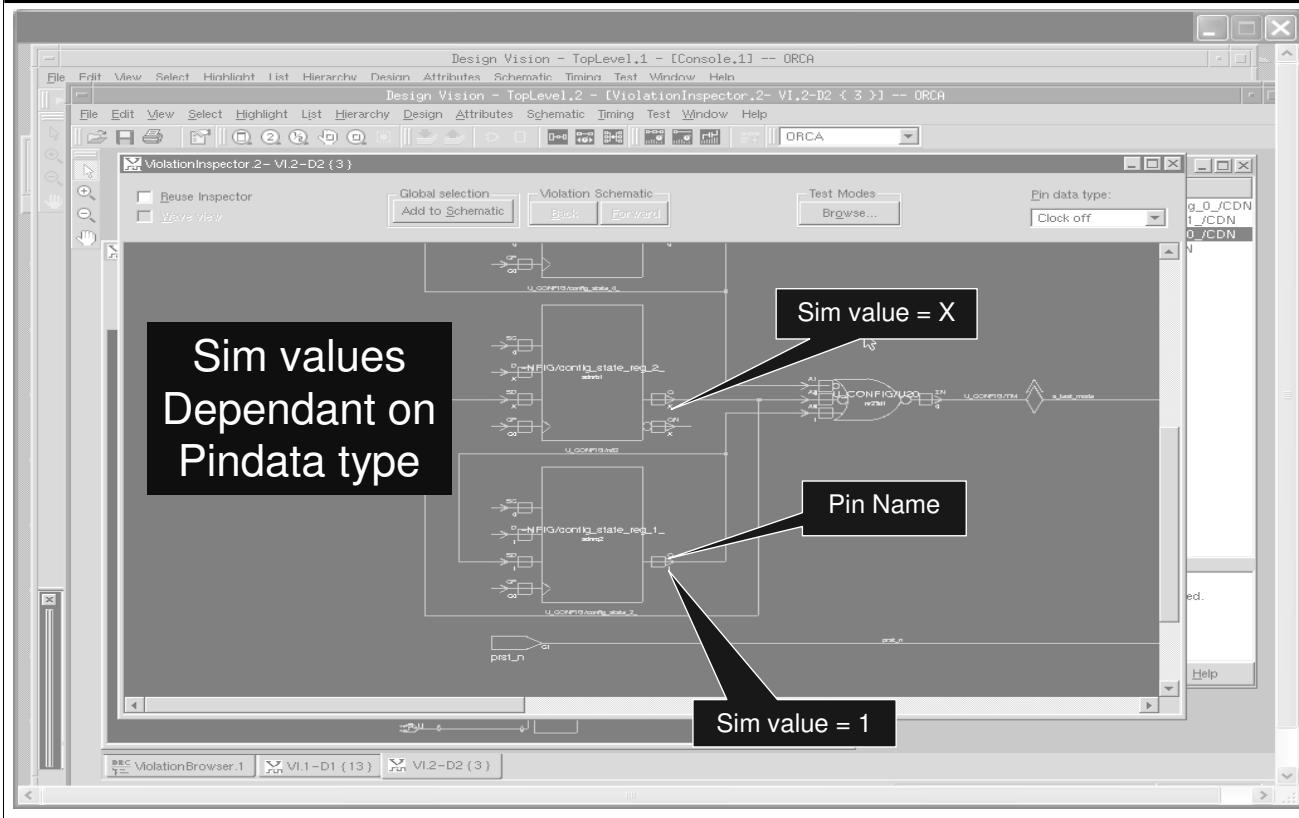
6-9

# Violation Inspector: D2 Analysis



6-10

# Violation Inspector: Schematic Viewer: Simulation Values



6-11

# DFT DRC GUI Debug: Agenda

**GUI Debug with Design Vision**

**Schematic Viewer Pindata**

**Waveform Viewer**

**GUI Commands**

**6-12**

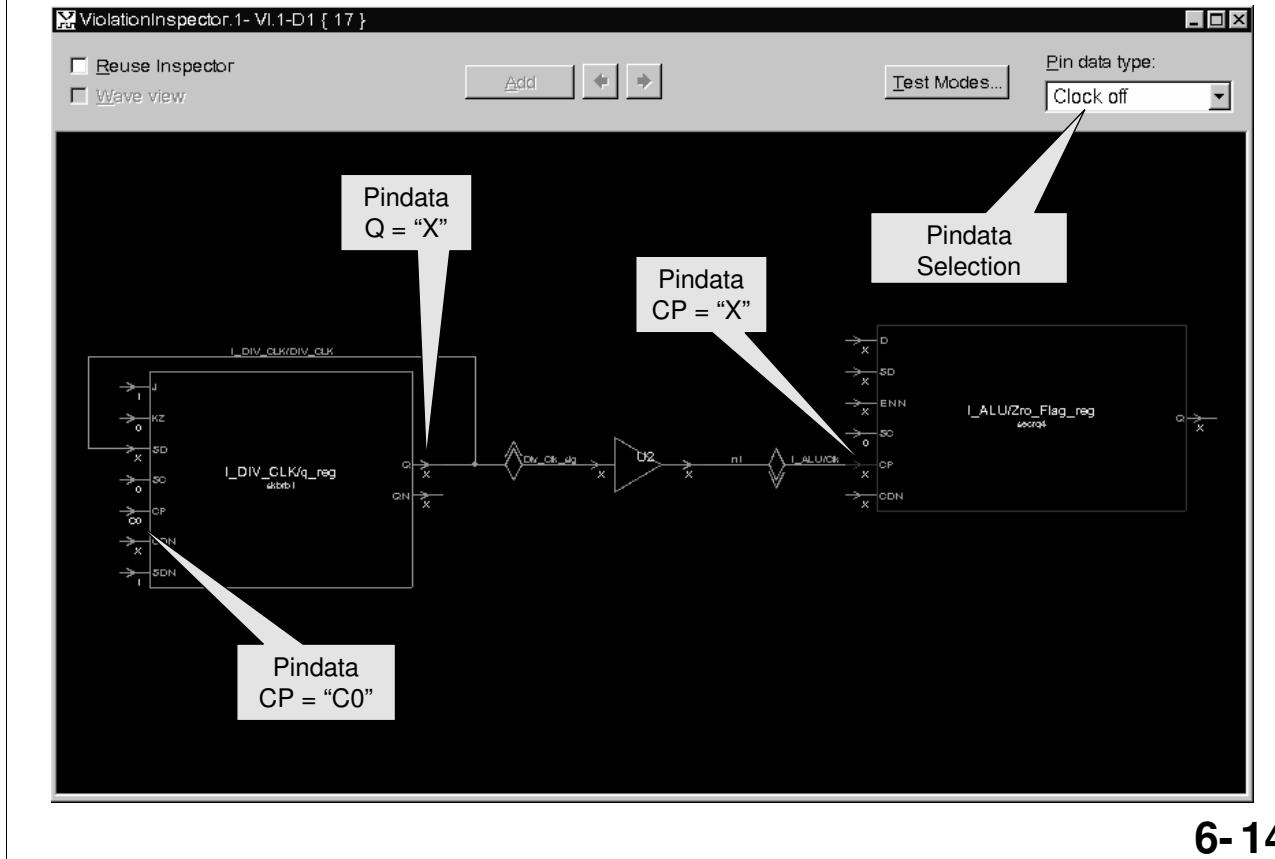
# What is Pindata?

- Pindata is the simulation value that is annotated on a schematic viewer for each cell pin
- There are several types of pindata annotations available
- Different types of pindata can be useful for debugging various DRC violations
- When debugging a DRC violation graphically, DFT Compiler will automatically select the pindata that should be most useful for debugging that violation type

6-13

Note: pindata does not account for **set\_test\_assume** specifications on internal nets.

# Pindata in Design Vision



6-14

# Pindata Types

- The following types of pindata are supported when using Violation Inspector

- Clock on
- Clock off
- Constrain value
- Load
- Master observe
- Shadow observe
- Shift
- Stability pattern
- Tie data
- Test setup

6-15

# Pindata: Clock On

- **Clock On**
  - Examples:
    - a) X-X b) 0-1 c) X-1
- **When the “Clock On” pindata is selected, it displays two bits separated by a “-” for each pin.**  
**The first bit is the simulated value that results when all clocks are set to off. All other inputs are set to X or to their constrained values. This is the same as the “Clock Off” pindata. The second bit is the simulated value that results when a specified clock is set to on. All other inputs are set to X or their constrained values.**

6-16

# Pidata: Clock Off

- **Clock Off**
  - Examples:
  - a) X    b) 1    c) 0
- **When “Clock Off” pidata is chosen the data displayed is the simulated values which occur when all defined clocks are set to their off values and all other inputs are set to X. Any 0 or 1 values are indications of logic with an unblocked combinational path back to a clock port. X's are a sign of logic with no path to a clock port or the path is blocked by gates which have X's for inputs.**

6-17

# Pindata: Constraint Data

## ■ Constraint Data

- Examples:
- a) X/-,X/-,X/- b) 0/-,0/B,0/- c) X/-,1/B,1/- d) X/-,~01/B,~01/B

## ■ When “Constraint Data” is chosen, the data consists of three pairs of characters "T/B,C/B,S/B" where:

- T = simulated value due to tied gates
- B = fault blockage due to tied gates indicated by T
- C = simulated value due to constraints for combinational ATPG: tied gates, constrained pins, constant value gates
- S = fault blockage due to constraints indicated by C
- S = simulated value due to constraints during sequential scan ATPG
- B = fault blockage due to any constraint indicated by S

**6-18**

The blockage fields are either the characters "B" for blocked, or "-" for not blocked.

A "~" followed by one or more characters is an indication of not allowed (restricted) values. For example, ~01 means not 0 or 1, or the same as restricted to X or Z. ~Z means restricted to 0,1, or X.

Example A "X/-,X/-,X/-" indicates no constraints and no blocks.

Example B "0/-,0/B,0/-" indicates the pin is constrained to 0 due to tied logic, there is also a constraint of 0 during combinational ATPG which contributes to a blockage. During sequential scan ATPG there is a constraint to 0 but no corresponding blockage.

Example C "X/-,1/B,1/—" indicates a constraint to 1 for both combinational and sequential scan ATPG, but a blockage due to that constraint only for combinational ATPG.

Example D "X/-,~01/B,~01/B" indicates there is no constraint or blockage due to tied gates but there is a constraint and a blockage for the combinational and sequential scan ATPG algorithm of not 0 or 1 (~01), in other words X or Z, due to constrained pins and constant value gates.

# Pindata: Load

## ■ Load

- Examples:
- a) X{}X b) 1{}1 c) 001{}1 d) X{}0{}1

■ When the pindata setting of “Load” is selected, the simulation events from the `load_unload` procedure are displayed. The curly braces "{}" represent a placeholder for the `shift` procedure. The values in front of the "{}" are from vectors defined in the `load_unload` procedure *prior* to the application of `shift`. The first value after the "{}" is the final simulated value at the end of the `shift` procedure. Any additional values after this character are from vectors defined within the `load_unload` procedure but *after* the application of the `shift` procedure.

**6-19**

Simulation is performed by setting all constrained ports to their constrained values, all constant value gates to their constant values, and all other input ports to X. Then each test cycle in the procedure is simulated, propagating its effect throughout the design.

Example A indicates the pin is at an X state during the `load_unload` procedure.

Example C indicates three time events prior to shift of "001" and after the shift the pin is still a 1.

Example D indicates there are two shift procedures and the pin begins at a value of X, is a 0 at the end of the first shift procedure and is a 1 at the end of the second procedure.

# Pindata: Shift

## ■ Shift

- Examples:
- a) XXX b) 010 c) 000 d) 00010 e) 010/00110

- When the pindata setting of “Shift” is selected, the simulation events due to test vectors defined in the shift procedure are displayed
- Simulation is performed by setting all constrained pins and constant value gates to their appropriate values and then simulating the test vectors in the load\_unload prior to the shift procedure to determine the initial values at the start of the shift procedure. Then the test vectors in the shift procedure are simulated to provide the values which are displayed.

6-20

Example A indicates the shift procedure contains 3 simulation events and the pin of interest is an X value for all three events.

Example B indicates a pin which is clocked during the shift procedure.

Example E is a special case which is often associated with JTAG related designs. The first three values prior to the slash "/" indicate the general case of the shift procedure. The values after the slash indicate an additional (non-general) application of a shift. to obtain this type of shift pattern the load\_unload procedure must contain test cycles after the Shift which pulse the shift clock. One or more additional shift operations is possible in which case each would be separated by a forward slash "/".

# Pidata: Master Observe

- **Master Observe**
  - Examples:
    - a) 010 b) 111
- **When the pidata setting of “Master Observe” is selected, the simulation events due to the test cycles defined in the `master_observe` procedure are displayed**
- **The `master_observe` procedure and “Master Observe” pidata are typically only seen when working with LSSD designs**

6-21

# Pindata: Shadow Observe

- **Shadow Observe**
  - Examples:
    - a) 010 b) 111
- **When the pin data setting of “Shadow Observe” is selected, the simulation events due to the test cycles defined in the `shadow_observe` procedure are displayed**
- **The `shadow_observe` procedure and “Shadow Observe” pindata are typically only seen when working with LSSD designs**

6-22

# Pindata: Stability Patterns

## ■ Stability Patterns

- Examples:
- a) XXXX/XXX b) 1111/111 c) 000011/111

■ The “Stability Patterns” setting of pindata will cause the simulation events due to the test vectors in the `load_unload`, `shift`, and `capture` procedures to be displayed. However, this data is only available for sequential devices that have been classified as stable with constant 1 or constant 0 values

6-23

Example A indicates a pin which is at an X value through the `load_unload` (values prior to "/") and through the capture procedure (values after "/").

Example C indicates a pin which is a 1 by the end of `load_unload` and remains so through a capture clock procedure.

# Pindata: Tie Data

- **Tie Data**
  - Examples:
  - a) X    b) 0    c) 1    d) Z
- **When the pindata setting of “Tie Data” is selected, the logic values resulting from tied logic are displayed. An X indicates that there is no value due to tied logic while a 0, 1, or Z indicates the affect of tied logic at that pin**

**6-24**

# Pindata: Test Setup

- **Test Setup**
  - Examples:
  - a) X    b) 0    c) 1    d) X0111
- **When the pindata setting of “Test Setup” is selected, the simulation events due to test cycles defined in the `test_setup` macro are displayed**
- **If there is no `test_setup` macro defined but PI constraints exist, then there is an implied `test_setup` macro consisting of one test cycle in which the constrained ports are assigned their constraint values**
- **“Test Setup” data can be viewed/debugged in the Waveform Viewer**

6-25

Example B indicates a logic value of 0 due to `test_setup` macro.

Example D indicates 5 simulation events with a final value of 1.

# DFT DRC GUI Debug: Agenda

**GUI Debug with Design Vision**

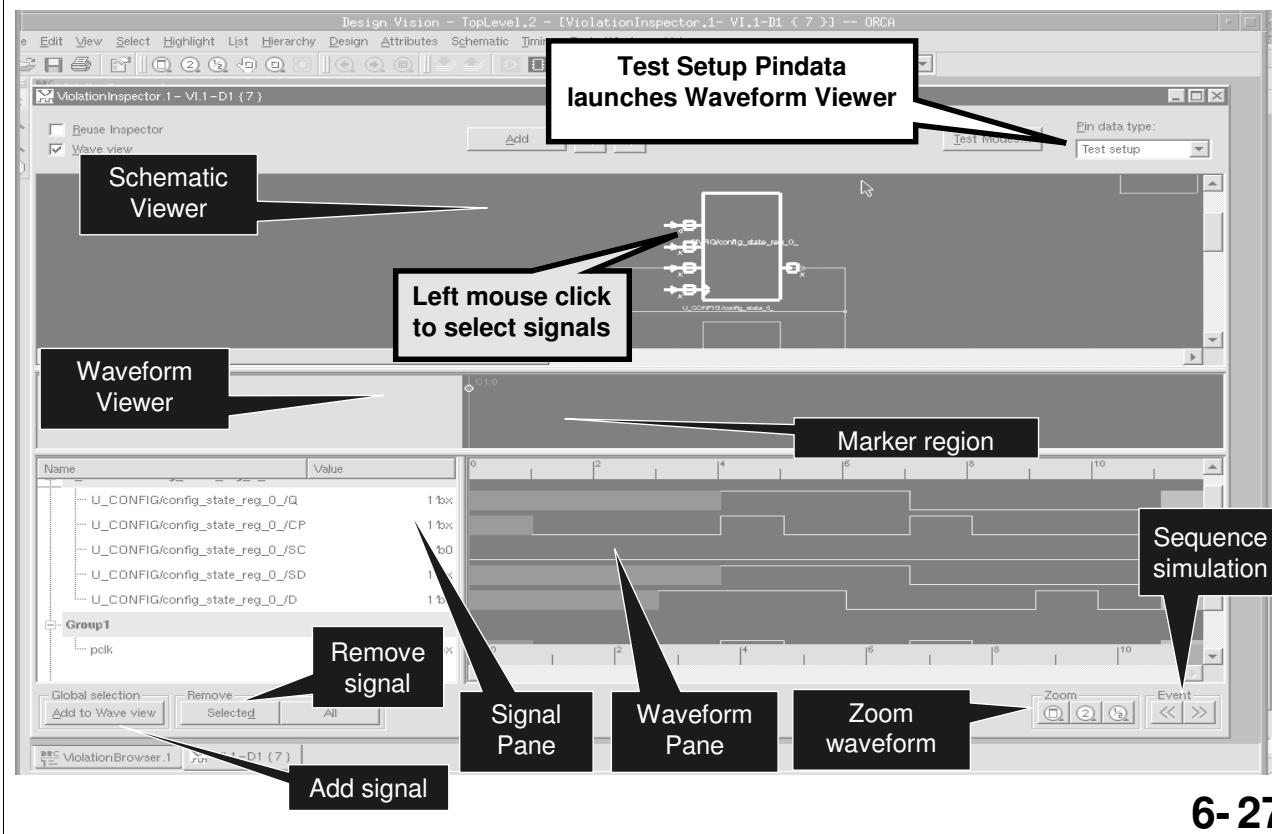
**Schematic Viewer Pindata**

**Waveform Viewer**

**GUI Commands**

**6-26**

## Violation Inspector: Waveform Viewer: test\_setup analysis



You can use the waveform viewer to display **test\_setup** patterns and debug initialization problems.

# DFT DRC GUI Debug: Agenda

**GUI Debug with Design Vision**

**Schematic Viewer Pindata**

**Waveform Viewer**

**GUI Commands**

**6-28**

# Helpful Commands

- **Zoom in: CTRL + =**
- **Zoom out: CTRL + -**
- **Find object and Zoom in to selection:**
  - Select -> By Name to pop up “Select by Name” box
    - ◆ Enter the object name then hit “Search” button
    - ◆ Highlight the object that was found, then hit “Select”
  - Zoom -> Zoom Fit Selection
- **Highlight an Object: CTRL + H**
  - ◆ Each consecutive CTRL + H changes the color
- **To report all the keyboard shortcuts:**
  - Select “Help” -> “Report Hotkey Binding”

6-29

# Helpful Commands

- **Move back in design hierarchy when in Schematic Viewer**
  - Right Click + “Back”
- **Move forward in design hierarchy when in Schematic Viewer**
  - Right Click + “Forward”
- **Move backwards in design hierarchy when in Schematic**
  - Right Click + “Back”
- **Close the Console Window:**
  - Right Click on left portion of console + “Undock”
  - Hit the X button on the upper left corner of the Console

**6-30**

Hot Keys are displayed next to menu items where available

# Helpful Commands

- List all tcl GUI commands: `help *gui*`
- Analyze violation from Violation Inspector command line:
  - `gui_inspect_violations [-type AType] AList`
- Add signals to Waveform Viewer:
  - `gui_wave_add_signal [-window AWnd] \ [-clct] AList`
- Add objects to Schematic Viewer:
  - `guiViolationSchematicAddObjects \ [-window AWnd] [-clct] AList`

6-31

Additional info on the `gui_*` commands can be found in the Appendix.

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- Use Design Vision to debug DFT DRC violations**
- Name several different kinds of Pindata used with the Design Vision Schematic Viewer**
- Use the Waveform Viewer to debug initialization sequences in `test_setup`**

**6-32**

# Lab 6: Introduction to Design Vision



45 minutes

**After completing this lab, you should be able to:**

- **Read a design and related files into Design Vision, the graphical interface to DFT Compiler (DFTC)**
- **Compile that design into a test-ready design**
- **Explore that design in Design Vision**
- **Save the test-ready design**

**6-33**

# Command Summary (Lecture, Lab)

<code>design_vision</code>	Runs Design Vision visualization GUI
<code>dft_drc</code>	Checks the current design against test design rules
<code>gui_inspect_violations</code>	Brings up specified DFT DRC violations in a new Violation Inspector window
<code>gui_wave_add_signal</code>	Add specified signals to waveform view of a specified Violation Inspector window
<code>guiViolation_schematic_add_objects</code>	Adds specified objects to the schematic view of a specified Violation Inspector window

6-34

# **Appendix A**

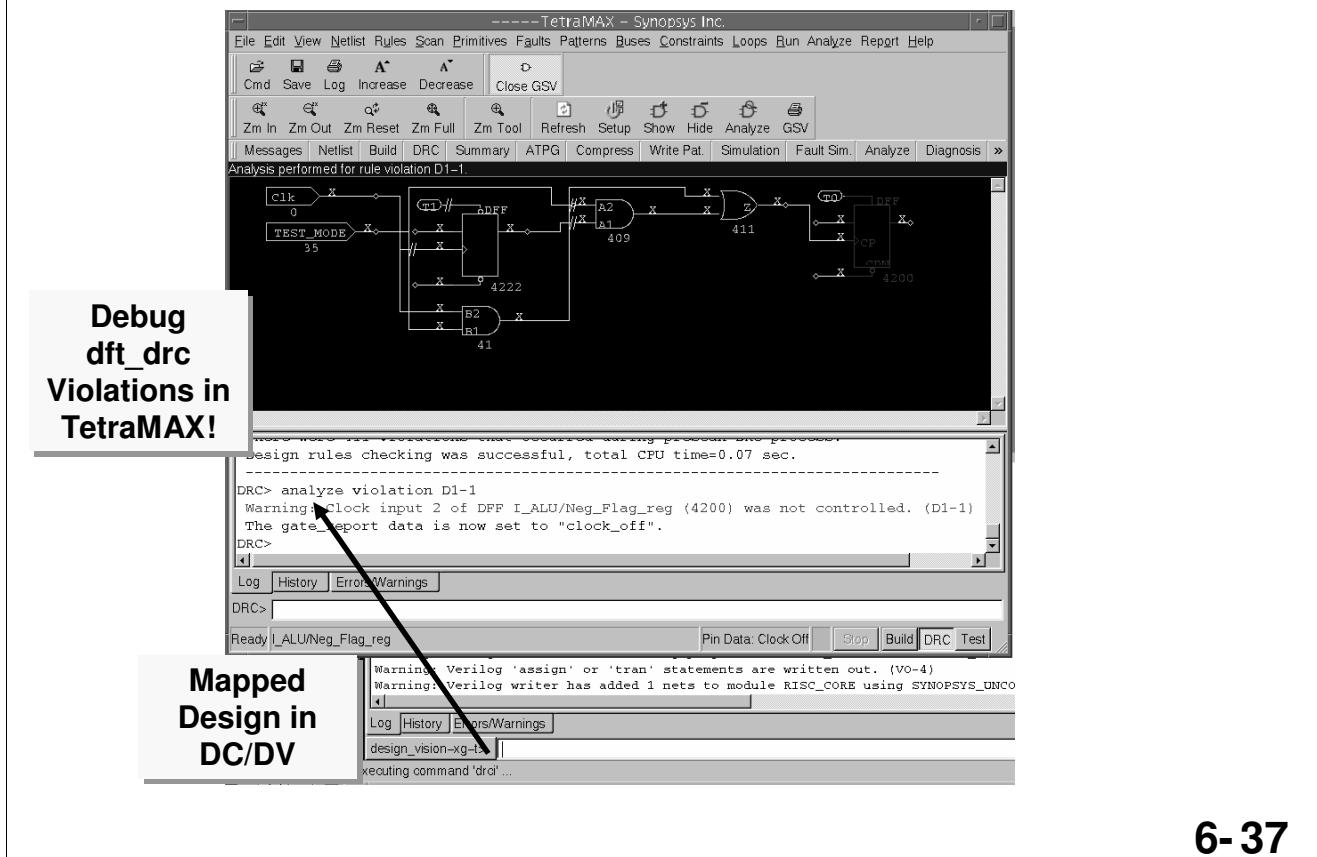
## **GUI Debug with TetraMAX**

# GUI Debug with TetraMAX

- TetraMAX is run “under the hood” for DFT DRC
- In many cases, TetraMAX can be run explicitly in order to debug DRC violation graphical with TetraMAX’s Graphical Schematic Viewer (GSV)
- TetraMAX is setup and run apart from DFTC
- Required files to debug DFT DRC violations in TetraMAX
  - Verilog netlist (`write -f verilog -o design.v`)
  - Verilog simulation library (used by TetraMAX)
  - Protocol file (for post-DFT DRC checks)
  - TetraMAX run script

6-36

# DFT DRC on Gates? Use TetraMAX GUI



6-37

# GUI Debug with dft\_drc\_interactive

- **TCL utility script to launch TetraMAX from dc\_shell**
- **Example script:**

```
# Point to the gate library, if available
set test_simulation_library [list /path/to/library/lsi_10k.v]
set_scan_configuration ...
set_dft_signal ...
create_test_protocol -capture_procedure multi
dft_drc
# A critical DRC violation has been uncovered
source dft_drc_interactive.tcl
dft_drc_interactive
# Problem is debugged inside TetraMAX
# Corrections are made inside DFT Compiler
dft_drc
preview_dft
insert_dft
```

- **For more info consult the SolvNet article (012388):**  
<https://solvnet.synopsys.com/retrieve/012388.html>

6-38

## **Appendix B**

### **GUI commands**

## **Usage: gui\_inspect\_violations**

```
gui_inspect_violations [-type AType] AList
```

- **Brings up specified DFT DRC violations in a new Violation Inspector window**
- **-type AtypE: This argument is used to specify the type of violation (e.g. D1) when inspecting multiple violations of same type**
- **AList: When specified with -type, this can be a list of one or more violation ids to inspect. Violation ids is the violation number shown in the first column of violation Browser.**

**6-40**

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

When –type argument is not used, this specifies a single violation instance to be inspected in violation inspector. Violation instances are named in the format (similar to TetraMAX) ViolationType-ViolationId e.g. D1-1.

## **Examples: gui\_inspect\_violations**

- To inspect multiple violations (5, 9 & 13) of type D1:

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

- To inspect a single violation 4 type D2:

```
gui_inspect_violations -type D2 4
```

Or ...

```
gui_inspect_violations D2-4
```

**6-41**

## **Usage: gui\_wave\_add\_signal**

```
gui_wave_add_signal [-window Awnd] [-clct] AList
```

- **Add specified signals to waveform view of a specified violation inspector**
- **-window Awnd: If Awnd is valid, the signal is added to the WaveForm Viewer of specified violation inspector window**
- **-clct: When specified, it implies that the command is to treat AList as a collection of object handles. If not, AList is treated as a list of object names.**
- **Alist: This is a list of object names if no -clct option is specified otherwise this is a collection of object handles.**

**6-42**

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

When “-window Awnd” is specified, If no such violation inspector window exists, the command exits with an error message. If no “-window” is specified, the signal is added to the waveform view of first ViolationInspector window created.

## Examples: `gui_wave_add_signal`

- To add a port object ‘i\_rd’ to the first created `ViolationInspector` window with a waveform view,

```
gui_wave_add_signal i_rd
```

- To add selected objects to the given window `ViolationInspector.3`

```
gui_wave_add_signal -window  
ViolationInspector.3 -clct [get_selection]
```

6-43

## **Usage: guiViolationSchematicAddObjects**

```
guiViolationSchematicAddObjects
[-window Awnd] [-clct] AList
```

- Adds the specified objects to the schematic view of a specified violation inspector window and selects them
- **-window Awnd:** If Awnd is valid, the AList objects are added to the Schematic Viewer
- **-clct:** When specified, it implies that the command is to treat AList as a collection of object handles. If not, AList is treated as a list of object names.
- **Alist:** This is a list of object names if no -clct option is specified otherwise this is a collection of object handles

**6-44**

When “-window Awnd” is specified, If no such violation inspector window exists, the command exits with an error message. If no “-window” is specified, the object are added to the current active window.

## **Examples: guiViolationSchematicAddObjects**

- To add a cell object to the active ViolationInspector Schematic window:

```
guiViolationSchematicAddObjects  
    gate_clock
```

- To add a cell “gate\_clk” and a port “i\_rd” to a specified ViolationInspector.2 window:

```
guiViolationSchematicAddObjects -window  
    ViolationInspector.2 {gate_clk i_rd}
```

- To add selected objects to a specified ViolationInspector.3 window

```
guiViolationSchematicAddObjects -window  
    ViolationInspector.3 -clct [get_selection]
```

**6-45**

This page was intentionally left blank

# Agenda

**DAY  
2**

**6 DFT DRC GUI Debug**



**7 DRC Fixing**



**8 Top-Down Scan Insertion**



# Unit Objectives



**After completing this unit, you should be able to:**

- **Name several available methods for fixing common DRC issues in a design**
- **State the DFT requirements for Asynchronous set or reset signals**
- **Use AutoFix to fix DRC issues relating to clocks, resets, sets, internal tristate, and bidirectional ports**

**7-2**

# DRC Fixing: Agenda

Fixing Common Violations

AutoFix Clocks/Resets/Sets

Internal Tristate Issues

AutoFix Internal Tristates

AutoFix Bidirectionals

7-3

# How to Fix DRC Violations

## ■ Four methods of correcting DRC violations in DFTC:

1. Edit RTL code and resynthesize design with DFT logic
2. Use AutoFix DFT to insert bypass or injection logic
3. Use User Defined Test Points to insert ad hoc test points
4. Edit netlist with `create_net` and other commands



### What Synopsys ACs Recommend:

Add testability to the design early on, preferably in the synthesizable **RTL code**. Use AutoFix as a workaround after RTL code-freeze, or for unfamiliar legacy designs.

7-4

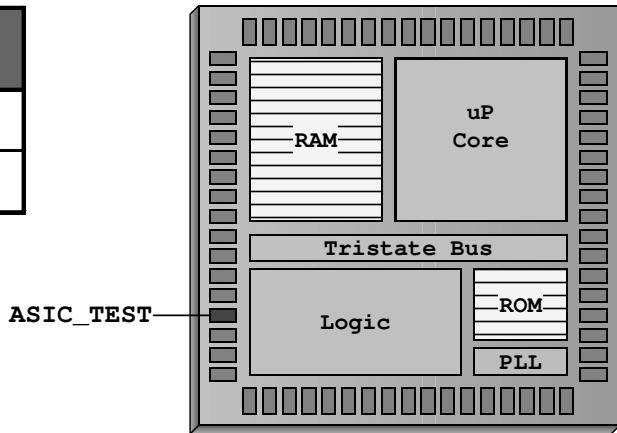
Synopsys recommends describing all DFT logic in the RTL source code, whenever possible. Later on, anyone can compile the RTL source, and obtain the synthesized DFT logic. This is the best strategy for design-for-reuse blocks, one of the keys to SoC design.

# Using a Test-Mode signal for DRC fixing

ASIC Mode	ASIC_TEST	SE
Normal	0	0
Test	1	0   1

**Test-Mode Pad**

Held at Logic 1



```
dc_shell> set_dft_signal -view exist -type Constant \
           -active 1 -port ASIC_TEST
```

- A test-mode can enable fixes to common DRC violations
- One test-mode pin can serve multiple on-chip test circuits
- Can be an input port or the output pin of an internal cell

7-5

Many DRC fixes use a **global signal** to enable DFT bypass or injection logic. This signal is held constant at **1** (unlike **SE**) during all scan-based testing. Many ASIC designs include a **similar** constant signal dedicated to  $I_{ddq}$  fault testing. This global signal—call it **ASIC\_TEST**—can be implemented as a primary **input**, or could be the **output pin** of an on-chip test-control block/register.

In this workshop:

The global signal will always be referred to via the name **ASIC\_TEST**.

Assume its polarity is **active-high**; this is not a current requirement of AutoFix.

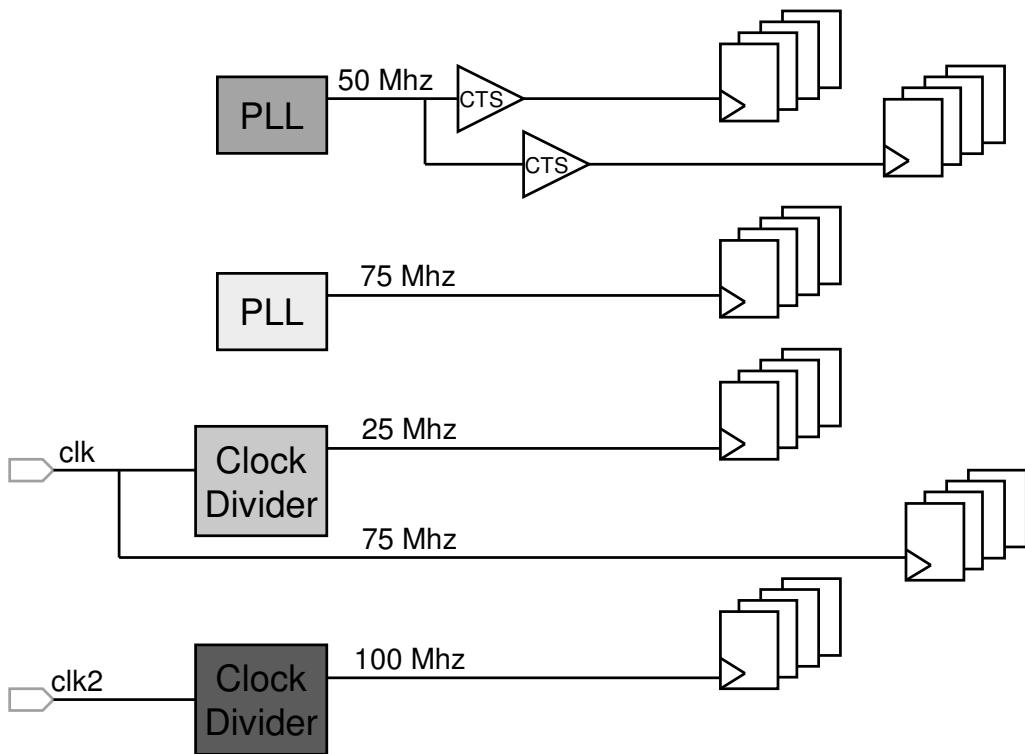
Inform ATPG it is **held high** all during test, using: **set\_dft\_signal**. Note that the signal type used is type “**Constant**” not “**TestMode**”

This has **no effect** on logic synthesis, and should not be confused with **set\_logic\_one**.

## Conclusion:

The global signal **ASIC\_TEST** is useful for DRC fixes that require a constant high value.

## How to handle multiple internal clocks for Test?

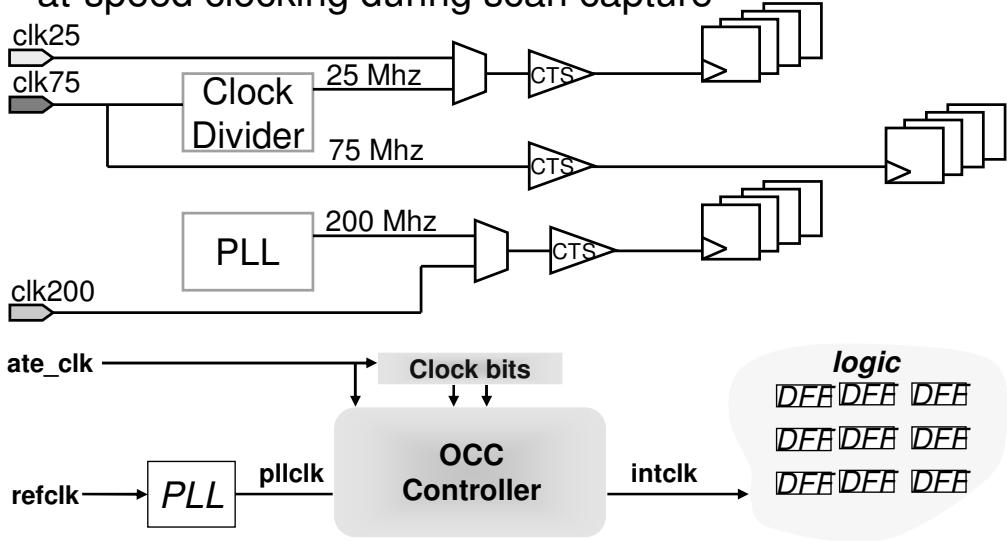


7-6

# Bypass or On-Chip Clocking (OCC)

- Internally generated clocks can be handled in one of two ways for Test

- Bypass internal clocks with an external clock
- Implement a clock controller to use the internal clocks for at-speed clocking during scan capture



7-7

This class doesn't cover implementation details for defining and inserting a On-Chip Clock (OCC) controller in DFT Compiler. That topic will be covered in a follow-on CES course, DFTC 2.

DFT Compiler supports On-Chip Clocking (OCC) for at-speed testing of delay defects. For more information on OCC consult SolvNet:

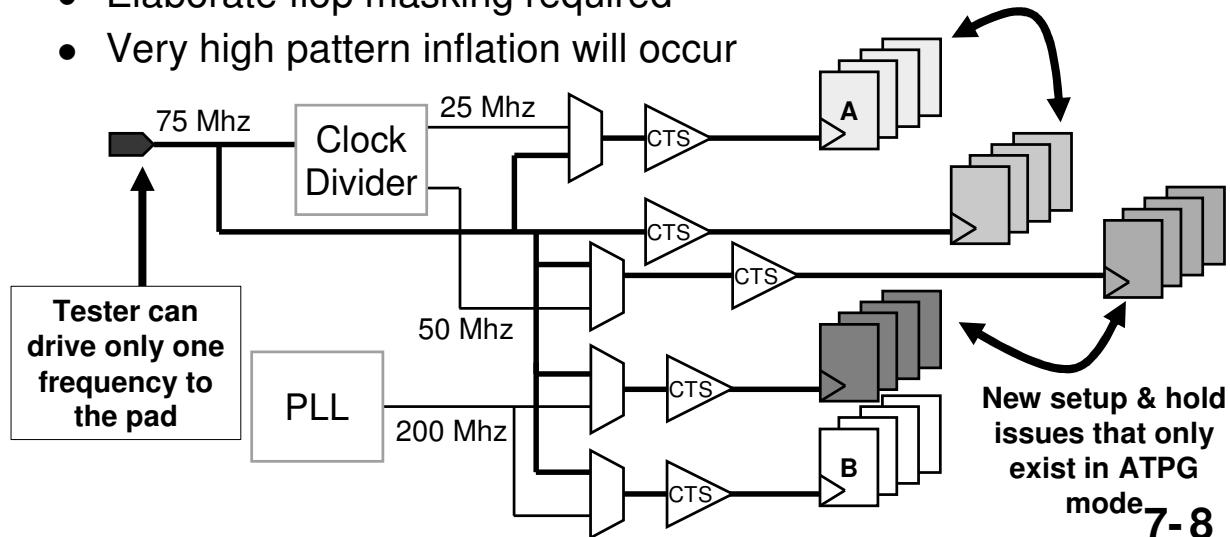
[https://solvnet.synopsys.com/dow\\_retrieve/A-2007.12/dftxg1/dftxg1\\_pll\\_support.html](https://solvnet.synopsys.com/dow_retrieve/A-2007.12/dftxg1/dftxg1_pll_support.html)

# Single Test Clock Issue

- Creates Scan specific hold time issues for shift and capture
  - For small geometries (250nm and below) process variation can make it impossible to close timing on a unified test clock

- At-speed ATPG becomes very difficult

- Elaborate flop masking required
  - Very high pattern inflation will occur



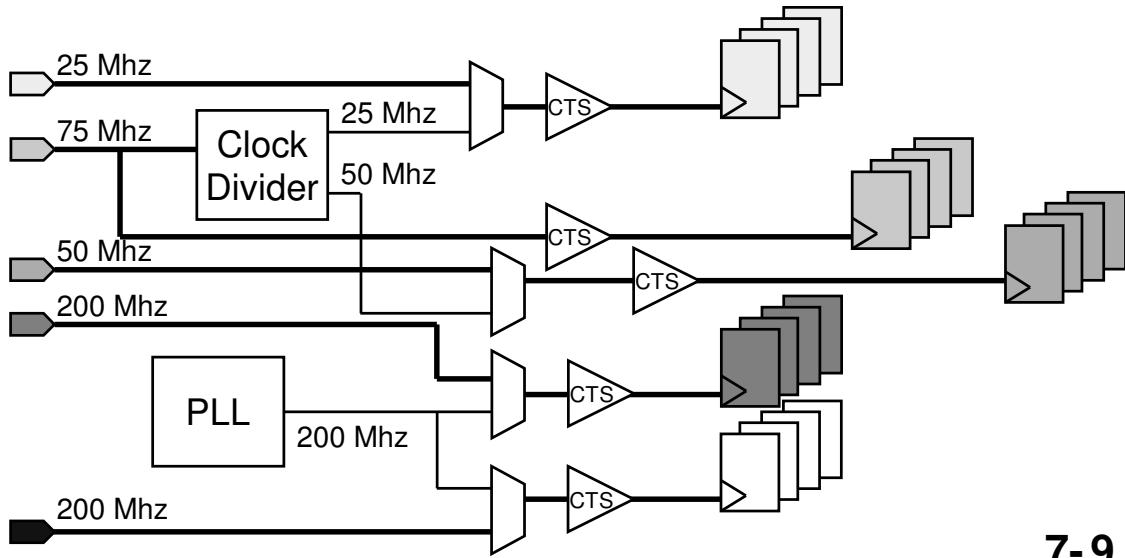
At larger geometries ( $>25\mu$ ) you could tie clocks together and get a pattern compaction advantage. However at smaller geometries, you cannot balance clock trees for hold if they communicate in both directions, and the non-common insertion delay is large enough.

As an example, if two clock trees each had a non-common insertion delay of 5ns, and an OCV of 5%, they would each vary by 250 ps, or 500 ps overall. If there is a hold margin in each direction  $<500$ ps (and there probably is), then it is impossible to fix timing by re-balancing the trees (all fixes would have to come from delaying data with buffers).

# Option 1: Ideal Solution

- Use separate top level ports for each CTS tree (if possible)

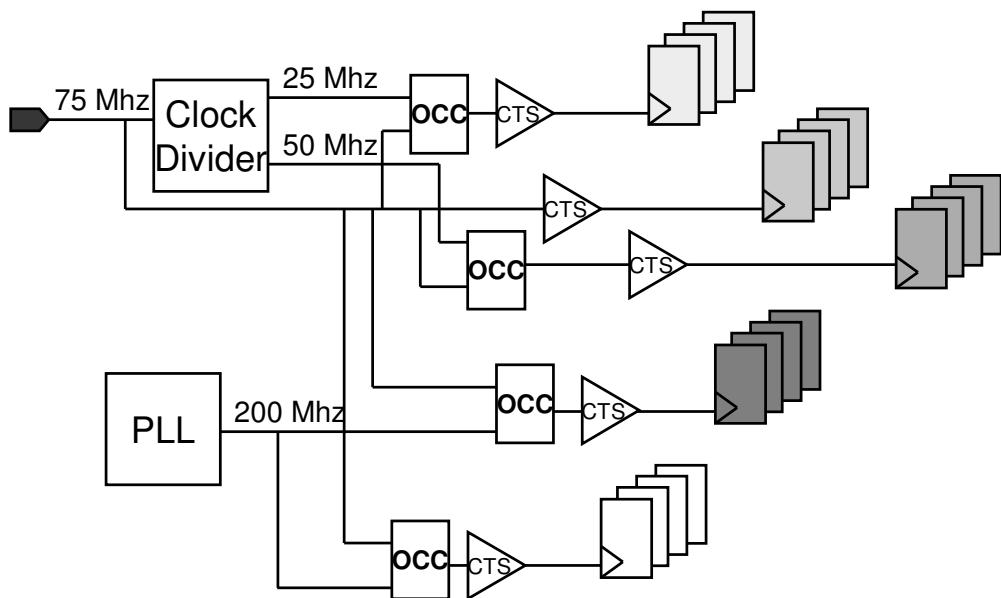
- Eliminates the need to fix setup/hold time issues between CTS trees that have different latencies
- Allows simultaneous testing of multiple frequencies



7-9

## Option 2: Multiple OCC Controllers

- Use multiple OCC Controllers to give ATPG separate control of each clock domain



7-10

# Related Clock Violations

Apply similar solutions to these related situations:

- On-chip PLL (phase-locked loop) clock generators
- Digital pulse generators—for example (A & delayed ~A)
- On-chip RAM with WRITE\_EN or WRITE\_CLK pins



## For More Information:

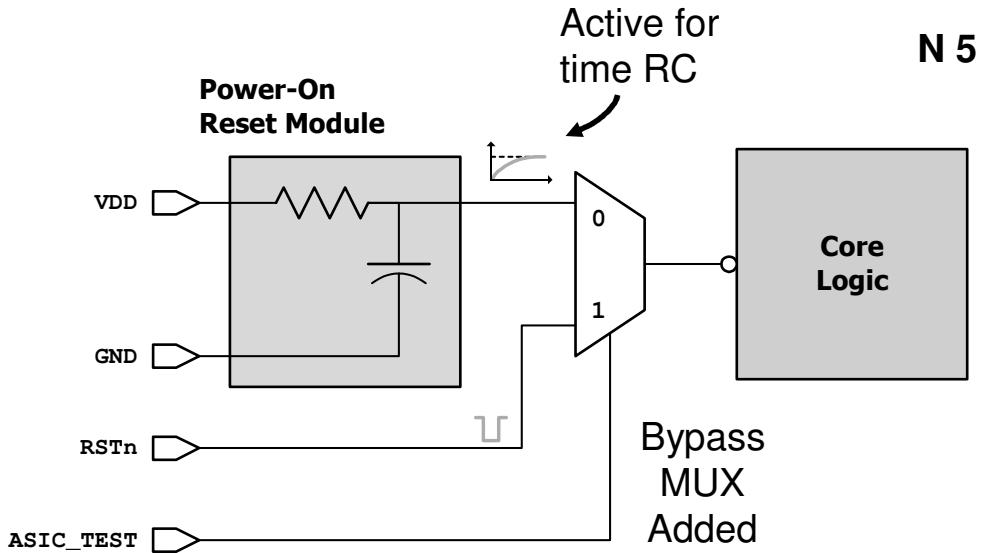
See the **ATPG Design Guidelines** appendix in the  
**Synopsys TetraMAX User Guide** for additional DFT  
violations and solutions

7-11

### On-Chip RAMs:

Embedded vendor specific memory blocks present many known testability problems. One issue is **random writes** to on-chip **RAMs** during the shifting in of serial scan data. Most ATPG tools do not simulate scan-shift, or keep track of the data written into **RAMs**. After the last scan-in cycle, the ATPG tool assumes all writeable RAM data is **XXX...X**. Add logic to **disable** or **bypass** **RAM** write-enables and clocks using **ASIC\_TEST** or **ScanEnable**. Using ScanEnable is preferred because it will disable random writes only during shift. The ability to “Scan through the RAM” during ATPG will be preserved.

# Power-On Reset Solution



- The reset recovery time cannot be controlled by ATE
- Bypass the power-on reset during entire test program
- Use an existing reset (RSTn) or other primary input

7-12

Power-on reset modules are generally implemented in one of two ways:

## Board-Level:

If **board-level** resistors and capacitors are used, this module is *not* in the DFTC netlist. If no such module appears in the gate-level netlist, then **RSTn** is just an input port. This *masks* a problem that will not show up in DFTC, but *might* resurface on the ATE.

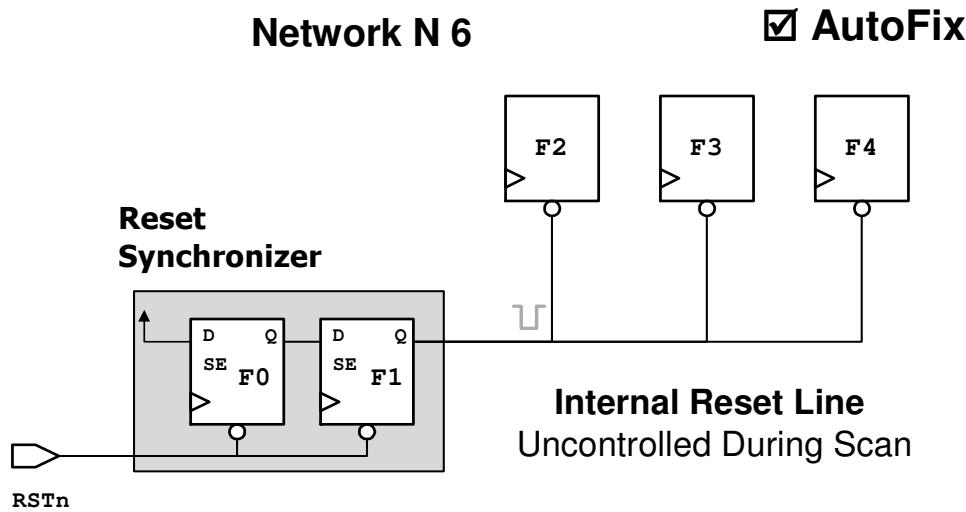
## On-Chip:

An on-chip **power-on reset** cell specific to a vendor lies **outside the scope** of DFTC. The **.lib** cell-library syntax read by DFTC cannot describe the function of such cells. If such a cell appears in the synthesized gate-level netlist, it is considered a **black box**. Black box cells have **no functional description** and generate **B5** DRC warnings. A simple way to spot black box cells in synthesized logic is **report\_cell** (attribute **b**).

## Conclusion:

- Add a bypass MUX as shown in N 5 to fix this problem—it is not addressed by AutoFix.
- It is a good idea to reserve an **ASIC\_TEST** pin for just such board-level contingencies!

# Internal-Reset Violation



- The problem is the uncontrolled asynchronous reset
- During scan, F1 can change state—resetting F2, etc
- All affected flip-flops are excluded from scan chains

7-13

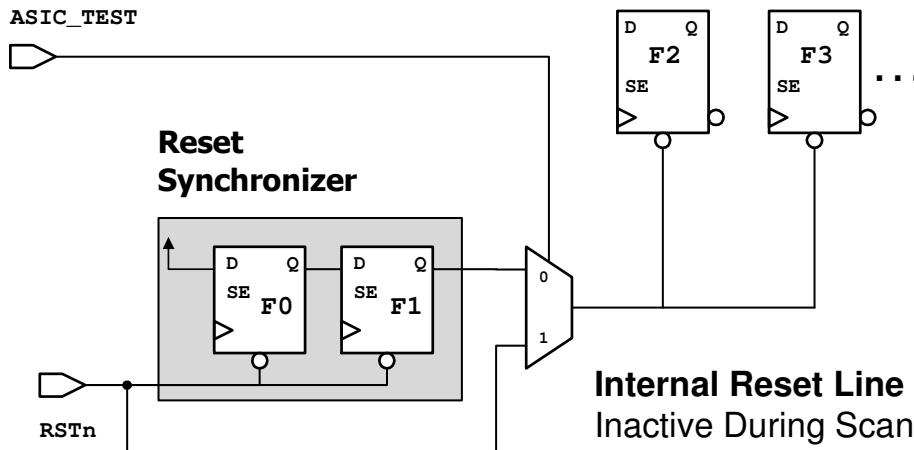
Risk-free scan requires **inactive** asynchronous set and reset signals during scan-shift. Network N 6 is a classic DRC violation which allows **F1** to reset flip-flops **F2**, etc. Unless the violation is fixed, all affected flip-flops are **excluded** from any scan path. If *many* flops are excluded, the impact of this violation is a **major** loss of coverage. The DFTC warning for this violation is **D2**, “asynchronous pin uncontrollable.” Use **AutoFix** as a fast workaround to add the correction logic during scan insertion.

## Initialization:

If the chip logic **can not be altered**, then the only solution is an **initialization** procedure. This is a classic example involving adding a **custom macro** to a STIL test-protocol file. The macro, called only once, initially clocks 1s into **F0** and **F1**, then holds **RSTn** high. With the uncontrolled-reset violation **fixed**, flip-flops **F2**, etc., all become **scannable**. This solution does **not** allow **F0** and **F1** to be on a scan path, because they would **toggle**. Another trade-off is that the internal reset line is held constant, and is thus **untestable**.

# Internal-Reset Solution

Network N 7



- Now all the flip-flops can be included in scan chains
- This implementation is what you get using AutoFix
- Muxing the **RSTn** provides the highest coverage

7-14

This violation has several solutions, each a tradeoff between DFT effort and coverage. The solution with **optimal coverage** is to multiplex **RSTn** around the synchronizer. This is now the default solution inserted by AutoFix. Faults on the internal reset line are thus fully **controllable**—but are they **observable**?

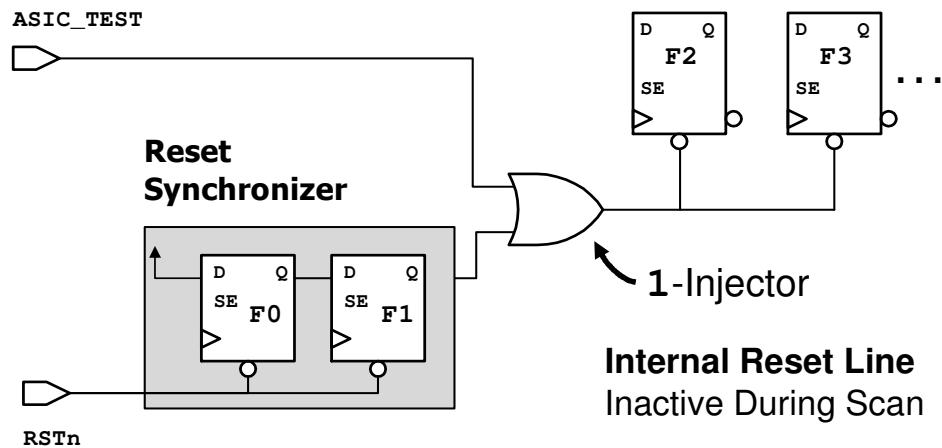
TetraMAX ATPG makes them observable by using **RSTn** in effect as a capture “clock.”

The reset synchronizer module itself is also fully **controllable**—but not **observable**.

As you will see later, you can readily fix this with an observe-only **test point** at **F1/Q**.

# Alternate Internal-Reset Solution

Network N 8



- Now all the flip-flops can be included in scan chains
- No longer the default implementation for AutoFix
- SA1 fault on internal reset line is still untestable

7-15

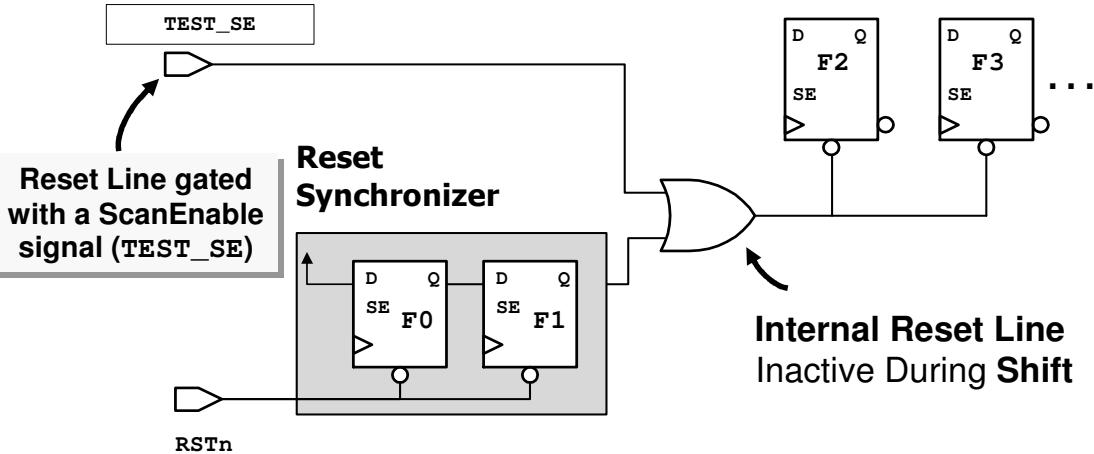
This violation has several solutions, each a trade-off between DFT effort and coverage. The solution in **N8** is implemented by AutoFix, but no longer implemented by default. With **ASIC\_TEST** constant, untestable faults remain—which could prevent resetting **F2**.

An alternative, with better coverage but more effort, is to use **ScanEnable** instead of **ASIC\_TEST**.

Both of these solutions allow flip-flops **F0**, **F1**, **F2**, etc., to be included in scan chains.

# Another Internal-Reset Solution

Network N 9



- All the flip-flops can be included in scan chains
- Full test coverage on the internal reset line
- Internal reset line is controlled by the Reset Synchronizer output during scan capture

7-16

With this solution, you may need to use the following commands:

To pass DRC in DFTC:

```
set_dft_drc_configuration -allow_se_set_reset_fix true
```

To pass DRC in TetraMAX:

```
set_drc -allow_unstable_set_reset
```

# DRC Fixing: Agenda

Fixing Common Violations

AutoFix Clocks/Resets/Sets

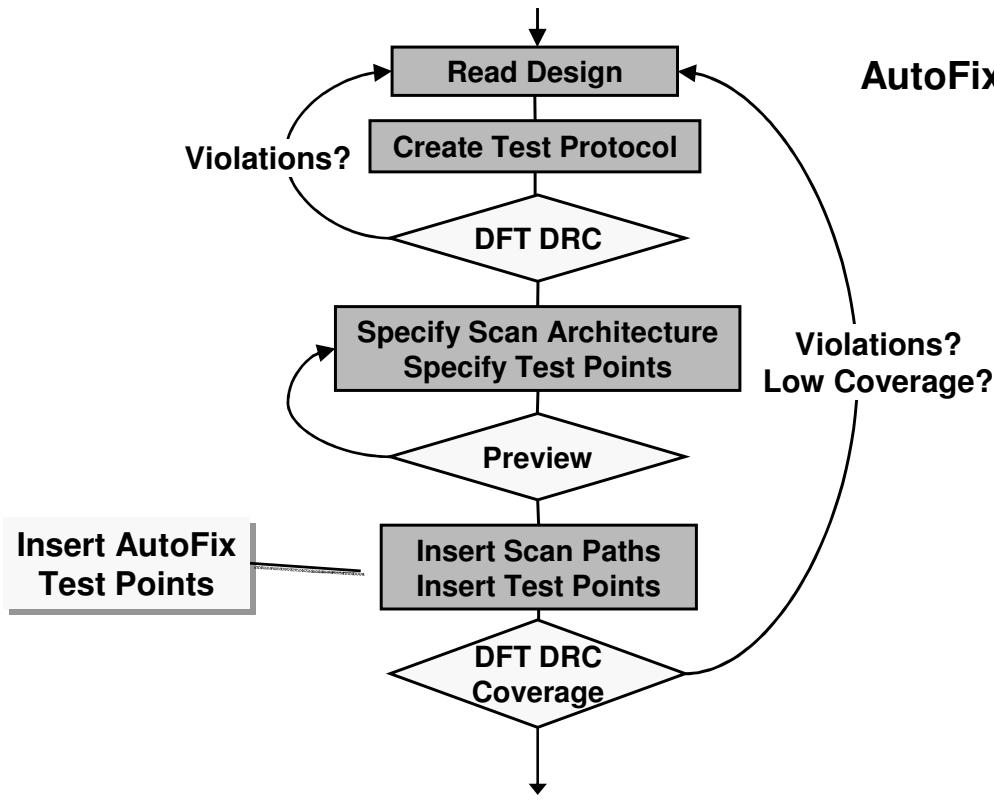
Internal Tristate Issues

AutoFix Internal Tristates

AutoFix Bidirectionals

7-17

# Typical AutoFix Flow



7-18

This flowchart is **identical** to one you saw earlier for gate-level DRC checking. The `insert_dft` command inserts and optimizes both **scan paths** and **test points**.

## Conclusion:

Use this flow to **globally AutoFix** the current design—including lower-level blocks. AutoFix corrects only the classes of violations indicated in networks N 1, N 3, and N 6. By default, **all** such violations will be corrected—**unless** you specify exclusions.

# When to Use AutoFix?

- **Modify the original RTL design to fix DFT violations whenever possible**
- **Sometimes modifying the RTL may not be an option**
  - Past code freeze
  - RTL not available
  - Not proficient at RTL coding
- **At the gate-level, AutoFix can solve the most important DFT problems:**
  - Uncontrollable clocks
  - Uncontrollable resets
  - Internal tristate buses
  - Bidirectionals

7-19

Note: AutoFix will not be able to performing logic fixing this cases:

- An explicit or implicit **dont\_touch** attribute is on the cell/design in question
- AutoFix can't fix logic issues inside of a library or leaf cell

# AutoFix for Uncontrolled Internal Clocks

- **set\_dft\_configuration**  
    **-fix\_clock enable | disable**
  - Enable or disable AutoFix feature to bypass uncontrolled internal clocks (default is **disable**)
  - Use **set\_ autofix\_configuration** to override default settings
- **set\_ autofix\_configuration**  
    **-type clock \**  
    **-method mux \**  
    **-control\_signal test\_mode \**  
    **-test\_data top\_clk**
  - Use **set\_ autofix\_element** to override global settings for specific clocks

7-20

The default fixing method for internal clock fixing is “mux”.

The **set\_ autofix\_configuration** command can be used to change this setting for a particular internal clock. The **-include\_elements** and **-exclude\_elements** options can be used for this. By default all internal clock violations are fixed when clock fixing is enabled.

The **set\_dft\_configuration** command has related commands **report\_dft\_configuration** and **reset\_dft\_configuration** which are used to report the current configuration settings and to reset the configuration back to defaults respectively.

The **set\_ autofix\_configuration** command has related commands **report\_ autofix\_configuration** and **reset\_ autofix\_configuration** which are used to report the current configuration settings and to reset the configuration back to defaults respectively.

# AutoFix for Uncontrolled Reset/Set

- **set\_dft\_configuration**
  - fix\_reset enable | disable**
  - fix\_set enable | disable**
    - Enables or disables the AutoFix feature to fix uncontrolled resets and/or sets (default is **disable**)
    - Use **set\_autofix\_configuration** to override default settings
- **set\_autofix\_configuration**
  - type <reset | set>**
  - method <mux | gate>**
  - control\_signal <test\_mode | scan\_enable>**
  - test\_data <bypass\_signal>**
    - Use **set\_autofix\_element** to override global settings for specific resets and/or sets

7-21

The default fixing method for reset/set fixing is “mux” (with a **TestMode** “control\_signal”).

The **set\_autofix\_configuration** command can be used to change this method and/or control for reset/set fixing. The **-include\_elements** and **-exclude\_elements** options can be used for applying AutoFix specifications to particular uncontrolled reset/set(s). By default all uncontrolled reset/set violations are fixed when reset/set fixing is enabled (with separate enables for reset and set fixing).

# Local Control of AutoFix

- **set\_autofix\_element**  
`<list_of_design_objects>`  
`-type <clock | reset | set>`  
`-method <mux | gate>`  
`-control_signal <test_mode | scan_enable>`  
`-test_data <bypass_signal>`
- The **set\_autofix\_element** is used to override global settings for specific clocks, resets and/or sets
- For example:

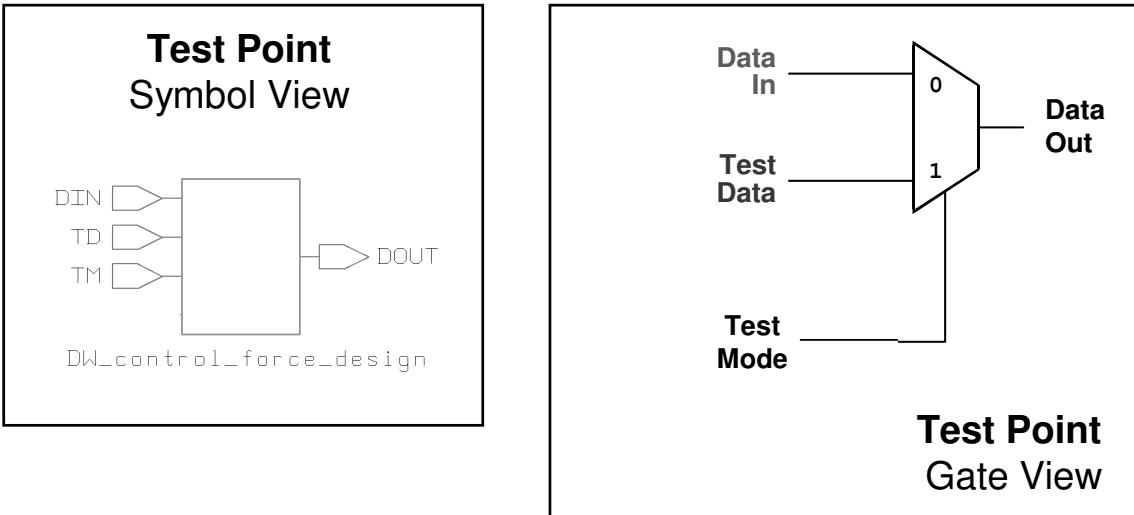
```
set_autofix_element [get_cells *myregs*] \
    -type reset -method mux
    -control_signal test_mode -test_data ext_rst
```

7-22

The **set\_autofix\_element** command has related commands **report\_autofix\_element** and **reset\_autofix\_element** which are used to report the current AutoFix element settings and to reset the AutoFix element settings back to defaults respectively.

Refer to the man page of the **set\_autofix\_element** command for information regarding which switches are available with each AutoFix “type”.

# Logic Added by AutoFix

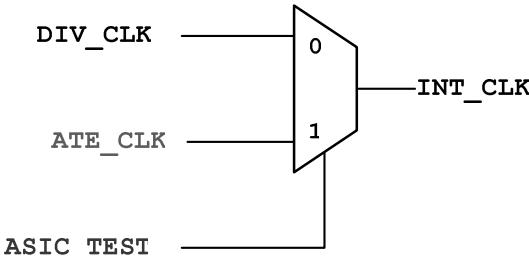


- AutoFix adds a configurable test-point architecture
- Exact implementation can vary during optimization

7-23

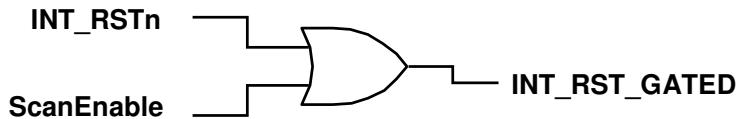
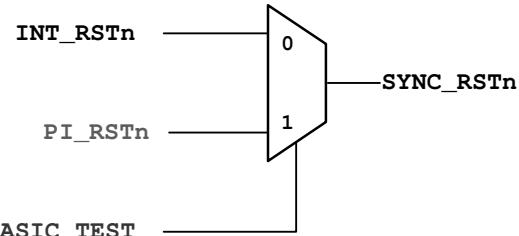
This slide shows the **test-point** architecture that is inserted by the **AutoFix** client. AutoFix test points are **configurable**—they behave much like **DesignWare** parts. For example, you can specify that **ASIC\_TEST** should drive all **TestMode** inputs. If left unspecified, AutoFix **default behavior** is to synthesize a **TestMode** type input signal. Gate-level test-point implementation is **optimized** during the **insert\_dft** step.

# Examples of AutoFix Test Points



**AutoFix MUXing**  
for Divided Clock  
or Asynch

**AutoFix Reset MUX**  
default option  
for Uncontrolled Reset-Bar



**AutoFix Reset Gate**  
-method gate

**7-24**

This slide has two optimized implementations of the AutoFix test-point architecture. In both cases, the user specified that **ASIC\_TEST** should drive the **test-mode** input.

## Conclusion:

Test-point architecture is configurable, depending in part on the user's specifications. Gate implementation can also vary with the synthesis constraints, mapping effort, etc.

# Simple AutoFix Script (1/2)

## DFTC Script

```
# Turn on AutoFix and fix all clock
#           and set/reset violations.

# Global command; applies to current design.

# Then report the present DFT configuration.

current_design ORCA

set_dft_configuration -fix_clock enable \
                      -fix_set enable -fix_reset enable

report_dft_configuration

# SET AutoFixing of CLOCKS:

set_dft_signal -view exist -type Constant \
               -active 1 -port ASIC_TEST

set_dft_signal -view spec -type TestMode -port ASIC_TEST
set_dft_signal -view exist -type ScanClock \
               -timing {45 55} -port CLK
set_dft_signal -view spec -type TestData -port CLK
set_automodify_configuration -type clock \
                             -control ASIC_TEST -test_data CLK

. . .
```

Fix All Clock, Set and  
Reset Violations  
in Current Design

Specify Signals  
Used to Fix  
Clocks

7-25

- For full details on commands, see the man pages.

### A few key insights:

#### **set\_dft\_configuration:**

The DFTC AutoFix tool supports several features: **clock**, **set**, **reset**, **xpropagation**, **bus** and **bidirectional**. By default, *bus* and *bidirectional* features are enabled, and all other AutoFix features are *disabled*.

#### **set\_dft\_signal:**

This command sets a constraint on **insert\_dft**. In this case, the constraint is to use existing port **ASIC\_TEST** as the test-mode signal. If **ASIC\_TEST** is not a port but a test-control block output pin, use option **-hookup\_pin**. Without this constraint, **insert\_dft** will synthesize a **new port** named **test\_mode**.

This script and command usage notes are continued on the next slide.

# Simple AutoFix Script (2/2)

AutoFix Logic  
Inserted Here

```
# SET AutoFixing of Sets and Resets:  
set_dft_signal -view exist -type Reset \  
    -active 0 -port RST_N  
set_dft_signal -view exist -type Reset \  
    -active 0 -port SET_N  
set_dft_signal -view spec -type TestData -port RST_N  
set_dft_signal -view spec -type TestData -port SET_N  
set_automation_script -type set \  
    -control ASIC_TEST -test_data SET_N  
set_automation_script -type reset \  
    -control ASIC_TEST -test_data RST_N  
  
# INSERT AutoFix LOGIC:  
# Uses dft_drc to spot fixable violations..  
create_test_protocol -capture_procedure multi_clock  
dft_drc  
preview_dft -test_points all > reports/autofix.pts  
insert_dft  
dft_drc -coverage  
# AutoFix done.  
# Details are in preview_dft -test_points all transcript.
```

Specify Signals  
Used to Fix Sets  
and Resets

7-26

For full details on commands, see the man pages.

A few key insights:

**set\_automation\_script:**

- Identifies an **ATE-controllable** clock to use in fixing uncontrolled clock violations.

Without this constraint, **insert\_dft** will look for an **existing** primary-input clock. If it finds none in the relevant input logic cone, **insert\_dft** will **create** a clock port. Typically, the list of cells is exhaustive—for example, **[all\_registers -clock CLK2]**. In effect, this command sets an **autofix\_clock** attribute on each specified cell.

To undo this setting, use: **reset\_automation\_script**.

# Preview of Test Points

## DFT Preview

Two of Nine  
Test Point  
Excerpts

Existing  
Test-Mode  
Port Used

```
dc_shell> preview_dft -test_points all
*****
Test Point Plan Report *****
.

Number of Autofix test points: 9
Number of test modes : 1
Number of data sources : 4
*****
TEST POINTS
-----
CLIENT      NAME      TYPE      LOCATIONS
-----
Autofix      test_point    F-01    ANGLE_reg[0]/CP
                                         ANGLE_reg[1]/CP
                                         ANGLE_reg[2]/CP
                                         ANGLE_reg[3]/CP
.
.
.
Autofix      test_point_10   F-1     ANGLE_reg[0]/CDN
                                         ANGLE_reg[1]/CDN
                                         ANGLE_reg[2]/CDN
                                         ANGLE_reg[3]/CDN
.
.
.
TEST MODES
-----
TAG          NEW/EXIST    TYPE      NAME
-----
handler_12   Existing    Port     ASIC_TEST
.
```

Clock Pins  
To Be Fixed

Asynch Pins  
To Be Fixed

*For clarity, this transcript has been abbreviated*

7-27

For full details on commands, see the man pages.

A few key insights:

**preview\_dft:**

The above data is generated **prior to** scan insertion by the **preview\_dft** command. By default, only the starred summary block, **Test Point Plan Report**, is shown. Use the **-test\_points all** option to print out **all** the details on the two test points.

**report\_dft\_configuration:**

- There is *no* reporting option for post-insertion AutoFix logic; refer to the previewed data.
- The **report\_dft\_configuration** option lists current configuration settings and enabled clients.

**reset\_dft\_configuration:**

- Undoes all settings specified earlier by **set\_dft\_configuration**.
- Resets the current design to default DFT behavior.

# Local Exclusion with AutoFix

## Optional Exclusions

```
# SET AutoFixing of Sets and Resets:  
.  
set_dft_signal -view spec -type TestData -port RST_N  
set_ autofix_configuration -type reset \  
-control ASIC_TEST -test_data RST_N  
  
# Exclude block U1/U2 from all AutoFixing.  
# Accepts full hierarchical instance names.  
# Any lower-level settings override higher.  
set_ autofix_configuration -type clock -exclude U1/U2  
set_ autofix_configuration -type reset -exclude U1/U2  
  
# INSERT AutoFix LOGIC:.  
# Excludes those flip-flops in block U1/U2.  
create_test_protocol -capture_procedure multi_clock  
dft_drc  
preview_dft -test_points all > reports/autofix.pts  
insert_dft
```

7-28

For full details on the **set\_ autofix\_configuration** command, see the man pages.

# General Test Point Insertion

- The AutoFix client can be used for specific types of logic fixing
- Other logic fixing requirements can be handled by User Defined Test Points (covered in the Appendix)

7-29

# DRC Fixing: Agenda

Fixing Common Violations

AutoFix Clocks/Resets/Sets

Internal Tristate Issues

AutoFix Internal Tristates

AutoFix Bidirectionals

7-30

# Identifying Tristate DFT Issues

## A. Undetectable Faults:

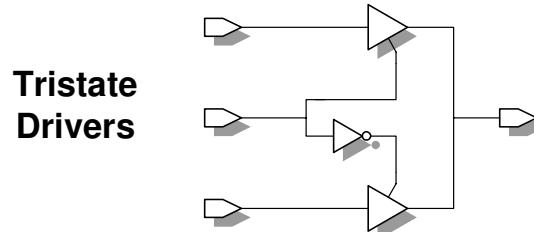
You will need added DFT logic—or ATE features—to achieve 100% coverage

## B. Contention During Scan Shift:

Scanning in a stream of 0s and 1s can enable tristate drivers at random

## C. Contention During Capture:

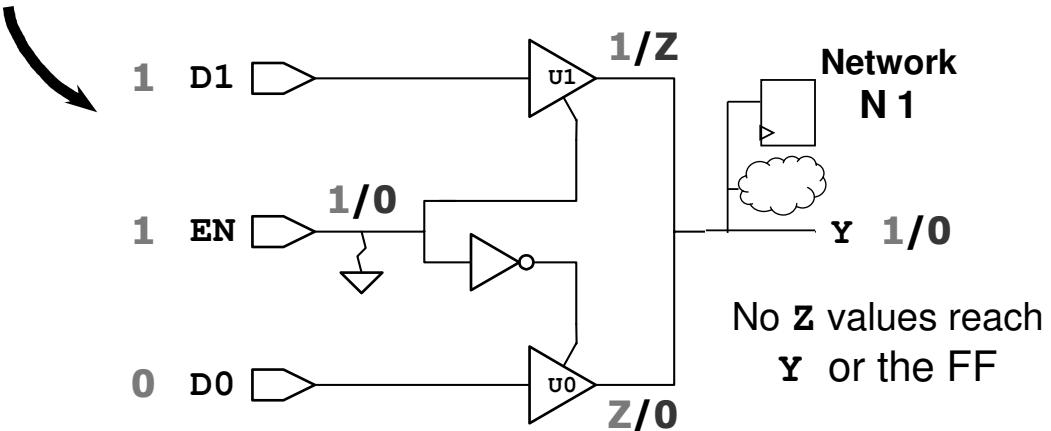
Changes in state at the capture-clock edge can enable drivers at random



7-31

# Some Tristate Faults are Testable

The SA0 fault on port **EN**  
is detected by pattern **011 1**



## Conclusion:

Some faults on tristate drivers are readily detectable

7-32

About 80% of the faults in tristate network **N 1** are **detectable** by the **D**-algorithm.

For example:

- TetraMAX can detect the **SA0** fault at **EN** using the pattern **(011 1)**.
- You will use this informal notation as shorthand for: (**D0 = 0**, **D1 = 1**, **EN = 1**, **Y == 1**).

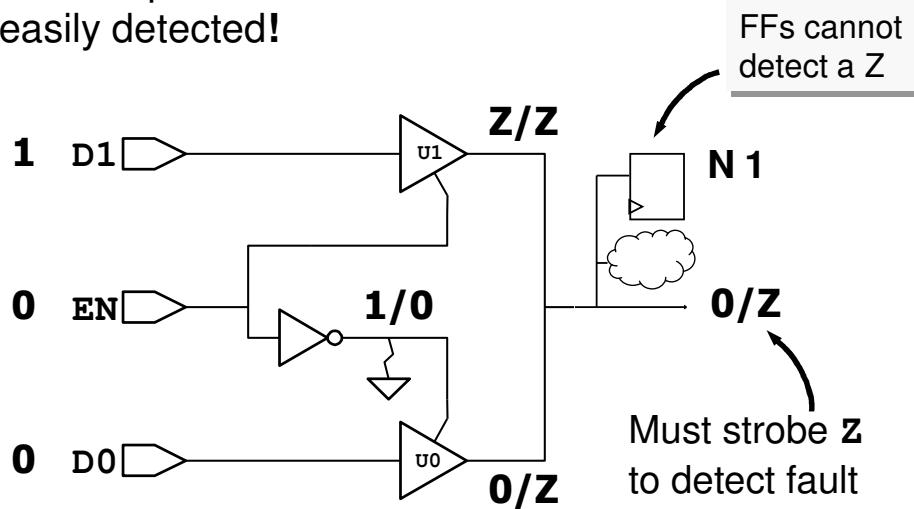
This test pattern works because a weak **Z** value at **Y** is **overridden** by a strong **0** or **1**.

The **difference** between fault-free and faulty outputs (**1 / 0**) is therefore **measurable**.

As you will see, this is *not* true for *all* possible stuck-at faults in tristate logic networks.

# An Undetectable Tristate Enable Fault

The SA0 fault on pin **U0/E**  
is not as easily detected!



## Conclusion:

But faults on tristate enables are often undetectable

7-33

In this slide, the SA0 fault was moved to a different site in the same network, **N 1**. The new fault site is the **enable** pin **U0/E**—or, equivalently, the inverter output pin. To *activate* this fault, you have to apply stimulus **EN = 0**, which *disables* driver **U1**. The resulting TetraMAX generated pattern to detect the SA0 fault is: **(010 0)**. In the presence of the SA0 fault, the response at primary output **Y** will always be **Z**.

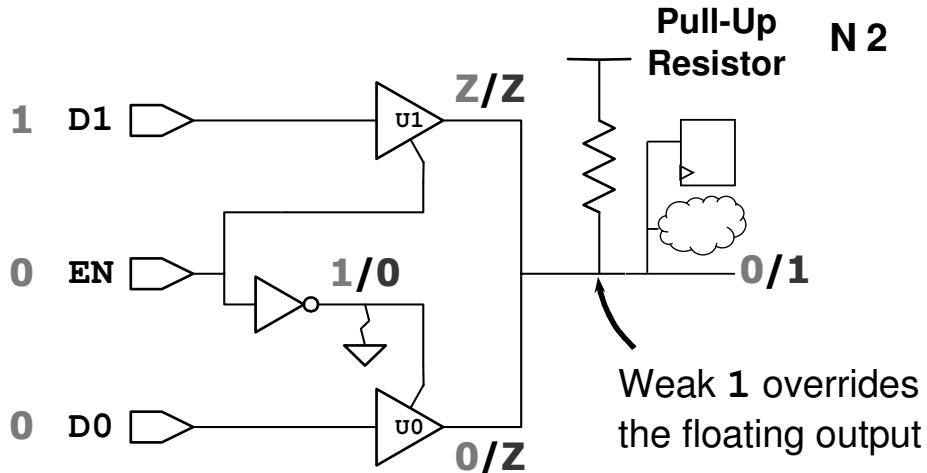
If **Y** was a primary output pin, some ATE hardware is capable of observing high-impedance **Z** outputs. Older or reduced-cost ATE hardware is not—rendering this fault **undetectable**.

## Conclusion:

If the target ATE can observe **Z** values, the faults in network **N 1** are 100% **detectable**. If not, faults on tristate enable pins are **undetectable**, lowering coverage to about 80%. Design and test engineers should decide early whether **Z** detection is a viable option. Note that if **Y** is not a **primary output**, as shown here, then **Z** detection is **not** an issue.

# Adding a Pull-Up Resistor

With added DFT logic, a floating bus is resistively pulled up to a weak 1



The SA0 fault at **U0/E** is now detectable in the usual way

7-34

Ad hoc DFT—adding testability hardware to a network—can improve coverage.

This slide shows a modified network, **N 2**, with an instantiated **pull-up resistor**. The resistor prevents output **Y** from ever floating to **Z**—pulling it up to a weak 1. Applying the same pattern (**010 0**) to network **N 2** will now detect the SA0 fault. Coverage increases to 90%, but a few undetectable faults are left on enable pins.

## Caveat:

This solution is **not recommended**.

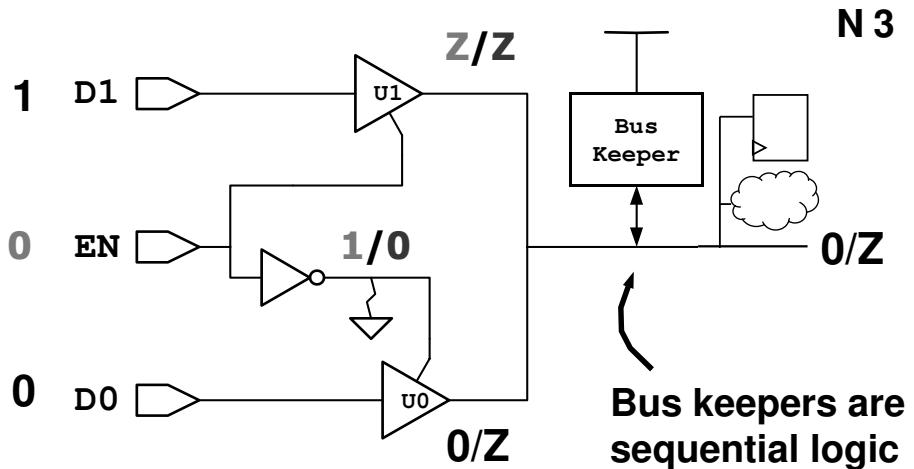
The resistor is **very slow**, possibly taking several clock cycles to pull up output **Y**. Worse, the resistor is not pure CMOS, and draws **static current** whenever **Y** is **0**. Unless you add cut-off transistors, this impacts low-power design and  $I_{ddq}$  testing.

## Conclusion:

Adding a pull-up resistor improves coverage on tristate buses by roughly 10%. This is an effective textbook illustration of ad hoc DFT, but has severe penalties.

# Adding a Bus Keeper

A keeper latches the *last* 0 or 1 on the bus overriding any z with a weak stored value



If you activate the target fault, the bus will float;  
thus you need to initialize the bus keeper first

7-35

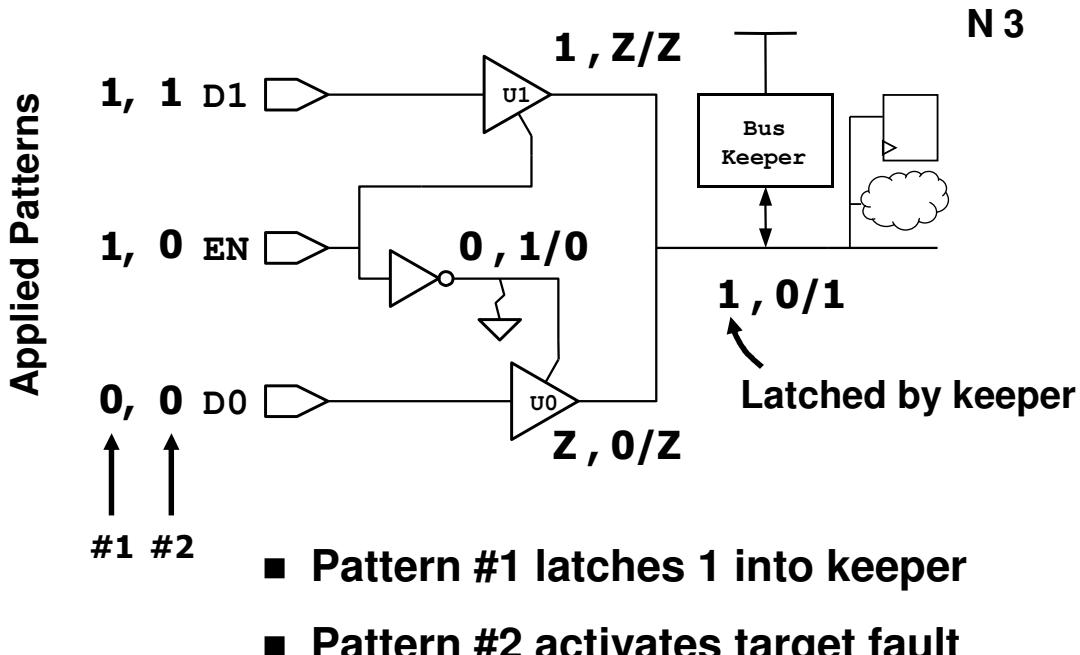
This slide shows a more complex ad hoc DFT modification to tristate network **N 3**. A sequential-logic primitive called a **bus keeper** or **holder** has been instantiated. A bus keeper acts like a one-bit, asynchronous, always-write-enabled SRAM cell. At the transistor level, it is a cross-coupled inverter pair with a **weak output** drive. Like the resistor, a keeper **prevents bus float** by driving the PO to a weak **0** or **1**. Unlike the resistor, the keeper is pure CMOS logic, and consumes **no static power**. But—it is **sequential logic**, and cannot be handled by **combinational ATPG tools**.

## Conclusion:

The bus-keeper solution is recommended by Synopsys, especially for TetraMAX. TetraMAX's fast- and full-sequential ATPG capabilities reduce the impact of this approach.

# ATPG with a Bus Keeper

A two-pattern sequence, generated by sequential ATPG, detects the fault:



7-36

Sequential ATPG is required in this case to utilize the bus keeper's storage capability. TetraMAX fast-sequential ATPG generated a two-pattern sequence to detect the fault. The sequence is: (011 X), (001 0) where X indicates a that the value latched by the keeper does not need to be observed for detection of the target fault.

## Dropped Faults:

TetraMAX ignores two potential faults on the bidirectional I/O pin of the bus keeper. This is a small drop in coverage for a large chip, even one with several bus keepers.

For more information on “dropped” faults, see TetraMAX **Help** for DRC rule **M28**.

## Conclusion:

Adding a bus keeper provides better than 90% coverage on tristate bus networks, but sequential ATPG is required, which can slow down test-program generation.

# DFT Improves Tristate Coverage

How to Boost Tristate Bus Coverage			
Added DFT Logic	Faults Detected	Fault Coverage	Drawbacks
None ( $N 1$ )	16/20	80%	Lowest fault coverage.
Pull-Up Resistor ( $N 2$ )	18/20	90%	Dissipates static power.
Bus Keeper ( $N 3$ )	18.5/20	92.5%	Needs sequential ATPG.



## What Synopsys Test Specialists Recommend:

For tristate logic, use bus keepers and Fast Sequential ATPG for high coverage in reasonable run times

7-37

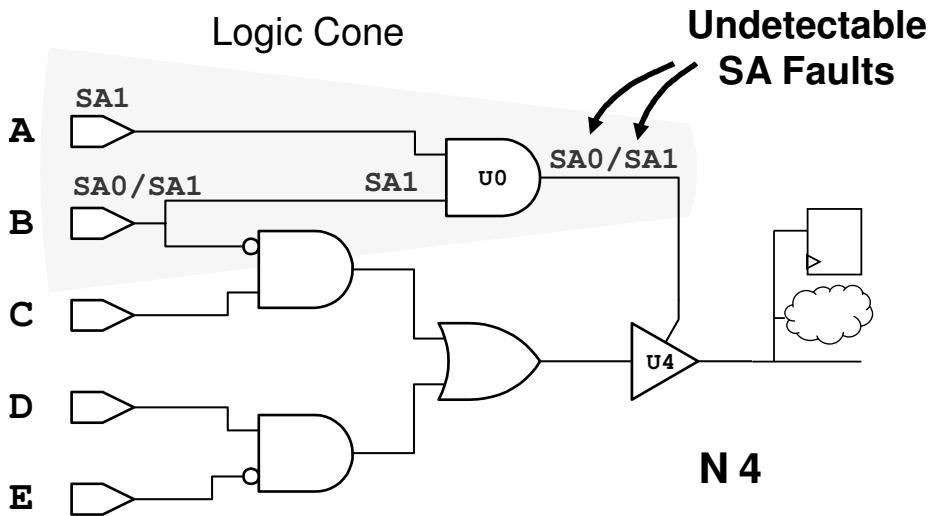
TetraMAX was used to generate the fault statistics in the above table, as follows:

- Network **N 1** has **20** potential SA faults, counting two tristate buffers and four IO ports.
- For simplicity, the inverter was *not* an actual cell in the Verilog netlist, and is not counted.
- TetraMAX classifies *some* faults as **possibly-detected**, assigning them *half-credit* by default.
  - This occurs in a few cases, and explains the *fractional* coverage figures in the above table.

Synopsys' Test Specialist ACs recommend the following **tristate strategy** for SoCs:

Wherever possible, replace internal tristate buses with ordinary **multiplexed** buses; elsewhere, use **bus keepers** and sequential ATPG to bring coverage close to 100%.

# Another Low-Coverage Case



- Applying stimuli to detect some faults will disable driver U4
- No other path exists to propagate fault effects to an output
- Most faults in the logic cone to U4/E are thus undetectable

7-38

Testability problems with tristate logic are by no means limited to bus structures.

Network **N 4** represents combinational **gate logic** driving a tristate output. For **42** total faults, assuming no **Z** strobing, TetraMAX achieved only 66% coverage for **N 4**. The low coverage is traced to 12 ATPG-untestable faults (TetraMAX code **AN**).

The slide shows these **AN** faults after collapsing, which leaves 6 nonequivalent faults. Code **AN** means that *under current ATPG conditions* a stuck-at fault is **not** detectable. The ATPG condition imposed here on **N 4** is that *no* strobing for **Z** values is allowed. TetraMAX therefore classifies these faults as **AN**, instead of just **undetectable**.

## Conclusion:

The only way to increase observability of these **AN** faults is to add **DFT** hardware. Instantiating a **bus keeper** at output **Y** may not be practical for an output pad cell. You could instead insert a noninvasive “observe-only” **test point** at pin **U0/Y**. You will see later on how to use **DFTC** to synthesize such test points automatically.

# Identifying Tristate DFT Issues

## A. Undetectable Faults:

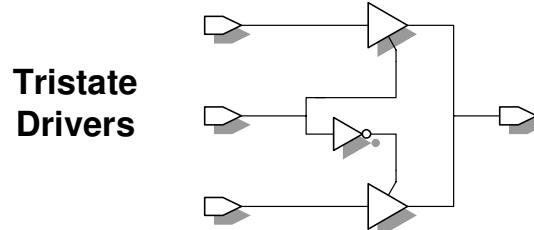
You will need added DFT logic—or ATE features—to achieve 100% coverage

## B. Contention During Scan Shift:

Scanning in a stream of 0s and 1s can enable tristate drivers at random

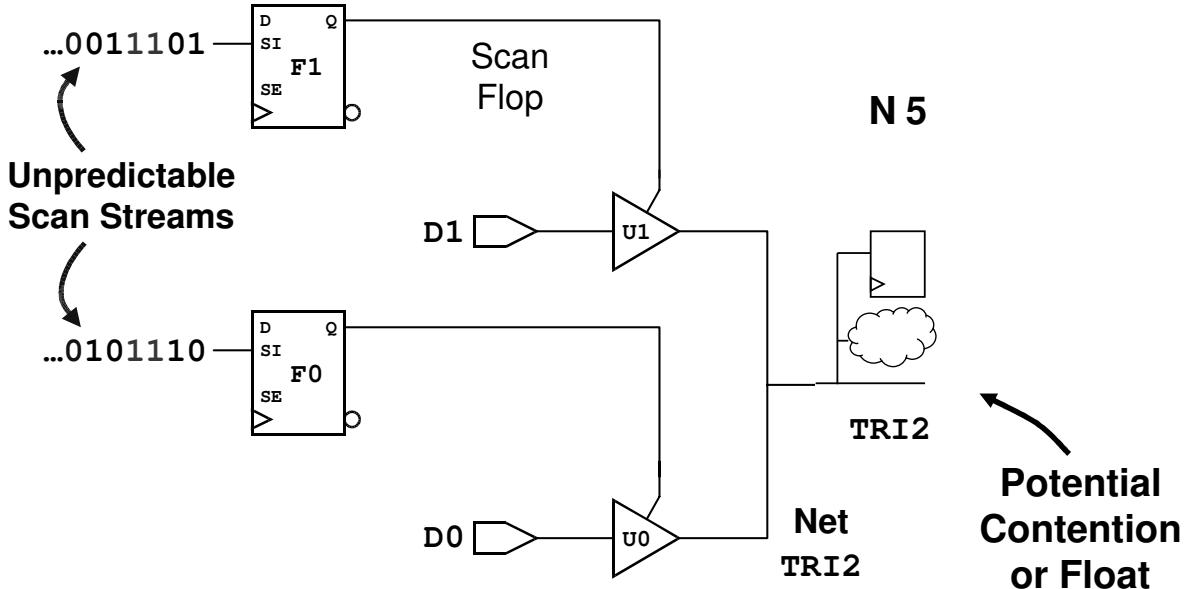
## C. Contention During Capture:

Changes in state at the capture-clock edge can enable drivers at random



7-39

# Contention in Scan Shift (1/3)



**The issue here is not low coverage—but potential damage!**

7-40

Tristate buses are subject to problems during scan-shift cycles because:

- The stream of bits shifted through scan paths for each test pattern is **unpredictable**.
- Scan flip-flops may, directly or indirectly, control tristate-driver enable lines, as in N5.
- These drivers may be **simultaneously enabled**—or all disabled—from cycle to cycle.

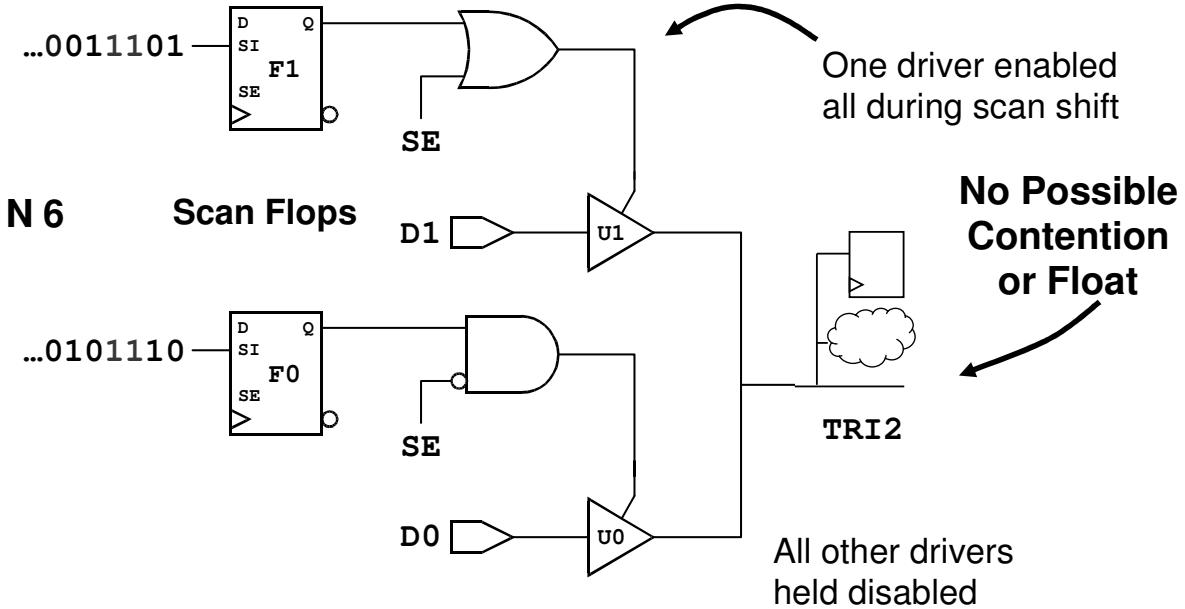
## Problems with Bus Contention:

Enabling **U0** and **U1 simultaneously** can turn on a pull-up **and** a pull-down transistor. Static current through net **TRI2** between them can cause **noise** or **reliability** problems. The **excess noise** can disturb nearby scan cells, causing spurious failures at the ATE. If excess current persists over clock cycles, the metal line could potentially **open up**.

## Problems with Bus Float:

Disabling **all** drivers lets **TRI2** float to a **Z** value—not damaging, but undesirable. If the **Z** value propagates directly to a PO, that port must be **masked (X)** by the ATE. If the **Z** value feeds a logic gate, the gate **output** is **X**, which propagates outwards. In either case, more PO ports have to be masked, tying up ATE per-pin resources.

## Contention-Free Scan (2/3)



### STD Rule:

Enable a single tristate driver per bus, during scan shift

7-41

The first solution resolves both contention and float on tristate buses during scan shift.

### Solution (1):

Use **SE** or another PI to **globally override** all internal tristate enables during scan shift; thus, a **single tristate driver** in N 6 is enabled while **SE** is high—the rest are disabled. At other times, including capture cycles, chip function is unaffected by the added logic. Instead of **SE**, some designers use a **separately-constrainable** input (e.g., **TRI\_SE**). Adding the **1**-injector (OR) gate and **0**-injector (INHIBIT) gates can be done two ways.

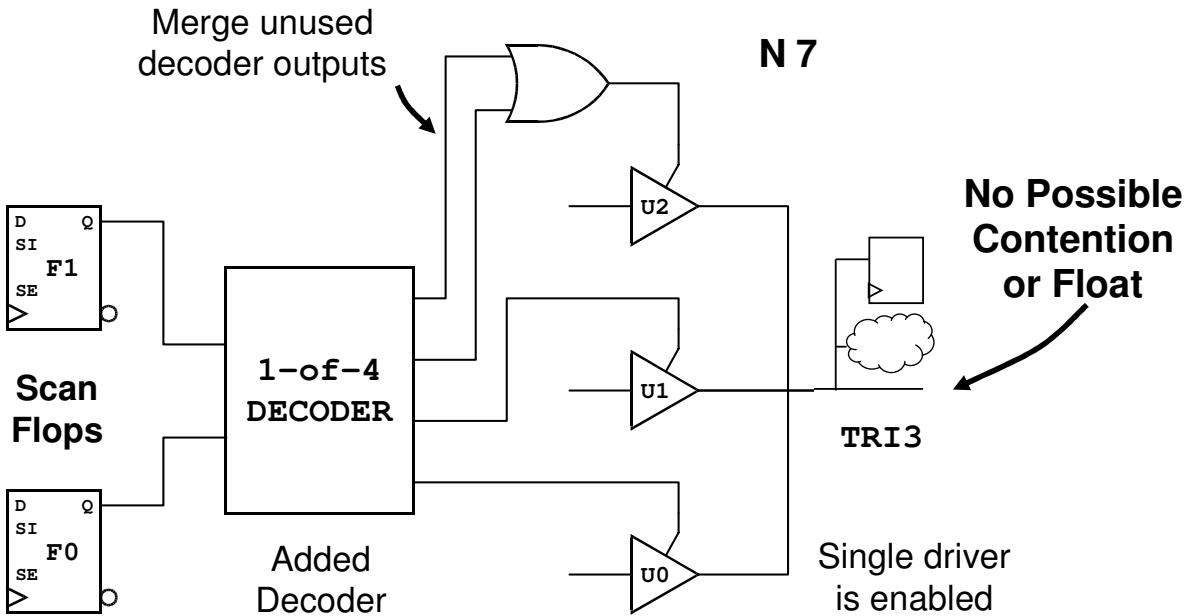
### HDL Method:

It is always best to describe this injection logic in your HDL code, and synthesize it. This allows **compile** to minimize the timing and area impact of the added DFT logic.

### DFTC Default:

By default, **insert\_dft** adds this disable/enable logic to **all** tristate-bus drivers. To explicitly restore this default: **set\_dft\_configuration -fix\_bus\_enable**.

# Contention-Free Scan (3/3)



## Driver-Enable Decoder:

Fully decode the tristate enables (1-of- $n$  hot) for each bus

7-42

The third solution also resolves contention and float, by decoding the driver enable lines.

### Solution (3):

Synthesize the 1-of- $n$  decoding logic from HDL code - **insert\_dft** cannot do this task. The 1-of- $n$  decoder ensures the DUT **always** satisfies the single-tristate-driver (STD) rule. The OR gate keeps one driver enabled, even if an **unused** input (**11**) occurs during scan. Beware of inefficient HDL constructs (such as **for** loops), especially for *wide* decoders. For VHDL, an efficient syntax for decoding 4 or more inputs is:

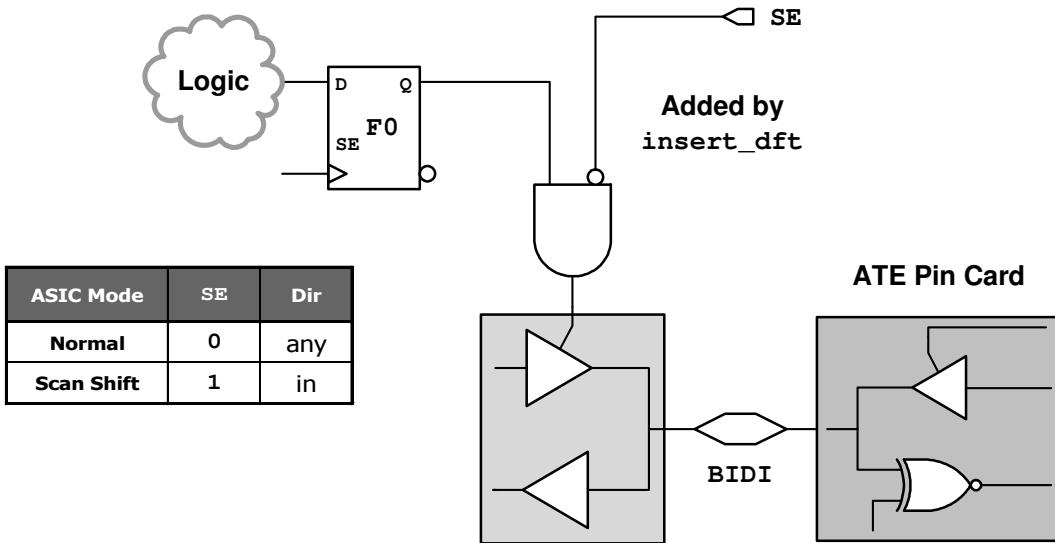
```
DEC_OUT <= (others <= '0');
DEC_OUT( CONV_INTEGER(DEC_IN) ) <= '1';
```

### Caveats:

Solution (3) may lose attractiveness if the bus has *too many* tristate drivers ( $n > 128$ ). The decoding logic may become prohibitive in terms of gate area or propagation delay. The **insert\_dft** algorithm does only **limited checking** for tristate-enable decoding. If you use this solution for net **TRI3**, prevent the addition of any disable/enable logic by insert DFT.

# Controlling Bidi Direction for Scan Shift

N 8



## Scan-Shift Control:

Direction is fixed during scan shift, but not during capture

7-43

This slide shows the simplest solution, which resolves contention for scan shift only. The added injection logic forces bidirectional pins into **input mode** during scan shift. Forcing **output mode** is also acceptable, provided board-level contention is no issue. This solution is implemented **automatically** during normal scan insertion, by default. The **insert\_dft** command will add this **disable/enable** logic for all tristate output enable of bidirectional pins.

To establish this default, specify:

```
set_dft_configuration -fix_bidirectional enable
```

To specify **input** direction:

```
set_ autofix_configuration -type bidirectional -method input
```

This setting will be applied **globally** to all the bidirectional ports in the current design.

To override this locally:

```
set_ autofix_configuration -type bidirectional \
    -method output -include BIDIS
```

If you describe the **disable/enable** logic in the HDL code,

```
set_dft_configuration -fix_bidirectional disable
```

# Identifying Tristate DFT Issues

## A. Undetectable Faults:

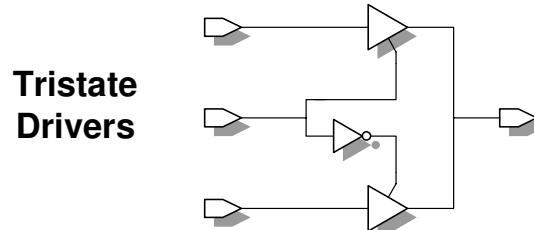
You will need added DFT logic—or ATE features—to achieve 100% coverage

## B. Contention During Scan Shift:

Scanning in a stream of 0s and 1s can enable tristate drivers at random

## C. Contention During Capture:

Changes in state at the capture-clock edge can enable drivers at random



7-44

# **Contention During Capture**

**How can you avoid contention during capture cycles,  
along with associated noise or reliability problems?**

Solution 1:

Let TetraMAX discard any patterns causing contention

Solution 2:

Use ASIC\_TEST instead of SE to control tristate enables,  
as in N 6

This lowers coverage but eliminates contention

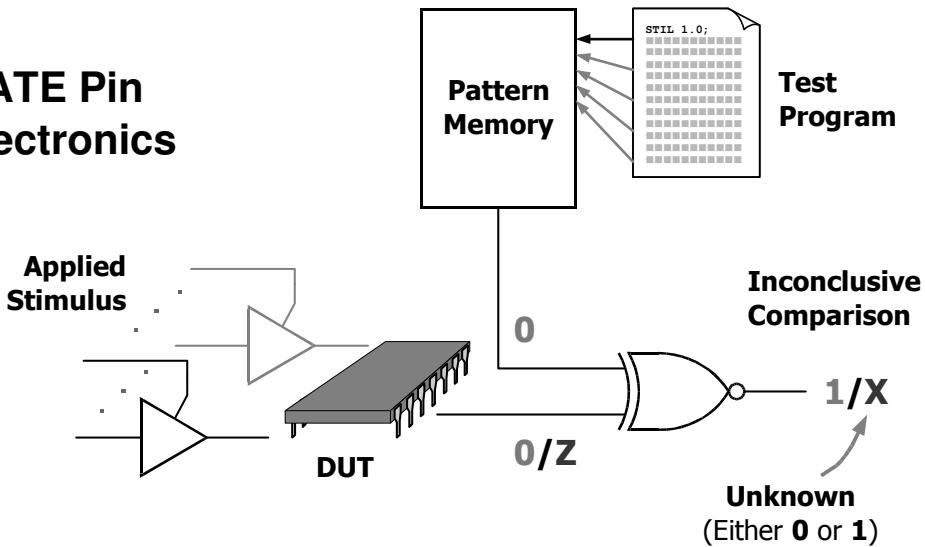
Solution 3:

Fully decode tristate enables

**7-45**

# Can Your ATE Observe Zs?

## ATE Pin Electronics



- Low-cost testers cannot observe high-impedance values
- The pin electronics is unable to compare 0 (or 1) with Z:
  - The ability of ATE to strobe a Z is tester-dependent

7-46

Checking for a high-impedance **Z** output is *not* an ordinary voltage measurement; thus, the output of the comparator logic in the diagram is the **unknown** value, **X**. **Unknown** means either logic **0** or **1**, but which one can not be definitely determined.

Some testers use **diode bridges** to verify the DUT output port is not actively *driven*. This is used by Credence in their Quartet Series: see [www.credence.com](http://www.credence.com). Other testers merely check for output voltages somewhere **between**  $V_{OL}$  and  $V_{OH}$ .

STIL defines an event **T (CompareOff)** for output voltages between  $V_{OL}$  and  $V_{OH}$ .

Complex ATE measurements, however, can stretch out the **time spent** on the tester.

### Conclusion:

Determine whether your target tester is able to reliably and quickly strobe **Z** values. If not, plan to improve testability of tristate networks by adding ad hoc **DFT logic**.

# DRC Fixing: Agenda

Fixing Common Violations

AutoFix Clocks/Resets/Sets

Internal Tristate Issues

AutoFix Internal Tristates

AutoFix Bidirectionals

7-47

# AutoFix for Fixing Internal Tristate Buses

## ■ **set\_dft\_configuration**

```
-fix_bus enable | disable
```

- Enable or disable AutoFix feature to prevent tristate shift violations on internal tristate busses (default is **enable**)
- Use **set\_ autofix\_ configuration** to override setting for particular tri-state buses

## ■ **set\_ autofix\_ configuration**

```
-type internal_bus \
-method no_disabling \
-include [get_nets TRI3]
```

- Use **set\_ autofix\_ element** to override settings for specific buses

**7-48**

For the types `internal_bus`, `external_bus` and `bidirectional`, specifies how to insert disabling logic.

By default, the fix method for `internal_busses` is **enable\_one**, the fix method for `external_busses` is **disable\_all** and the default for `bidirectional` is **input**.

Note: if a “black box” model is connected to an internal tristate bus, it will not be AutoFixed. In these cases, a model of the black box (that includes TSD’s on the interface) can be referenced with the “**test\_simulation\_library**” variable.

# Override Global Internal Bus Fixing

```
# Fixing of buses is enabled by default  
# All buses are fixed using enable_one method  
set_dft_configuration -fix_bus enable
```

**Priority:**

Overrides **global** DFT configuration settings.

```
set_ autofix_element [get_nets TRI3] \  
-type internal_bus -method no_disabling
```

List of Nets

Other Options:  
enable\_one  
disable\_all

You can control insertion of disable/enable logic per net

7-49

# DRC Fixing: Agenda

Fixing Common Violations

AutoFix Clocks/Resets/Sets

Internal Tristate Issues

AutoFix Internal Tristates

AutoFix Bidirectionals

7-50

# AutoFix for Bidirectionals

- **set\_dft\_configuration**

- fix\_bidirectional enable | disable**

- Enables or disables the AutoFix feature to fix bidirectional buses (default is **enable**)
  - Use **set\_ autofix\_ configuration** to override default settings

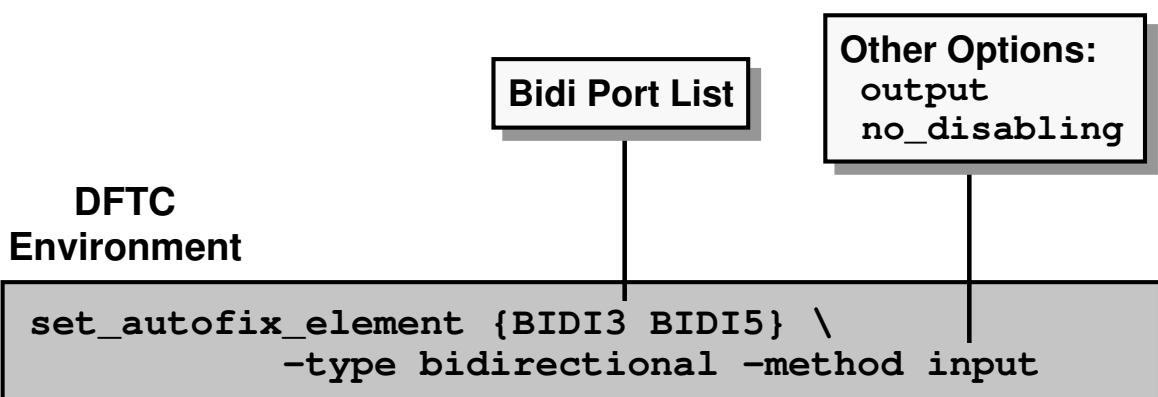
- **set\_ autofix\_ configuration -type bidi \**

- method output**

- Use **set\_ autofix\_ element** to override settings for specific bidirectionals

7-51

# Customizing Bidirectional Control DFT Logic



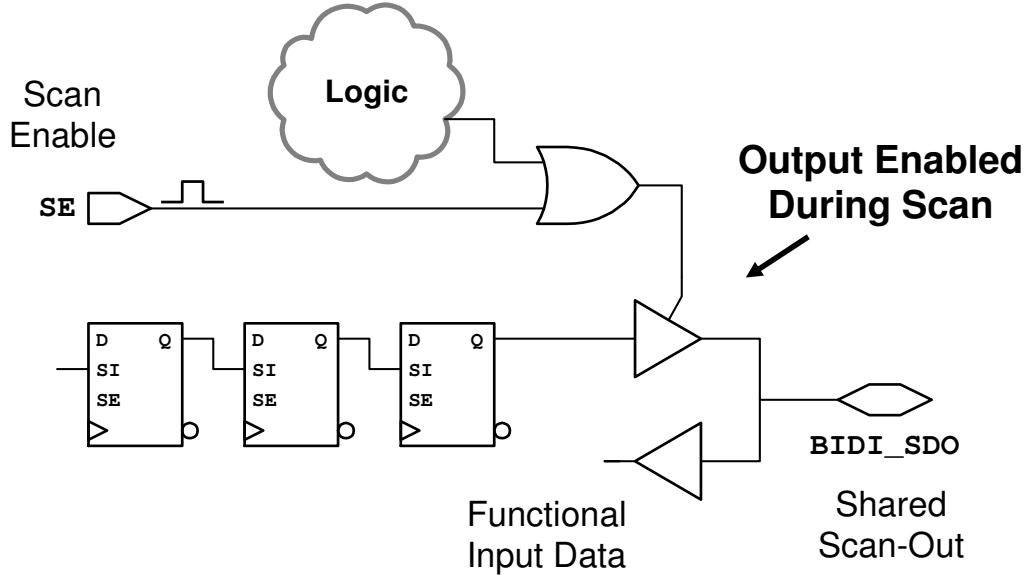
Priority:  
Overrides **global** AutoFix configuration settings

You can control insertion of disable/enable logic per port

7-52

# Enabling Output During Scan

N 9



7-53

The DRC issue is resolved by the OR gate, which injects a 1 whenever **SE** is active. This logic is added *automatically* by **insert\_dft** when it sees the above constraint. Port direction during capture cycles is *not* an issue, since the DUT is in functional mode.

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- **Name several available methods for fixing common DRC issues in a design**
- **State the DFT requirements for Asynchronous set or reset signals**
- **Use AutoFix to fix DRC issues relating to clocks, resets, sets, internal tristate, and bidirectional ports**

**7-54**

# Lab 7: DFT for Clocks and Resets



60 minutes

**After completing this lab, you should be able to:**

- **Analyze Pre-DFT and Post-DFT test violations and determine which ones could be addressed by using AutoFix**
- **Given a scan insertion script, modify it to enable AutoFix**

**7-55**

# Command Summary (Lecture, Lab)

<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>set_dft_configuration</code>	Sets the DFT configuration for the current design
<code>report_dft_configuration</code>	Displays options set by <code>set_dft_configuration</code>
<code>reset_dft_configuration</code>	Resets the DFT configuration for the current design
<code>set_ autofix_configuration</code>	Controls automatic fixing of violations
<code>report_ autofix_configuration</code>	Displays options set by <code>set_ autofix_configuration</code>
<code>reset_ autofix_configuration</code>	Resets the AutoFix configuration on a type basis
<code>set_ autofix_element</code>	If one of the AutoFix features is enabled , it makes specifications on a type and design object basis
<code>report_ autofix_element</code>	Displays options set by <code>set_ autofix_element</code>
<code>reset_ autofix_element</code>	Reset the current AutoFix configuration of the design for a particular type and a list of design objects
<code>preview_dft</code>	Previews, but doesn't implement, the scan architecture
<code>insert_dft</code>	Adds scan circuitry to the current design

7-56

# **Appendix**

## **User Defined Test Points (UDTP)**

# User Defined Test Points

- UDTPs direct DFTC to insert control and observe points at user specified locations in the design
- Why use UDTP?
  - Fix uncontrollable clocks and/or asynch pins
  - Increase the test coverage of the design
  - Reduce the pattern count

7-58

# Types of User Defined Test Points

- The following types of user defined test points are supported in DFT Compiler
  - Force
    - ◆ `force_0`, `force_1`, `force_01`, `force_z0`,  
`force_z1`, `force_z01`
  - Control
    - ◆ `control_0`, `control_1`, `control_01`,  
`control_z0`, `control_z1`, `control_z01`
  - Observe
    - ◆ `observe`

7-59

# Specifying a Test Point Element

- Use the command:

`set_test_point_element`

- This command allows the users to specify the location and the type of test points along with a set of options in order to achieve their test point requirements

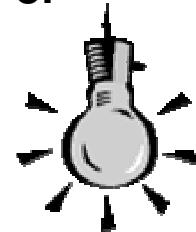
7-60

# Test Point Types

- The type of test point to be inserted can be specified as follows:

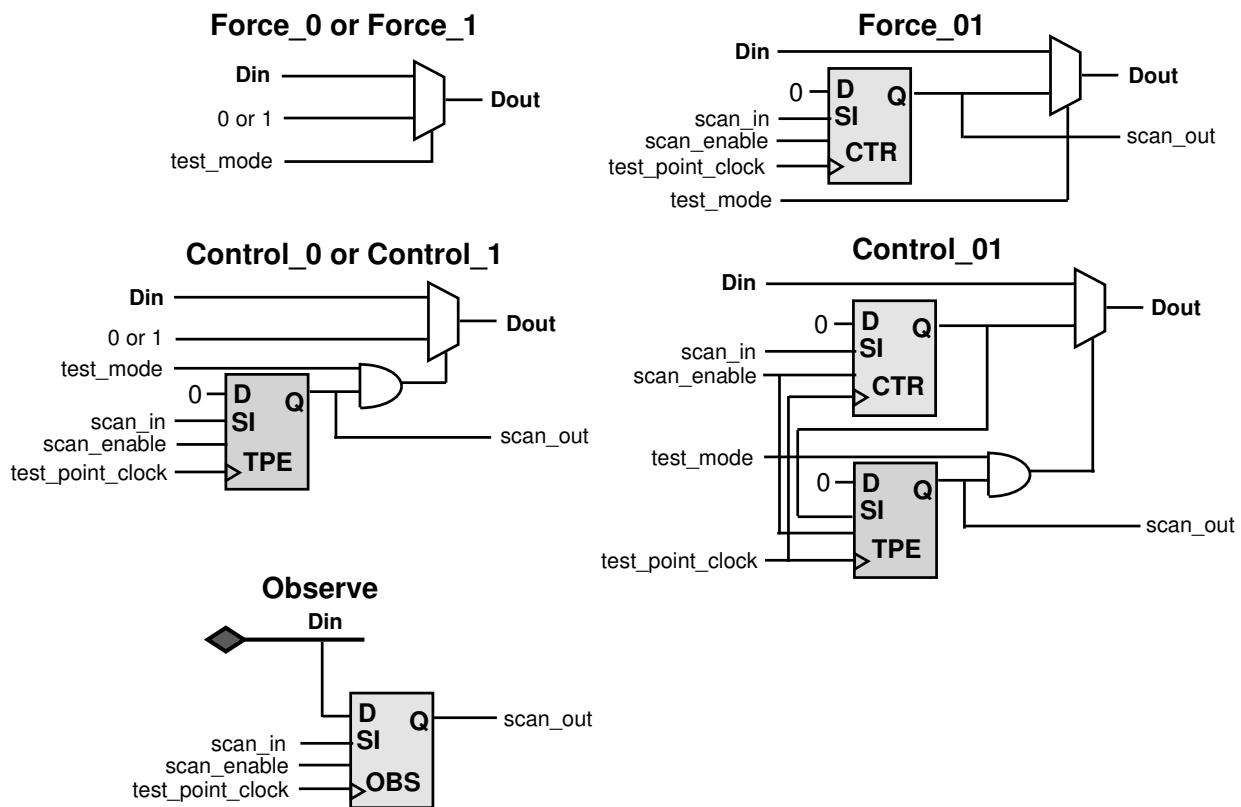
```
set_test_point_element [pin list]  
    -type <test_point_type>
```

- Pin list specifies the location at which UDTP will be inserted. It is a required argument
- The type of test points can be force, control or observe
- Remember: Only one “type” of test point can be specified with each invocation of the `set_test_point_element` command



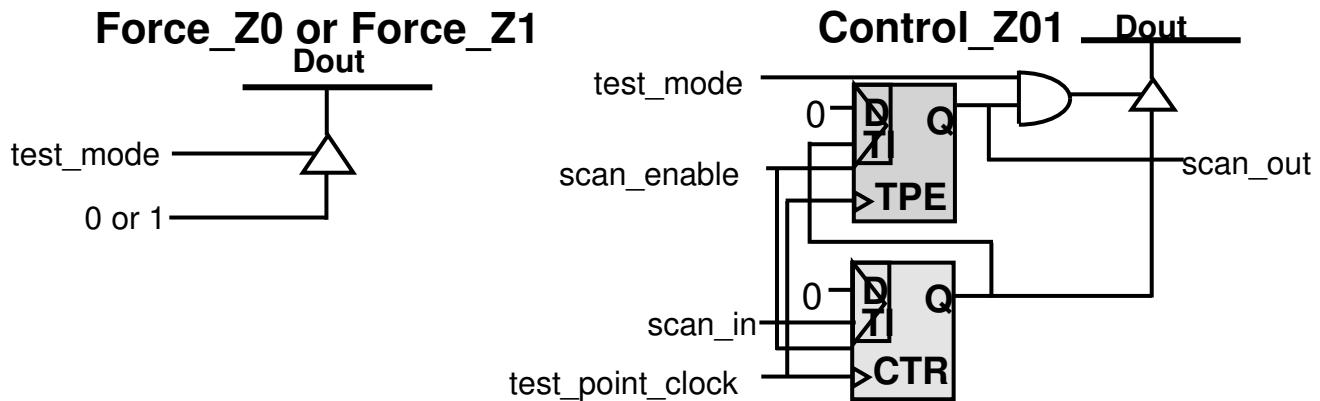
7-61

# UDTP Types



7-62

The Force\_Z0, Force\_Z1, Force\_Z01, Control\_Z0, Control\_Z1, and Control\_Z01 test points are the same concept but where the test points are on an internal tri-state bus. Some examples:



# How to specify Control and Clock signals

## ■ Control Signal:

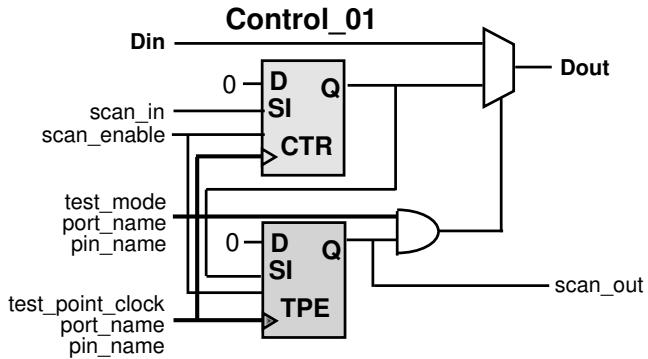
Must have been previously defined as a **TestMode** or **ScanEnable**

```
set_test_point_element pin_list -type control_01 \
-control_signal port_name|pin_name
```

## ■ Clock Signal:

Must have been previously defined as a **ScanClock**

```
set_test_point_element pin_list -type control_01 \
-clock_signal port_name|pin_name
```



7-63

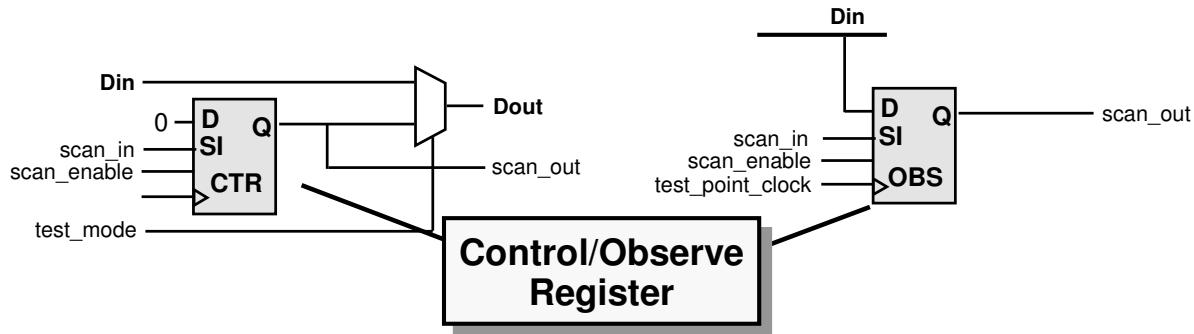
- Control signal can be an existing signal or created by the tool
- An existing port must have signal type of **TestMode** or **ScanEnable**
- If no control signal is specified, any existing port with the signal type **TestMode** is used
- If no TestMode port exists, DFTC will create a control port called “**test\_mode**” of type **TestMode**
  
- Any existing clock in the design can be specified
- An existing clock must be defined as a **ScanClock** before using with **set\_test\_point\_element**
- If no clock is specified, DFT Compiler will create a new clock port called “**tpclk**”

# Enabling Control or Observe Registers

- Enables or disables the insertion of control, force, or observe scan registers

```
set_test_point_element  
[-scan_source_or_sink enable | disable]
```

- Default is “enable”



7-64

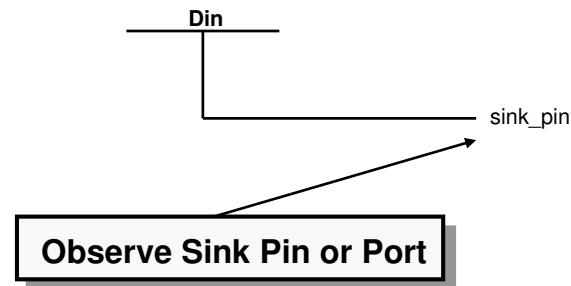
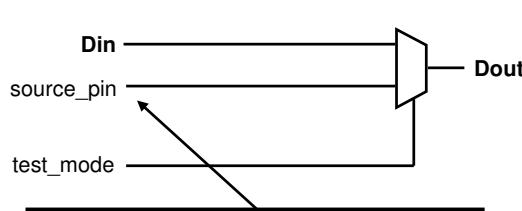
# Specifying a Control Source or Observe Sink

- The user can specify the name of the source signal (for control or force points) or sink signal (for observe points)

```
set_test_point_element
```

```
[-source_or_sink existing_pin_or_port]
```

- This option is valid only for control or observe test point types, when register insertion is disabled



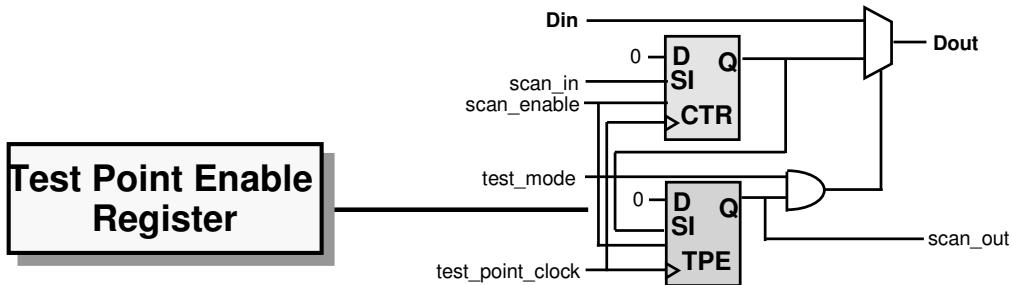
7-65

# Enabling Scan Register Test Point Enables

- Command option `-scan_test_point_enable` enables or disables the insertion of test point enable scan registers

```
set_test_point_element pin_list -type test_point_type  
[-scan_test_point_enable enable|disable]
```

- Default is “enable”
- This option is valid only for control test points



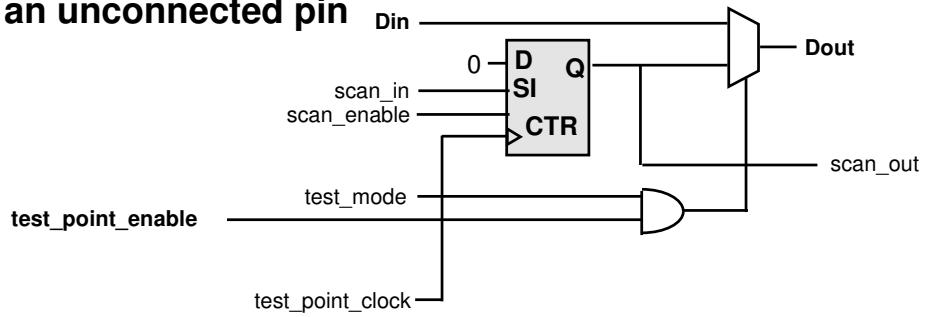
7-66

# Specifying a Test Point Enable

- User can optionally specify a test point enable pin or port for control points with the `-test_point_enable` option

```
set_test_point_element pin_list -type test_point_type  
[-test_point_enable existing_pin_or_port]
```

- If the name of the enable point is not supplied then DFTC creates a new port called `test_point_enable`
- This option is valid only for control test points
- Must supply an unconnected pin

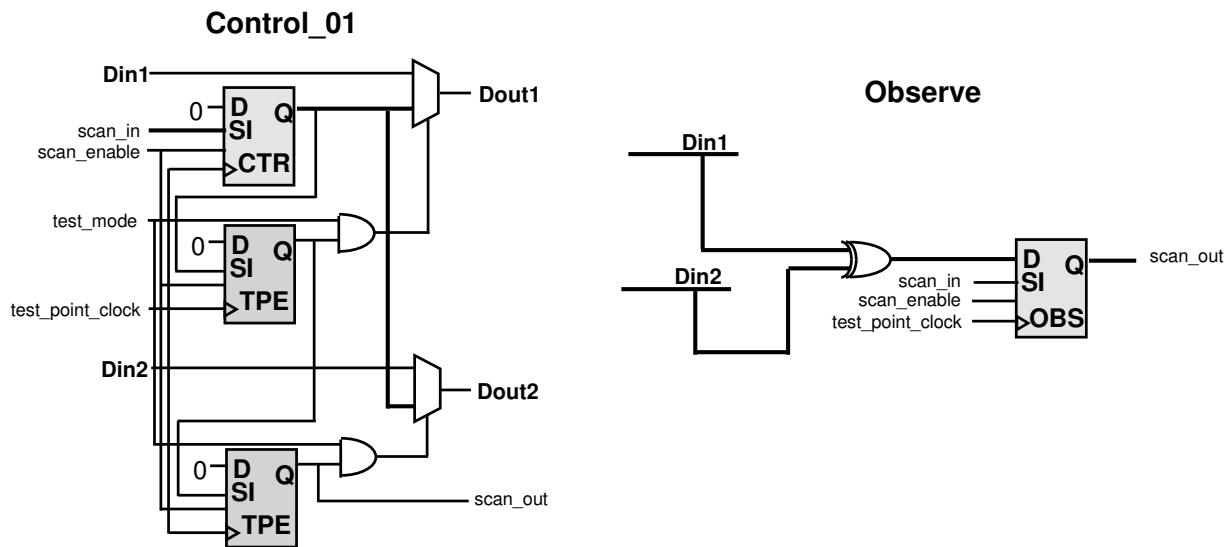


7-67

# Test Point Register Sharing

## ■ Sharing Control or Observe scan registers

```
set_test_point_element pin_list -type test_point_type \
-test_points_per_scan_source_or_sink 2 (default: 8)
```



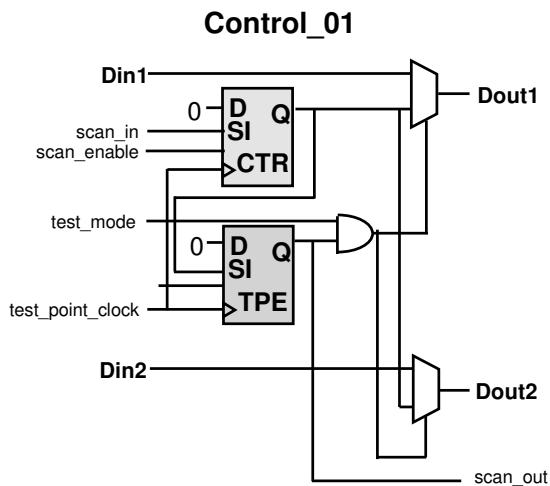
7-68

# Test Point Enable Register Sharing

## ■ Sharing Control Test Point Enable Registers

### ■ This option is valid only for the control test point types

```
set_test_point_element pin_list -type test_point_type \
-test_points_per_test_point_enable 2 (default: 1)
```



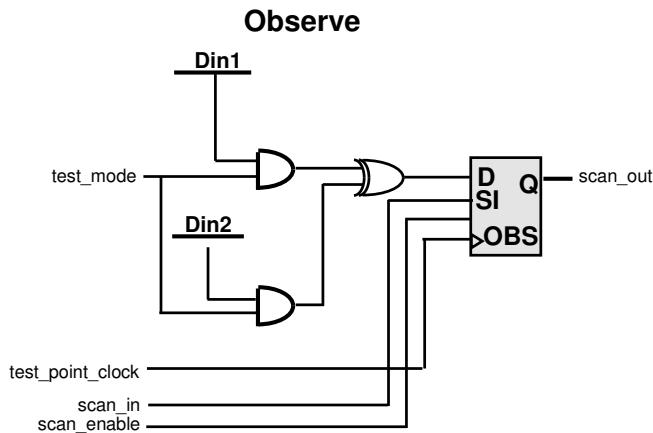
7-69

# Saving Power

- The user can specify insertion of power saving logic structures (AND gates) in the XOR tree for observe test points

`set_test_point_element`

`[-power_saving enable | disable]`



7-70

Note: The power saving feature does not gate the clock of the observe flops that are added, it only gates the XOR tree.

# Example UDTP Flow Script (1/2)

```
# Read in the design and synthesize it
acs_read_hdl -hdl_source ./rtl/Verilog A_DESIGN
link
compile -scan

# Define the clocks and asynchs in the design
set_dft_signal -type ScanClock -view exist -timing {45 55} -port CLK1
set_dft_signal -type ScanClock -view exist -timing {45 55} -port CLK2
set_dft_signal -type Reset -view exist -active 0 -port RST_N
set_dft_signal -type TestMode -active 1 -port TM
set_scan_configuration -chain_count 10 . . .

# UDTP specification. Specify observe and control TPs
# Turn power saving ON for observe test points
set_test_point_element -type observe -power_saving enable \
                        -clock_signal CLK1 {alpha_1/Q beta_5/out gamma_3/Q}
# Use an existing clock as clock signal and
#           existing port as control signal for control TP
set_test_point_element -type control_1 {u3/omega_2/q alpha_7/Q} \
                        -clock_signal CLK2 -control_signal TM
```

7-71

## Example UDTP Flow Script (2/2)

```
# Run pre-DRC
create_test_protocol -capture_procedure multi_clock
dft_drc -verbose
# Preview the scan chains
preview_dft -show all -test_points all
# Scan insertion
insert_dft
# Run post-DRC
dft_drc -verbose
report_scan_path -view exist
# Write out the netlist
change_names -rules verilog -hier
write -hier -f ddc -out A_DESIGN_udtp_scan.ddc
write -hier -f verilog -out TEST_udtp_scan.v
```

7-72

# **Recommendation for At-speed Tests**

- All inserted Test Points should be disabled during at-speed testing
- Either use a dedicated `TestMode` signal or use a dedicated clock to make it easy to disable the test points when performing at-speed testing

**7-73**

This page was intentionally left blank.

# Agenda

**DAY  
2**

**6 DFT DRC GUI Debug**



**7 DRC Fixing**



**8 Top-Down Scan Insertion**



# Unit Objectives

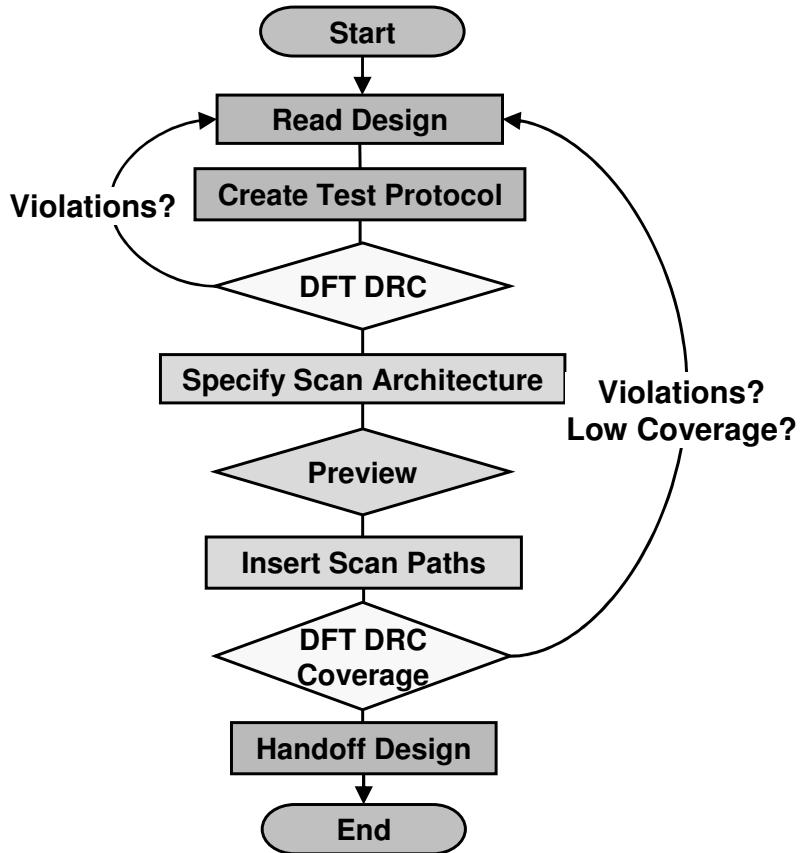


After completing this unit, you should be able to:

- Insert scan chains into a design using a top-down flow
- Specify and insert balanced top-level scan chains
- Reuse existing functional top-level pins for scan
- Specify and preview scan chain architectures

8-2

# Typical Scan Insertion Flow



8-3

# Top-Down Scan Insertion: Agenda

Specify Scan Configuration

Preview Scan Configuration

Insert DFT Logic

Balance Scan Chains

Finer Control

8-4

# Specifying Scan Architecture

- Scan architecture can be specified using `set_scan_configuration` command
- This command is used to specify the scan properties such as:
  - Scan style
  - Number of scan chains or scan chain length
  - Handling of multiple clocks
  - Lock-up Elements
  - Registers to be omitted from scan chains

8-5

Many key details of scan-path insertion are managed by `set_scan_configuration`. The configuration settings you specify are **global**, and apply across the current design.

This command supports **over 20** options—only the most commonly used options will be covered. The slide shows an assortment of settings, with the user-specified arguments italicized.

Unlike most other `dc_shell` commands, `set_scan_configuration` is **additive**. As the slide shows, you can specify the various configuration settings **consecutively**. Previous settings are not overridden—unless, of course, they contradict each other.

Proceed with scan-path configuration only **after** you have fixed all known DFT problems. The primary role of this command is to set up and guide the **scan-insertion** algorithms. A few options (like `-style`) will also influence pre-scan DRC.

To report all current configuration settings, use: `report_scan_configuration`.

## Using set\_scan\_configuration (1/5)

```
-style multiplexed_flip_flop | lssd |
    clocked_scan | aux_clock_lssd |
    combinational | none
```

- Specifies the type of scan in the design

```
-clock_mixing no_mix |
    mix_edges |
    mix_clocks |
    mix_clocks_not_edges
```

- Specifies whether scan synthesis will build scan chains with different clock domains on the same chain
- Mixing clock domains on the same chain usually requires lock-up latches to be inserted at the domain crossings

8-6

## Using `set_scan_configuration` (2/5)

### `-chain_count integer_or_default*`

- Specifies the number of scan chains `insert_dft` is to build
- The default will build the minimum # of scan chains given the clock mixing constraints

### `-max_length integer_or_default*`

- If this option is set with a positive integer, `insert_dft` builds scan chains without exceeding the specified length

### `-exact_length integer_or_default*`

- If this option is set with a positive integer, `insert_dft` builds, as much as possible, scan chains with the specified length

8-7

\*default means that this option has not been set by the user to an integer value.

If left set to the default, `insert_dft` builds the minimum number of scan chains consistent with clock mixing constraints.

**Note:** The `-chain_count` and `-max_length` and `-exact_length` options are mutually exclusive. If used together, the following is the order of preference:

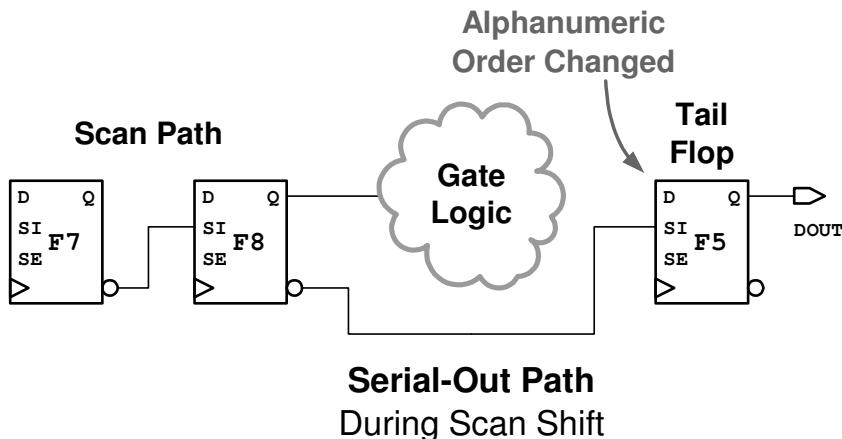
max\_length  
chain\_count  
exact\_length

# Using set\_scan\_configuration (3/5)

`-create_dedicated_scan_ports true | false`

## ■ Default Behavior:

- Looks for flip-flop **directly driving** an output port
- Places this flop at **tail end** of the scan chain
- Overrides default alphanumeric ordering scheme
- Set the option to TRUE to prevent this change in ordering



8-8

Sharing scan-access ports with functional data ports is largely **automated** by DFTC.

### Default Behavior:

By default, DFTC looks for a flip-flop that **directly drives** a functional output port. If such a flip-flop exists, **preview\_dft** will order it at the **tail** of the scan chain. If there is **more** than one, **preview\_dft** uses the **last** flop in alphanumeric order. The port is thus driven by the contents of the flop, whether **scan** or **functional** data. Buffers or inverters inserted between the flop and the port do not alter this default. Use **preview\_dft** to see when a scan flop has been moved to the end of the path.

For either TRUE or FALSE, DFTC will use user specified ports first (specified with **set\_dft\_signal**).

To avoid unexpected moving of cells, **Synopsys recommends setting the option to TRUE**.

## Using `set_scan_configuration` (4/5)

`-internal_clocks none | single | multi`

- \*global\* flag: identifies internal clocks (clock sources from single or multi-input combinatorial gates) that are to be treated as *separate* clock domains when creating scan chains (more on this topic later)

`-test_mode <mode_name>`

- Multiple scan configurations can be supported in one `insert_dft` task where each scan configuration has a unique test\_mode name

`-exclude_elements <element_list>`

- List of elements that should be excluded from the scan chains

8-9

## Using set\_scan\_configuration (5/5)

**-add\_lockup true | false**

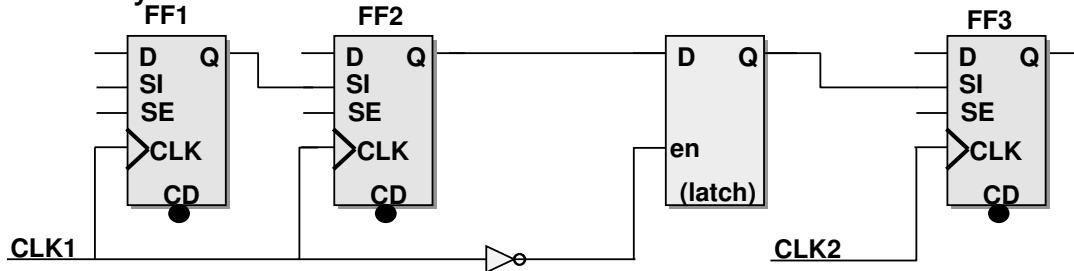
- enables/disables synthesis of lock-up elements when crossing different clock domains during scan synthesis

**-insert\_terminal\_lockup true | false**

- When *true*, insert\_dft inserts lock-up elements at the end of scan chains (default is **false**). Ignored if the scan style is not `multiplexed_flip_flop`

**-lockup\_type latch | flip\_flop**

- Specifies whether to use a latch or a FF as a lock-up synchronization element. Default is **latch**.

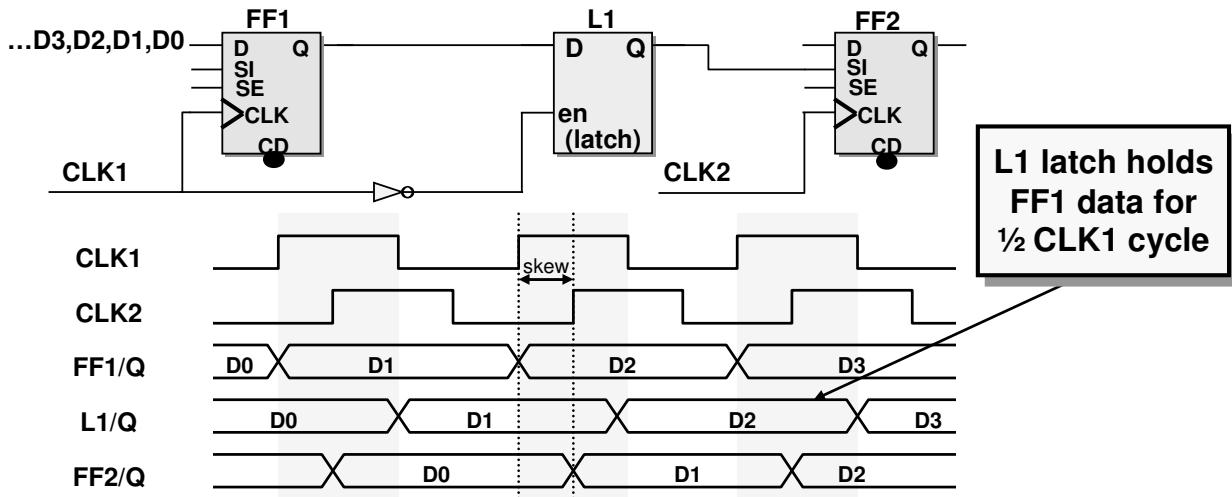


8-10

Terminal lock-up latches are often used when it is known that chains will be concatenated at a higher level later in the flow.

# Using Lock-up Elements

- Lock-up latches (or Flip-flops) are used to allow scan chains to cross clock domains
- They mitigate the skew between two clock domains to ensure data is shifted reliably on the scan chain



8-11

This diagram shows how a transparent **lock-up latch** gets inserted into a scan path. The latch is **inserted** at the point where the scan path **exits** the first clock domain. If the two clock domains are in **different blocks**, the latch is added to the **first block**. The **active-low** enable makes the latch **transparent** when **CLK1** is **inactive**; thus the scan bit shifted into **FF1** is enabled into the latch about a half-cycle later. On the **next** edge of **CLK1**, the latch **locks in** this scan bit, though **FF1** changes. The locked-up scan bit **remains available** all during the high time of **CLK1**. This provides almost an **extra half-cycle** for the scan bit to cross to the next domain. Without the latch, late arrival of **CLK2** due to skew might cause a lost scan bit.

# Reporting & Resetting Scan Configurations

- To report the scan configuration settings for the current design

`report_scan_configuration`

- To reset the scan configuration settings for the current design back to the defaults

`reset_scan_configuration`

8-12

# Top-Down Scan Insertion: Agenda

Specify Scan Configuration

Preview Scan Configuration

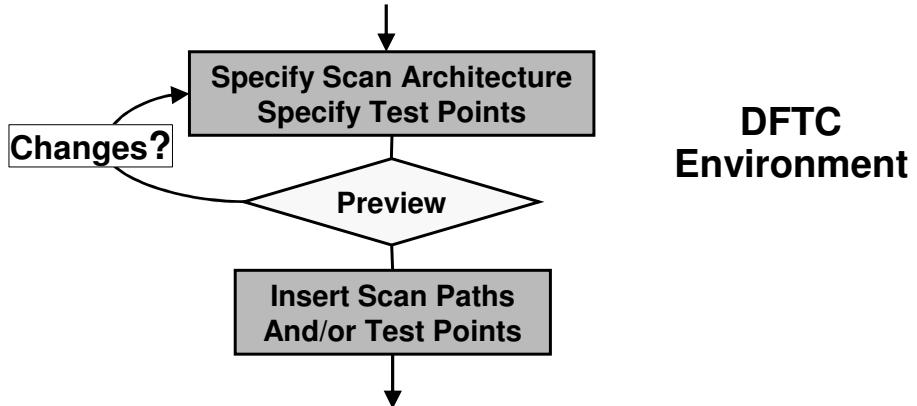
Insert DFT Logic

Balance Scan Chains

Finer Control

8-13

# A Fast Iteration Loop



**Use `preview_dft` to check scan architectures quickly!**

**8-14**

This slide is a zoomed in version of the **Specify-Preview-Insert** loop from the previous flowchart. By default, `insert_dft` involves compute-intensive algorithms. That is why scan insertion can be run **in the background** on a high-speed platform.

For fast debugging of scan paths, use `preview_dft` instead. Previewing does no synthesis—it is similar to **Print Preview** button in a personal computer application.

You will see a transcript summarizing the features of the expected scan architecture. If you do not like the results, edit global configuration options or explicit settings. This avoids running a long scan insertion just to find out the results were not right.

If you skip previewing, `insert_dft` runs the preview silently.

# What Does Scan Preview Do?

1. Checks Scan-Path Consistency:  
Flags issues like overlapping paths (same cell in two chains)
2. Determines Chain Count:  
The default is the minimum number of scan paths consistent with clock domains, clock-mixing, and rebalancing options
3. Allocates and Orders Scan Cells:  
Builds explicit paths first, then assigns other flops to paths grouped by common clock, then by block (default order is alphanumeric)
4. Adds Connecting Hardware:  
Inserts scan links (wiring between segments, latches, MUXes) and finds specified scan-access ports, or creates dedicated ones

```
dc_shell> preview_dft
```

8-15

Scan previewing is done by the **preview\_dft** command.

Previewing permits quick iterations, letting you fine tune the target scan architecture. Previewing in DFTC is far more than reporting a bunch of settings to the screen. During preview, an **internal representation** of the specified scan architecture is built. This is roughly comparable to an elaborated HDL description of the scan architecture; thus **no logic** is synthesized during preview, and **no database annotation** occurs.

## Allocation and Ordering:

Scan paths specified explicitly by **set\_scan\_path** commands are elaborated first. The remaining scan flops are then allocated, keeping **related** cells in the **same** path. For example, flops on a specific **test clock** are preferentially kept in the same path. Flops located in the same **subdesign** are preferentially kept in the same path. Cells within a path are ordered alphabetically by **full hierarchical pathname**. This default ordering is overridden by risk-free clock requirements, port sharing, etc.

# Typical Scan Preview Log

```
dc_shell> preview_dft
preview_dft
  Loading design 'ORCA'
  Loading test protocol
  ...
  Architecting Scan Chains
  ...
  Number of chains: 6
  ...
  Scan enable: scan_en (no hookup pin)
  Scan chain 'chain0' (pad[0] --> sd_A[0]) contains 488 cells
  Scan chain 'chain1' (pad[1] --> sd_A[1]) contains 488 cells
  Scan chain 'chain2' (pad[2] --> sd_A[2]) contains 488 cells
  Scan chain 'chain3' (pad[3] --> sd_A[3]) contains 488 cells
  Scan chain 'chain4' (pad[4] --> sd_A[4]) contains 487 cells
  Scan chain 'chain5' (pad[5] --> sd_A[5]) contains 487 cells
```

Protocol  
Read

Forming Scan  
Architecture

Desired  
Ports?

Balanced  
Scan Chains?

8-16

This slide shows a typical **dc\_shell** transcript of the scan preview process.

Confirm the following scan architecture details:

number of scan chains

scan chain balance

correct scan ports (enable(s), all scan ins, scan outs)

```
preview_dft -show [-show argument_list] keywords:
  bidirectionals - bidirectional conditioning,
  cells - scan cells,
  scan - scan attributes,
  scan_clocks - scan clocks,
  scan_signals - scan signals,
  segments - scan segments,
  tristates - tristate disabling,
  scan_summary - condensed scan information,
  all - everything)
```

## Using preview\_dft -show all

```
Scan chain 'chain1' (scan_in0 --> scan_out0) contains 19 cells
  cnt_n_reg[0]                      (ck_dp, 45.0, falling)
  cnt_n_reg[1]
  cnt_n_reg[2]
  cnt_n_reg[3]
  cnt_n_reg[4]
  ....
  cnt_n_reg[6]
  cnt_p_reg[5]                      (ck_dp, 45.0, rising)
  cnt_p_reg[6]
  cnt_p_reg[7]
  tc_p_reg[0]
  tc_nn_reg
  tc_p_reg[2]

  Scan signals:
    test_scan_in: scan_in0 (no hookup pin)
    test_scan_out: scan_out0 (no hookup pin)
```

All scan chain cells are listed

Clock domain crossing are shown

8-17

Use `preview_dft -show all` to see clock domain crossings.

# Top-Down Scan Insertion: Agenda

Specify Scan Configuration

Preview Scan Configuration

Insert DFT Logic

Balance Scan Chains

Finer Control

8-18

# Specifying DFT Insertion Parameters

- **DFT Insertion parameters can be specified using `set_dft_insertion_configuration`**
- **This command is used to specify:**
  - Design name preservation
  - Optimization
  - Whether to unscan violating FFs left off scan chains

8-19

# Using `set_dft_insertion_configuration`

## `-synthesis_optimization none | all`

- Controls cells upsizing, downsizing, buffer insertion

**all** = Full optimization

**none** = No optimization (i.e. “stitch only”)

Rapid Scan Synthesis (RSS)

## `-preserve_design_name true | false`

- Prevents sub-designs from being renamed `_test_1` and from being unqualified

## `-unscan true | false`

- Specifies whether to unscan cells during DFT insertion

**8-20**

**Note:** unscanning of flops does not occur if either synthesis optimization is disabled (`-synthesis_optimization none`) or if using DCT

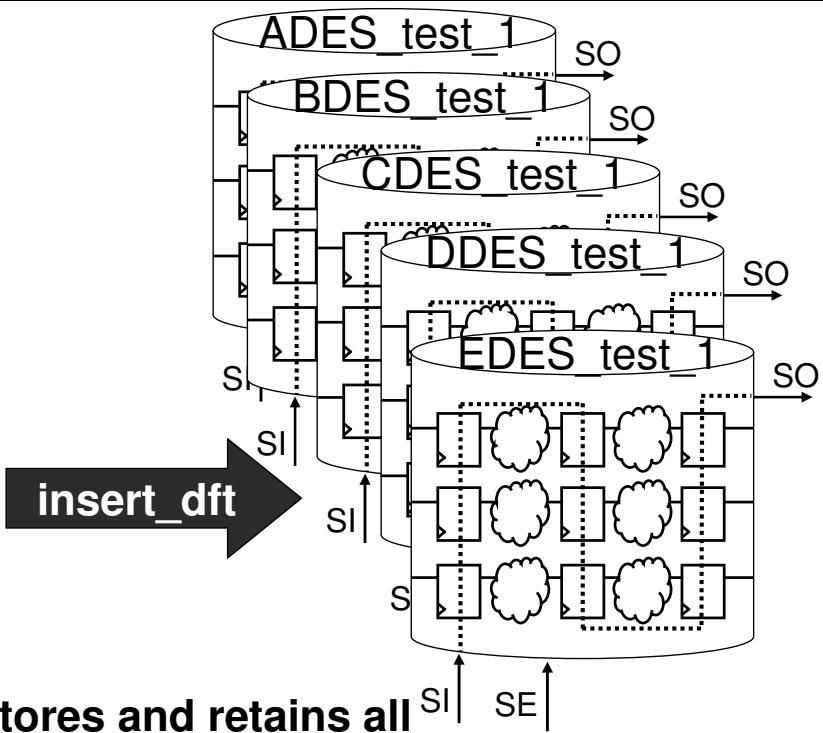
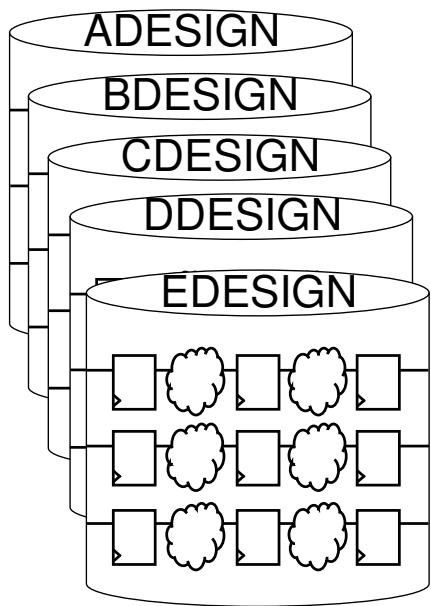
# Rapid Scan Synthesis (RSS)

- **Rapid Scan Synthesis (RSS) improves both capacity and runtime**
  - Avoids design duplication during scan insertion
  - Stitches scan chain without any optimization
  - User defined scan replacement
  - User controlled unscanning

**8-21**

Refer to SOLD and/or SolvNet for more information on User Defined Scan Replacement and User Controlled Unscanning.

# Avoiding Default “Test Uniquification”



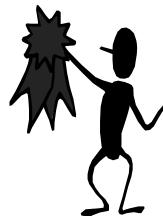
- DFTC by default stores and retains all the original and the “Test unquified” designs in the hierarchy of the current design

8-22

```
set_dft_insertion_configuration -preserve_design_name true
```

# Preserving Design Names

- Stops the changing of design names from “xyz” to “xyz\_test\_1”
- Avoids design renaming issue with verification flows
- Avoids “cloning” the multiply instantiated designs (and all their subdesign hierarchy):
  - Saves time
  - Saves memory



8-23

# Synopsys Test Tip

## 6.1 Insertion Without Optimization

Gate-level remapping is a key default feature of `insert_dft`. But to **disable** remapping, for final layout or other purposes, run:

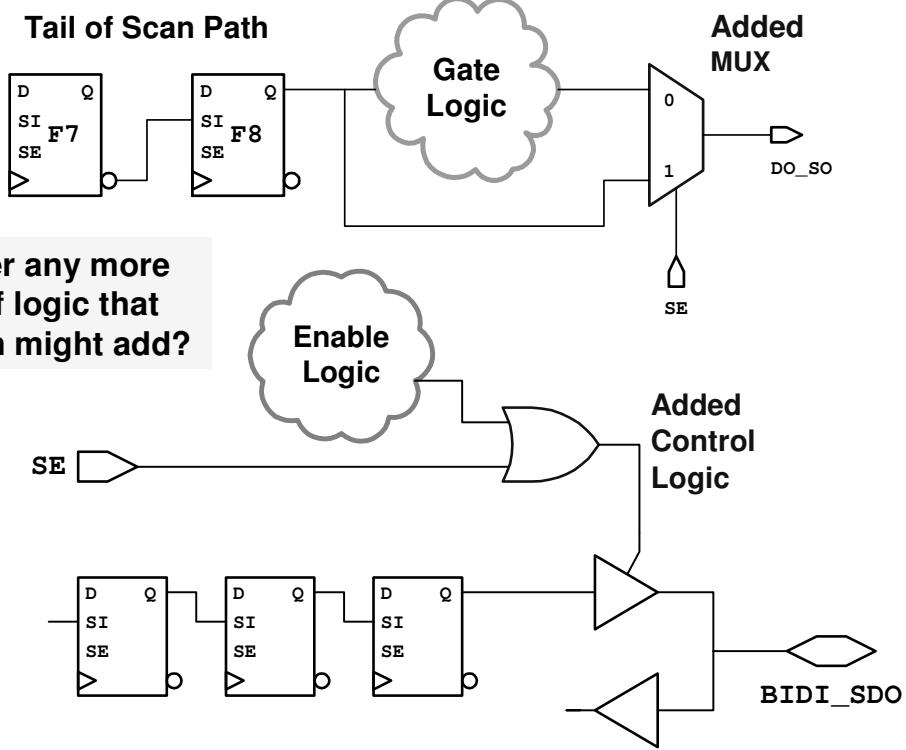
```
dftc> set_dft_insertion_configuration \
           -synthesis_optimization none \
           -preserve_design_name true
```

The first option skips fixing of all **user** timing and area constraints and also avoids fixing **foundry** design rules (including hold violations, for clock objects with a `fix_hold` attribute set.)

The second option prevents proliferation of unqualified subdesigns, generated by scan insertion whenever block logic is altered.

8-24

# Note: Inserted DFT Logic Not Optimized



8-25

Optimization controls cell upsizing, downsizing, buffer insertion

Added Scan Out Muxes, Bidi Mode Logic, etc. may be on paths that require a subsequent optimization.

# What Does Scan Insertion Do?

1. **Reads Previewed Representation:**  
Silently runs a scan preview, if needed
2. **Does Needed Scan Replacement:**  
Optionally replaces scan-compliant nonscan flip-flops
3. **Ensures No Contention:**  
Adds generic logic to enforce single-tristate-driver (STD) rule, and to maintain bidirectional paths during scan-shift cycles
4. **Inserts Test Points:**  
If AutoFix or Test Points are enabled, adds test-point logic
5. **Assembles Scan Paths:**  
Stitches access ports, scan links, and scan cells together
6. **Minimizes Violations:**  
Maps generic logic, optionally fixes violations by gate-level remapping, optionally “unscans” excluded scan flops

```
dc_shell> insert_dft
```

8-26

Scan insertion is done by the **insert\_dft** command.

If you omitted an **explicit** previewing step, then previewing is done **silently**. The previewed scan architecture is **synthesized** and optimized in six distinct phases.

Phase 1 A preview\_dft only if you did not run it explicitly.

Phase 2 If you did not run compile –scan, scan replacement occurs here.

Phase 3 TSD and Bidi Rules.

Phase 4 includes AutoFix and Test Points.

Phase 5 assembles the scan paths—but can leave the design with **timing violations**.

Phase 6 **fixes** violations, performing **compile**-like reoptimization at the gate-level.

This optimization can be controlled using **set\_dft\_insertion\_configuration**.

## Checklist:

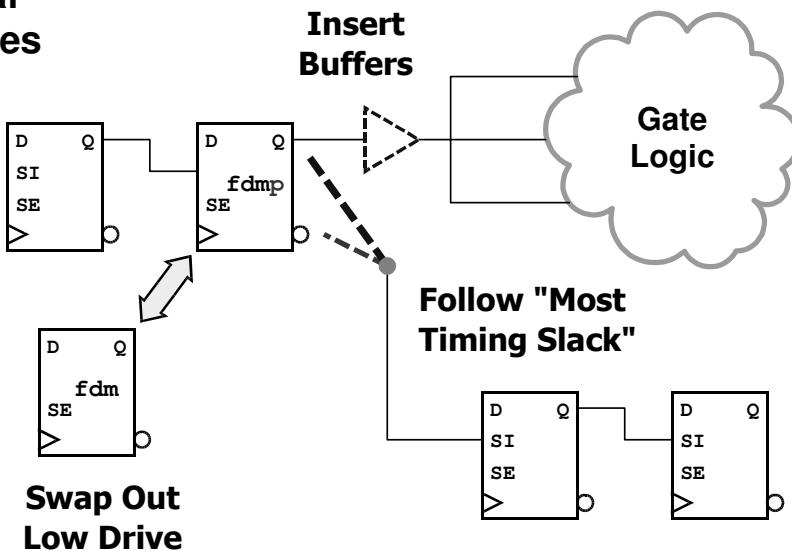
Prior to scan insertion, ensure **no timing violations** are left in the application logic.

Do not expect **insert\_dft** to do the job of **compile**—long run times may result!

Remove any **dont\_touch** attributes left behind after application-logic synthesis.

# Gate-Level Remapping

## Typical Examples



- These transformations can occur during scan insertion if optimizations are enabled

```
dc_shell> insert_dft
```

8-27

Phase 6 of scan insertion is like running `compile -incremental` on gate logic—only faster. The gate-level remapping algorithms focus only on scan-oriented transformations.

### Examples:

**Resizing** flip-flops or buffers to increase the **drive** strength along a scan path.

**Inserting** inverting or non-inverting **buffers** to boost drive or split fanout loads.

Bringing out the scan path via the pin (example: **QB**) that has the **most timing slack**.

### Options:

Scan-chain inversion—e.g., the use of **QB** pins—is not always supported by vendors.

To disable the use of **QB** pins, set `test_disable_find_best_scan_out true`.

For practical tips on using this variable, see: SOLV-IT! article [900784.html](#).

Optimization can be controlled or disabled using `set_dft_insertion_configuration`.

```
set_dft_insertion_configuration
  -synthesis_optimization none: No optimization (i.e. Just stitch it)
  -synthesis_optimization all:   Full optimization, the default
```

**Note:** this type of gate-level remapping during scan insertion is not available in DCT

# Design Compiler Topographical Mode (DCT)

- Design Compiler Topographical (DCT) enables accurate prediction of post-layout timing, area, and power during RTL synthesis without the need for wireload model-based timing approximations
- Uses Synopsys' placement and optimization technologies to drive accurate timing prediction within synthesis, ensuring better correlation to the final physical design
- Built in as part of the “DC Ultra” feature set and is available only by using the `compile_ultra` command in topographical mode (`dc_shell -topo`)

8-28

# DFT Flow in DCT

- The DFT flow in DCT is similar to the DFT flow in Design Compiler wire load model mode, except that the `insert_dft` command does not perform an optimization step (i.e. “stitch-only”)

```
compile_ultra -scan
create_test_protocol
dft_drc
preview_dft
insert_dft
dft_drc

#User must rerun optimization after scan insertion
compile_ultra -incremental -scan
```

8-29

## DCT Details

- DCT only supports multiplexed flip-flop scan style
- When you run scan insertion in topographical mode, the `preview_dft` and `insert_dft` commands print the following message:  

```
Running DFT insertion in topographical mode.
```
- DFTC performs topographical ordering when running scan insertion in DCT. The virtual layout information is utilized, where available, to ensure that there is no severe impact on functional timing
- You should still use the scan chain reordering flow after scan insertion in DCT to obtain optimal results

8-30

# Top-Down Scan Insertion: Agenda

Specify Scan Configuration

Preview Scan Configuration

Insert DFT Logic

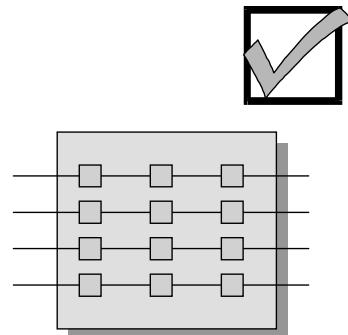
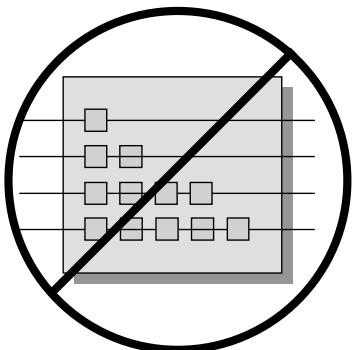
Balance Scan Chains

Finer Control

8-31

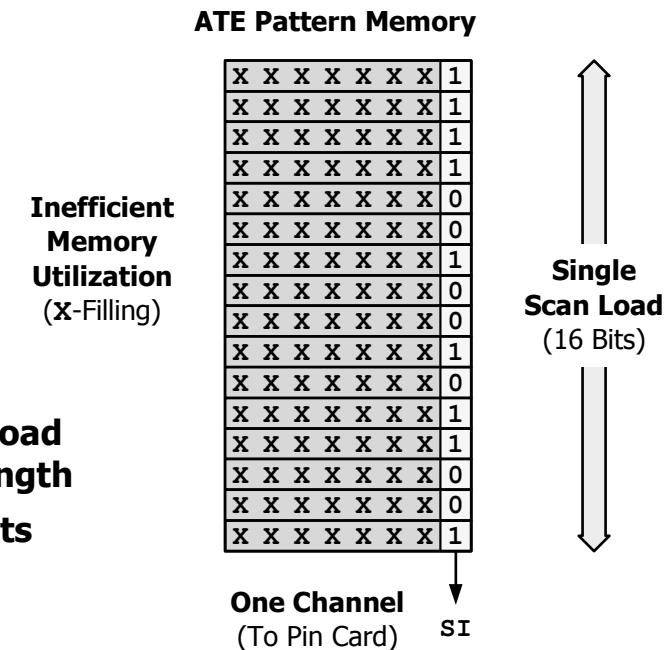
# Why Balanced Scan Chains?

- Always strive to build as balanced a set of top-level scan chains as possible
- Unbalanced chains lead to:
  - Wasted test application time on tester
  - Wasted pattern memory on tester
  - Wasted \$\$



8-32

# Wasted ATE Pattern Memory (1/3)



One long scan path underutilizes ATE pattern memory

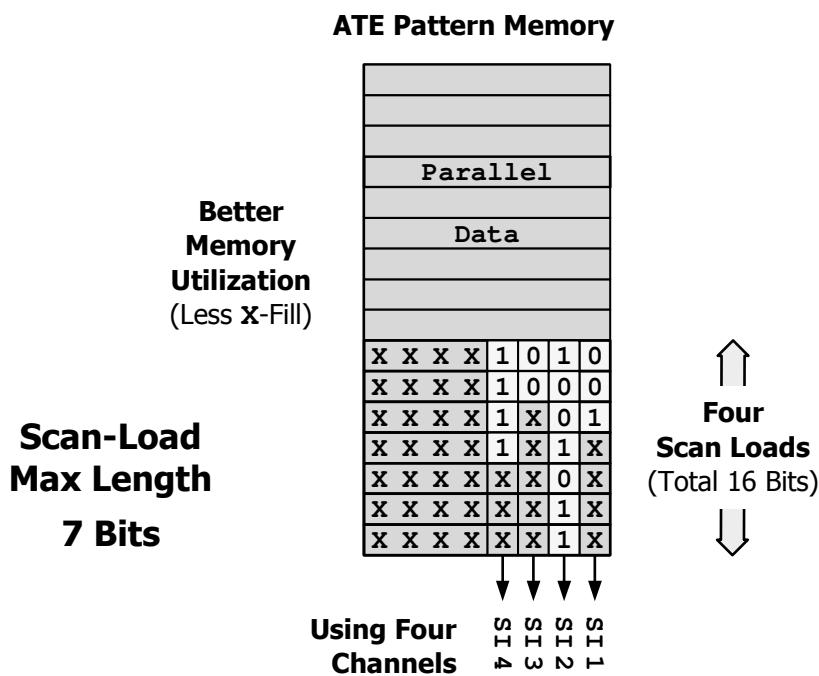
8-33

Another tradeoff involving scan-chain length is ATE pattern-memory limitations. In an earlier unit you were shown that the test program is stored vector by vector in **pattern memory**. Every test pattern includes the necessary serial data for loading all the scan chains. The string of **1s** and **0s** shifted into a chain to execute a pattern is called a **scan load**. The ATE must reserve one **tester channel** per scan load to store and shift in this data. A tester channel corresponds to one **tester pin**, along with the storage bits behind it. In the diagram, just **one channel** is utilized—which requires a very **deep memory**!

## Conclusion:

A **single** long scan path is undesirable because ATE memory may lack needed depth. In addition, too many memory cells are filled with **Xs** (don't cares), and thus wasted.

# Wasted ATE Pattern Memory (2/3)



These four paths are better, though imbalanced

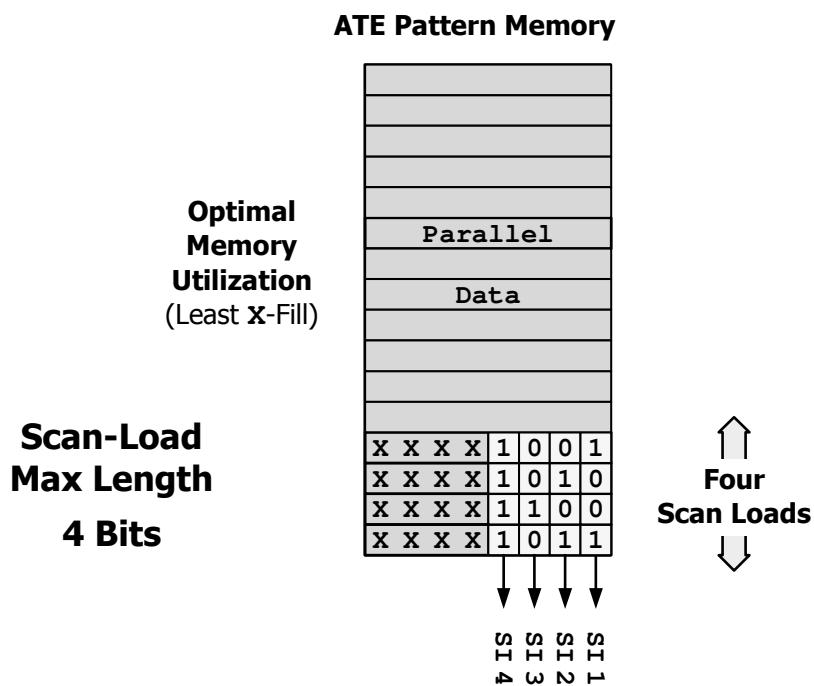
8-34

This slide shows the single 16-bit scan load broken up into four, still totaling 16 bits. This avoids tester memory-depth limitations, but the scan paths are still imbalanced. Because the scan loads **differ** in length, the **shorter** bit strings must be filled with **Xs**. The **Xs** are shifted **first** during scan-in, ensuring all chains are loaded by **last** scan-in. But the **Xs** use up bits of tester memory that could have been used for parallel data.

## Conclusion:

This is still inefficient use of tester memory, and your test program might be too long. You do not want to **pause** the test program to **reload** memory for every unit you test. Plan instead to have **multiple** scan paths on the chip with reasonably **balanced** lengths. Balanced lengths tend to yield the shortest scan loads, thus minimizing tester time.

# Wasted ATE Pattern Memory (3/3)



**Four balanced paths are best**

8-35

This slide shows the **optimal** scenario, with four balanced paths totaling 16 bits. This uses the least memory—and requires the fewest scan vectors per test pattern.

#### Caveat:

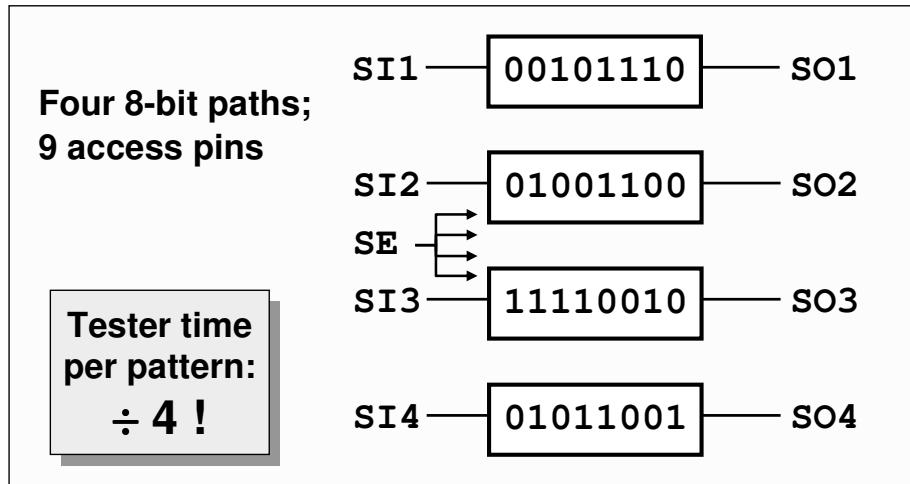
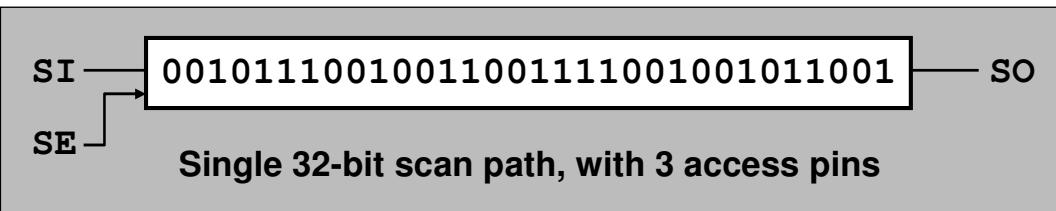
A **deep scan-memory** option, supported by some ATE models, changes this picture. A number of channels, usually  $2^n$ , will offer significantly **greater depth** for scan loads. With this option, the optimal **chain count** is a **power of 2**.

#### Conclusion:

In general, plan to insert **multiple** scan paths on the chip with **balanced** lengths. This minimizes tester time, utilizes memory efficiently, and avoids depth limits.

Exception: Use **fewer** chains if you are pin-limited and the ATE has deep memory.

# Scan Access Pins Vs. Tester Time



8-36

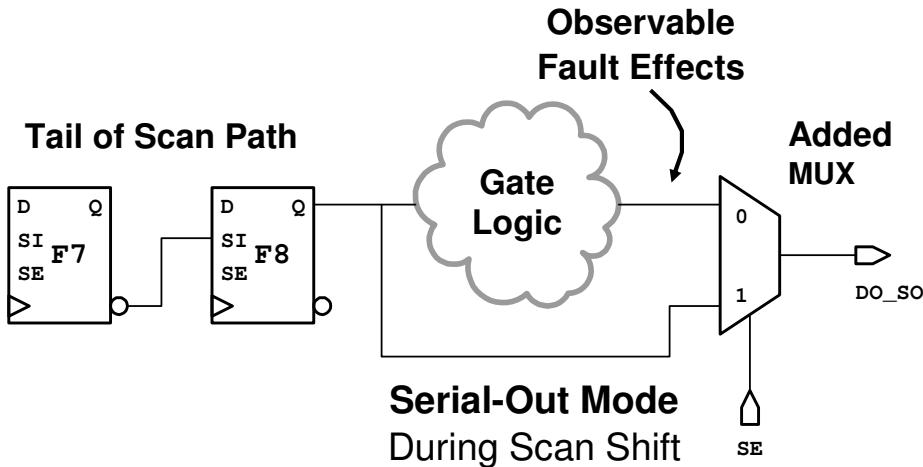
Choosing optimal scan-path length involves an inherent tradeoff:

The **longest** scan path in your design determines the number of **vectors** per pattern; thus tester time per pattern **increases directly** with the longest scan path on the chip. Tester time in this example is reduced by a factor of 4, just by partitioning a long path. But—each added scan path requires **two more** package pins, **SI** and **SO**.

## Conclusions:

For most SoC designs, plan on supporting **as many** scan paths as available pins allow. Share **SI** or **SO** with functional pins as necessary, using **insert\_dft** to add the logic. The scan-chain count does not affect how many **test patterns** are generated by ATPG; thus you can optimize tester time versus pin count, without compromising coverage.

# Using Functional Output as Scan-Out Port



## Declaring a Functional Output as a ScanOut:

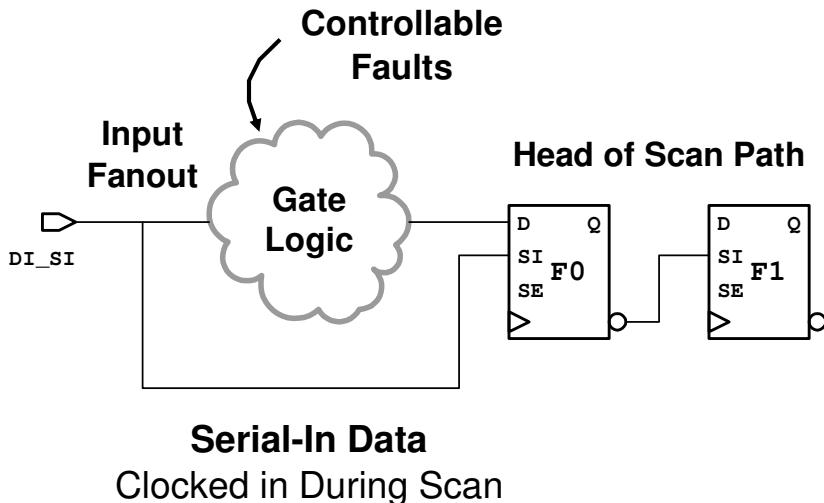
```
set_dft_signal -type ScanDataOut \
    -port DO_SO -view spec
preview_dft
```

8-37

You can still share an output port even if the nearest flop drives **intervening gate logic**. **insert\_dft** will add a **multiplexer** to bypass the gate logic during scan shift. The **set\_dft\_signal** command is used declare the existing functional port as a ScanOut. The required multiplexer is synthesized **automatically** during scan insertion. It uses **SE** instead of **ASIC\_TEST** as a MUX select, to keep the gate logic **observable**. The MUX is not reported by **preview\_dft**, but will be synthesized by **insert\_dft**.

Tristate or bidirectional ports can also be shared—DFTC will add the necessary logic. Directional control logic is inserted to enable output mode during scan-shift cycles. The tristate output buffer must, however, reside in the block where insertion is done.

# Using Functional Input Pin as Scan-In Port



## Declaring a Functional Input as a Scan-In:

```
set_dft_signal -type ScanDataIn \
               -port DI_SI -view spec
preview_dft
```

8-38

By **default**, `preview_dft` always synthesizes a **dedicated scan-in** port on a design. This slide shows how to **override** this default behavior and **share** a scan input port.

To share scan-in with an existing functional port, use `set_dft_signal` syntax. Pick an input with **minimal fanout** which feeds into the **D** pin of a scannable flop. The example above shows that an additional, nonfunctional, input **fan-out** is synthesized. This fan-out, in effect, **demultiplexes** the scan-in data to the flip-flop when **SE** is high.

You can customize the default **naming conventions** for ports created by insertion.

Example: set the variable: `test_scan_enable_port_naming_style = "test_se%_s"`

# Specify Chain Count

To explicitly specify four scan paths, overriding the default minimum chain-count:

```
set_scan_configuration -chain_count 4  
preview_dft
```

- Four scan paths are previewed, if possible
- All four paths are enabled by the global SE port
- By default `preview_dft` tries to balance the scan chain lengths of the 4 requested chains

8-39

The default behavior of `preview_dft` is to minimize the number of scan paths. It gives you the minimum chain count consistent with risk-free scan requirements.

As you will see, these requirements can include clock-mixing and rebalancing options.

The explicit `-chain_count` option shown above overrides the default behavior. If the count cannot be honored within requirements, warnings like **TEST-355** occur.

Example: requesting 4 single-clock scan chains for a chip with 5 different clocks.

To revert to the default, set the `-chain_count` argument to the string **default**.

## Balancing Chains:

By default, `preview_dft` tries to produce balanced chains that meet requirements.

# How DFTC Balances Scan Chains

## Default Behavior

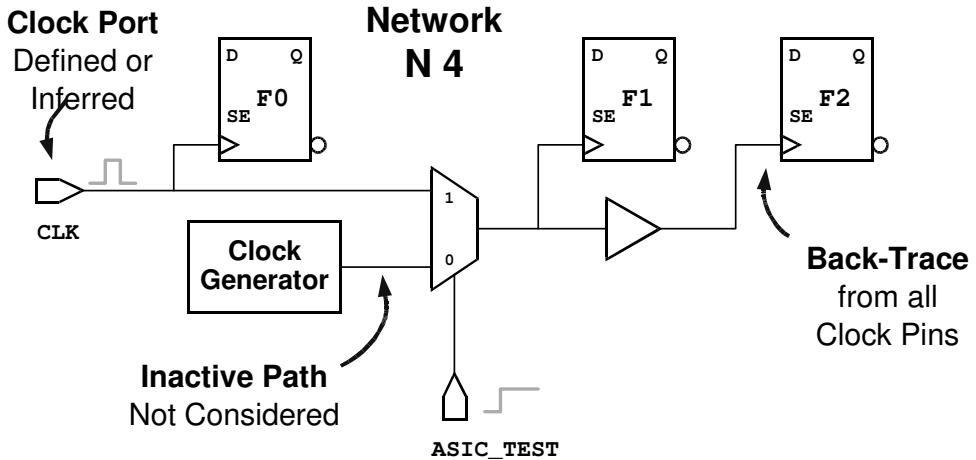
- **By default `preview_dft` will safely balance paths:**
  - Never uses more than one clock in a single chain
  - Will not break up existing scan-chain segments

## Nondefault Clock-Mixing

- **Better balance because it now allows:**
  - Multiple clocks in a single chain
  - Multiple clock edges in a single chain

8- 40

# What Does DFTC Consider a Clock Domain?



- When test clocks are declared with `set_dft_signal`, this identifies the test clock domains
- If clocks are inferred, DFTC must trace the netlist to make its best “guess” for the top-level clock sources
- There is only one test-clock domain for network N 4

8-41

This slide describes how DFTC identifies distinct **clock domains** within a design.

In the inference flow (not recommended when clocks and resets are known), DFTC traces **back** from clock pins on all sequential cells, `create_test_protocol -infer_clocks -infer_asynch` identifies the **ports**.

Only **active** paths are considered during back-tracing—all **inactive** paths are ignored. Inactive paths arise from **constants** (e.g. -type Constant) which prevent the flow of signals. Back-tracing from all the flip-flops in network N 4 leads to the same input port, **CLK**; thus by default, `create_test_protocol -infer_clock` infers only **one** test-clock domain for this entire network.

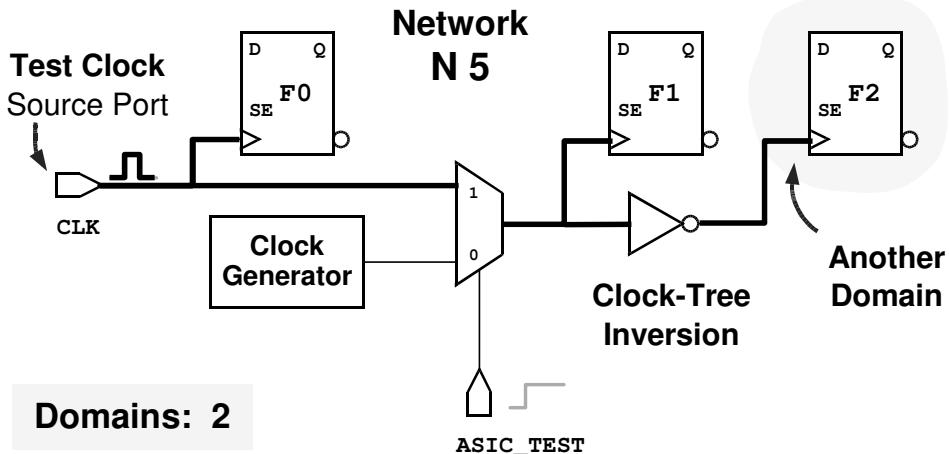
## Default Test Clocks:

The best practice is to **explicitly** declare clock waveforms using `set_dft_signal`.

If you forget, `create_test_protocol` uses **default** test-clock waveforms for inferred clock ports!

For multiplexed flip-flop scan style, the **default** waveform with a 100 ns period is: `-timing {45 55}`.

# How DFTC Considers Dual-Edge Clocking



- Network N 5 flip-flops use both edges of the clock CLK
- preview\_dft sees two distinct clock domains
- By default, DFTC inserts a separate scan path for flip-flop F2

```
set_scan_configuration -internal_clocks none
```

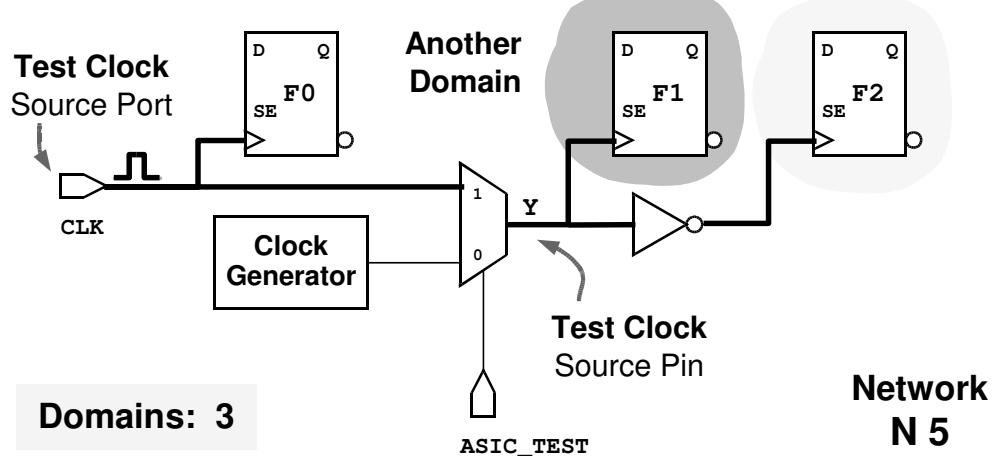
8-42

Testability issues arise when flip-flops are clocked on **both edges** of the same clock. Network N 5 contains **two subsets** of flip-flops, positive- and negative-edge-triggered. During the basic tester cycle, these two subsets change state at **different** time points. This is a **departure** from the rule that all scan flops clock in new data at the same time.

## Conclusions:

When **both edges** are used, **preview\_dft** will always see **two** distinct clock domains; thus **preview\_dft** sees one **clock domain** for each **active edge** of each **test clock**. The total number of **clock domains** will in turn impact the default number of scan paths.

# An Internal Clock Domain



- Clock gating or MUXing does not by itself create a domain
- You must apply configuration option `-internal_clocks`
- When `-internal_clocks` is set to “multi”, network N5 has three test-clock domains

```
set_scan_configuration -internal_clocks multi
```

8-43

This slide shows a design with an **internal clock domain not identified** by default.

For **DFTC** to see an internal clock domain, apply the option `-internal_clocks`.

You will see detailed syntax for setting this option, globally or locally, on the next slide.

As a result, **preview\_dft** identifies a **third** clock domain, whose source is MUX pin **Y**, otherwise, network N 5 would still default to **two** individual test-clock domains.

## Conclusions:

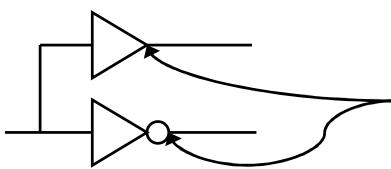
Option `-internal_clocks` is important for managing clock skew due to DFT logic. If your goal is avoiding hold time violations in scan paths due to skew, this is a **key tool**.

# Specifying Internal Clocks

- Gate or MUX logic can drive an internal clock signal
- Global `-internal_clocks` option applies across a design
- Local option applies only to a specific test-clock tree

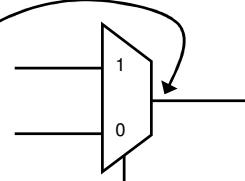
```
Global: set_scan_configuration \
          -internal_clocks single|none|multi

Local: set_dft_signal -type ScanClock \
          -internal_clocks single|none|multi ...
```



`-internal_clocks single`

Test Clock Source Pin



`-internal_clocks multi`

8-44

This slide summarizes DFT risk management using `-internal_clocks` syntax. Internal clocks are often created by DFT hardware that has been inserted in a clock network. An inserted MUX or injection logic **adds skew**, causing potential hold time issues. This option **reduces** the risk by forcing the internal clock line into a **separate** domain. For scan styles other than multiplexed flip-flop—like **lssd**—the option has no effect.

## Local Scope:

To set this option on a clock: `set_dft_signal -type ScanClock . . . -internal_clocks single|none|multi`. This clock-specific **local** setting will take precedence over the global configuration.

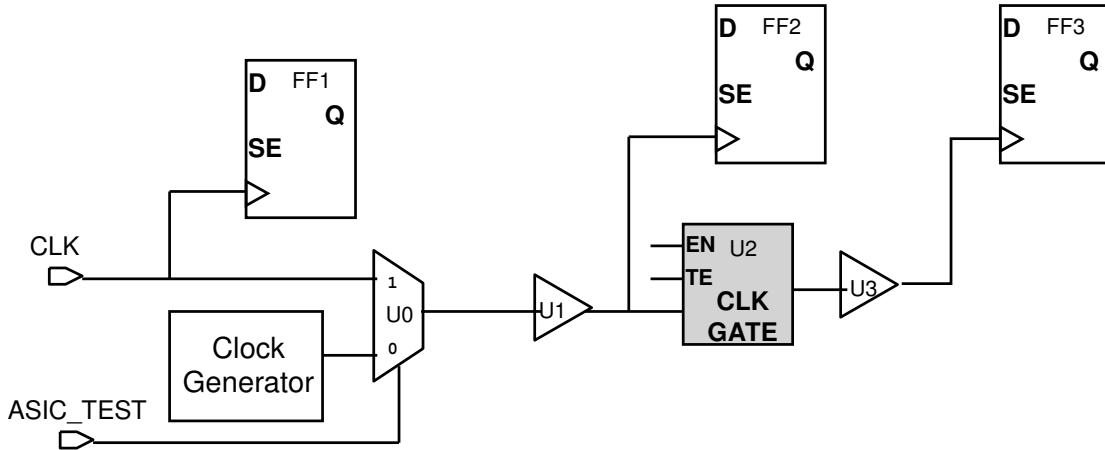
## Conclusions:

Setting `-internal_clocks single|multi` helps DFTC avoid timing risks during scan shift. The only tradeoff is that you may get more scan paths on the chip than you planned. This option does not address clock skew problems affecting **capture** cycles.

NOTE: Clock gating cells are not seen as creating new clock domains – Integrated Clock Gate (ICG) cell outputs will not be treated as an internal clock source.

# Internal Clocks and Clock Gating

- Clock gating elements are not considered when identifying internal clock domains with -internal\_clocks single | multi



How many clock domains with -internal\_clocks multi?  
How many with -internal\_clocks single?

8-45

# Mixing Clocks: Conservative

- If you use multiplexed flip-flop scan style, then:

- By default, no clock mixing along a scan path is allowed
- Default chain count is thus one path per clock domain
- This restricts balancing in chip designs with many clocks
- For better balancing at a low risk, mix edges of the same clock (`-clock_mixing mix_edges`)

## Low-Risk Clock Mixing:

```
set_scan_configuration -clock_mixing no_mix  
OR -clock_mixing mix_edges  
preview_dft
```

8-46

The global default `set_scan_configuration` setting is `-clock_mixing no_mix`. With this default setting, `preview_dft` generates **one scan path** per clock domain. This is the **safest** strategy—but it can mean **too many**, or **poorly balanced**, scan paths. It is hard to balance a **two-clock** IC with a majority of the flip-flops on **one** clock tree.

## Mixing Edges:

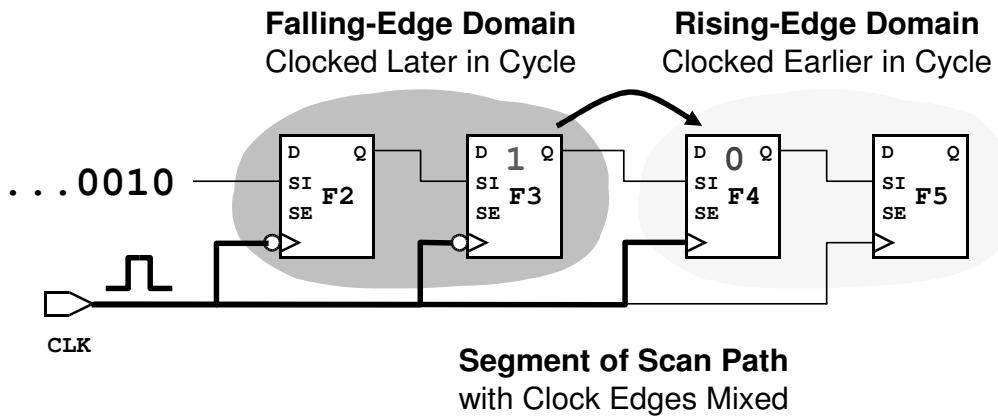
Option `mix_edges` allows mixing of **both edges** of the **same clock** along a scan path. Cells are ordered along a scan path in accordance with the NICE rule on the next slide. This guarantees that each scan bit is read by the next flip-flop **before** it is overwritten. With this option, the chain count defaults to the number of clock **signals** (not **domains**).

This option is **low risk**—unless clock skew becomes comparable to clock pulse-width.

Mixing edges allows more freedom for balancing, but chains can still be very unequal.

# Scan Ordering by NICE Rule

This risk-free ordering is guaranteed by `preview_dft`:



## NICE Rule:

The scan bit in **F3** must be read by **F4** before it is overwritten; thus Next Instance must be clocked Concurrently or Earlier

8-47

The slide shows part of a scan path in which both edges of the same clock are **mixed**. The path **crosses** over from one test-clock domain to another—separate—domain. The scan bit in **F3** must be read by **F4** *before* it is overwritten by the next scan shift. This rule applies to mixing both edges of the same clock, or to two different clocks. For the RTZ clock waveform shown, the **rising** edge occurs **earlier** in the tester cycle. Because rising-edge flops are clocked earlier, they must be put **further down** the chain.

## NICE Rule:

This risk-free ordering, based on test-clock waveforms, is created by `preview_dft`. It “parses” each pair of cells to ensure the **NICE** rule is met—if not, it reorders them. The rule is: the **next** instance in a scan chain must be clocked **concurrently** or **earlier**.

Previewing is based on **ideal** test-clock edge times—it does not consider clock skew.

# Mixing Clocks: Need Lock-ups

- Balanced paths with many clocks require lock-ups
- Greater potential for hold time or clock-skew issues if lock-up elements are not used
- As a precaution, lock-up latches (or Flip-Flop's) will be inserted
- Option `mix_clocks` is the most aggressive of all

```
set_scan_configuration -clock_mixing mix_clocks  
preview_dft
```

## Recommendation:

Always let DFTC insert lock-up latches!

```
set_scan_configuration -add_lockup true
```

8-48

These global settings permit better **balancing** at a higher **risk** of scan timing issues. Option `mix_clocks_not_edges` allows mixing **different clocks**, but on **one edge**. With this option, you could get just two chains, even with many different clock signals.

## Mix Clocks:

The most **aggressive** strategy, `mix_clocks`, allows full mixing of **all** clock domains. Flops clocked on different edges are ordered automatically according to the **NICE** rule. There is a potential risk—**hold violations** could occur if different clocks are **skewed**. This can happen from one scan cell to the next when  $t_{CLK-Q} + t_{net} < t_{hold} + t_{skew}$ . To prevent this, `preview_dft` inserts a **lock-up latch** before exiting a clock domain; thus a risk-free scan path is guaranteed—even in the presence of significant skew. Mixing clocks allows **optimal** balancing, but with the **overhead** of lock-up latches.

The following command determines whether latches or flip-flops are use as lock-up elements. The default is latches:

```
set_scan_configuration -lockup_type latch | flip_flop
```

# Rules for Inserting Lock-up Latches

- When a scan path crosses clock domains, two scenarios can occur:
  1. Different `set_dft_signal -timing`, No Skew:  
If the `-timing` phase changes, then scan flops are placed in NICE order and DFTC assumes the `-timing` phase difference is greater than worst-case clock skew between the clocks  
**No lock-up latches are inserted**
  2. Same `set_dft_signal -timing`, Risk of Skew:  
If the `-timing` phases are identical, then clock skew is a potential problem and clock-tree branches could be skewed  
**By default, a lock-up latch is inserted at the clock domain crossing for ½-cycle extra hold time margin**

8-49

This slide applies only to chip designs using **multiple** test clocks in the scan paths. It describes the assumptions DFTC makes when a scan path **crosses** clock domains. Keep in mind that total skew between test clocks may include **ATE-added** skew.

## Different Phases:

If you rely on clock **timing differences** instead of lock-up latches, then be aware of skew. The phase difference between test-clock domains must always **exceed** total clock skew. Here **phase difference** means the difference in arrival time of the active test-clock edges.

## Identical Phase:

If all test clocks have **identical** edge times, then by default **lock-up latches** are inserted. For a minimal overhead, these retiming latches provide an extra **half-cycle** of hold time. This prevents hold time violations on any scan path from one clock domain to another.

# Rules for Inserting Lock-up Latches

- + edge to + edge triggered devices
  - Lock-up Latch OK
- - edge to - edge triggered devices
  - Lock-up Latch OK
- + edge to - edge triggered devices
  - Will never happen by default in DFTC since it will always order later triggered devices earlier in the chain
- - edge to + edge triggered devices
  - No lock-up latch needed
  - Potential hold time issue if added

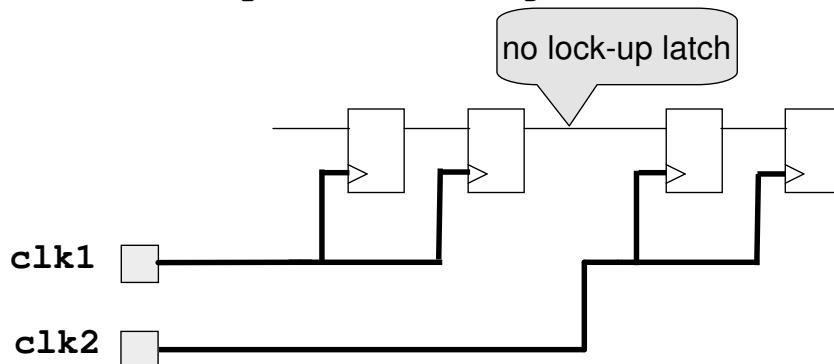
8-50

# Clock Equivalence

## ■ **set\_dft\_equivalent\_signals**

- Allows you to define clocks as equivalent for architecting purposes
- **insert\_dft** will NOT insert lock-up latches between clock domains defined as equivalent
- Example:

```
set_dft_equivalent_signals [list clk1 clk2]
```



8-51

# Test - How Many Scan Chains?

```

-chain_count      DEFAULT
-clock_mixing    no_mix
-internal_clocks none

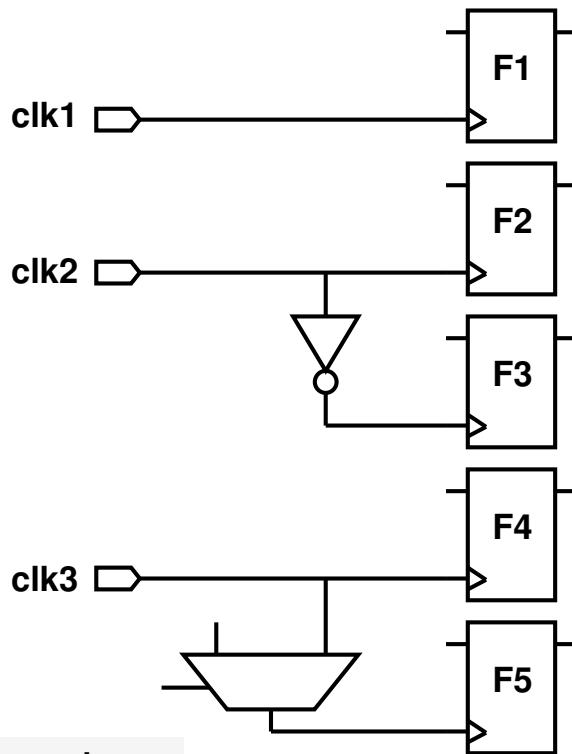
-chain_count      3
-clock_mixing    mix_edges
-internal_clocks none

-chain_count      1
-clock_mixing    mix_clocks
-internal_clocks multi

-chain_count      3
-clock_mixing    mix_clocks_not_edges
-internal_clocks multi

-chain_count      3
-clock_mixing    mix_clocks
-internal_clocks none

```



In what order are the elements of each chain?

8-52

→ 4 chains {F1} {F2} {F3} {F4, F5}

→ 3 chains {F1} {F3, F2} {F4, F5}

→ 1 chain {F3, F1, F2, F4, F5}

→ 3 chains {F1, F2} {F3} {F4, F5}

→ 3 chains {F1} {F3, F2} {F4, F5}

The number of scan chains being generated depends on several specifications. By default (no clock mixing) DFT Compiler generates one scan chain per clock domain. A clock domain contains all FFs clocked by the same edge of a clock. DFT Compiler treats a dedicated external source (input port) as a clock. If you specify a chain count greater than the number of clock domains, DFT Compiler will split the longest chains into smaller chains. Note that the scan chains probably do not have equal length without clock mixing.

If you allow mixing of clock edges within a scan chain, the default chain count equals the number of clocks. Again, you can specify a greater number of scan chains. With mixing of edges, the probability that you get equal length scan chains is higher than without any clock mixing.

If you allow any arbitrary mixing of clocks within scan chains, DFT Compiler will generate equal length scan chains. This results in a minimum scan chain length, given a specific chain count. DFT Compiler orders the elements within a chain by clock domain, and inserts lock-up latches between all adjacent elements clocked by different clocks (but with identical waveform), unless you have specified `-add_lockup false`.

By enabling recognition of internal clocks DFT Compiler will create new clock domains. Whenever a clock net passes a cell with at least two input pins, the cell output will be considered a separate clock domain.

# Top-Down Scan Insertion: Agenda

Specify Scan Configuration

Preview Scan Configuration

Insert DFT Logic

Balance Scan Chains

Finer Control

8-53

# Scan Path Exclusion

**Valid, scan-compliant (DRC clean) flip-flops are excluded from scan paths if:**

1. The cell has been “excluded” with  
`set_scan_configuration -exclude_elements`
  
2. The cell has a `scan_element` attribute set to false:  
`set_scan_element false { F59 F60 F71 }`



**Alert:**

Flip-Flops **excluded** from scan paths appear as **black boxes** to combinational ATPG tools—only sequential ATPG handles them.

**8-54**

`set_scan_configuration -exclude_elements` will not stop `dft_drc` from reporting violations on these cells.

You can purposely **exclude** a few critical flip-flops from being included in scan paths. For example, those flip-flops might be on a critical path with tight timing constraints, or the flip-flops might be part of the TestMode entry logic. Use can use `set_scan_element false` or `set_scan_configuration -exclude_elements` to deliberately exclude a flip-flop or a whole block. High fault coverage is still attainable with a full-sequential ATPG tool like TetraMAX.

## Caveats:

**Be careful of `dont_touch` attributes—they are intended for nonscan synthesis only. Unexpected `dont_touch` attributes in logic can hinder scan replacing and autofixing.**

# How to Control Unscanning

- Unscanning refers to returning a DFT-violating scan flip-flop back an ordinary flip-flop

```
set_dft_insertion_configuration  
    -unscan true | false
```

- To specify cells to be excluded from the scan chain but remain as a scan flip-flop (not unscanned)

→ use `set_scan_configuration -exclude`

- To specify cells to be excluded from the scan chain that also can be “reverted back to non-scan” (unscanned) during `insert_dft`

→ use `set_scan_element false`

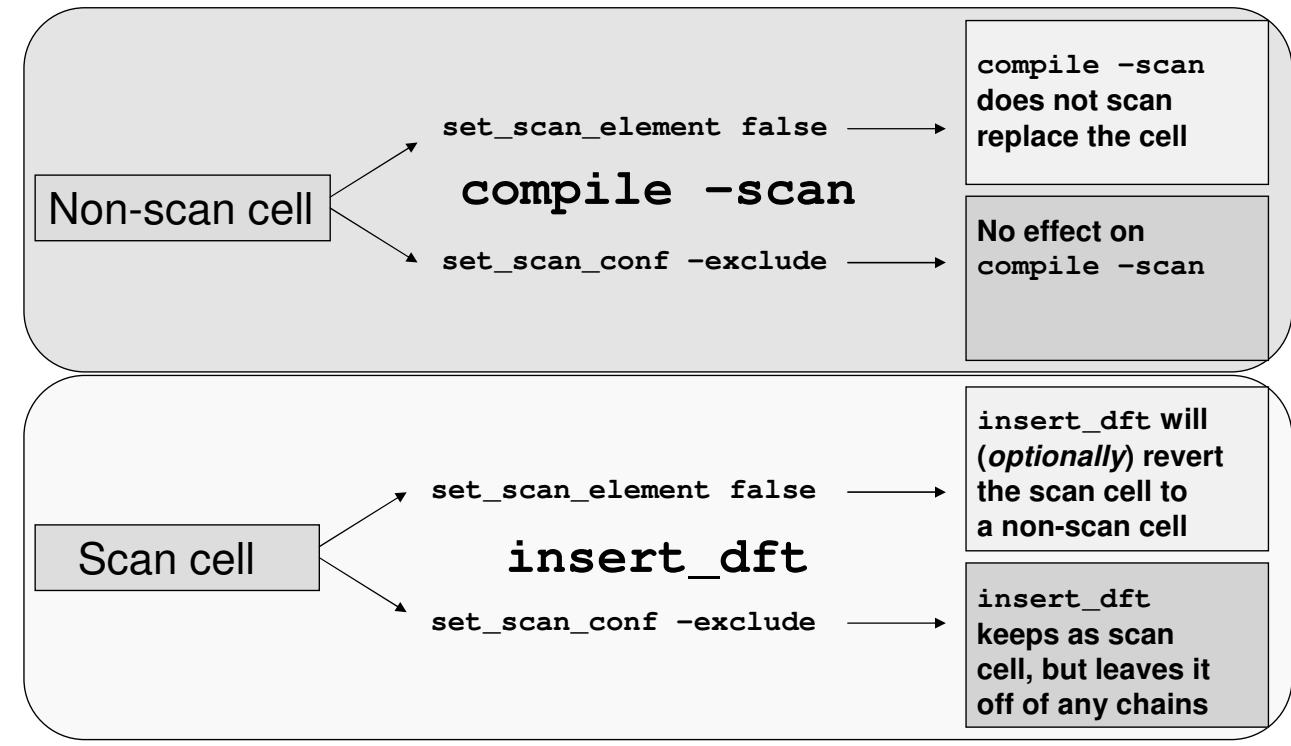
8-55

You can purposely **exclude** scan flip-flops from being included in scan paths and have them remain as scan flip-flops. For example, those flip-flops might be spare registers.

**Note:** unscanning of flops does not occur if either synthesis optimization is disabled (`-synthesis_optimization none`) or if using DCT

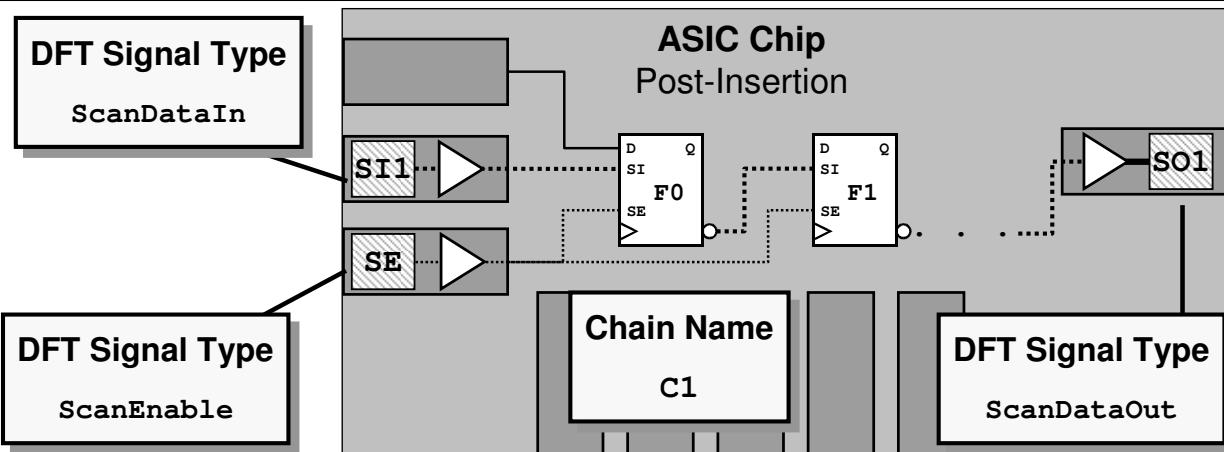
# Unscan/Exclude Behavior

`set_scan_element vs. set_scan_configuration -exclude`



8-56

# What Defines a Scan Chain?



```
set_dft_signal -view spec -type ScanDataIn -port SI1  
set_dft_signal -view spec -type ScanDataOut -port SO1  
set_dft_signal -view spec -type ScanEnable -port SE \  
    -active_state 1  
set_scan_path -view spec C1  -scan_data_in SI1 \  
    -scan_data_out SO1 \  
    -scan_enable SE
```

8-57

This slide shows part of a **silicon chip**, looking down on some I/O cells and core logic.

## Default Scan Ports:

By default **insert\_dft** synthesizes all scan-access ports it needs on the current design. In this example, the dedicated ports **SI**, **SE**, **SO** in the diagram were already in the RTL design.

## Scan-Port Declarations:

Port declarations **steer** a scan data or control signal through designated **ports** or **pins**. Declaring scan ports is an **optional** step. In the example above, **preview\_dft** is instructed by **set\_dft\_signal** to use **SI1** for scan-in. Due to the declaration, inserted scan path **C1** will start from the designated port **SI1**.

## Conclusion:

You can declare port **SE** in RTL code, declaring it as the **ScanEnable** port. You can **share** scan-out, scan-in, etc, with existing **functional** ports to reduce pin count.

In addition, the **set\_scan\_path** command can be used to further specify how the scan chain will be built by declaring which ports to use for the scan access and which cells are to be placed on a particular chain

# The `set_scan_path` command

- The `set_scan_path` command is used to control how the scan chains are built on a chain-by-chain basis

```
set_scan_path <name> [-view spec | exist] ...
```

- The `set_scan_path` command can specify:

- The scan chain name
- Which scan ports are used for a particular chain
- Which scan cells are used for a particular chain
- Whether a terminal lock-up will be added to the chain
- The length of the chain
- Existing scan chains

8-58

# Scan Chain Access Ports

- The `set_scan_path` command can control the scan access ports with the following options

```
set_scan_path <scan_chain_name>
  -view spec
  -scan_data_in <ScanEnable_port>
  -scan_data_out <ScanOut_port>
  -scan_enable <ScanEnable_port>
  -scan_master_clock <ScanClock_port>
```

- Example:

```
set_scan_path c1 -view spec -scan_data_in SI1
  -scan_data_out SO1 -scan_enable SE
  -scan_master_clock CLK
```

8-59

# Scan Chain Elements

- The `set_scan_path` command can control the elements of the scan chain with the following options

```
set_scan_path <scan_chain_name>
  -view spec
  -include_elements <include_list>
  -ordered_elements <ordered_list>
  -head_elements <head_list>
  -tail_elements <tail_list>
  -complete [ true | false ]
```

- A **TCL** list, *not a “collection”*, must be provided with the `*_elements` options

8-60

# How to Specify a Scan Path Using Lists

- The `set_scan_path` command does not accept a collection, you must pass it a list when using `-include_elements`, `-ordered_elements`, `-head_elements`, or `-tail_elements`
- Example:

```
set core1 [get_cells u_core1/*reg*]
set_scan_path my_chain1 -view spec \
    -include_elements [get_object_name $core1]
```

8-61

Notes:

Please see <https://solvnet.synopsys.com/retrieve/015281.html>

If user ran:

```
set_scan_path my_chain1 -view spec \
    -include_elements [get_cells u_core1/*reg*]
```

User will not get an error but `preview_dft` will ignore this specification

Another example of how to look for registers under a certain hierarchy path:

```
set my_cell_core1 [filter_collection [all_registers -cells] \
    "full_name=~u_core1/*"]
```

## Example: -ordered\_elements

- How to use `-ordered_elements` to fix the order of elements in a scan chain

```
set_scan_path core_chain2 -view spec \
    -ordered_elements [list \
        my_modB/D_reg \
        my_modB/A_reg \
        my_modB/C_reg \
        my_modA/F_reg \
        my_modA/B_reg \
        my_modA/E_reg ] \
    -complete true
```

Specify if the  
chain is  
“complete”

8-62

Note:

If `-complete true` is omitted (or `-complete false` is used), then other cells could be added to chain `core_chain2` to get balanced chains.

# Explicit Chain Length and Clock Control

```
set_scan_path scan_chain_name
```

```
[-exact_length <integer>]
```

```
[-scan_master_clock <clock_name>]
```

- You can specify the length and the clock associated to a particular scan chains
- If chain length specification cannot be met, a warning message will be printed and the specification ignored

```
set_scan_path -insert_terminal_lockup true
```

- **preview\_dft** shows where terminal lock-up latches (1) will be inserted

```
Scan chain '1' (test_sil --> test_so1) contains 1 cell:  
CURRENT_STATE_reg (1) (CLK, 45.0, rising)
```

8-63

Local setting for terminal lock-up per chain

**-insert\_terminal\_lockup true | false**

Specifies a lock-up latch to be associated at the end of the scan chain. Scan architect takes the user-defined lock-up latch specification into consideration while building scan chains.

# Multiple Scan Enables

- DFT Compiler allows more than one dedicated scan enable for a design
- You can specify multiple scan enables and associate them with particular scan chains with the `set_scan_path` command

## Example:

```
set_dft_signal -type ScanEnable -port sel ...
set_dft_signal -type ScanEnable -port se2 ...
set_scan_path c1 -scan_enable sel
set_scan_path c2 -scan_enable se2
```

8-64

A dedicated port can be associated to one or more specified scan chains.

All scan enable pins of scan elements, not belonging to any specification, will get connected together.

# Example Multiple Scan Enable Preview

```
Preview DFT report
```

```
...
```

```
Scan chain 'c1' (test_sila --> rsv[22]) contains 6 cells:
```

```
...
```

```
Scan signals:
```

```
test_scan_out: rsv[22] (no hookup pin)
```

```
test_scan_enable: sel (no hookup pin)
```

```
Scan chain 'c2' (test_si2a --> test_so2a) contains 4  
cells:
```

```
...
```

```
test_scan_enable: se2 (no hookup pin)
```

8-65

# Managing Scan Paths

- You can report on specified scan paths using

```
report_scan_path -view spec
```

- You can remove scan path specification using

```
remove_scan_path -view spec -chain <name>
```

- After scan insertion (`insert_dft`), the scan chains that were actually *implemented* will be seen as part of the “`existing_dft`” view

```
report_scan_path -view existing_dft
```

8-66

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- **Insert scan chains into a design using a top-down flow**
- **Specify and insert balanced top-level scan chains**
- **Reuse existing functional top-level pins for scan**
- **Specify and preview scan chain architectures**

**8-67**

# Lab 8: Top-Down Scan Insertion



45 minutes

**After completing this lab, you should be able to:**

- **Preview the results of different scan specifications before implementing one that yields well-balanced top-level chains**
- **Perform the Mapped DFT flow for a test-ready design through scan insertion and test coverage estimation**

**8-68**

# Command Summary (Lecture, Lab)

<code>set_scan_configuration</code>	Specifies the scan chain design
<code>report_scan_configuration</code>	Displays options set by <code>set_scan_configuration</code>
<code>reset_scan_configuration</code>	Resets the scan configuration for the current design
<code>set_scan_element</code>	Sets the <code>scan_element</code> attribute on specified design objects, to determine whether <code>insert_dft</code> replaces them with scan cells
<code>preview_dft</code>	Previews, but doesn't implement, the scan architecture
<code>set_dft_insertion_configuration</code>	Sets the DFT insertion configuration for current design
<code>insert_dft</code>	Adds scan circuitry to the current design
<code>set_dft_equivalent_signals</code>	Sets a given list of DFT signals as equivalents
<code>set_scan_path</code>	Specifies a scan chain for the current design
<code>report_scan_path</code>	Displays scan paths and scan cells in the scan paths set by <code>set_scan_path</code> command. Also displays scan paths inserted by <code>insert_dft</code>
<code>remove_scan_path</code>	Remove the scan path specification for current design

8-69

# **Appendix**

## **Scan Groups**

# Scan Groups

- **Scan group: A group of scan cells that should be grouped together during scan stitching**
  - No other cells will be inserted between elements of the group
  - Scan architect may optionally order the cells within the group
- **Benefit: gives users more flexibility in customizing their scan architecture**

8-71

Scan group:

- cells within the group will be grouped together
- other cells not inserted inside the group
- Ordering within the group is up to DFT Compiler

# Scan Group Commands

- To specify a new scan group:

```
set_scan_group <name_of_group>
```

- To remove previously specified scan group, before insert\_dft:

```
remove_scan_group <name_of_group>
```

- To report on a previously specified scan group:

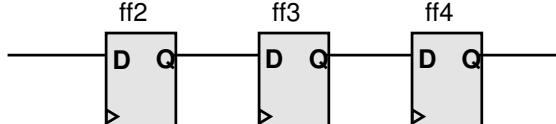
```
report_scan_group
```

- Scan group specifications are not cumulative

- Previously specified scan group will be overwritten by a new scan group of the same name

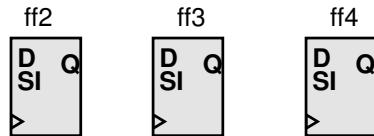
8-72

# Using set\_scan\_group



**Examples:**

```
set_scan_group existing_segment1
  -access [list ScanDataIn ff2/D ScanDataOut ff4/Q]
  -serial_route true
  -include_elements [list ff2 ff3 ff4]
```



```
set_scan_group keep_together
  -serial_route false
  -include_elements [list ff2 ff3 ff4]
```

**8-73**

Scan segments are a special case of scan group where the serial connections are already routed

Scan order within the segment will be preserved

Must specify elements and access pins

A scan segment is defined as a pre-existing set of flops where the serial connection already exists.  
A typical scenario is when a group of flops is hooked up as a shift register. DFTC will reuse the existing shift register for scan insertion.

# Using set\_scan\_group

- To put individual flops into a scan group:

```
set_scan_group group1 -include {U018 U011 U002}
```

- To put a core segment into a scan group:

```
set_scan_group group1  
-include {U018 U011 U000/1}
```

- To put a previously defined group as part of a scan path:

```
set_scan_path chain0  
-ordered_elements [list group1 U003]
```

8-74

Scan group specifications are not cumulative

Previously specified scan group will be overwritten

Example:

```
set_scan_group group1 -include {U018 U011 U002}  
set_scan_group group1 -include {U008 U001 U007}  
⇒group1 is now {U008 U001 U007}
```

Scan group limitations:

- Cells in a scan group must be within same clock domain  
`preview_dft` will issue TEST-400 warning saying a scan group has elements clocked by different clocks, or same clock but different edges
- Collections cannot be part of a scan group
- Nested scan groups not allowed
- A scan path cannot be part of a scan group
- Groups cannot be mode specific. A specified group will be used in all test modes.

# What is Supported for Members of a Group?

## ■ YES:

- Cells
- Design instance names
- Core segments from test models

## ■ NO:

- Another scan group
- A collection
  - ◆ Workaround:  
`set_scan_group <name_of_group>`  
`-include [get_object_name <name_of_collection>]`
- Chains defined by `set_scan_path`
- Elements previously defined in another scan group or scan path
  - ◆ Second scan group specification will be discarded

8-75

This page was intentionally left blank

# Agenda

**DAY  
3**

**9 Export**



**10 High Capacity DFT Flow**



**11 Multi-Mode DFT**



**12 DFT MAX**



**13 Conclusion**

# Unit Objectives



After completing this unit, you should be able to:

- Name 2 files that DFTC must write to handoff a scan design
- Write a SCANDEF file that can be taken to ICC for scan chain reordering
- Generate a SVF file that can be used by Formality for formal verification

9-2

# Exporting Design Files: Agenda

ATE Concepts

Writing Test Protocols

SCANDEF

Formality

9-3

# ATE: Final Destination of Handoff



- The objective of this unit is to demystify common ATE-related issues
- Focus will be on tester-neutral features common to all ATE
- SOC designers must understand ATE-specific limitations

9-4

- Verigy's 93K ATE is shown here to illustrate state-of-the-art SOC test equipment.
- This slide is not intended as a Synopsys endorsement of any ATE vendor or product.
- The ATE industry is attempting to meet growing system-on-chip testing requirements.
- High frequency, high pin count, and on-chip analog signals are critical SOC test issues.
- The **test head** in the illustration contains the actual high-speed **pin electronics** cards.
- **Production ATE**, unlike prototype testers, is aimed at **high throughput**, not ease of use.
- A popular alternative: lower-cost **prototype testers**, optimized for use by **designers**.

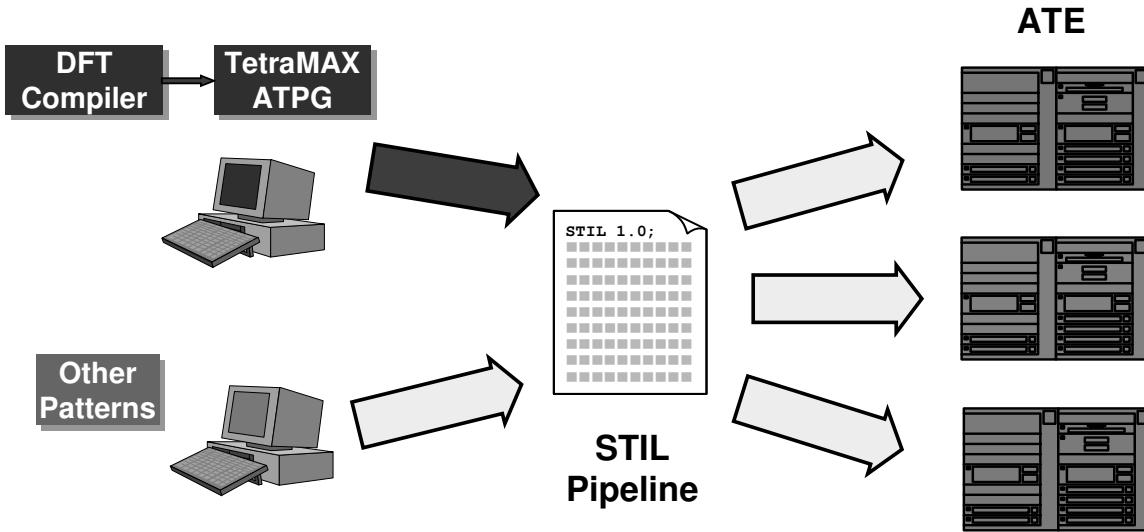
## ATE Vendor Information:

Verigy: [www.verigy.com](http://www.verigy.com).

Credence Systems: [www.credence.com](http://www.credence.com).

Teradyne Inc.: [www.teradyne.com](http://www.teradyne.com).

# Protocol Path from DFTC to ATE



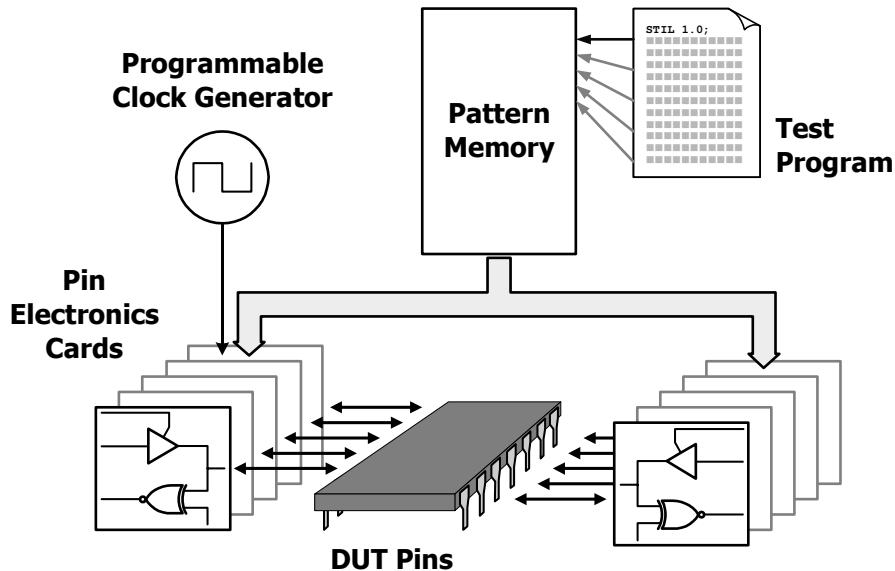
**STIL is a vendor-neutral IEEE standard pipeline between Test Tools and ATE**

**9-5**

Standard Test Interface Language (STIL) is an **industry standard** (IEEE Std 1450-1999).

It allows **tester-neutral** description of test patterns, pin-timing protocol, and scan paths; thus STIL is a single language for moving data between multiple ATPG tools and testers. TetraMAX supports STIL (with extensions yet to be standardized) as its **native language**. A STIL program can be fed—**without translation**—to any IEEE-1450-compliant tester. For noncompliant testers, you can use a third-party translator like Source III VTRAN. The STIL syntax is optimized for **fast** loading of **large** volumes of test data into the ATE. It is **not** optimized for human readability—but most users will not need to edit STIL code.

# Inside the ATE



Generic ATE Architecture

9-6

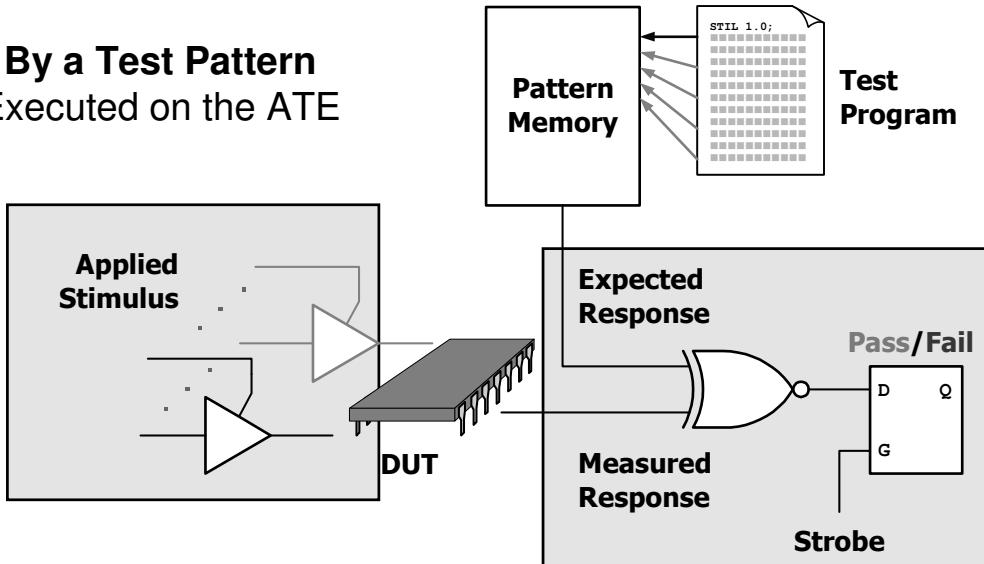
This block diagram emphasizes the parts of an IC tester most relevant to the logic designer. The **device under test** (DUT)—either a packaged IC, or a die still on the original wafer. In either case, the DUT is held in a **test fixture** providing access to all of its primary ports. Local **pin-electronics cards** enable individual PIs to be driven and POs to be observed. Programmable **clock generators** synchronize the DUT to the ATE during the test program. All clocks fed into the DUT are ordinarily of the **same frequency**, though phase may differ. The **test program** is generated by an ATPG tool and the patterns are loaded into memory. In turn, each pattern is read out of **pattern memory** and parsed into stimulus and response. The pattern is **executed** by applying the stimulus to the DUT and observing the response.

## Conclusion:

Designers should always be aware of the various assumptions made by the target ATE. For example, pattern-memory **depth** may limit the **file length** of a loaded test program.

# How a Fault is Detected

By a Test Pattern  
Executed on the ATE



## Failure Criteria:

Measured DUT response fails to match the expected response

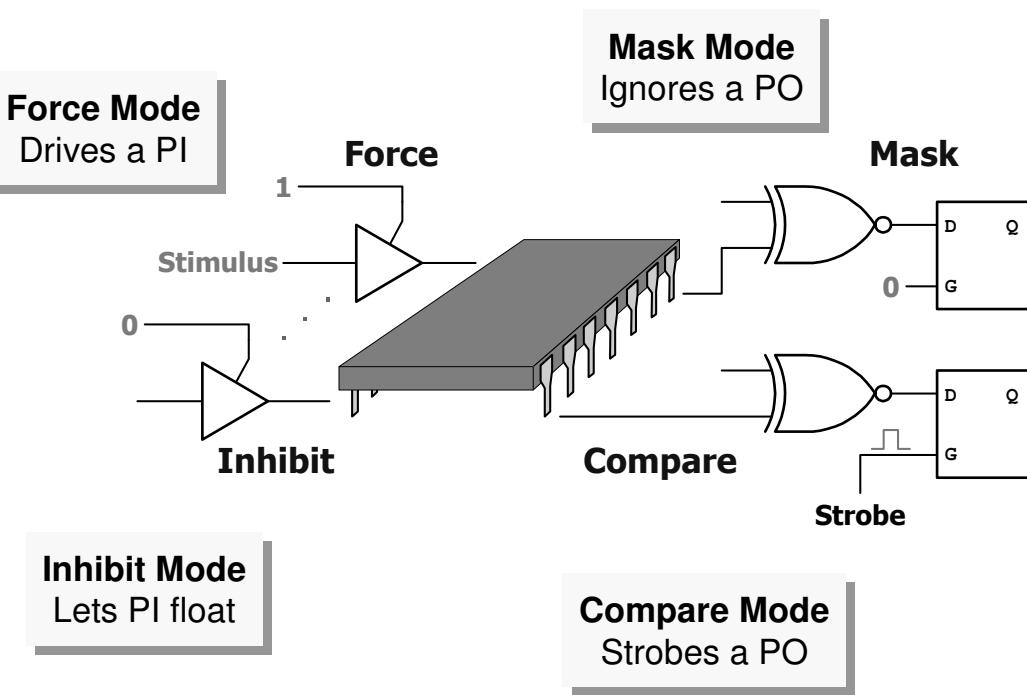
9-7

Earlier in Unit 1, you saw a textbook **criterion** for detecting a target fault in combinational logic. An applied input  $i$  detected fault  $f$  if and only if output  $Y_f(i)$  differed from output  $Y(i)$ . This slide suggests how this fault-detection criterion is implemented in ATE hardware. The **pin-electronics** cards, which lie physically close to the DUT are emphasized for you.

The diagram is only **conceptual**—thus, differential comparators replace XNOR gates. The flow of events begins when the current test pattern is read out of pattern memory. From the pattern, the ATE derives the **input stimulus** and applies it to the DUT PI ports. After an adequate settling time, the steady-state **output response** at the POs is measured. If the measured DUT response **differs from** the expected response, the fault is **detected**. The **pass/fail** result is strobed into local pin memory, and the failure is logged to a file.

In such case, the defective DUT is sent to the **fault diagnosis** lab—or simply discarded.

# Individual Pin Modes



9-8

- Inside the ATE, the **pin-electronics card** is the basic interface to each DUT pin.
- This slide shows four basic modes that can be individually applied to any pin.
- This slide emphasizes the **function** of each mode—event **timing** will be covered later.

## Modes Defined:

In **force mode**, the ATE drives the PI to a strong 0 or 1 (STIL characters D or U).

In **inhibit mode** the ATE leaves the PI pin undriven and floating (STIL character Z).

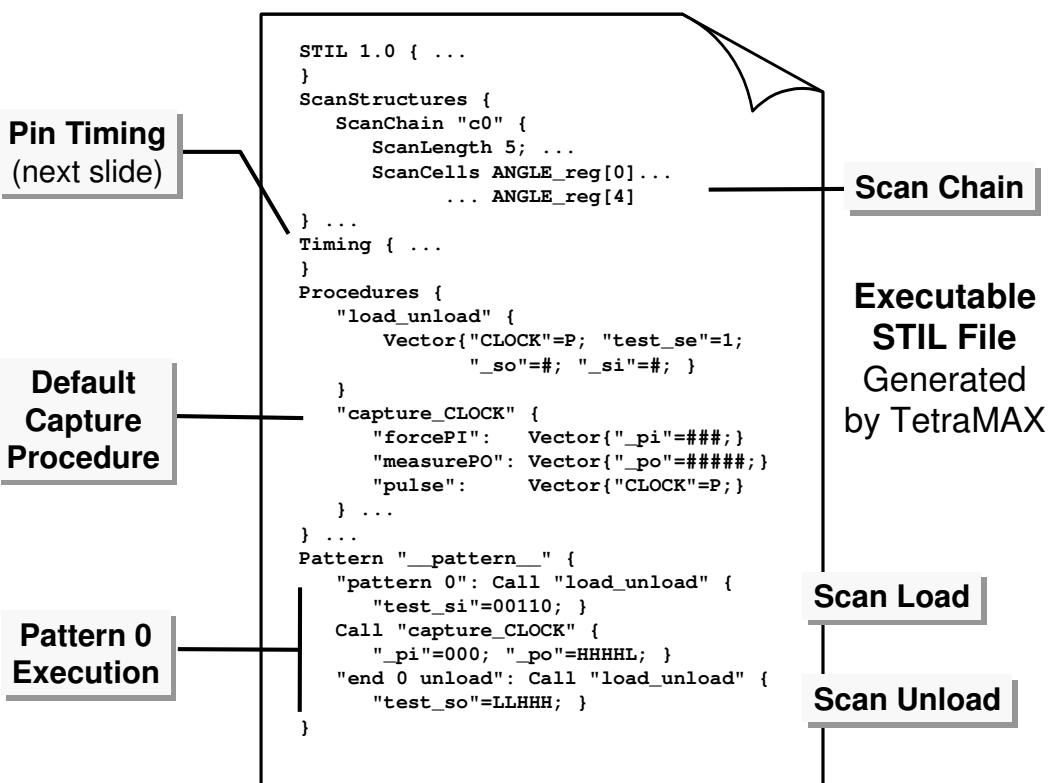
In **compare mode**, the ATE strobes the PO for an expected response (STIL L or H).

In **mask mode**, the ATE ignores the PO pin, and does not compare it (STIL X).

The above STIL **waveform event** characters are predefined by IEEE Standard 1450.

See, for example, IEEE 1450, Section 18.2, Tables 9 and 10, for more information.

# ATE “Executes” STIL Test Program



9-9

This slide shows a typical **test program** written out by DFTC or TetraMAX in STIL format:  
The goal is not to learn the syntax, but to understand several key **STIL concepts**.

This sample DUT has a five-bit scan chain, labeled **c0**.

This STIL test program uses **procedures** to concisely describe repetitive actions.

Procedure **load\_unload** asserts SE and shifts serial stimulus data into chain **c0**.

After loading, the scan chain **ANGLE\_reg[4:0]** will contain the scan data **00110**.

Pattern 0 targets a **SA1** fault that, if present, freezes the **MSB** of the counter at **1**.

The structure of the **STIL** code shows that pattern 0 is executed in **three phases**.

The phases involve calls to **load\_unload**, **capture\_CLOCK**, and **load\_unload**.

The **expected** data captured into the scan chain, and later scanned out, is **LLHHH**.

If the ATE measures **H** instead of **L** for the **MSB**, then the **SA1** fault is **present**.

## Conclusion:

The test program describes every detail needed for the ATE to execute patterns.  
Even the **pin timing** is included in the **STIL** program—as the next slide shows.

# Pin Timing from Protocol in STIL Format

```
STIL 1.0 { ...
}

...
Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            Force Mode --> "all_inputs" { 0 { '0ns' D; } }
            Force Mode --> "all_inputs" { 1 { '0ns' U; } }
            Force Mode --> "all_inputs" { Z { '0ns' Z; } }

            Inhibit Mode --> "all_outputs" { X { '0ns' X; } }
            Inhibit Mode --> "all_outputs" { H { '0ns' X; '40ns' H; } }
            Inhibit Mode --> "all_outputs" { L { '0ns' X; '40ns' L; } }
            Inhibit Mode --> "all_outputs" { T { '0ns' X; '40ns' T; } }

            "CLOCK" {P { '0ns' D; '45ns' U; '55ns' D; } }
        }
    }
} ...
```

- Letters in blue are predefined STIL waveform events
- Time points in green are DFTC pin-timing parameters
- Signals like CLOCK are specific to the design under test      **9-10**

This excerpt from the previous **test program** was also written out by DFTC. It describes the **pin-timing protocol** used by **load\_unload**, **capture\_CLOCK**, etc. **Waveform event** characters like **U** (up) and **H** (high) are predefined in the STIL spec. Line items for inputs and outputs correspond to the four **pin modes** you learned earlier. This waveform table tells you **when** events happen, and **where** (at which primary ports).

## Examples:

- During scan-shift cycles, **SI** is always forced (**U** or **D**) at exactly **0 ns** into the cycle.
- An expected **low** or **high** (**L** or **H**) at **SO** is measured or strobed at **40 ns** into the cycle.
- During scan-shift cycles, the active edge of **CLOCK** (**U**) occurs at **45 ns** into the cycle.
- During capture cycles, ignored outputs are **masked** (**X**) from **0 ns** to the next compare.

# Exporting Design Files: Agenda

ATE Concepts

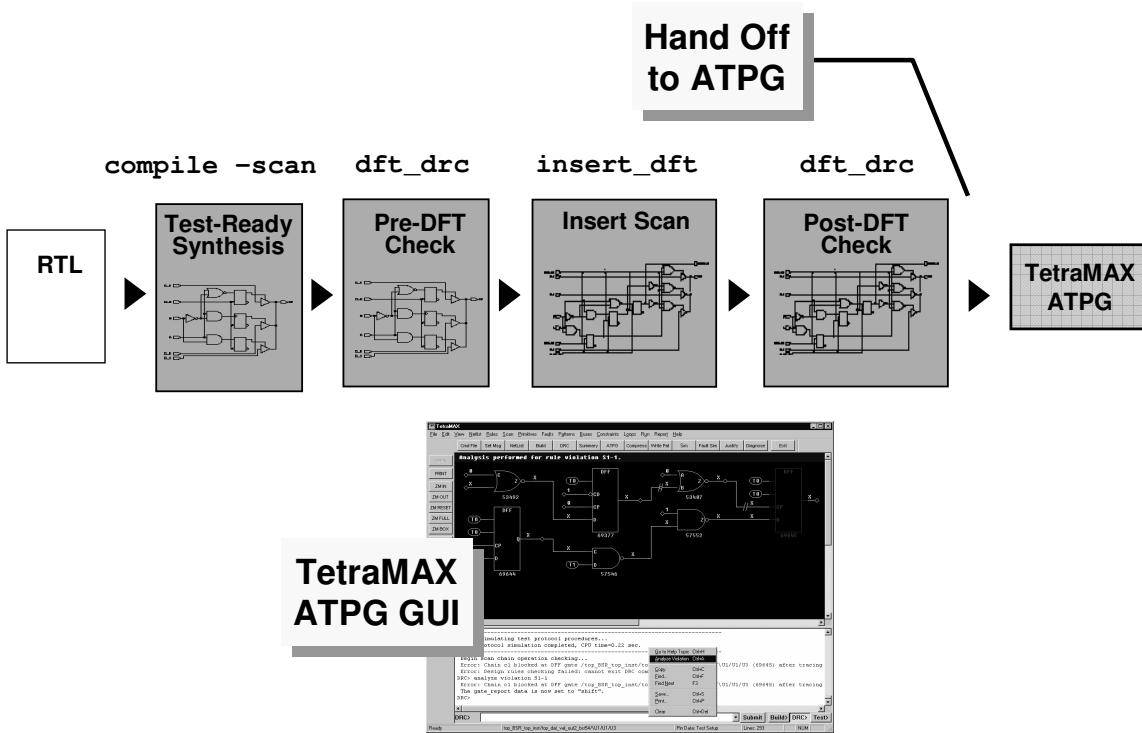
Writing Test Protocols

SCANDEF

Formality

9-11

# DFTC-to-TetraMAX Flow



**9-12**

Guidelines for preparing to export a design to TetraMAX include:

Fix as many **dft\_drc** warnings as you can while still in the DFTC environment.

TetraMAX requires valid scan paths. Prior to export, confirm that paths are intact using **report\_scan\_path -view existing**

Document the presence of nonscan flip-flops and latches, alerting the test team to the need for sequential ATPG.

# DFTC Handoff Script

```
# DFTC-to-TetraMAX Handoff Script
# Write out gate-level netlist in Verilog format.
# Save the final test protocol in STIL format.

change_names -rules verilog -hierarchy
write -format verilog \
      -hier RISC_CORE \
      -out mapped_scan/RISC_CORE.v
write -f ddc -hier -out mapped_scan/RISC_CORE.ddc

set test_stil_netlist_format verilog

write_test_protocol \
      -out tmax/RISC_CORE.spf
```

**Write Netlist**

**Save Protocol**

**9-13**

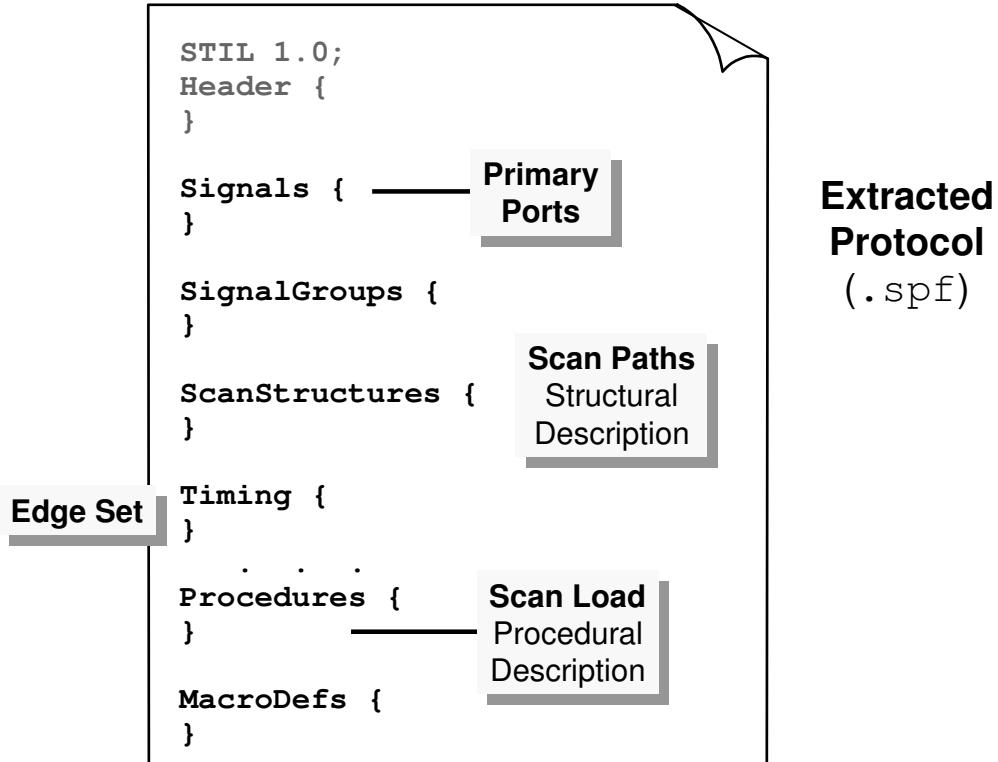
- TetraMAX supported netlist formats are: Verilog, VHDL and EDIF.
- TetraMAX reads neither **.db** nor **.ddc** formats; doing so results in an **N1** error.
- Specifying a **netlist format** allows for **HDL-specific** naming conventions—for example: Verilog **escaped characters** in module and signal names.
- Similarly, buses and bus syntax (brackets vs. parentheses) can be HDL-specific.
- Remember that **DFTC** variables can affect the format of the generated netlist.
- TetraMAX cannot handle VHDL primary ports using **array of array** types.

# What Affects the Test Protocol?

- Design features like clocks and resets define with `set_dft_signal`
- Constants, Scan In/Outs/Enables defined with `set_dft_signal`
- Scan specifications such as `set_scan_path`
- Test variables (`test_default_*`) such as `test_default_strobe`

9-14

# STIL Protocol File Structure



9-15

The protocol file conveys the following information from DFTC to TetraMAX:

Declarations of all **scan-access** ports (**SI**, **SO**, **SE**, etc);

Declarations of state-changing inputs (**clocks**, in TetraMAX terminology).

**Pin timing** parameters for clocks, input stimuli, and output measures.

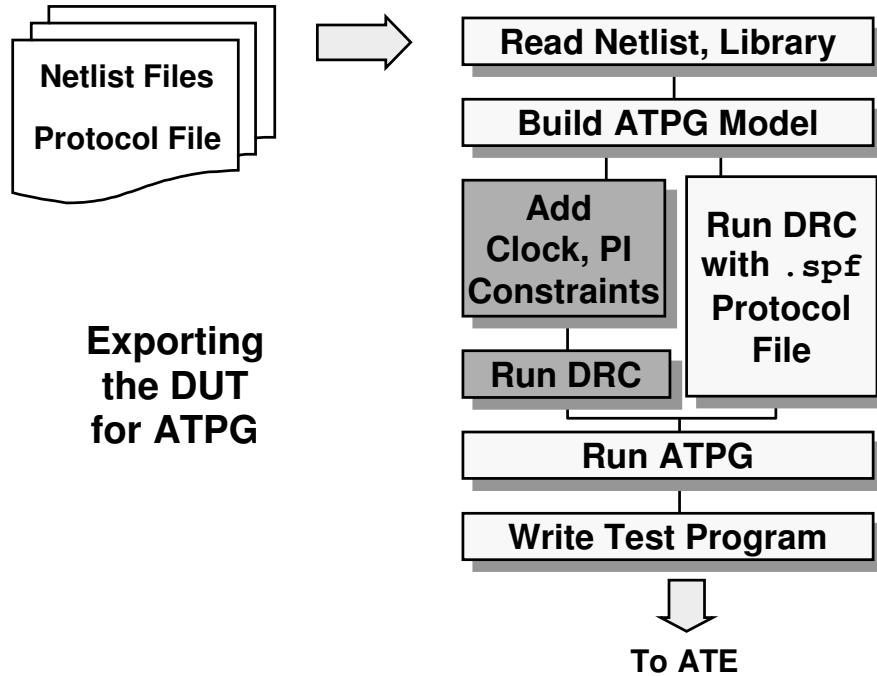
**Constraints** on ports, including equivalence relationships.

Custom initialization sequence (**test\_setup** macro definition).

Scan-shift sequence (**load\_unload** and **shift** procedures).

Optional custom procedures (e.g., **master\_observe**, **shadow\_observe**).

# TetraMAX Baseline Flow



9-16

The easiest way to export to TetraMAX is using the STIL protocol file:

Synopsys fully supports the intent of the IEEE 1450 STIL standard.

The STIL language is, in effect, the **native language** of TetraMAX.

As a leading-edge tool, however, TetraMAX uses STIL **extensions**.

These extensions are part of a **preliminary** standard, IEEE P1450.1.

## Technical References:

See **TetraMAX User Guide**, App. E, for details on the use of STIL P1450.1 syntax.

See **Exporting to Other Tools User Guide**, Chapter 1, for more on exporting to TetraMAX.

Also see **TetraMAX User Guide**, App. B.

## Alternative Flow:

You can describe some protocol details **manually**, using TetraMAX commands.

For example, **add\_clocks** can be used to identify clock input ports and resets.

Similarly, **add\_pi\_constraints** is like DFTC's **set\_dft\_signal -type Constant** command.

# Exporting Design Files: Agenda

ATE Concepts

Writing Test Protocols

SCANDEF

Formality

9-17

# Physical DFT

## ■ Scan synthesis done in DC

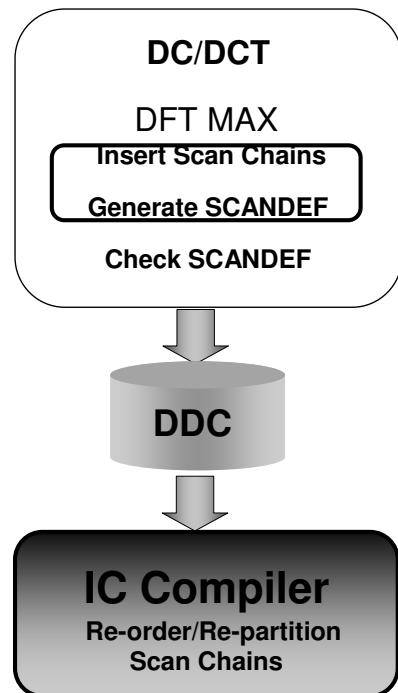
- SCANDEF flow to ICC
- `check_scan_def` command validates SCANDEF in DC/DCT to reduce iterations between DCT-ICC

## ■ Physical Optimizations in IC Compiler

- Scan chain re-partitioning and re-ordering (Placement and clock tree based)
- Reduced scan wire length, congestion & routability
- Hierarchical design support

## ■ Supported SCANDEF flows

- Basic scan & adaptive scan
- Multi-mode scan
- Multi-voltage support
- Internal pins
- Memories with test-models



9-18

# Example: DFT Optimization

## NO DFT Optimization

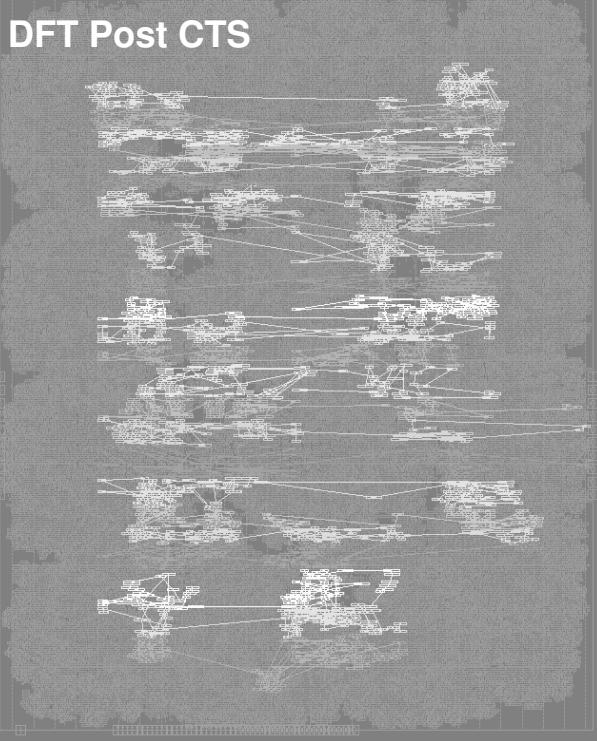
- Largely arbitrary interconnect

## Post Placement

- Chains globally optimized

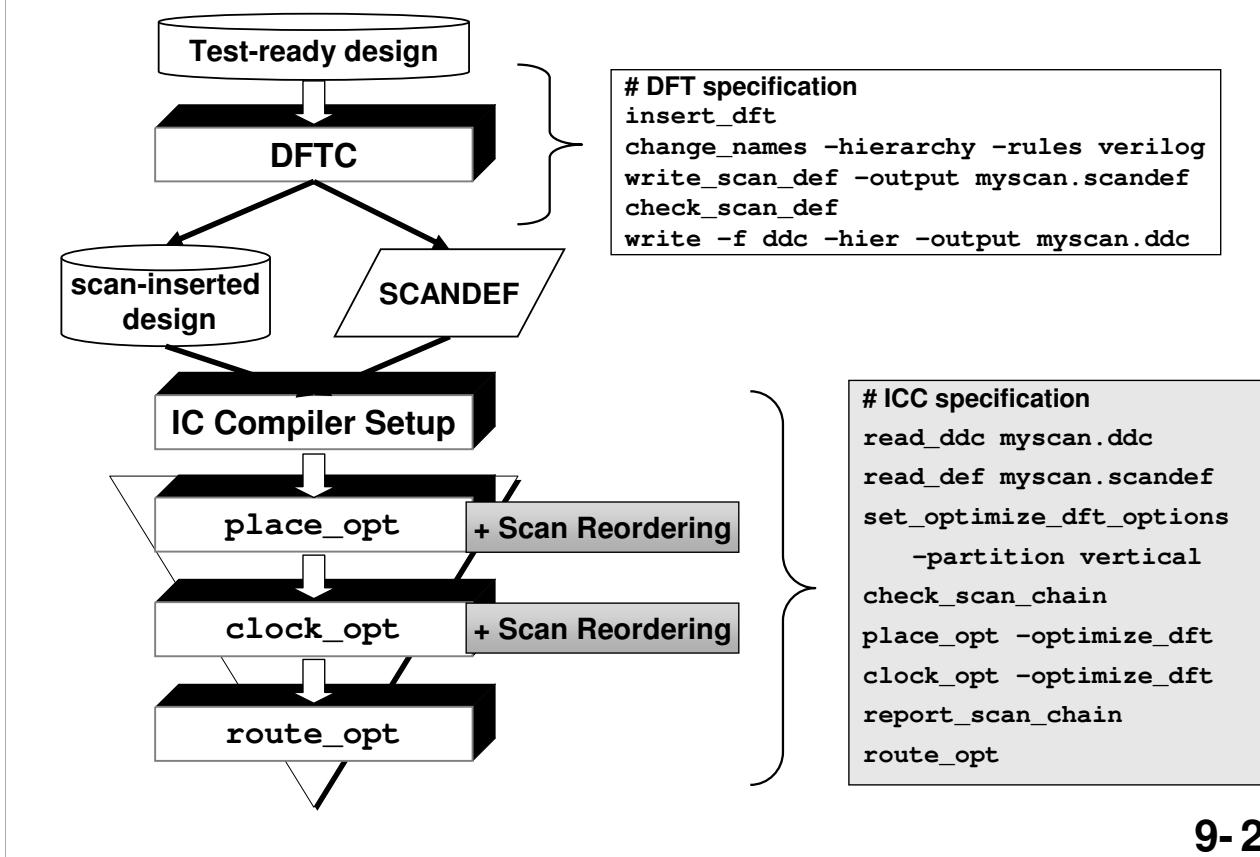
## Post CTS

- Chains locally re-optimized
- CTS Crossings : 1591 -> 706



**9-19**

# DC / IC Compiler DFT Flow

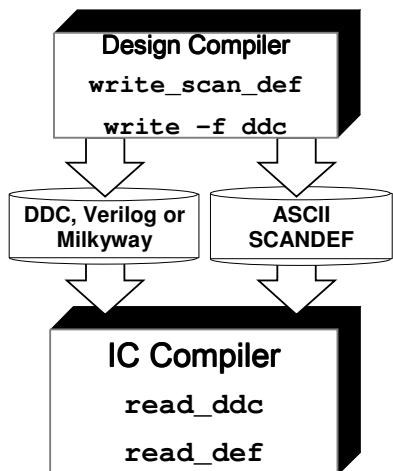


9-20

# DDC Based SCANDEF Data

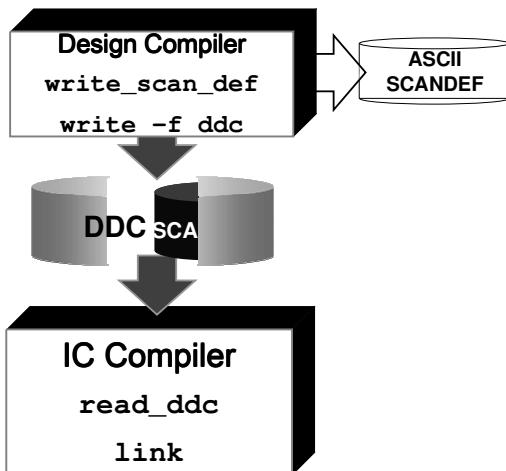
- *Eases Management of Scan Chain Data*

Prior to 2007.12-SP2



IC Complier required ASCII  
SCANDEF for scan chain  
reordering

2007.12-SP2



ASCII SCANDEF no longer  
required; part of DDC database

9-21

# DDC Based SCANDEF Benefits

- **Scan reordering data is part of the DDC database**
  - Eases file management
  - Streamlines the DC hierarchical flow for scan chains
  - Replaces use of extra ASCII-based SCANDEF files
- **Scan reordering data attached to each DDC block**
  - Streamlines hierarchical DC flow for scan chains
  - Top-level DDC contains ‘expanded’ scan data
  - ASCII SCANDEF is optionally available
- **IC Compiler automatically extracts scan data from the DDC during link**
  - Optimally reorders and repartitions scan chains based on physical placement
  - Failure to extract SCANDEF from DDC does not disrupt design loading

9-22

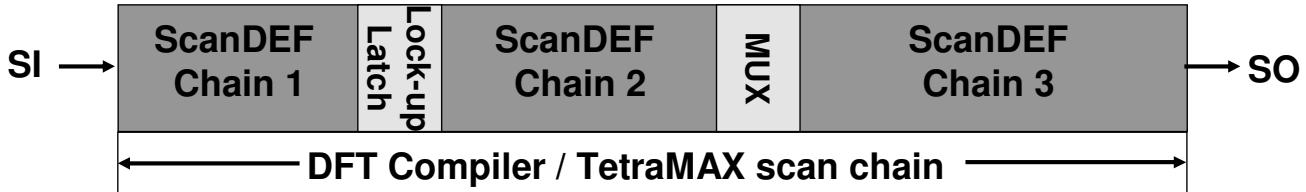
Attaches to DDC (not Milkyway) and top-level blocks top when writing hierarchical SCANDEF

SCANDEF is an open ASCII standard to transfer scan chain data for physical-based reordering

DDC based SCANDEF is a binary alternative to reading ASCII SCANDEF in ICC

# What is SCANDEF?

- **SCANDEF is a subset of the LEF/DEF spec that defines scan chain information**
  - A 'stub' chain is between an IO port, lock-up latch or mux (this is different from a DFTC or TetraMAX chain)
  - Allows backend tools to reorder the 'short' SCANDEF chains without running a Test DRC



**9-23**

The SCANDEF file is a subset of the Cadence Design Exchange format (DEF), which is an open standard available from OpenEDA. In addition to CDN, it's supported by most of the competitors (i.e. Magma, LV, MENT).

SCANDEF is a data file created by the scan insertion tool. It's an ASCII file that specifies scan chains that will be reordered by another tool. SCANDEF specifies a list of 'stub' chains that can be reordered by another tool. The boundaries of these stub chains is I/O port, lock-up latch or multiplexer. The PARTITION syntax adds the ability to exchange flops between 'stub' chains. In the figure in the slide the following 3 scan chains are defined:

Chain 1: SI – SCANDEF Chain 1 – Lock-up Latch  
Chain 2: Lock-up Latch – SCANDEF Chain 2 – MUX  
Chain 3: MUX – SCANDEF Chain 3 – SO

Q: Does SCANDEF include the complete design & floorplan information (i.e. is it the same as the floorplan DEF?)  
A: No, it's a standalone DEF file, only containing the scan information

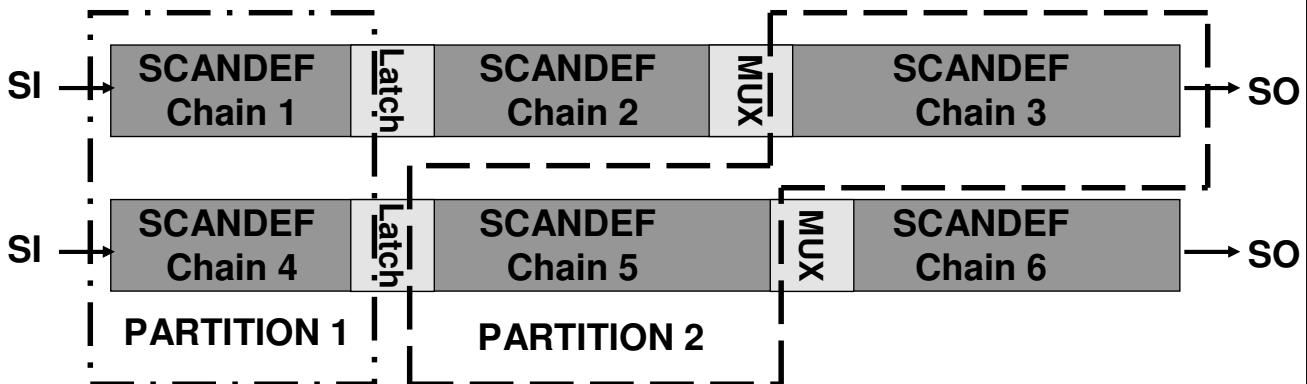
Q: Where comes SCANDEF from?  
A: It is written from DC after test insertion

Q: Is it compliant with attributes stored in CEL view?  
A: Yes.

Q: After reordering and repartitioning in ICC will it be consistent with the test protocol generated in DC?  
A: Yes, constraints in SCANDEF and ICC respecting those will make sure it's compliant

# Partitioning with SCANDEF

- The SCANDEF specification also supports reordering with repartitioning
- A PARTITION is a group of “SCANDEF chains” that may exchange flops during reordering



9-24

Note that what the SCANDEF file calls a “chain” is not that same as the number of top-level scan chains. For SCANDEF the chains are actually **chain segments** where the segments are determined by the existence of multi-input gates (Lock-up elements, MUXes, etc.) on the scan chain.

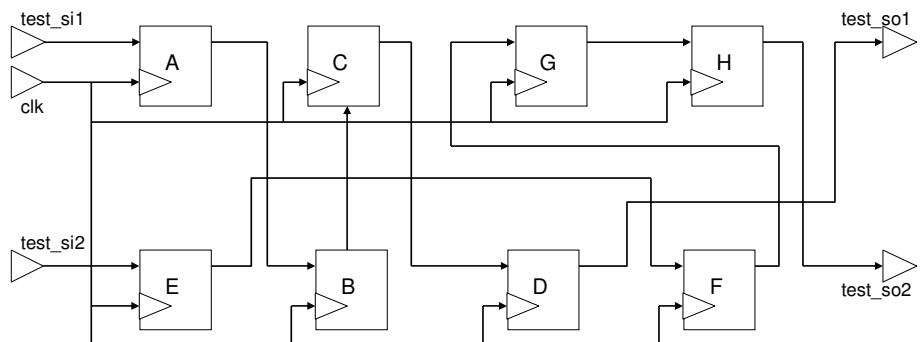
In the above example there are **2** scan chains and **6** SCANDEF chain segments.

# Example: SCANDEF File

```
DESIGN my_design ; → Design name
SCANCHAINS 2 ; → Number of chain stubs in
- 1
+ START PIN test_si1
+ FLOATING A ( IN SI ) ( OUT Q )
    B ( IN SI ) ( OUT Q )
    C ( IN SI ) ( OUT Q )
    D ( IN SI ) ( OUT Q ) } → “FLOATING” indicates that these
                                flops can be reordered
+ PARTITION CLK_45_45 → PARTITION keyword in SCANDEF
+ STOP PIN test_so1
                                Flops can be swapped between two
                                partitions with same name
- 2
+ START PIN test_si2
+ FLOATING E ( IN SI ) ( OUT Q )
    F ( IN SI ) ( OUT Q )
    G ( IN SI ) ( OUT Q )
    H ( IN SI ) ( OUT Q )
+ PARTITION CLK_45_45
+ STOP PIN test_so2
```

9-25

## Alpha Numeric Ordering

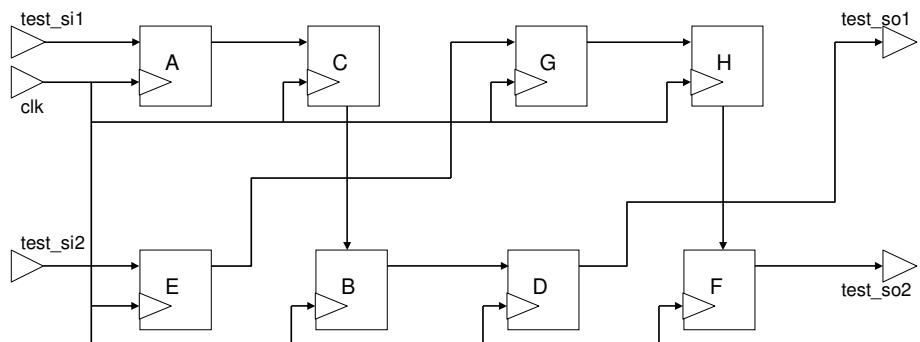


<i>Chain Order</i>	DC	ICC reorder	ICC reorder + repartition
Scan Chain 1	ABCD		
Scan Chain 2	EFGH		

9-26

With SCANDEF, we don't need to settle for simple alpha-numeric ordering...

## Reordering Within Scan-Chain



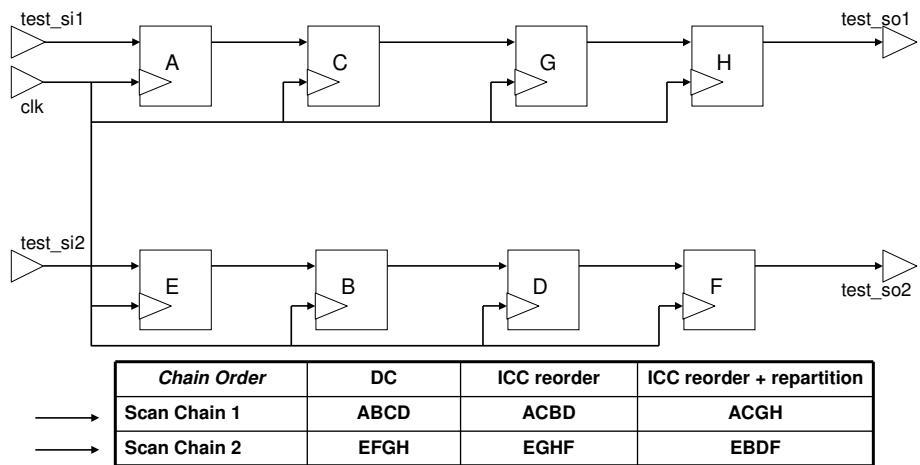
Chain Order	DC	ICC reorder	ICC reorder + repartition
Scan Chain 1	ABCD	ACBD	
Scan Chain 2	EFGH	EGHF	

9-27

This SCANDEF file allows ICC to reorder in a seamless manner.

Re-ordering scan chains reduces wire length and congestion, and improves routability.

## Reordering Across Scan-Chains



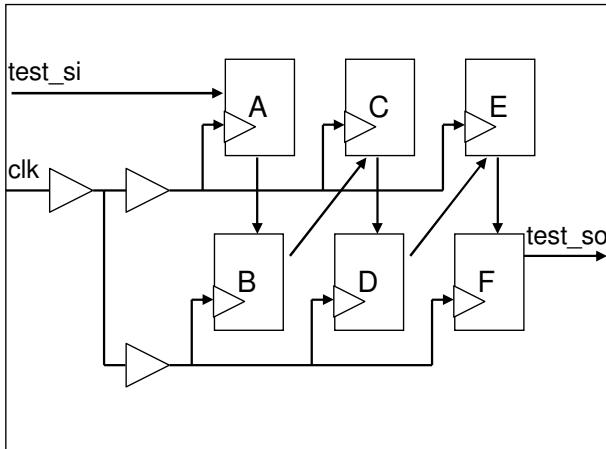
9-28

Repartitioning using SCANDEF allows even better results.

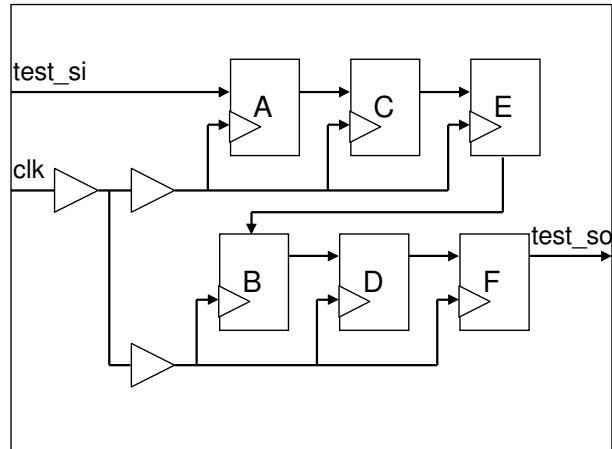
# Clock Tree Based Reordering

`clock_opt -optimize_dft`

- Reorders to minimize crossings between clock buffers
- Impacts wire-length



Without clock tree based reordering



With clock tree based reordering

**9-29**

IC Compiler's timing based scan ordering after Clock Tree Synthesis minimizes hold time violations and avoids buffer insertion.

Clock buffer based reordering helps accommodate On-Chip Variation.

# Example Script

```
## Reading top-level design
read_verilog top_test_ready.v
current_design top
link

set_dft_signal -view existing_dft -type ScanClock \
    -port clock -timing [45 55]
create_test_protocol -capture_procedure multi_clock
dft_drc
preview_dft
insert_dft

change_names -rules verilog -hier
write_scan_def -o top.scandef
check_scan_def

# Write the DDC *after* write_scan_def
write -f ddc -hier -o top_with_scandef.ddc
write_test_protocol -o test_mode.spf
write -f verilog -hier -o top.v
```

9-30

# SCANDEF Notes

- The SCANDEF generated will not be consistent to the design if design modifications are done on the scan path after `insert_dft`
- “SCANCHAINS” as it appears in SCANDEF and as seen by IC Compiler is *not* the same as the traditional “scan-chain” definition in DFT Compiler & TetraMAX
  - Example: If DFTC inserts one scan-chain with a lock-up latch, SCANDEF & `report_scan_chain` in ICC will refer to this as 2 scan-chains (as chain is broken at lock-up latch in SCANDEF file)

9-31

# Exporting Design Files: Agenda

ATE Concepts

Writing Test Protocols

SCANDEF

Formality

9-32

# Formality Support

- **Formality is Synopsys' Formal Verification tool**
- **Test Information is automatically passed to the Setup Verification File (SVF) file, eliminating the need to use `set_constant` to disable the ScanEnables, TestModes etc.**
- **The default SVF file generated in DC is named `default.svf`**
  - Use `set_svf <file_name>` to change name/path

9-33

# What DFT Data is Passed to Formality?

- **The following Test Information will only be recorded in the SVF file after `insert_dft`:**
  - `ScanEnable` ports set to disable scan mode operation
  - `TestModes` ports set to disable whenever used i.e. in AutoFix/ Adaptive Scan etc.
  - Dedicated `ScanDataOut` ports have a don't verify
  - Core wrapping : `wrp_shift` (type of `ScanEnable`) is disabled
  - BSD: `TCK`, `TMS`, `TRST` ports are held at 0 and the `TDO` port is not verified
- **Report setup in assumptions summary report**
- **Scan extraction flows and compliance check flows won't record the information in the SVF file**

9-34

The requirement was to be able to guide Formality to automatically disable the DFT inserted logic in scan-inserted designs, not check for any mismatches on dedicated scan outputs and have all the assumptions reported.

You can see this information in the summary report.

# Formality Sample Script

## ■ DC

```
# Command to define your svf file  
set_svf ./my_svf_file
```

## ■ Formality

```
# Use the SVF file  
set synopsys_auto_setup true  
set_svf ./my_svf_file  
  
# Read Reference Design  
create_container pre_dft  
read_ddc ./outputs/des_unit.prescan.ddc  
set_top des_unit  
set_reference_design pre_dft:/WORK/des_unit  
  
# Read Implementation Design  
create_container post_dft  
read_ddc ./outputs/des_unit.post_scan.ddc  
set_top des_unit  
set_implementation_design post_dft:/WORK/des_unit  
  
# Match compare points and Verify  
match  
verify  
exit
```

9-35

In DC, the SVF filename is specified as my\_svf\_file

In Formality, the previously generated SVF file is read.

The **set synopsys\_auto\_setup true** variable should be set prior to **set\_svf** in order for this information to be passed on.

# Limitations for SVF

- **The following flows are not supported for the SVF**
  - Internal pins
  - DBIST
  - RTL DRC
- **Note: If Test logic is added at the RTL level and the logic is controlled by a `TestMode` and/or `ScanEnable` signal, the logic won't be verified because the SVF will put the `TestMode` and `ScanEnable` in their inactive states**

9-36

# **Unit Summary**

---

**Having completed this unit, you should now be able to:**

- **Name 2 files that DFTC must write to handoff a scan design**
- **Write a SCANDEF file that can be taken to ICC for scan chain reordering**
- **Generate a SVF file that can be used by Formality for formal verification**

**9-37**

# Lab 9: Export Design Files



30 minutes

**After completing this lab, you should be able to:**

- Given a gate-level scan design export the files required by downstream tools such as ATPG and P&R

**9-38**

# Command Summary (Lecture, Lab)

<code>change_names</code>	Change the names of ports, cells, and nets in a design
<code>write</code>	Writes a design netlist from memory to a file
<code>set test_stil_netlist_format</code>	Indicates to the <code>write_test_protocol</code> command the netlist format to use when writing STIL protocol files
<code>write_test_protocol</code>	Writes a test protocol file
<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>write_scan_def</code>	Writes scan chain information in SCANDEF format for performing scan chain reordering with P&R tools
<code>read_def</code>	Annotates the design with the data from a file in Design Exchange Format (DEF)
<code>set_optimize_dft_options</code>	Define options for physical DFT optimization in ICC
<code>check_scan_chain</code>	Scan chain structural consistency check based on the scan chain information stored in the current design
<code>place_opt -optimize_dft</code>	Enables placement aware scan reordering
<code>clock_opt -optimize_dft</code>	Enables clock-aware scan reordering
<code>report_scan_chain</code>	Report scan chains defined on the current design
<code>set_svf</code>	Generates a Formality setup information file for efficient compare point matching in Formality
<code>synopsys_auto_setup</code>	Enables the Synopsys Auto Setup Mode

**9-39**

This page was intentionally left blank.

# Agenda

**DAY  
3**

**9 Export**



**10 High Capacity DFT Flow**



**11 Multi-Mode DFT**



**12 DFT MAX**



**13 Conclusion**

# Unit Objectives

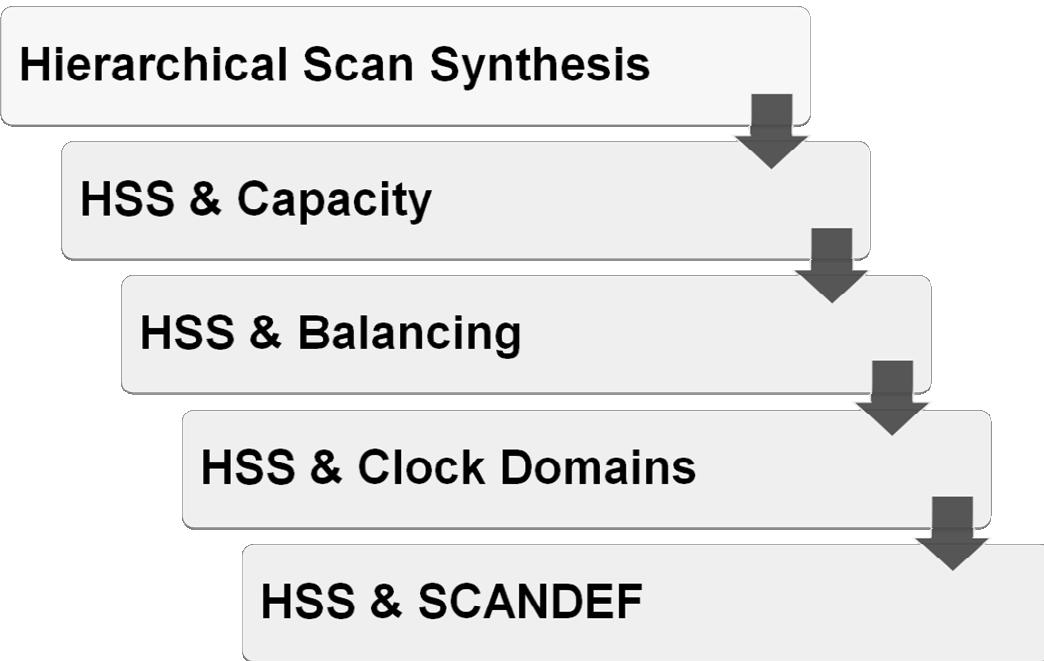


After completing this unit, you should be able to:

- Identify the capacity and run-time bottlenecks using full gate-level designs in both top-down and bottom-up scan insertion flows
- Describe how Test Models improve scan insertion capacity and run-time in a bottom-up flow
- State the two methods for implementing Test Models in a scan insertion flow
- Explain two aspects of scan chain architecture that are more difficult to achieve in a bottom-up versus top-down scan insertion flow

10-2

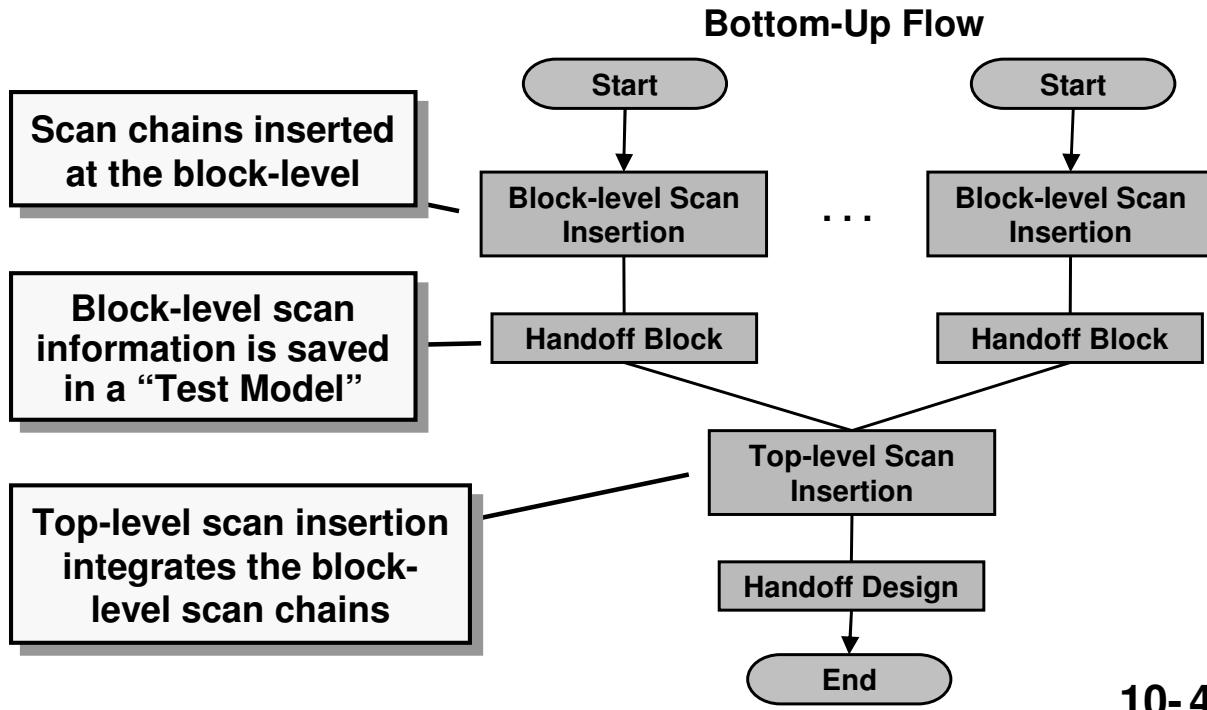
# High Capacity DFT Flow: Agenda



10-3

# Hierarchical Scan Synthesis (HSS)

- Hierarchical Scan Synthesis (HSS) is another name for the “Bottom Up” scan insertion flow



# Test Models

- A Test Model is an abstraction of the Test architecture for a given block
- DFT Compiler Test Models are written in Core Test Language (CTL)
- CTL is an extension of STIL (Standard Test Interface Language), IEEE 1450
- Core Test Language (CTL) is an IEEE standard (1450.6)
  
- More information on CTL syntax can be found in the Appendix

10-5

If you need to create a Test Model but don't have a netlist (RAM with embedded scan chains, for example), you can use a script, CTLGEN, to generate the Test Model. See the following SolvNet article (**018658**) for more information:

<https://solvnet.synopsys.com/retrieve/018658.html>

# Test Model Contents

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

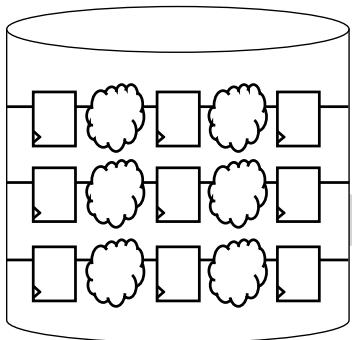
10-6

Note:

The Test Model is only an **abstraction** of the Test interface and structures (scan chains) of the design. You cannot use the Test Model for ATPG. i.e. TetraMAX cannot read in a test model.

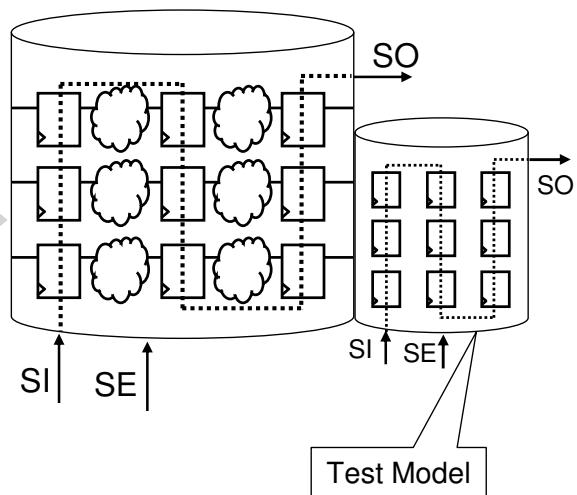
# How to Create Test Models

Non Scan Design



`insert_dft`

Scanned Design



- DFTC automatically creates a test model during scan insertion that abstracts the test behavior

10-7

# How to Write/Read Test Models

- **Write a CTL DDC model**

```
write_test_model -o counter.ctlddc -format ddc
```

- **Write an ASCII CTL model**

```
write_test_model -o counter.ctl -format ctl
```

- **Write a design DDC (with Test Model attached)**

```
write -format ddc -hier -o counter1_scan.ddc
```

- **Read a Test Model**

```
read_test_model counter.ctlddc -format ddc
```

- **Read a design DDC (with Test Model attached)**

```
read_ddc counter.ddc
```

**Note:**

The default write/read format is ddc

10-8

# How to Enable/Disable Test Models

- By default, if a Test Model is read, it will be used during top-level integration
- The `use_test_model` command can be used to turn on or off Test Model usage for specified blocks
  - Turn on Test Model use for all design blocks

```
use_test_model -true [get_designs *]
```
  - Turn on Test Model use for a specific block

```
use_test_model -true counters1
```
  - Turn off Test Model use for a specific block

```
use_test_model -false adderx2
```

10-9

Note: When Test Models are enabled/disabled for a given block with `use_test_model`, the sub-blocks for that block are enabled/disabled as well

# How to Report Test Models

- **list\_test\_models**  
The **list\_test\_models** command will report a simple list of the test models currently loaded
- **report\_test\_model [-design <design\_name>]**  
The **report\_test\_model** command will report information about a particular test model
- **report\_use\_test\_model**  
The **report\_use\_test\_model** command lists the sub-designs in **dc\_shell** that are instantiated within the current design. For each sub-design, it indicates whether a test model or gates are used

10-10

# Test Model Usage Reported During dft\_drc

## ■ Standard dft\_drc reporting:

```
dc_shell> dft_drc
In mode: all_dft...
Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
Loading test protocol
Information: Using DFT model for cell(s): I_ORCA_TOP/I_BLENDER(BLENDER),
    I_ORCA_TOP/I_CONTEXT_MEM(CONTEXT_MEM), I_ORCA_TOP/I_PARSER(PARSER),
    I_ORCA_TOP/I_PCI_CORE(PCl_CORE), I_ORCA_TOP/I_PCI_READ_FIFO(PCl_RFIFO),
    I_ORCA_TOP/I_PCI_WRITE_FIFO(PCl_WFIFO), I_ORCA_TOP/I_RISC_CORE(RISC_CORE),
    I_ORCA_TOP/I_SDRAM_IF(SDRAM_IF), I_ORCA_TOP/I_SDRAM_READ_FIFO(SDRAM_RFIFO),
    I_ORCA_TOP/I_SDRAM_WRITE_FIFO(SDRAM_WFIFO)
...basic checks...
...basic sequential cell checks...
    ...checking for scan equivalents...
...checking vector rules...
...checking pre-dft rules...
```

## ■ Enhanced dft\_drc reporting:

```
dc_shell> dft_drc
In mode: all_dft...
Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
Loading test protocol
...basic checks...
...basic sequential cell checks...
    ...checking for scan equivalents...
...checking vector rules...
...checking pre-dft rules...
-----
Cores and Modes used for DRC in mode: all_dft
-----
I_ORCA_TOP/I_BLENDER (BLENDER) : Internal_scan
I_ORCA_TOP/I_CONTEXT_MEM (CONTEXT_MEM) : Internal_scan
I_ORCA_TOP/I_PARSER (PARSER) : Internal_scan
I_ORCA_TOP/I_PCI_CORE (PCI_CORE) : Internal_scan
I_ORCA_TOP/I_PCI_READ_FIFO (PCI_RFIFO) : Internal_scan
I_ORCA_TOP/I_PCI_WRITE_FIFO (PCI_WFIFO) : Internal_scan
I_ORCA_TOP/I_RISC_CORE (RISC_CORE) : Internal_scan
I_ORCA_TOP/I_SDRAM_IF (SDRAM_IF) : Internal_scan
I_ORCA_TOP/I_SDRAM_READ_FIFO (SDRAM_RFIFO) : Internal_scan
I_ORCA_TOP/I_SDRAM_WRITE_FIFO (SDRAM_WFIFO) : Internal_scan
```

10-11

# Linking a Test Model to a Library Cell

- An ASCII CTL Test Model can be linked to library elements like RAMs
- If a library cell (or hardmacro) has built-in scan chains, a CTL Test Model must be linked to the library .lib or to the design
- To link a Test Model to a library, use this command

```
read_lib <lib_name>.lib \
    -test_model <cell_name>:<model_file>.ctl
```
- To link a Test Model to a design (which must be done if it has not been linked in the library), use this command

```
read_test_model -format ctl -design \
    <full_path_to_cell_name> <model_file>.ctl
```

10-12

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

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

## Checking if a Library Contains CTL Models

To determine if a library has CTL models attached to it, read in the library with the link command or the compile command, then run report\_lib. If the library has CTL models attached, the report will indicate ctl in the Attributes list as shown below

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

# How to Verify a Test Model?

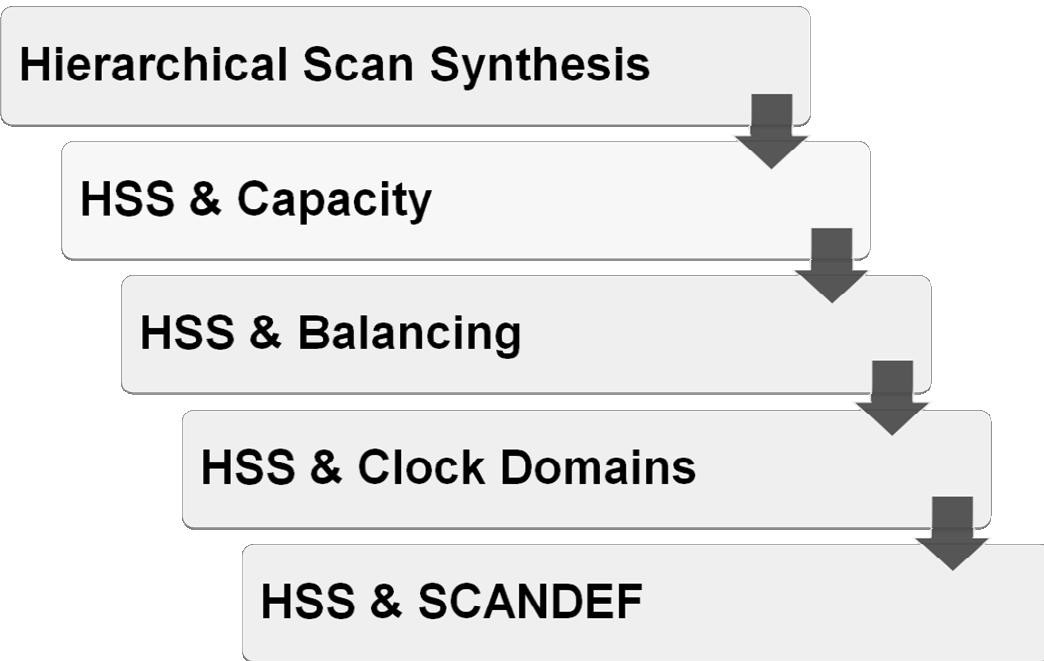
- You can verify information inside of a test model:

```
read_test_model -format ddc counter.ct1ddc  
current_design my_ctl_counter  
report_scan_path -view exist  
report_dft_signal -view exist
```

- The report commands will list the existing DFT signals and scan chains described in the Test Model

10-13

# High Capacity DFT Flow: Agenda



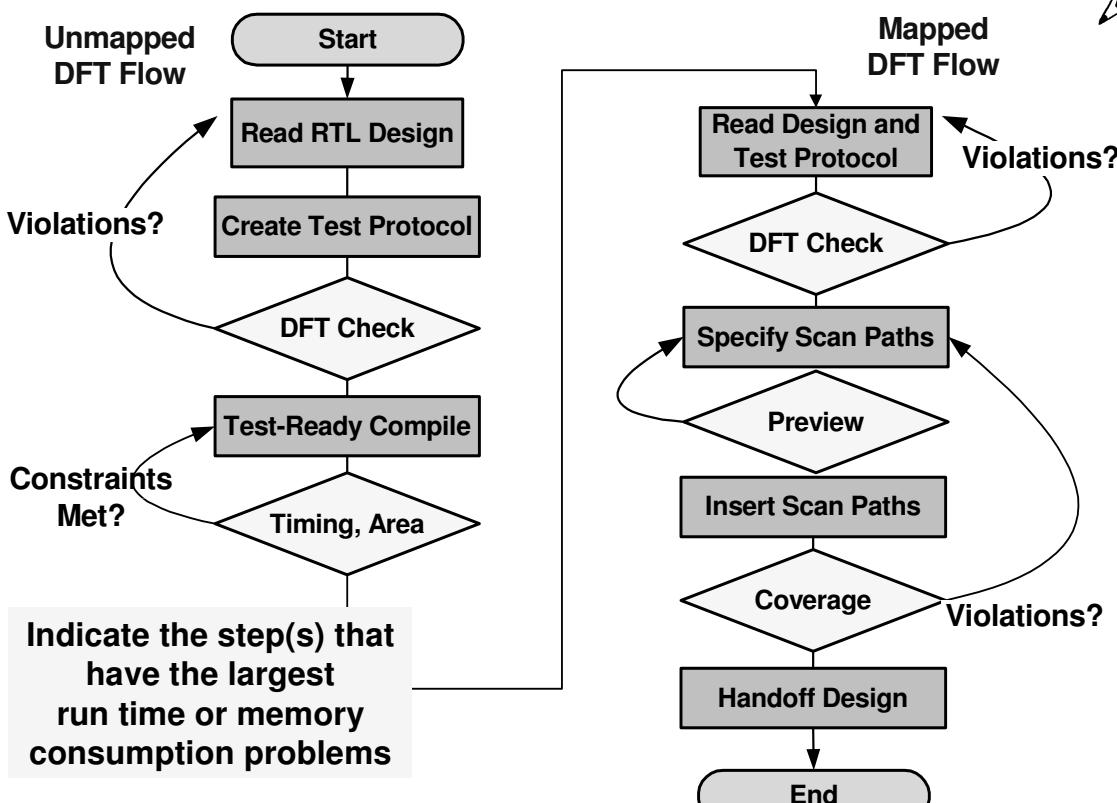
10-14

## **Bottom-Up Capacity Issues**

- A Bottom-Up flow by itself does not solve all capacity issues regarding scan insertion
- The Bottom-Up flows allow for a “divide and conquer” approach at the block level
- However, at some point ALL the block-level designs need to be read for top-level integration
- The use of block-level Test Models and/or Interface Logic Models (ILMs) for top-level integration will increase capacity and reduce run-time for top-level integration

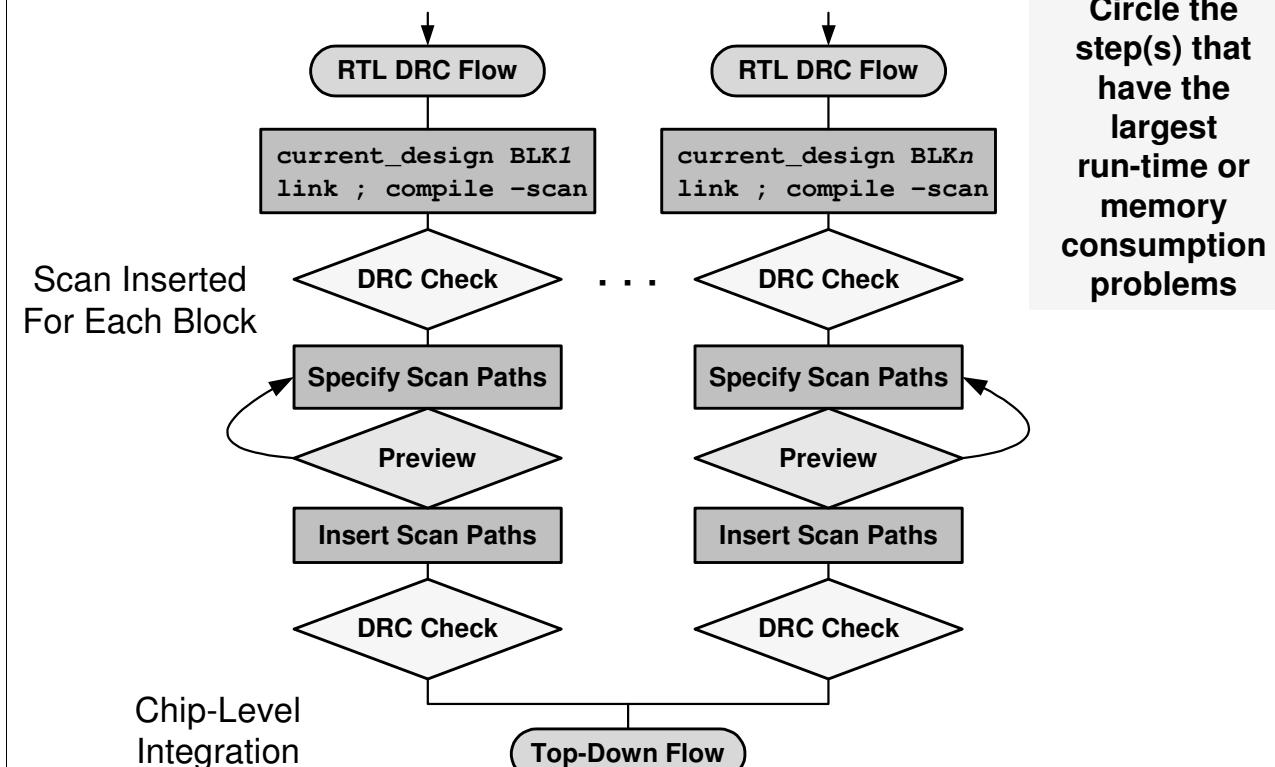
**10-15**

# Recall: Top-Down Scan Insertion Flow



10-16

# Bottom-Up Scan Insertion



10-17

Bottom-up scan insertion gives you greater manual control, but requires more effort. Scan paths are inserted on a block-by-block basis, then integrated at the chip-level.

### Advantages:

Bottom-up insertion allows block designers to maintain ownership until block sign-off. You can ensure that your block still meets timing even after the scan paths are inserted. Since insert\_dft runs on individual blocks, overall run times are conveniently short.

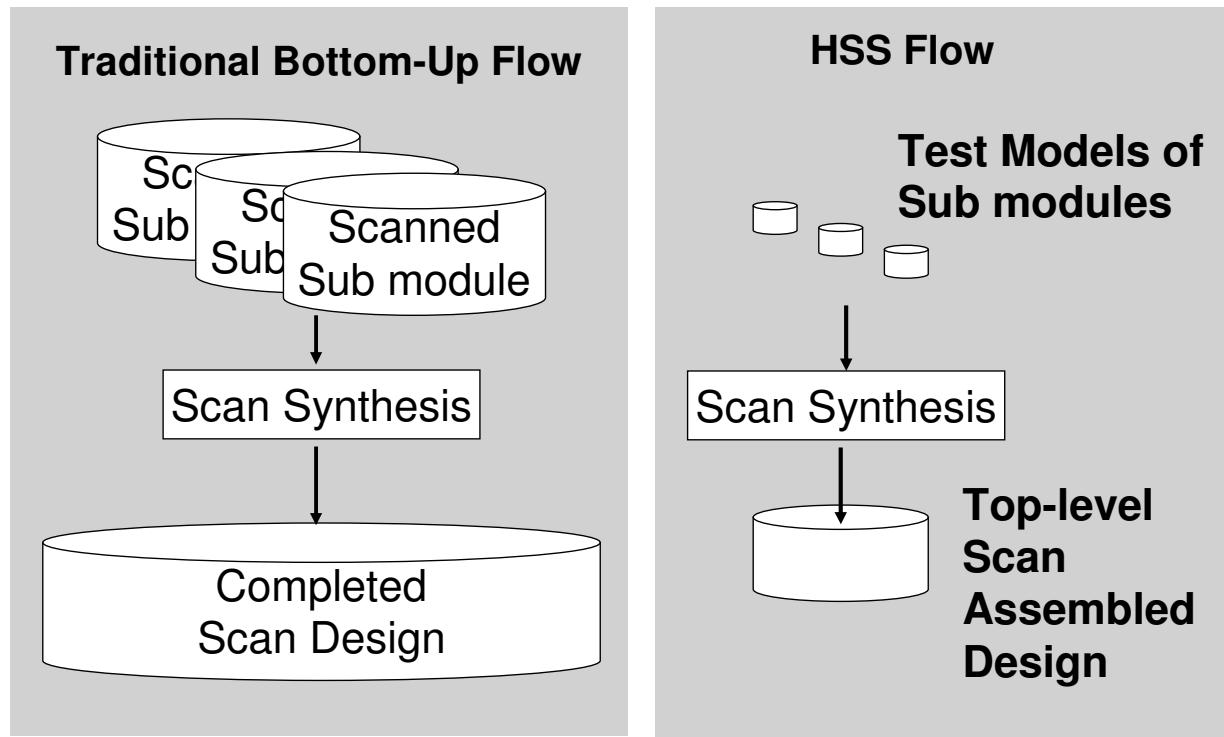
### Disadvantages:

Block-level clock mixing and tristate disabling may need revision later at the chip-level. The bottom-up approach usually requires more planning and scripting by the design team.

### Conclusion:

Use the approach that fits your SOC design flow—DFTC fully supports both techniques. Mix top-down and bottom-up by inserting scan top-down into major subsystem blocks.

# Solving Bottom-Up Bottleneck with HSS



10-18

HSS: Hierarchical Scan Synthesis

DFT Compiler capacity significantly increased by using an abstracted view (or a test model) of the scanned netlists.

Test models based on the IEEE Standard Core Test Language (CTL).

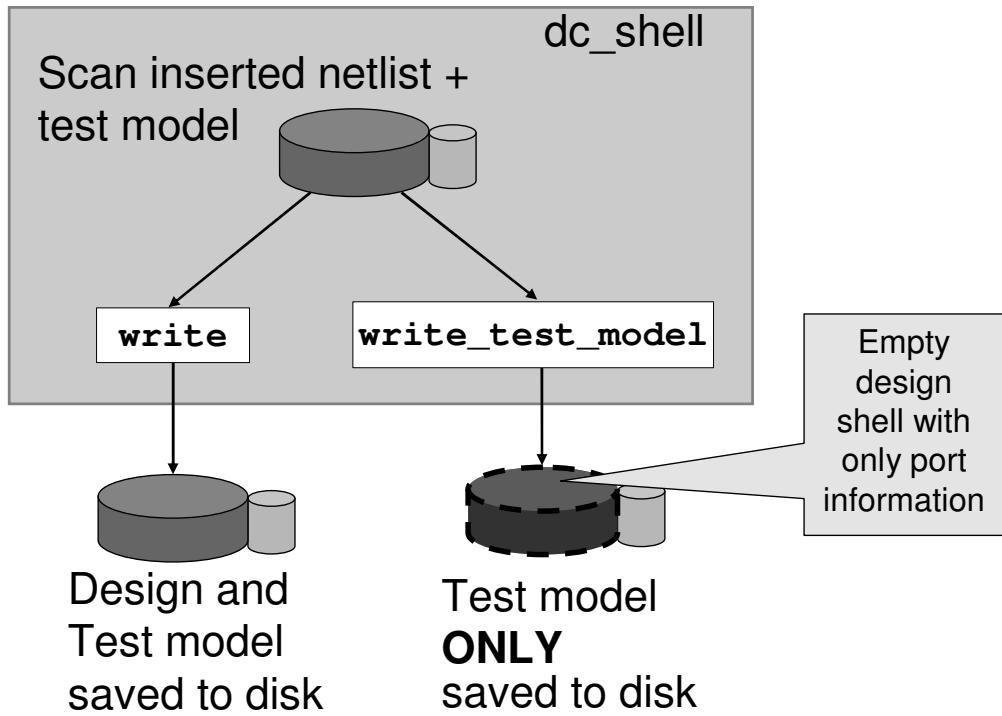
Provides significant reduction of memory usage for scan synthesis and design rule checking.

Easy learning curve enables quick adoption.

Uses existing bottom-up scan insertion flow and commands.

Use of test models is transparent to users.

# Saving Test Models



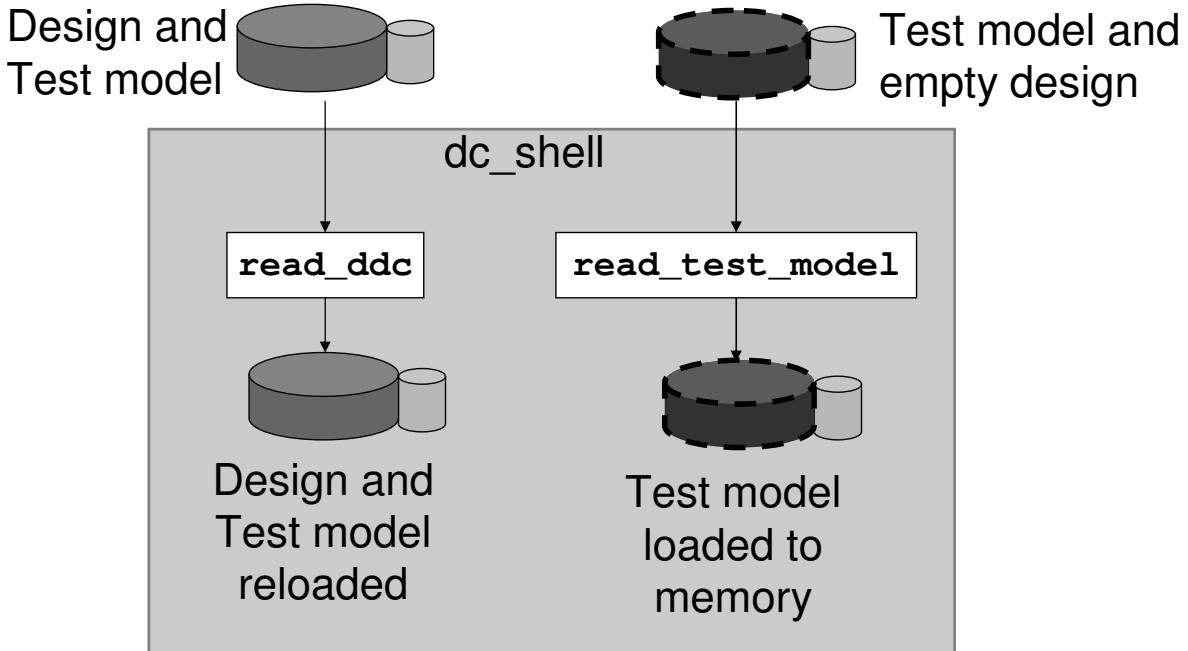
10-19

Once a test model is created it stays connected to the design. As such, the test model information can be saved off to a DDC file in two different ways.

Whenever a design is saved as a ddc the Test Model information is stored in the ddc file.

The **write\_test\_model** command can be used to write out just the test model along with a skeleton description of the module's ports and directions.

# Reading Test Models for Top-Level Stitching



10-20

Test models can be loaded back into memory through `read_ddc` or `read_test_model`.

The `read_ddc` or `read_file` command can be used to load the entire design back into `dc_shell`. During this process the design and the test model will be loaded into memory.

The `read_test_model` command can be used to read just the test\_model into memory.

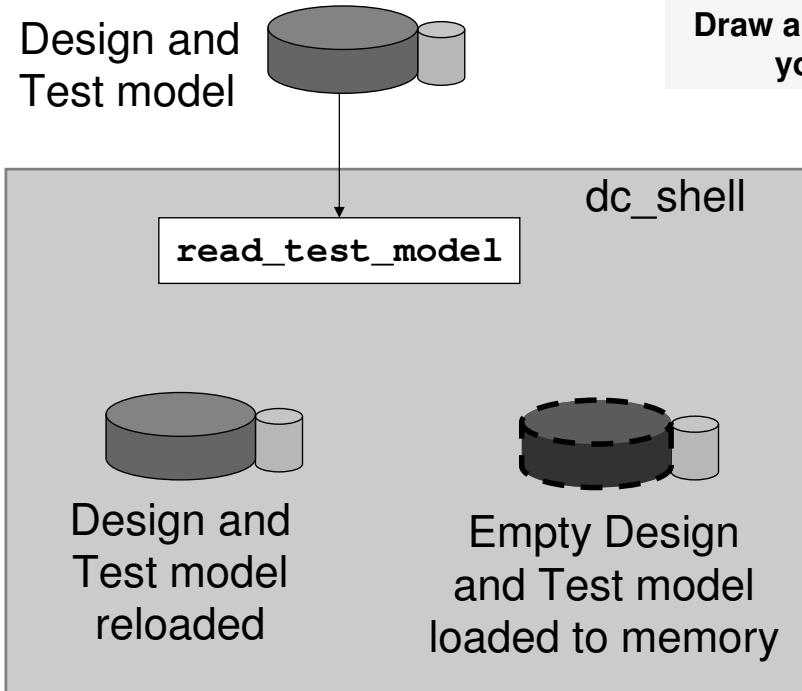
The `read_test_model` command will either accept a full ddc design or a test model only ddc.

# Test for Understanding



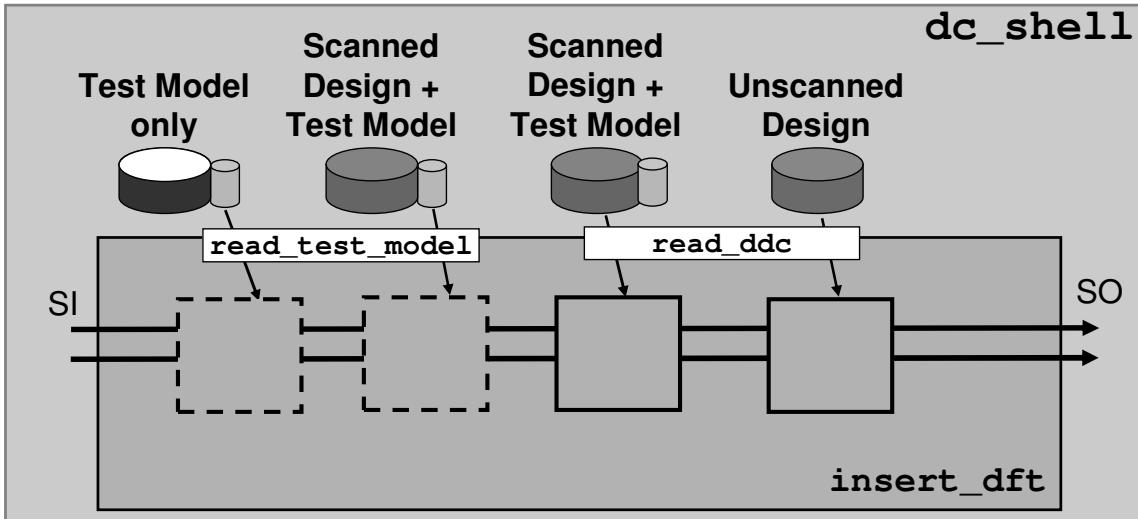
What is the result?

Draw a link to indicate  
your choice



10-21

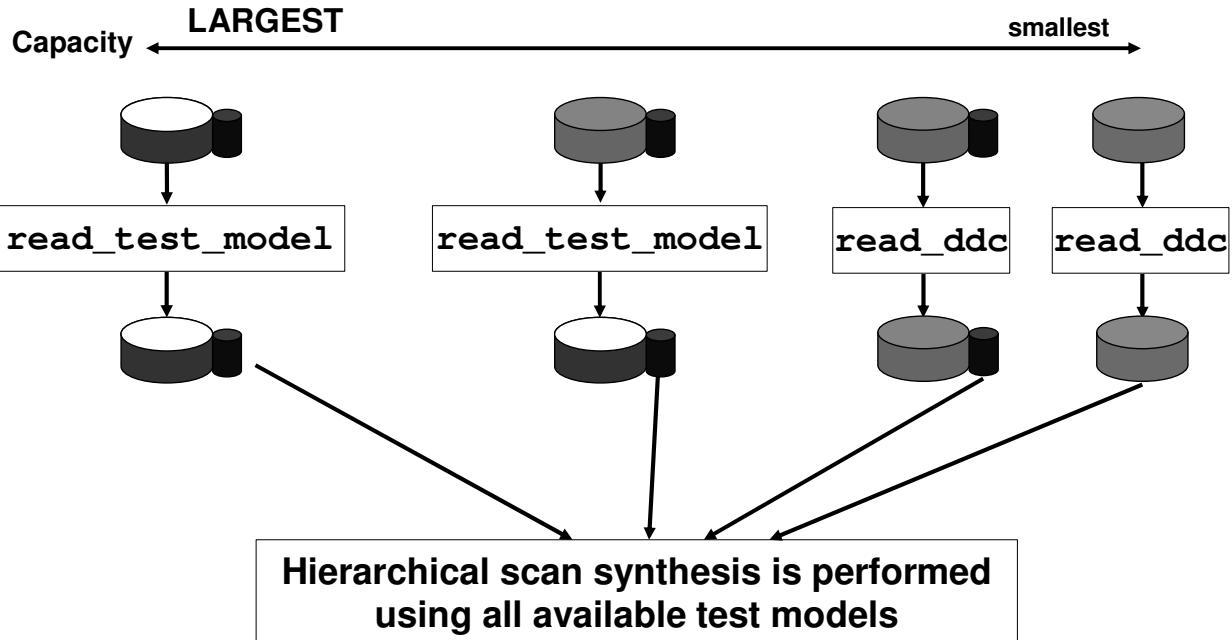
# Test Models Versus Complete Designs



- DFT Compiler can handle a mixture of full gate-level netlists and test models
- `insert_dft` will use Test Models for scan synthesis INSTEAD of the full designs if the Test Model exists (by default)

10-22

# Getting the Most Benefit from Test Models



10-23

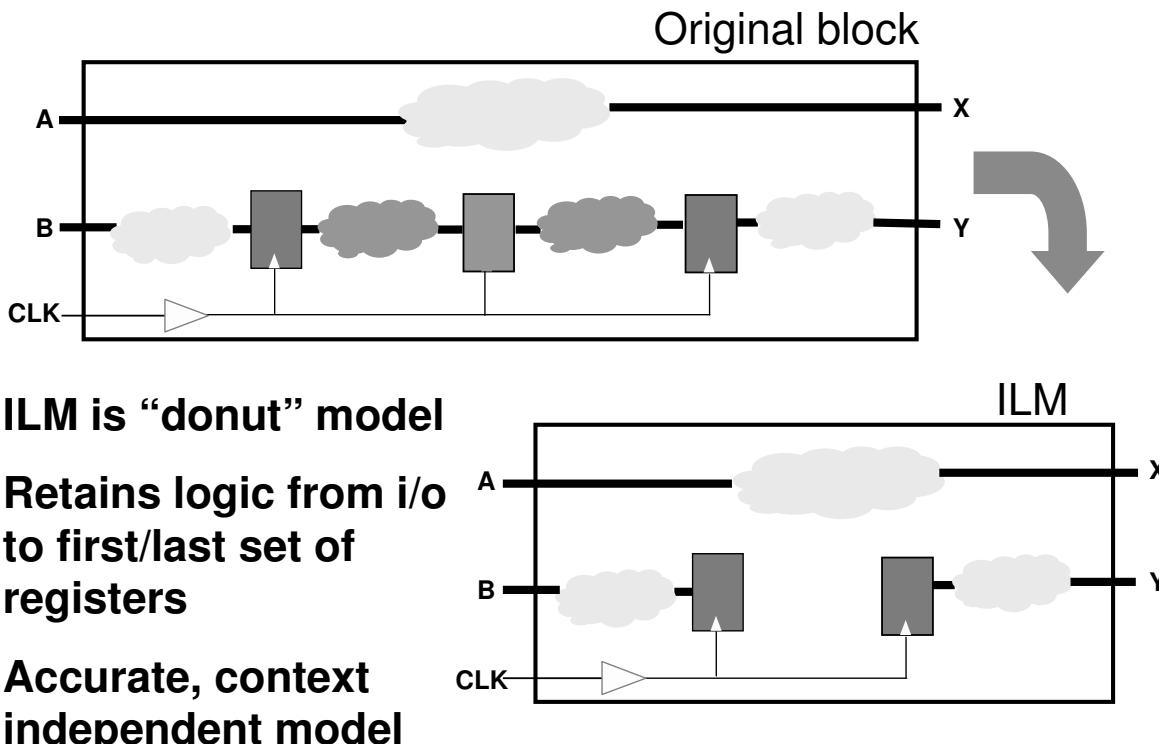
The **read\_ddc** or **read\_file** command can be used to load the entire design back into dc\_shell. During this process the design and the test model information will be loaded into memory

The **read\_test\_model** command can be used to read just the test\_model information into memory.

The **read\_test\_model** command will either accept a full design ddc or a test model only ddc.

A designer can load an un-scanned netlist and use **insert\_dft** to stitch the chains and create the test model.

# Solving Bottom-Up Bottleneck with ILMs



- ILM is “donut” model
- Retains logic from i/o to first/last set of registers
- Accurate, context independent model

**10-24**

In contrast to ETMs, Interface Logic Models take a structural approach to model generation, where the original circuit is modeled by a smaller circuit that represents the interface logic of that circuit.

Interface logic contains all the circuitry from :

- 1) Input ports to output ports (combinatorial input to output paths). (Note that if a transparent latch is encountered in a timing path then it is treated as a combinational logic device and path tracing continues through the latch until an output port or an edge-triggered register is encountered). The number of latch levels that are included is under user control
- 2) Input ports to edge-triggered registers
- 3) Edge-triggered registers to output ports and
- 4) Clock trees that drive interface registers (e.g. registers mentioned in 2) and 3) above).

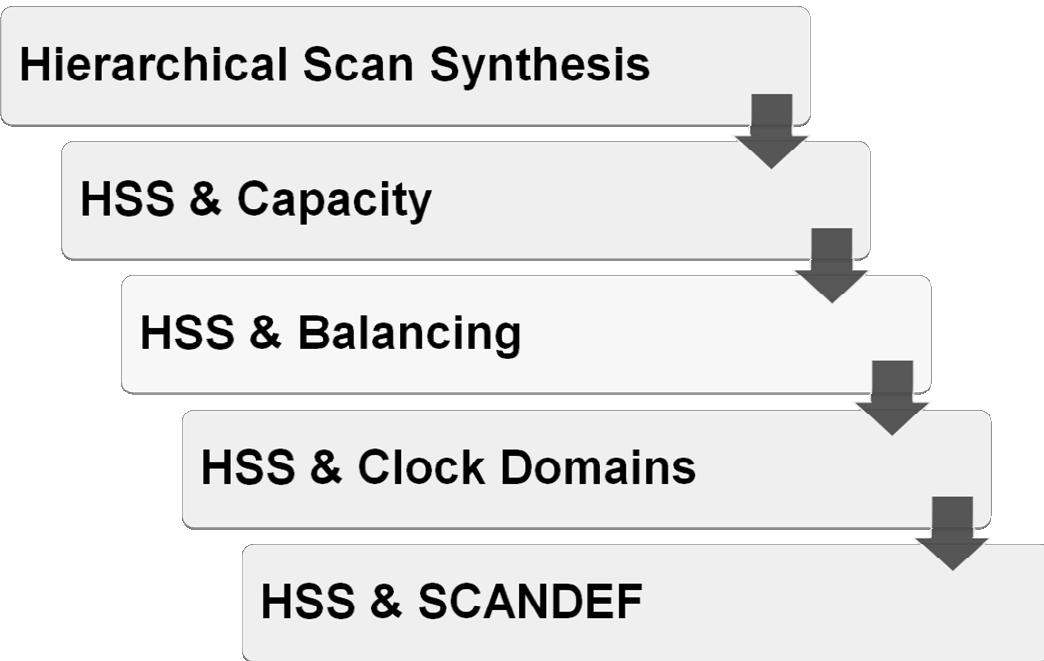
# Creating an ILM

- An ILM is generated with the `create_ilm` command in DC/DCT
- A Test Model can be associated with an ILM
- The following example creates an ILM for the current design with the specification that the fanin/fanout of ports `reset` and `scan_enable` should be ignored

```
create_ilm -ignore_ports {reset scan_enable} -verbose  
write -format ddc -hier -output foo_ilm.ddc
```

10-25

# High Capacity DFT Flow: Agenda



10-26

# **Bottom-Up Balancing Issues**

- A Bottom-Up flows need to be aware of issues at the block level that may effect top-level scan chain balancing efficiency
- Need to choose an appropriate number of scan chain (and ultimately scan chain length) at the block level
- The block level chain lengths will have an direct impact on top-level scan chain balancing in HSS flows

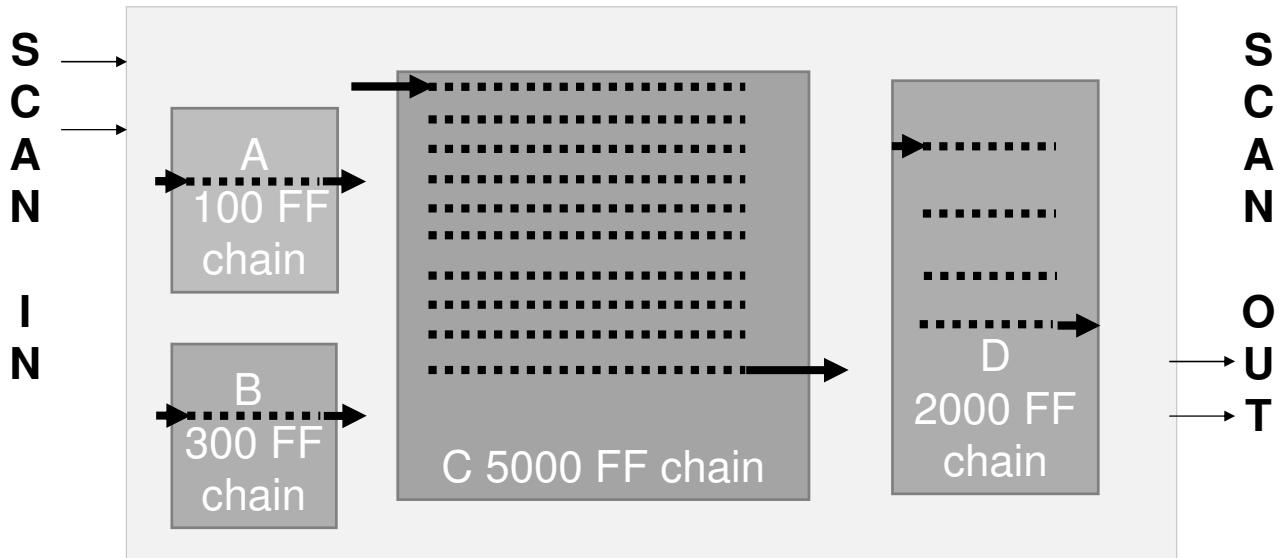
**10-27**

# Avoid Very Long Subdesign Chains



When inserting scan at the chip/top-level DFTC preserves subdesign chains by default

Use your Markup tools to draw 2 top-level scan chains as balanced as possible given that the block level chains cannot be changed

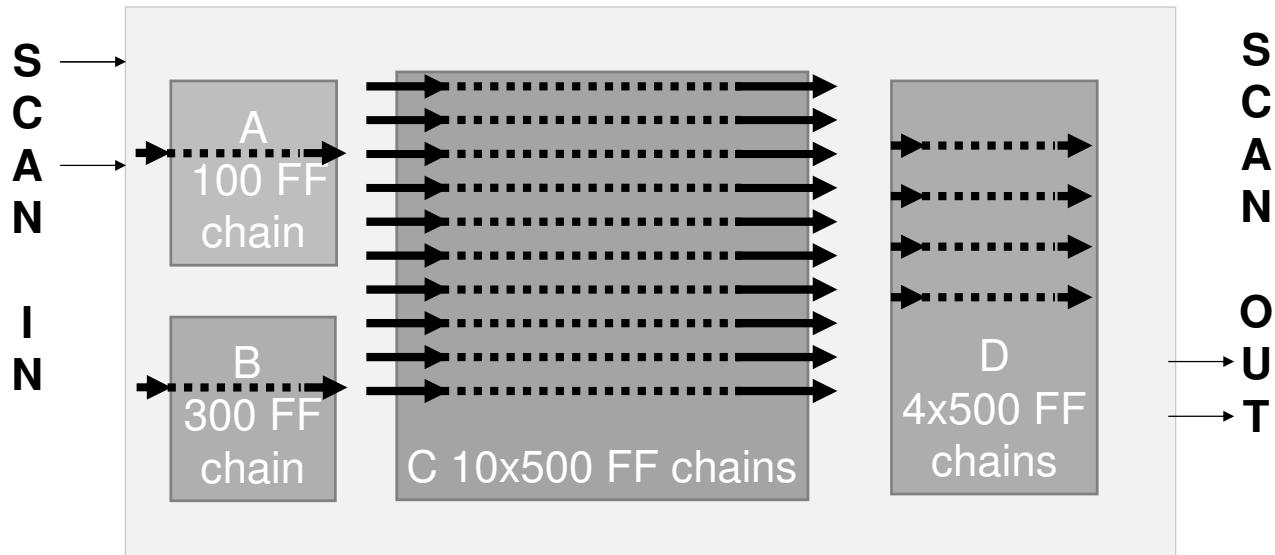


10-28

# Create Short Block Level Chains



Given the revised blocks C and D determine the most balanced length of the two scan chains  
Chain 1 length:                                   Chain 2 length:



10-29

# How to Control Block Chain Length

## Avoid

How to predict  
optimal count  
for each block?

```
dc_shell> set_scan_configuration -chain_count 10
```

Recommended option...

What is a good value?

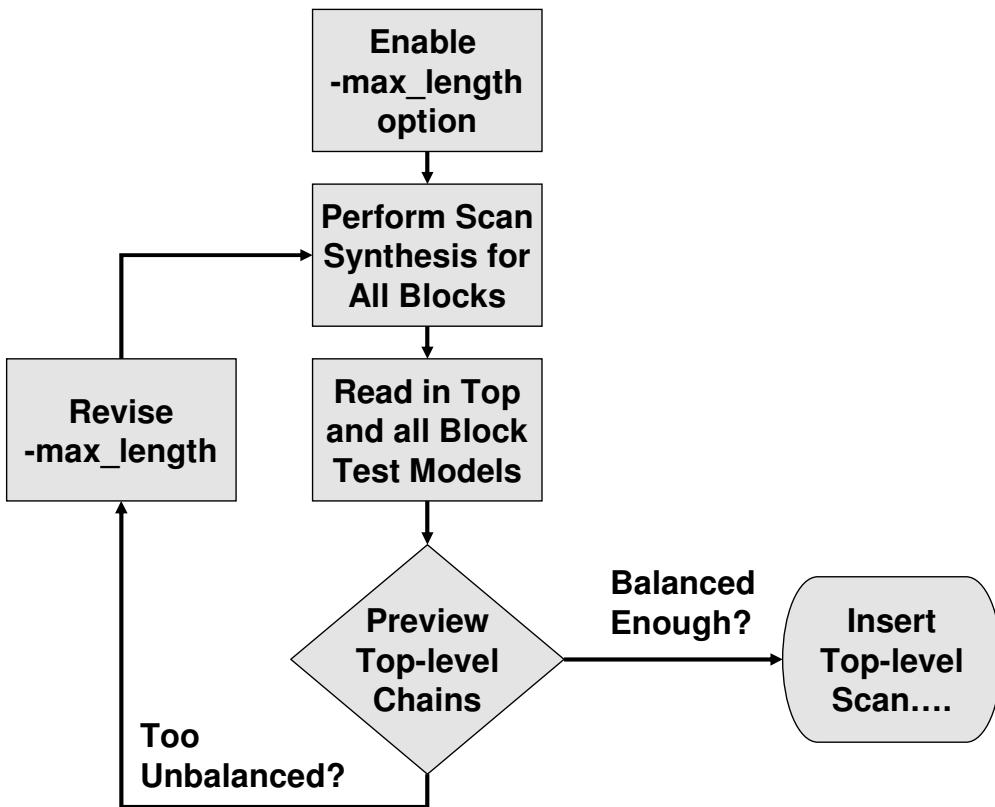
```
dc_shell> set_scan_configuration -max_length 500
```

Or ..

```
dc_shell> set_scan_configuration -exact_length 500
```

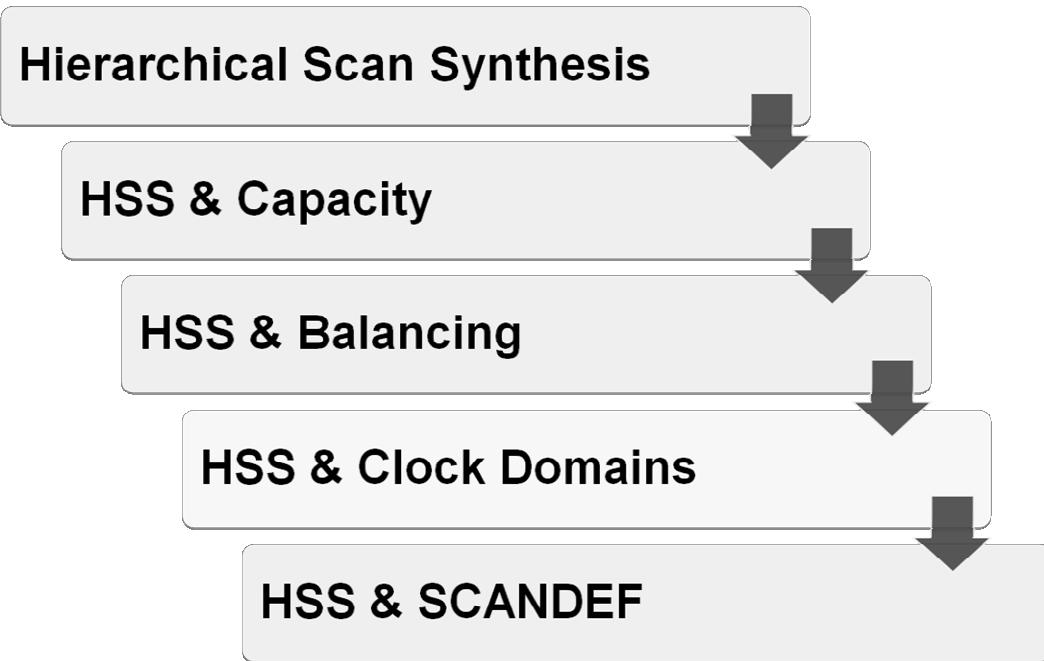
10-30

# Bottom-Up Flow for Better Top-Level Balance



10-31

# High Capacity DFT Flow: Agenda



10-32

# **Bottom-Up Clock Domain Issues**

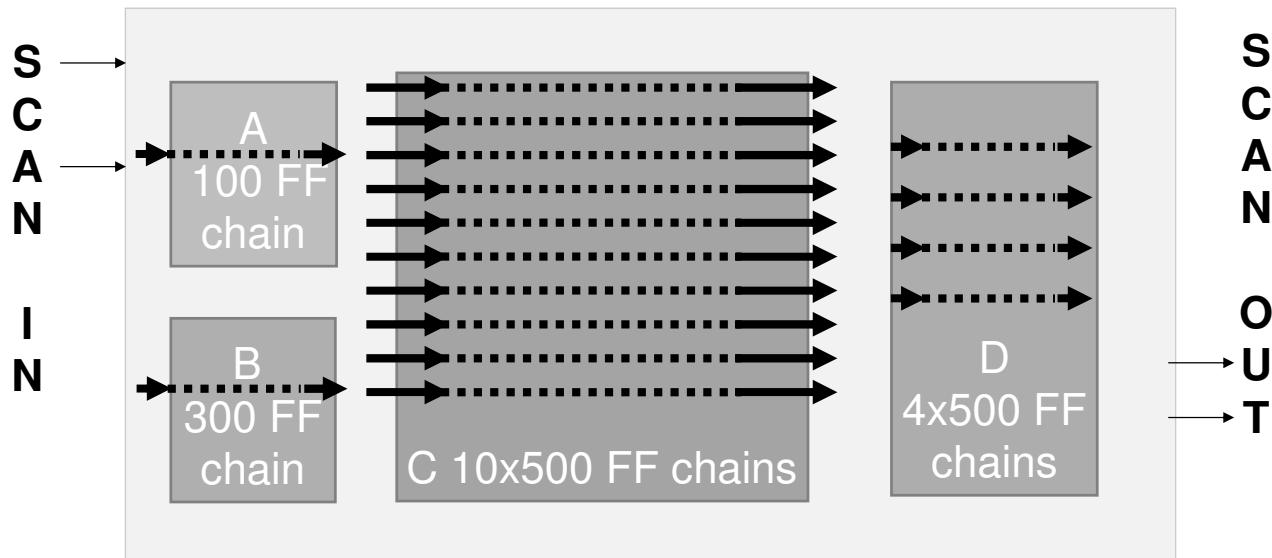
- A Bottom-Up flows need to be aware of issues at the block level that may effect top-level scan chain integration with multiple clock domains
- These issues include:
  - Lock-up element usage
  - Where the lock-up element are to be inserted (block level or top level)

**10-33**

# Multiple Clocks Limit Balancing



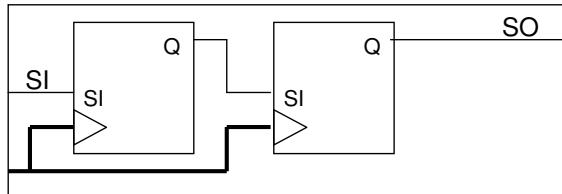
With three different clock domains, what is the default top-level scan chain architecture?



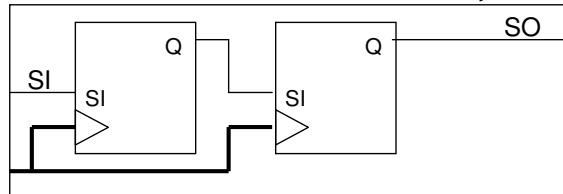
What is normally added to allow top-level clock domain mixing in the scan chains?

10-34

# No Block Level Lock-up Latches

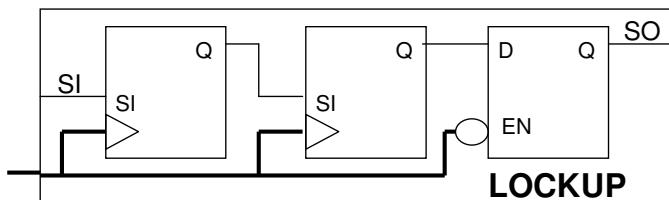


BLKA

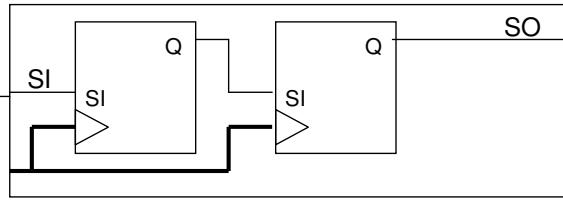


BLKB

Top-level  
insert\_dft



BLKA

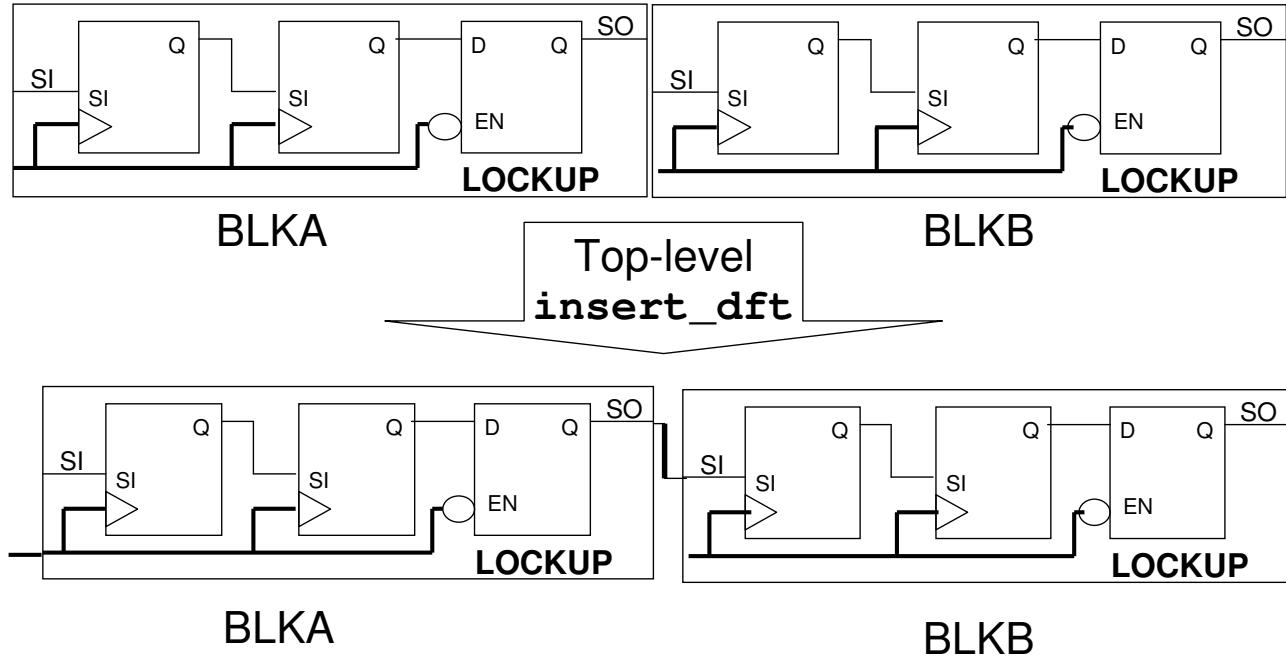


BLKB

If BLKA were a Test Model in the top-level design,  
where could the lock-up latch be inserted?

10-35

# End of Chain Lock-up Latches



10-36

If you know clock mixing will be done at the top-level, you can add “terminal lock-up” elements during block level scan insertion. Then, no lock-up elements will be required at the top-level.

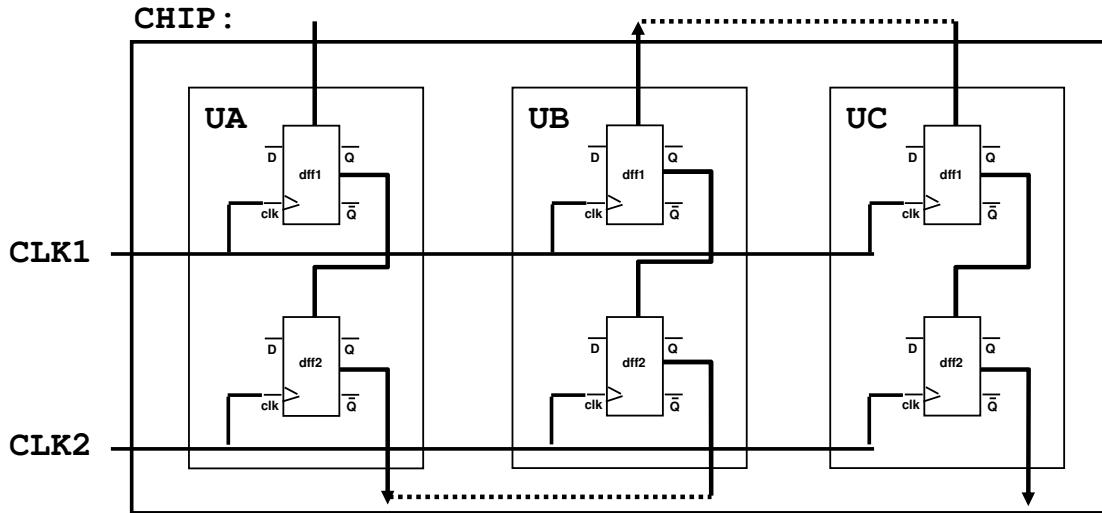
```
set_scan_configuration -insert_terminal_lockup true
```

# Block Level Tip: Avoid Clock Mixing



Circle where the lock-up  
latches would be added

What if you mix clocks for each individual subdesign?

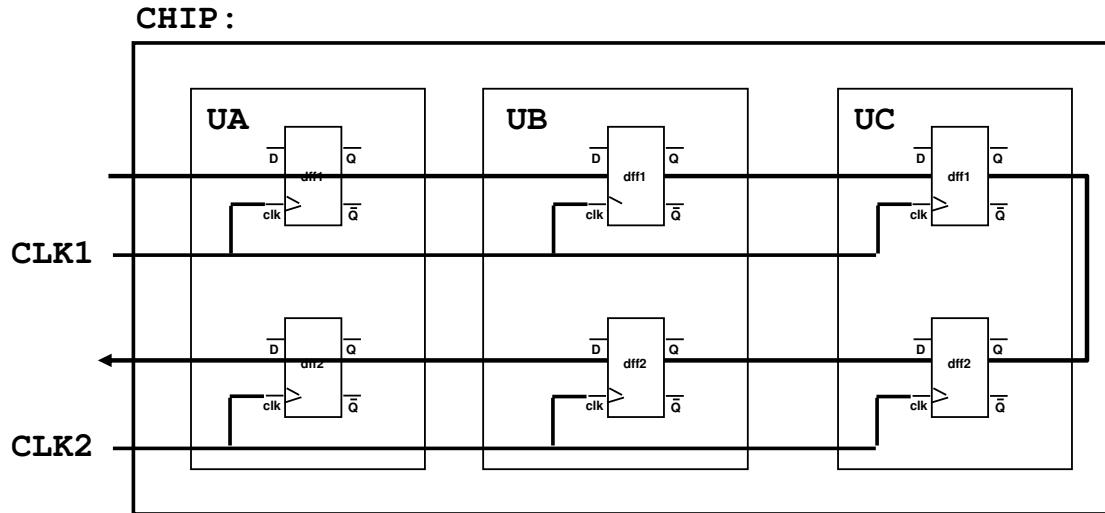


At chip-level, you mix clocks—with no rebalancing

10-37

# Top-Level Tip: Allow Clock Mixing

Instead specify no mixing for each individual subdesign



Now the scan path has a single clock-domain crossing!

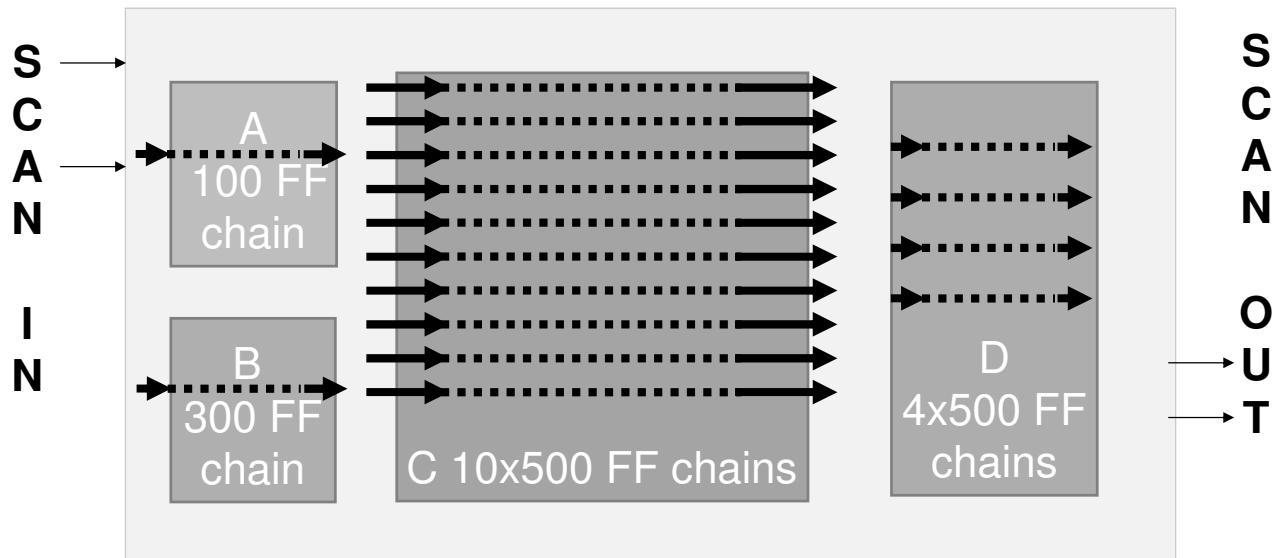
10-38

If you specify “no\_mix” at the block level no block-level lock-up elements are required. Balancing with “clock mixing” can be done at the top-level and will require fewer total lock-up elements.

# Test for Understanding



During block level scan synthesis,  
how many end of chain lock-up latches  
would be inserted?



If the blocks had no end of chain lock-up latches, how many top-level lock-up latches would be needed for 2 chains with clock mixing enabled?

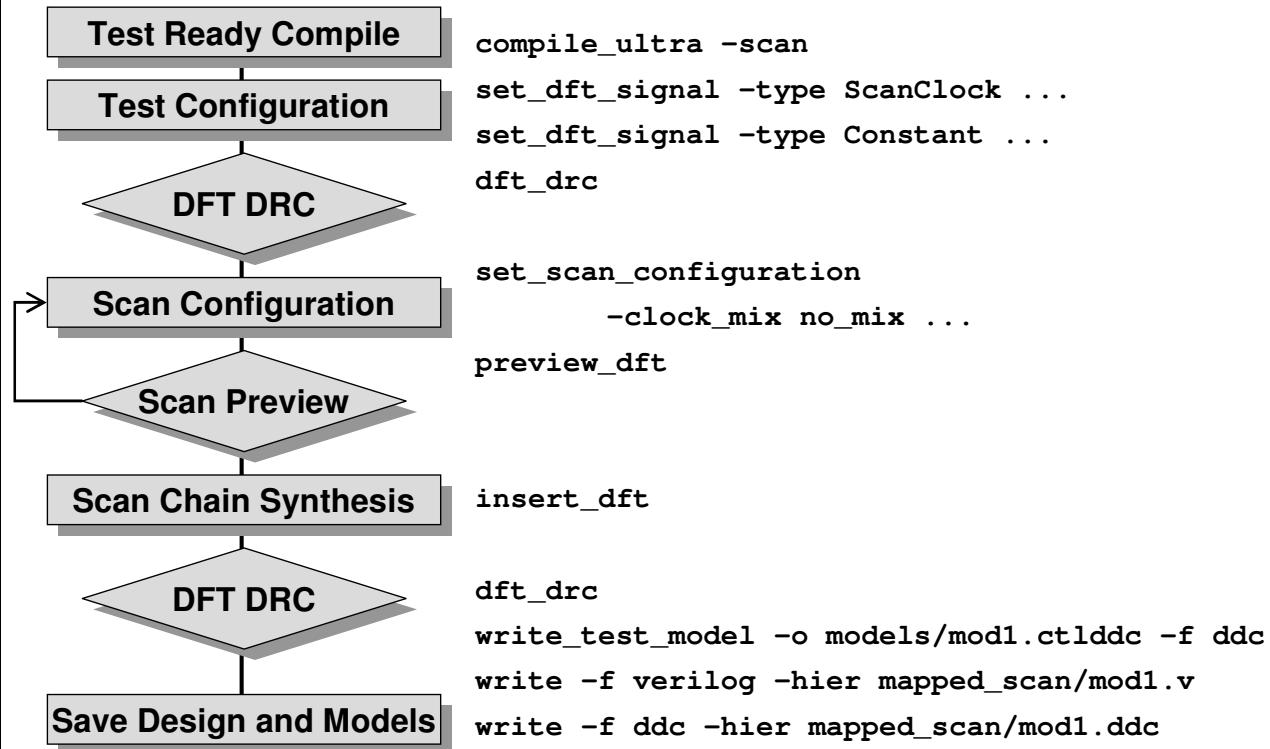
10-39

In UNIX you can try:

```
unix % fgrep LOCKUP *_scan.v | wc -l
```

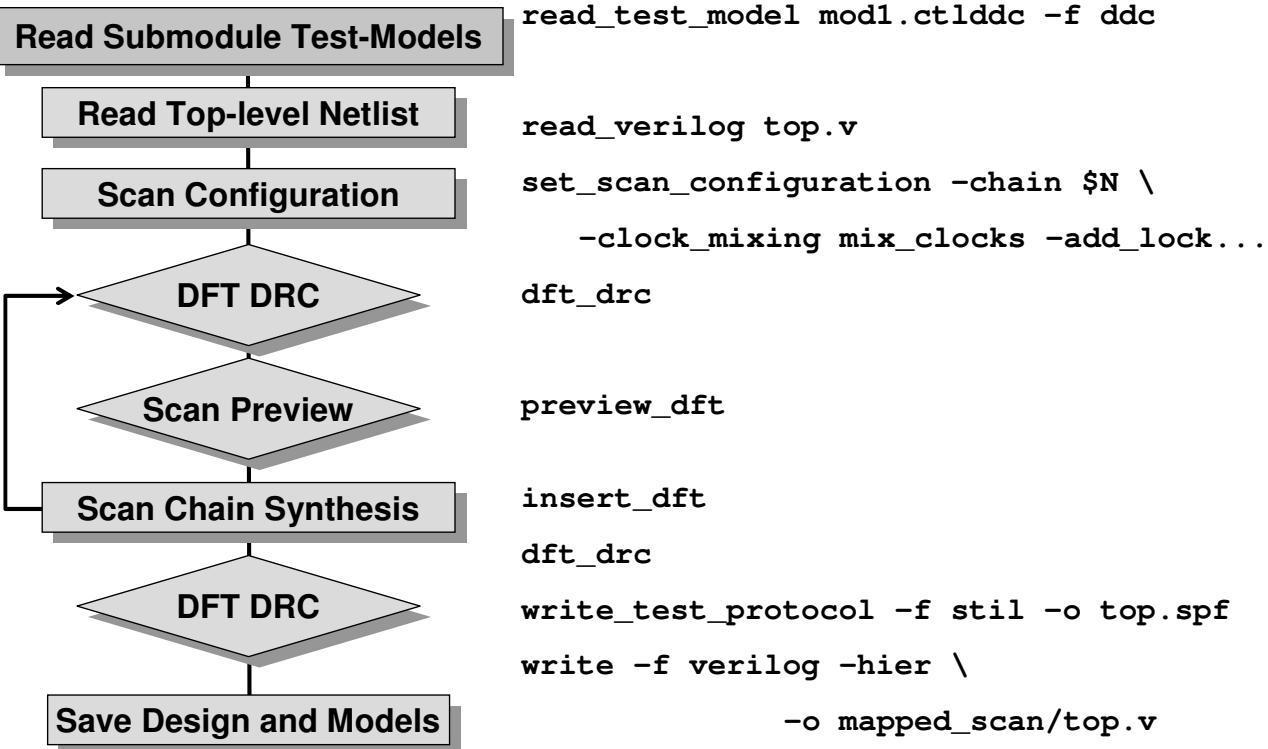
to get the number of lock-up latches added to your design.

# Script for Block Level HSS



10-40

# Script for Top-Level HSS



10-41

# Enhanced DFT DRC Reporting With HSS

- When running an HSS flow, dft\_drc report:

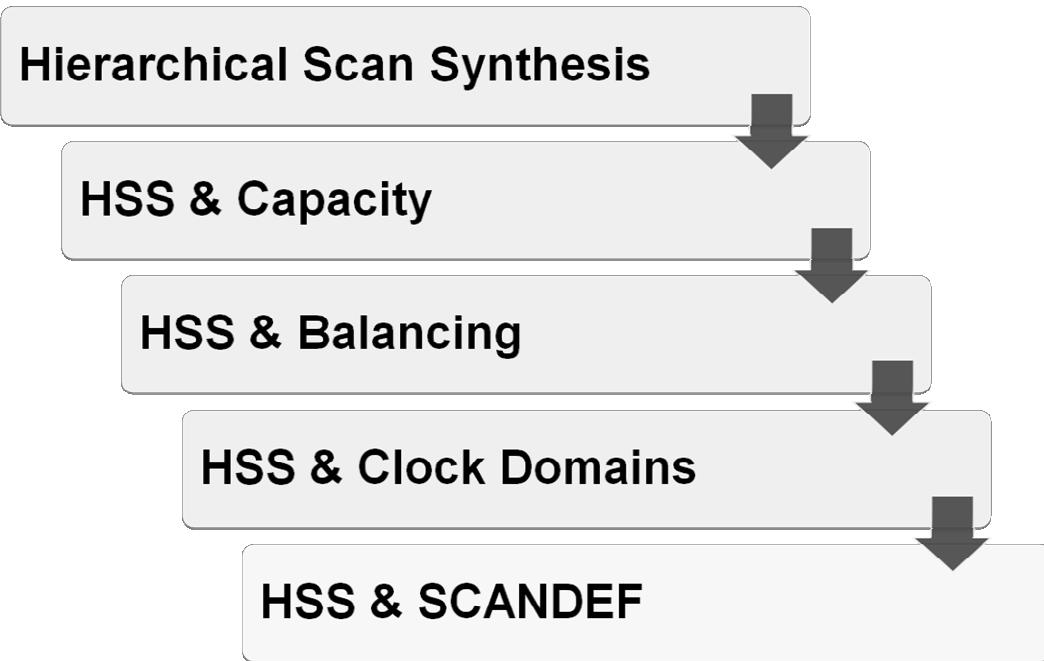
Sequential Cell Report:	Sequential Cells	Core Segments	Core Segment Cells
Sequential Elements Detected:	2961	27	2949
Clock Gating Cells :	0		
Synchronizing Cells :	4		
Non-Scan Elements :	0		
Excluded Scan Elements :	0	0	0
Violated Scan Elements :	0	0	0
(Traced) Scan Elements :	12 ( 0.4%)	27	2949 (100.0%)

- In preview\_dft report, core segments noted by (s)

```
Scan chain 'chain1' (pad[1] --> sd_A[1]) contains 528 cells:  
I_ORCA_TOP/I_SDRAM_IF/1 (s) (sdr_clk, 55.0, falling)  
I_ORCA_TOP/I_SDRAM_IF/4 (s)  
I_ORCA_TOP/I_SDRAM_IF/6 (s)
```

10-42

# High Capacity DFT Flow: Agenda



10-43

# Example Script (DC Block-level)

```
read_ddc block_test_ready.ddc
set_dft_signal -view existing_dft -type ScanClock \
                -port clock -timing [45 55]
create_test_protocol -capture_procedure multi_clock
dft_drc
preview_dft
insert_dft
dft_drc
change_names -rules verilog -hier
write_scan_def -o blk.scandef
# Write the DDC *after* write_scan_def
write -f ddc -hier -o block_with_scandef.ddc
write_test_model -o blk.ctlfdc
write -f verilog -hier -o block.v
```

10-44

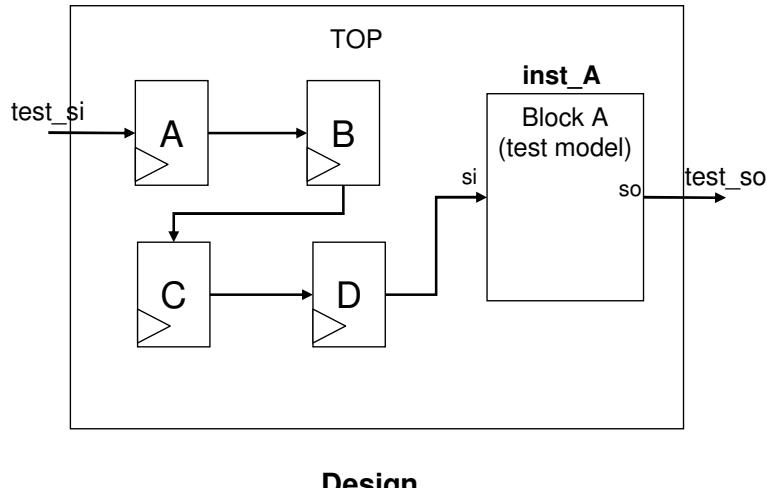
# Example Script (DC Top-level)

```
## Reading top-level design
read_verilog top_test_ready.v
## Reading test-models
read_test_model blk.ctlddc
current_design top
link
set_dft_signal -view existing_dft -type ScanClock \
    -port clock -timing [45 55]
create_test_protocol -capture_procedure multi_clock
dft_drc
preview_dft
insert_dft
dft_drc
change_names -rules verilog -hier
write_scan_def -o top.scandef
write_test_protocol -o test_mode.spf
## Remove the test-model blocks so that DC doesn't write empty modules
## for the blocks
remove_design block
# Write the DDC *after* write_scan_def
write -f ddc -hier -o top_with_scandef.ddc
write -f verilog -hier -o top.v
```

10-45

- Remember that you have to generate separate SCANDEF files at block & top level.
- CTL (test-models) are NOT used in IC Compiler

# SCANDEF Example with Test Models



SCANDEF

```
DESIGN TOP ;  
SCANCHAINS 1 ;  
- 1  
+ START PIN test_si  
+ FLOATING A ( IN SI ) ( OUT Q )  
B ( IN SI ) ( OUT Q )  
C ( IN SI ) ( OUT Q )  
D ( IN SI ) ( OUT Q )  
inst_A ( IN si ) ( OUT so ) (BITS 5)  
+ STOP PIN test_so
```

10-46

# Unit Summary

**Having completed this unit, you should now be able to:**

- **Identify the capacity and run-time bottlenecks using full gate-level designs in both top-down and bottom-up scan insertion flows**
- **Describe how Test Models improve scan insertion capacity and run-time in a bottom-up flow**
- **State the two methods for implementing Test Models in a scan insertion flow**
- **Explain two aspects of scan chain architecture that are more difficult to achieve in a bottom-up versus top-down scan insertion flow**

**10-47**

# Lab 10: High Capacity DFT Flows



75 minutes

After completing this lab, you should be able to:

- Given existing scan insertion scripts, implement Rapid Scan Synthesis (RSS) in a top-down scan insertion flow achieving well balanced scan chains
- Modify a bottom-up scan insertion script for full gate-level designs to use Test Models with RSS and run it
- Preview top-level chain balance using Test Models after block level scan insertion and revise block level scan architecture as needed to improve top-level scan chain balance

10- 48

# Command Summary (Lecture, Lab)

<code>insert_dft</code>	Adds scan circuitry to the current design
<code>write_test_model</code>	Writes a test model file
<code>read_test_model</code>	Reads a test model file
<code>write</code>	Writes a design netlist from memory to a file
<code>use_test_model</code>	Specifies for which sub-designs the CTL models should be used during scan architect and insertion
<code>read_lib</code>	Reads a technology, physical, or symbol library
<code>current_design</code>	Sets the working design
<code>report_scan_path</code>	Displays scan paths and scan cells in the scan paths set by <code>set_scan_path</code> command. Also displays scan paths inserted by <code>insert_dft</code>
<code>report_dft_signal</code>	Displays options set by <code>set_dft_signal</code> command
<code>read_ddc</code>	Reads in one or more design files in Synopsys DC database (.ddc) format
<code>set_scan_configuration</code>	Specifies the scan chain design

10-49

# **Appendix A**

## **CTL Syntax Examples**

# CTL Syntax: Scan Chain

- A ScanStructure block defines the chains

```
ScanStructures Internal_scan {  
    ScanChain "1" {  
        ScanLength 3;  
        ScanCells "temp1_reg" "uclkgen" "out_reg";  
        ScanIn "test_si";  
        ScanOut "out";  
        ScanEnable "test_se";  
        ScanMasterClock "clk";  
    }  
}
```

10-51

## P1450.6

IEEE Standard Test Interface Language (STIL) for  
Core Test Language (CTL) Extension

There should be a section for each scan chain listing **ScanDataIn**, **ScanDataOut**,  
**ScanEnable** and **ScanMasterClock**.

# CTL Syntax: Procedures

```
Procedures Internal_scan {
    "capture" { .. }
    "capture_clk" {...}
    "load_unload" {
        W "_default_WFT_";
        C {
            "all_inputs" = ONN1NN;
            "all_outputs" = XX;
        }
    }
    "Internal_scan_pre_shift" : V {
        "_clk" = 0;   "_si" = N;   "_so" = X;   "test_se"=1;
    }
    Shift {
        V {
            "_clk" = P;"_si" = #;   "_so" = #;
        }
    }
}
```

10-52

There should be a procedure called **Internal\_scan** that specifies the scan chain shift info, and capture clocks in the test model

# CTL Syntax: Scan Signals (1/2)

## ■ Scan Clock

```
"clk" { DataType ScanMasterClock MasterClock; }
```

## ■ Test Mode

```
"tm" { DataType TestMode; }
```

## ■ Scan Out

```
"test_so" { LaunchClock "clk" {LeadingEdge; }
             DataType ScanDataOut {
               ScanDataType Internal;
             }
             ScanStyle MultiplexedData;
           }
```

10-53

Example of how a **ScanClock**, **TestMode** pin and scan out are specified

Notice the **test\_so** (**ScanDataOut**) specification also lists the polarity of the clock. This is referred to as the launch clock for the chain.

**LeadingEdge** means the chain ends with a positive polarity flop

# CTL Syntax: Scan Signals (2/2)

## ■ Scan In

```
"test_si" { IsConnected In {
    Signal "temp1_reg/TI";
}
CaptureClock "clk" {
    LeadingEdge;
}
DataType ScanDataIn {
    ScanDataType Internal;
}
ScanStyle MultiplexedData;
}
```

## ■ Scan Enable

```
"test_se" {
    DataType ScanEnable {
        ActiveState ForceUp;
```

**10-54**

The **test\_si** (**ScanDataIn**) info includes captureclock info for first flop on the chain,  
**LeadingEdge** vs **TrailingEdge**

# CTL Syntax: Terminal Lock-ups

## ■ Terminal Lock-up Element

```
"scan_out0" {
    LaunchClock "ck_dp" {
        LeadingEdge;
    }
    OutputProperty SynchLatch;
    DataType ScanDataOut {
        ScanDataType Internal;
    }
    ScanStyle MultiplexedData;
}
```

10-55

If chain end with a terminal lock-up (end of chain lock-up) you should see the **SynchLatch** **OutputProperty**

# CTL Syntax: Feedthroughs

## ■ Feedthrough Signal

```
"clkout" {  
    IsConnected Out {  
        Signal "clk";  
    }  
}
```

**10-56**

For any feedthroughs you should see the above syntax

This means that the output port “**clkout**” is connected to the input port “**clk**”.

## **Appendix B**

### **Test Model Extraction**

# How to Extract a Test Model?

- You can extract a test model from an `scan_existing` block
- Identify Clocks, Resets, ScanEnable, Constants, Scan In's/Out's, and Scan Paths (`set_scan_path`)
- Run `dft_drc` to extract existing chains
- Verify with `report_scan_path`
- Write out test model

**10-58**

This “extraction” flow is not necessarily a recommended flow. It is normally not required because Test Models are automatically created when performing block-level scan insertion.

# Flow for Extraction

```
read_verilog blockA.v
set_dft_signal -type ScanClock ...      -view exist
set_dft_signal -type Constant ...       -view exist
set_dft_signal -type ScanDataIn ...     -view exist
set_dft_signal -type ScanDataOut ...    -view exist
set_dft_signal -type ScanEnable ...     -view exist
set_scan_path chain1 -view exist \
               -scan_data_in test_si -scan_data_out out
set_scan_state scan_existing
create_test_protocol
dft_drc
report_scan_path -view exist

write_test_model -o blockA.ctlddc -f ddc
```

10-59

Note the use of “**-view exist**” for the **set\_scan\_path** command.

Also note the use of the **set\_scan\_state** command to set the state to “**scan\_existing**”.

# **Appendix C**

## **CTLGEN Utility**

# Script to Generate CTL Test Models

```
# Source the ctlgen.tcl script
source ctlgen.tcl

# Read in the target design
read_verilog -netlist my_module.v
current_design my_module

# Declare the Test interface signals
set_ctlgen_signal -type scanclock -port clk
set_ctlgen_signal -type ScanEnable -port se
set_ctlgen_signal -type ScanDataIn -port si1
set_ctlgen_signal -type ScanDataIn -port si2
set_ctlgen_signal -type ScanDataOut -port so1
set_ctlgen_signal -type ScanDataOut -port so2

# Declare the scan chains
set_ctlgen_path -name chain_1 -scan_data_in si1 -scan_data_out so1 \
-length 5 -scan_master_clock clk -scan_enable se
set_ctlgen_path -name chain_2 -scan_data_in si2 -scan_data_out so2 \
-length 2 -scan_master_clock clk -scan_enable se

# Generate the CTL model
ctlgen -output my_module.ctl
```

TCL Script “ctlgen” can be used to generate CTL Test Models

See SolvNet Article:

<https://solvnet.synopsys.com/retrieve/018658.html>

10-61

This page was intentionally left blank

# Agenda

**DAY  
3**

**9 Export**



**10 High Capacity DFT Flow**



**11 Multi-Mode DFT**



**12 DFT MAX**



**13 Conclusion**

# Unit Objectives



**After completing this unit, you should be able to:**

- **Name two motivations for using multiple test modes in a design**
- **Know how to declare multiple test mode configurations in DFT Compiler**
- **Export the necessary files to run ATPG on a Multi-Mode design**

**11-2**

# Multi-Mode DFT: Agenda

Multi-Mode Motivation

Declaring Multiple Modes

Multi-Mode Specifications

Other Considerations

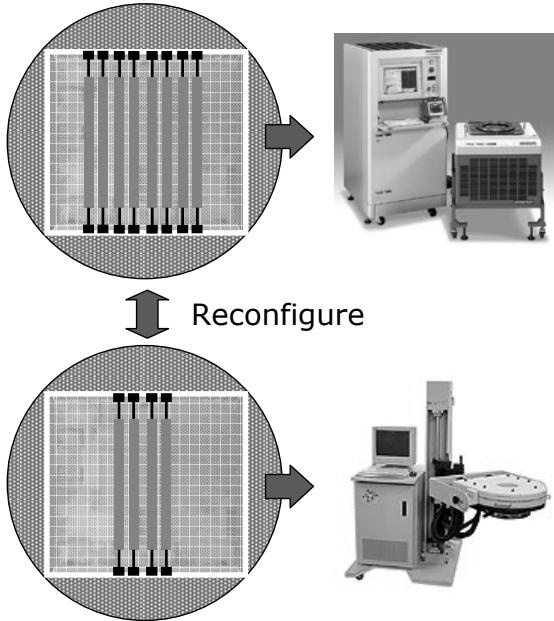
11-3

# Why Use Multi-Mode?

- **Different levels of access to the scan elements of a design may be required for various production testing steps**
  - Examples: wafer probe, package test, and burn-in
- **To accommodate these different test requirements, multiple scan architectures must be provided on the same scan design**
- **These different Test requirements can be specified to DFTC by defining multiple “modes”**

11-4

# Multiple Test Configurations

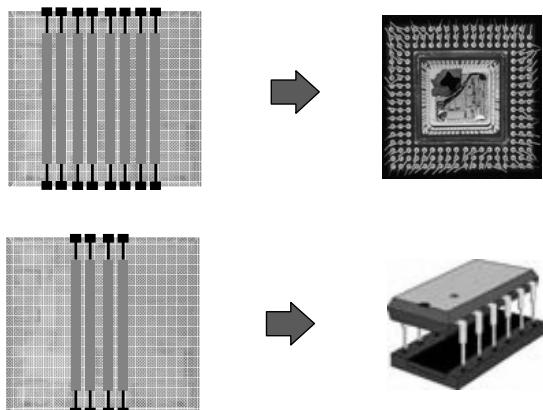


- There may be different Test Configurations (wafer probe, package test, burn-in) that have different scan interface required depending on available pins, ATE resources, multi-site testing plans etc.

11-5

# Multiple Package Configurations

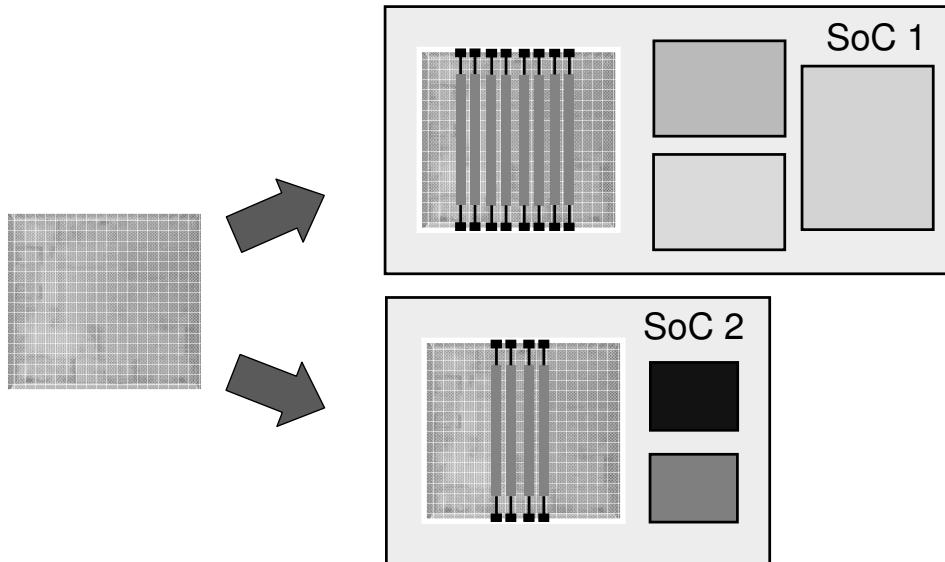
- A chip may have multiple package configurations that require a varying number of scan interface signals to take full advantage of the available pin resources



11-6

# Embedded Core Applications

- Embedded cores sold as IP may need to be flexible regarding the scan chain architecture depending on the target SoC



11-7

# Multi-Mode DFT: Agenda

Multi-Mode Motivation

Declaring Multiple Modes

Multi-Mode Specifications

Other Considerations

11-8

# Using Multi-Mode

- Must first declare TestMode ports  
(`set_dft_signal -type TestMode`)
- The `define_test_mode` command is used to declare the user specified test modes
- The declared test modes can be decoded “one hot” or binary
- DFT Compiler automatically inserts the TestMode decode logic

11-9

May declare up to  $2^{** \#TestMode\_ports}$  different modes (binary decode)

All unspecified combinations of values applied on declared test mode ports will not be decoded

# Specifying Mode Control Ports

- Specify the mode pin decoding style using:

```
set_dft_configuration \
    -mode_decoding_style <one_hot | binary>


- binary – binary decode modes from TestMode ports
- one_hot – each mode requires a separate TestMode port

```

- Specifying a port to be used as TestMode signal:

```
set_dft_signal -type TestMode \
    -port <name_of_top_level_port> \
    -hookup_pin <hookup_point>


- NOTE: Mode control pin cannot be shared with AutoFix

```

11-10

# Using `define_test_mode`

## ■ Usage

```
define_test_mode mode_name  
-usage [scan|scan_compression|wrp_if|wrp_of  
|wrp_safe|bist|bist_controller|bist_scan]  
-encoding [<port_name> <0|1> ...]
```

## ■ Example: Create a scan test mode

```
define_test_mode my_test_mode
```

## ■ Example: Create an Adaptive Scan mode

```
define_test_mode my_comp_mode \  
-usage scan_compression
```

11-11

### Usage notes for `define_test_mode`:

- Note: Mode names are **CASE SENSITIVE**
- Test Mode Ports must be defined with `set_dft_signal -Type TestMode`
- Specifications of ports and values must be done across **all modes**
- If 2 modes have the same encoding, `preview_dft` or `insert_dft` will give an **Error**
- The encoding data of a `define_test_mode` command which is issued overrides any previous encoding data for that mode
- Specifications of encoding must be done for **all modes** defined with `define_test_mode`. Any mode which does not have encoding specified will cause `preview_dft` and `insert_dft` to error out
- After the `define_test_mode` command, the defined test mode will be set as the “current test mode”. To change the current test mode, use the `current_test_mode` command

# Encoding Example

- Goal: Insert a test mode controller in a scan design encoding the modes using pins DBC, IBN, and SBN as shown on the right

Test Mode	DBC	IBN	SBN
T0	0	0	0
T1	0	0	1
T2	0	1	0
T3	0	1	1
T4	1	0	0
T5	1	0	1
T6	1	1	0
T7	1	1	1

```
set_dft_signal -type TestMode -port [list DBC IBN SBN] -view spec  
define_test_mode T0 -usage scan -encoding {DBC 0 IBN 0 SBN 0}  
define_test_mode T1 -usage scan -encoding {DBC 0 IBN 0 SBN 1}  
define_test_mode T2 -usage scan -encoding {DBC 0 IBN 1 SBN 0}  
define_test_mode T3 -usage scan -encoding {DBC 0 IBN 1 SBN 1}  
define_test_mode T4 -usage scan -encoding {DBC 1 IBN 0 SBN 0}  
define_test_mode T5 -usage scan -encoding {DBC 1 IBN 0 SBN 1}  
define_test_mode T6 -usage scan -encoding {DBC 1 IBN 1 SBN 0}  
define_test_mode T7 -usage scan -encoding {DBC 1 IBN 1 SBN 1}
```

11-12

# Encoding Example: preview\_dft Report

```
=====
Test Mode Controller Information
=====

Test Mode Controller Ports
-----

test_mode: DBC
test_mode: IBN
test_mode: SBN

Test Mode Controller Index (MSB --> LSB)
-----

DBC, IBN, SBN
Control signal value - Test Mode
-----

000 T0 - InternalTest
001 T1 - InternalTest
010 T2 - InternalTest
011 T3 - InternalTest
100 T4 - InternalTest
101 T5 - InternalTest
110 T6 - InternalTest
111 T7 - InternalTest
```

11-13

# Multi-Mode DFT: Agenda

Multi-Mode Motivation

Declaring Multiple Modes

Multi-Mode Specifications

Other Considerations

11-14

# Performing Mode-Specific Specifications

- Certain DFTC commands can be made mode specific. There are two ways to declare mode specific commands

- Use **-test\_mode**

```
set_scan_configuration \
-test_mode <name_of_test_mode> ...
```

- Use **-test\_mode all** to perform specifications that apply in all modes

- Use **current\_test\_mode**

```
current_test_mode <name_of_test_mode>
set_scan_configuration ...
```

11-15

Note: the **define\_test\_mode** command will change the “current test mode” to the mode just defined.

After the **define\_test\_mode** command, to set the “current test mode” to cover all modes, use:  
**current\_test\_mode all\_dft**

# Multi-Mode Scan Specification

- The following commands can be mode specific:
  - `set_scan_configuration`
  - `set_scan_compression_configuration`
  - `set_scan_path`
  - `set_dft_signal`
  - `write_test_protocol`
- For the `set_scan_configuration` command, the following options can be mode specific:
  - `-chain_count`
  - `-max_length`
  - `-exact_length`
  - `-clock_mixing`
  - `-exclude`

11-16

## **preview\_dft report in Multi-Mode scan**

- **preview\_dft prints mode specific scan chains for each defined test mode**
- **Symbol (m) next to scan element indicates location of reconfigurable multiplexing logic**
- **preview\_dft will print information on all test modes to be inserted**

**11-17**

# Example: preview\_dft report

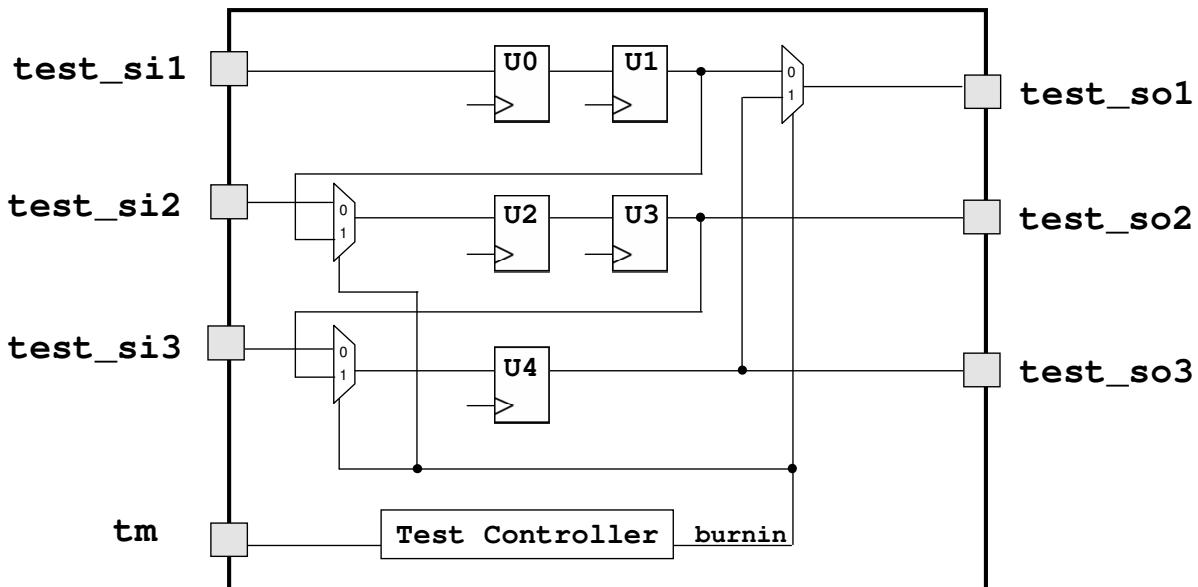
```
*****
Current mode: burnin <----- [Chains in different modes]
*****
(m) shows cell scan-out drives a multi-mode multiplexer
Scan chain '1' (test_si1 --> test_so1) contains 5 cells
Active in modes: burnin :
U0
U1 (m)
U2
U3 (m)
U4 (m)
*****
Current mode: Internal_scan
*****
Scan chain '1' (test_si1 --> test_so1) contains 2 cells
Active in modes: Internal_scan :
U0 (clk, 45.0, rising)
U1 (m)
Scan chain '2' (test_si2 --> test_so2) contains 2 cells
Active in modes: Internal_scan :
U2 (clk, 45.0, rising)
U3 (m)
Scan chain '3' (test_si3 --> test_so3) contains 1 cells
Active in modes: Internal_scan :
U4 (m) (clk, 45.0, rising)
```

[Location of multi-mode multiplexer]

11-18

# Example: Multi-mode Multiplexers

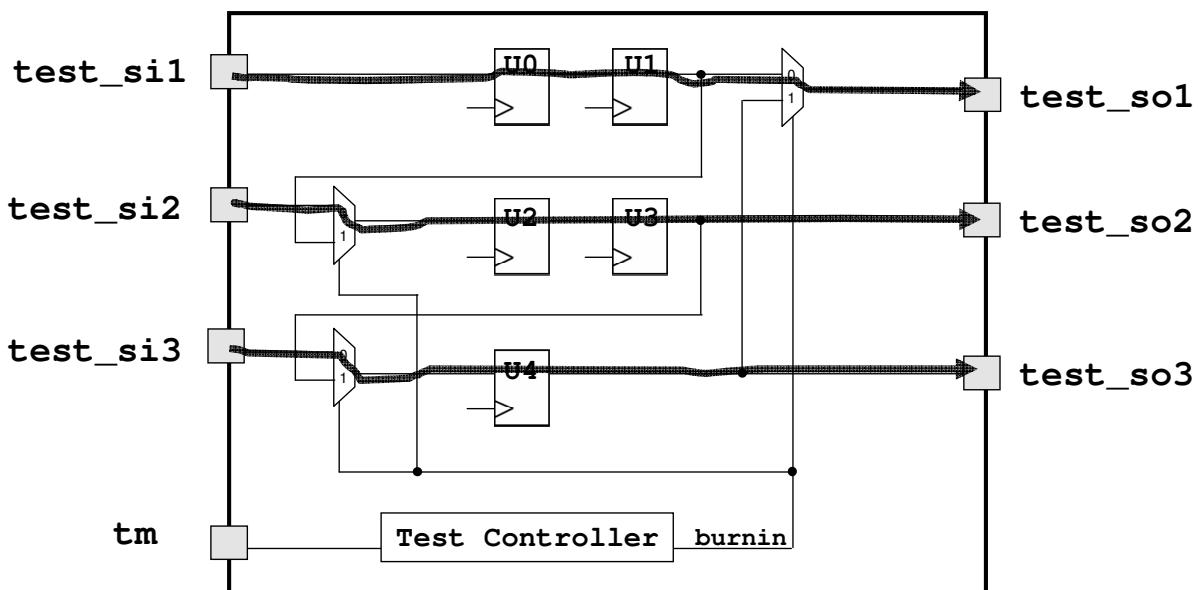
- Multi-mode multiplexer locations for the multi-mode example design



11-19

## Example: Internal\_scan Mode

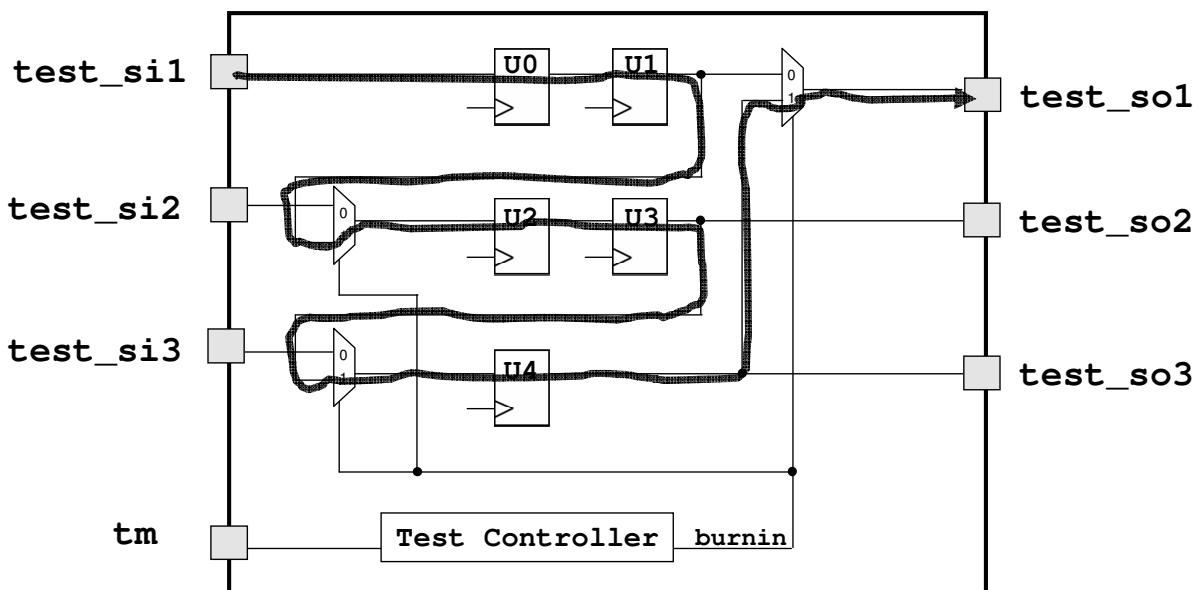
- 3 scan chains traced in Internal\_scan mode  
( $t_m = 0$ )



11-20

## Example: burnin Mode

- 1 scan chain traced in burnin mode ( $tm = 1$ )



11-21

# Multi-Mode DFT: Agenda

Multi-Mode Motivation

Declaring Multiple Modes

Multi-Mode Specifications

Other Considerations

11-22

# Running DRC in Multiple Modes

- Post-DFT DRC checks are not run against all modes simultaneously
- Must run DRC mode by mode
- Example:

```
current_test_mode <mode_1>  
dft_drc  
current_test_mode <mode_2>  
dft_drc
```

11-23

For some tips on running **dft\_drc** with multi-mode design, check out the following SolvNet article (016684):

<https://solvnet.synopsys.com/retrieve/016684.html> (**Multimode Scan - some hints and tips on its usage in DFT Compiler & DFT MAX**)

# Writing Test Protocol

- Need to specify a test mode while writing out test protocol to disk

- Command:

```
write_test_protocol \
    -test_mode <name_of_mode> \
    -o <protocol_file_for_test_mode>
```

- Note: Need to write out all test modes as separate protocol files for TetraMAX

11-24

# Multi-Mode: Example Script

```
# Define how test modes are decoded
set_dft_configuration -mode_decoding_style binary

# Define Mode Control ports (3 modes = 2 ports)
set_dft_signal -view spec -type TestMode -port Mode1
set_dft_signal -view spec -type TestMode -port Mode2

# Define test modes to be used in multi-mode
define_test_mode Scan_mode1 -view spec -usage scan
define_test_mode Scan_mode2 -view spec -usage scan
define_test_mode Single_chain -view spec -usage scan

# Define mode specific scan constraints
set_scan_configuration -chain_count 16 -test_mode Scan_mode1
set_scan_configuration -chain_count 8 -test_mode Scan_mode2
set_scan_configuration -chain_count 1 -test_mode Single_chain \
    -clock_mixing mix_clocks
```

11-25

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- **Name two motivations for using multiple test modes in a design**
- **Know how to declare multiple test mode configurations in DFT Compiler**
- **Export the necessary files to run ATPG on a Multi-Mode design**

**11-26**

# Command Summary (Lecture, Lab)

<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>define_test_mode</code>	Defines an existing test mode for model inference or a new test mode to create during DFT synthesis
<code>set_dft_configuration</code>	Sets the DFT configuration for the current design
<code>set_scan_configuration</code>	Specifies the scan chain design
<code>current_test_mode</code>	Sets or gets the working test mode for the current design
<code>set_scan_path</code>	Specifies a scan chain for the current design
<code>preview_dft</code>	Previews, but doesn't implement, the scan architecture
<code>write_test_protocol</code>	Writes a test protocol file
<code>dft_drc</code>	Checks the current design against test design rules

11-27

This page was intentionally left blank

# Agenda

**DAY  
3**

**9 Export**



**10 High Capacity DFT Flow**



**11 Multi-Mode DFT**



**12 DFT MAX**



**13 Conclusion**

# Unit Objectives



**After completing this unit, you should be able to:**

- **State which steps in the DFT flow are involved with Adaptive Scan (DFT MAX)**
- **Explain how DFT MAX maintains test coverage while reducing test application time and test data volume**
- **Modify a Mapped Flow DFT script to include Adaptive Scan**

**12-2**

# DFT MAX: Agenda

**Scan Compression Basics**

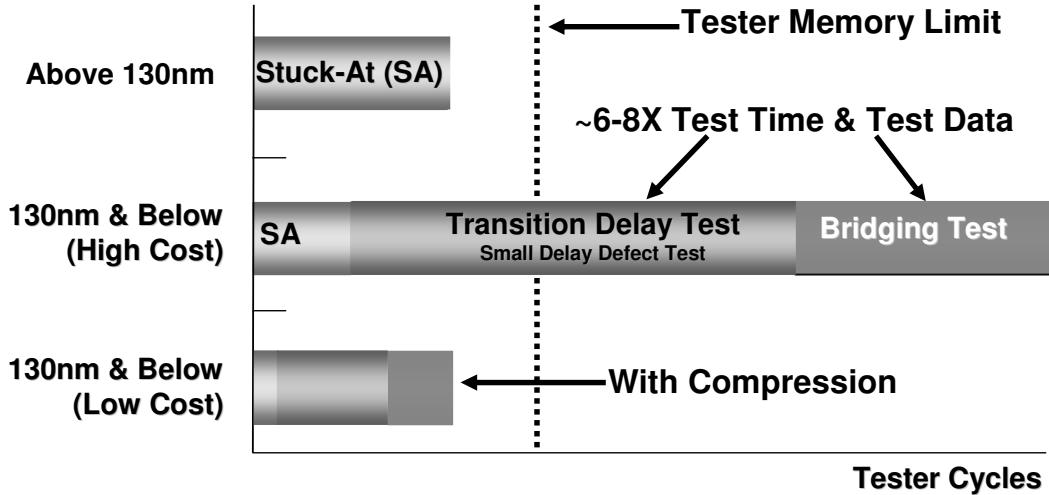
**Adaptive Scan Commands**

**Hierarchical Adaptive Scan**

**Multi-Mode Adaptive Scan**

**12-3**

# Why Use Compression?



Test Pattern Compression Needed!

12-4

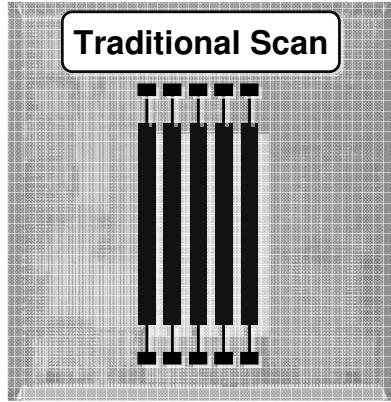
At 130nm and above, designers generally use Stuck-At tests and some Transition Delay testing up to the memory limits of their tester.

Below 130nm, Transition delay and Bridging tests become critical. The Transition Delay tests can replace some of our Stuck-At tests, but tester memory is usually far exceeded. More advanced ATPG techniques such as "Small-Delay Testing" and "Power Aware ATPG" will further increase the Test Data problem.

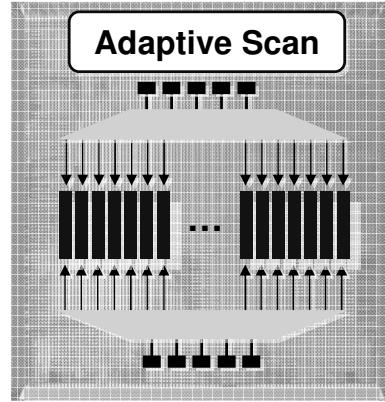
By adding compression, test quality is maintained and the cost of upgrading the tester configuration is avoided

# How Compression Works

- Traditional scan chains are split into shorter chain
- Shorter chains require less time to load and less data to be loaded on tester



One Scan chain for each  
scan-in/scan-out pin



Multiple Scan chains  
share same scan-in/  
scan-out pins

12-5

# Test Compression Concepts

## ■ Test Application Time Reduction (TATR)

- Used to reduce tester execution cost

$$TATR = \frac{\text{Regular Scan Patterns} \times \text{Regular Chain Length}}{\text{DFT MAX Patterns} \times \text{DFT MAX Chain Length}}$$

## ■ Test Data Volume Reduction (TDVR)

- Used to ensure test program resides within tester memory
- Avoids time-consuming reloads
- Used if adding more patterns to increase test quality

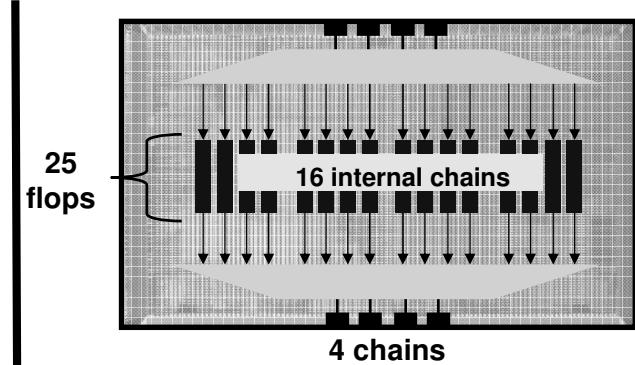
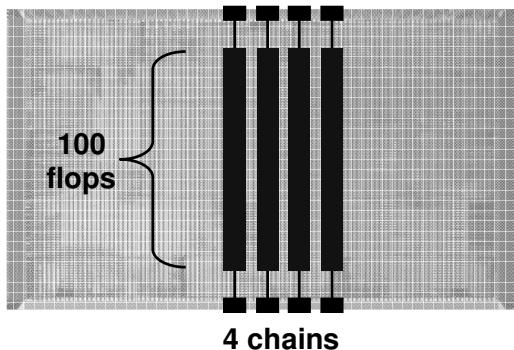
$$TDVR = \frac{\text{Regular Scan Patterns} \times \text{Regular Pin Data}}{\text{DFT MAX Patterns} \times \text{DFT MAX Pin Data}}$$

12-6

The number of tester cycles required to test a design is equal to the number of patterns times the length of the longest scan chain.

Test data volume is equal to the number of patterns times the number of scan channels times the longest chain.

# Test Cycle Savings



**Test Application Time = Patterns x ScanLength**

## Example:

- 50 patterns
  - 100 shifts to load each pattern
- =>  $50 * 100 = 5,000$  cycles

## Example:

- 60 patterns
  - 25 shifts to load each pattern
- =>  $60 * 25 = 1,500$  cycles

$$\text{TATR} = 5000 / 1500 = 3.33 \times$$

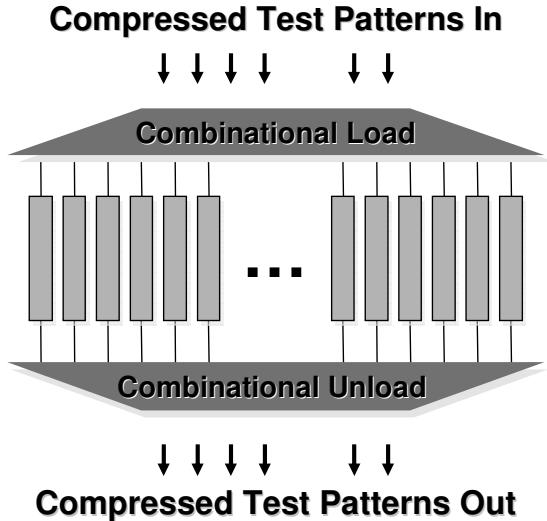
**12-7**

How does this compression work?

We know from Scan that Test Application Time is equal to the number of patterns times the length of the longest scan chain. With Adaptive Scan, we are shrinking the length of that longest chain.

For Scan, Test Data Volume is proportional to the longest chain times the number of pins we use to push the data through. Of course we then multiply by the number of patterns, but lets say that for Adaptive Scan, the pattern count is the same. Again, since the scan chain length is shorter in Adaptive Scan, but the pin count is identical, we get a lower Test Data Volume.

# Adaptive Scan Compression



- Built into Design Compiler
- As easy as scan
- Selectable compression rate
- Virtually Zero impact on timing
- Less than 0.5% area overhead
- Single synthesis command
- Tight links with physical synthesis

**Reduce chip cost  
Increase product quality  
No extra design cycles**

**12-8**

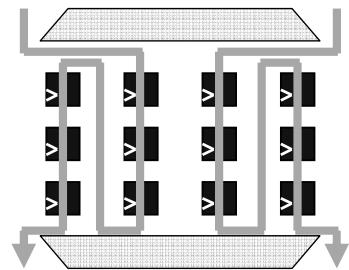
The Synopsys Compression tool is called “Adaptive Scan” (also known as “DFT MAX”)

Uses same flow as traditional scan.  
Gets same coverage as scan.  
Has similar runtime as scan.

# Test Modes Created During Insertion

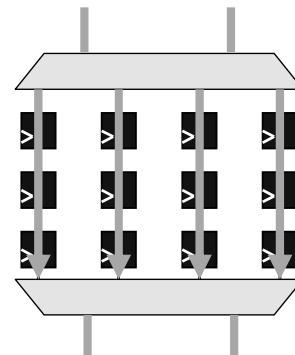
## ■ Internal Scan Mode

- Re-configures the chip to look like traditional scan
  - `dc_shell> write_test_protocol \  
-out internalscan.spf \  
-test_mode Internal_scan`
- The short compression chains are reconfigured into longer scan chains
- Also called:
  - ◆ Reconfigured scan mode
  - ◆ Fall back mode
  - ◆ Regular scan mode



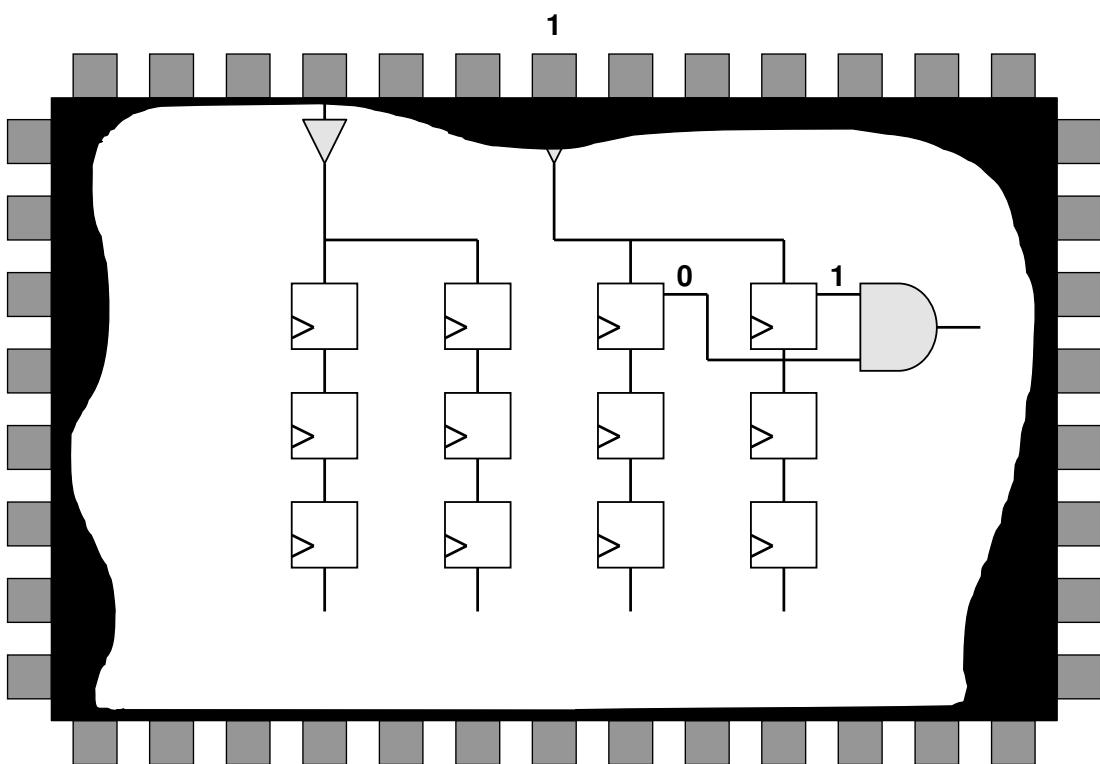
## ■ Scan Compression Mode

- Uses the shorter chains directly
  - `dc_shell> write_test_protocol \  
-out scancompression.spf \  
-test_mode ScanCompression_mode`
- Also called:
  - ◆ Adaptive scan mode
  - ◆ Scan Compression mode
  - ◆ DFT MAX



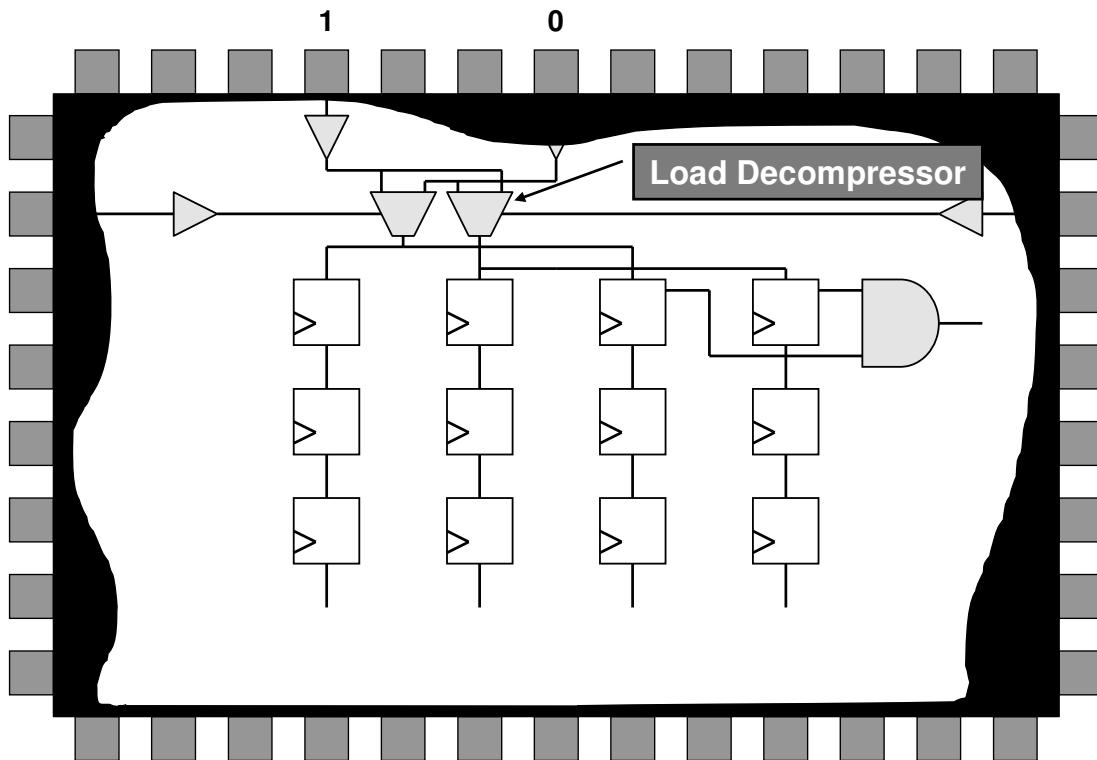
12-9

# Shared Scan-Ins Can Reduce Coverage



12-10

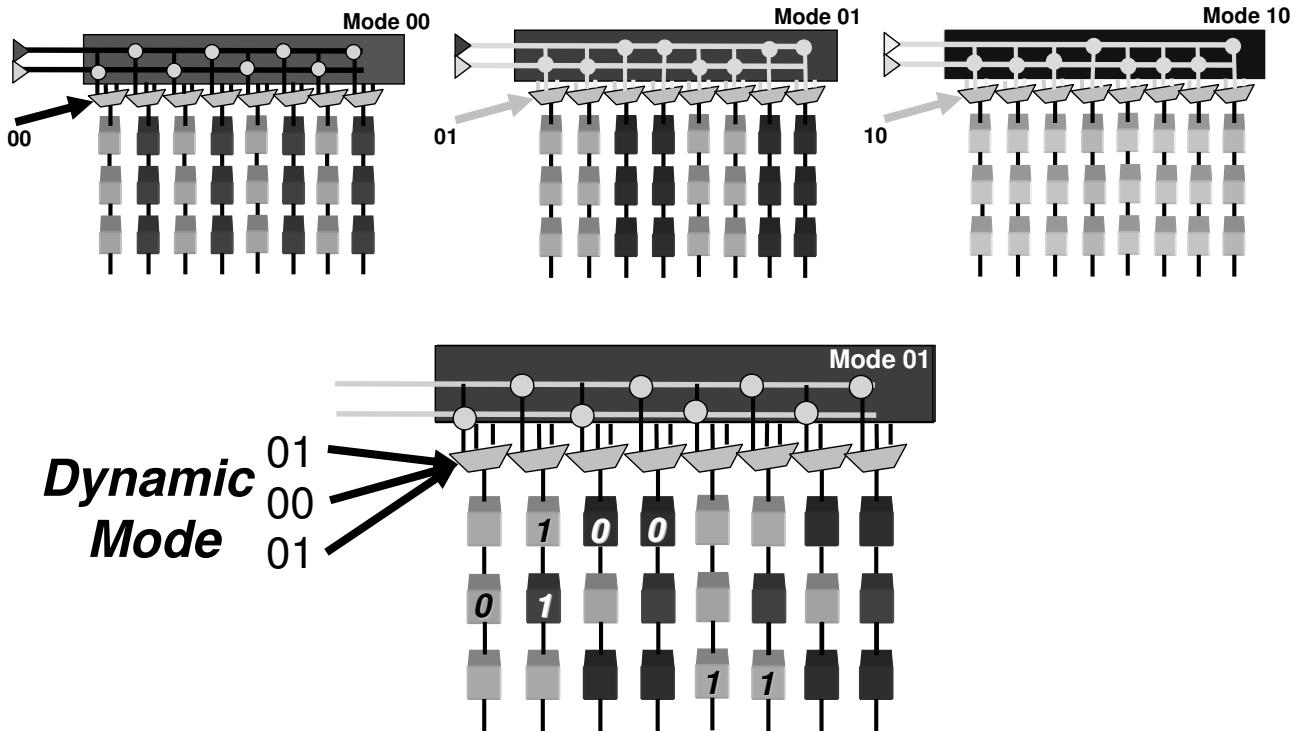
# Adaptive Scan Reduces Dependency



12-11

# Load Decompressor Principle

## Multiple Shared Scan-in Configurations



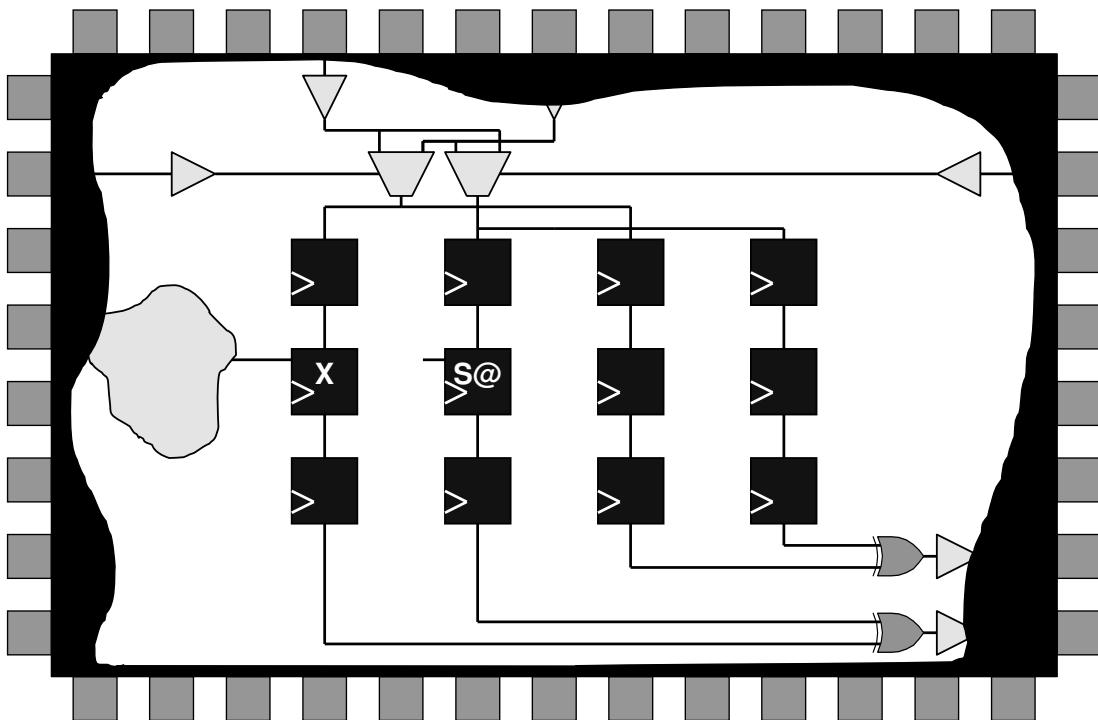
12-12

DFTMAX compression takes some of the scan inputs and uses them as control logic for the adaptive scan inputs.

From the tester, the stimulus all looks the same and the user is unable to distinguish between scan inputs for control and scan inputs for data. The reconfiguration is done on a shift by shift basis.

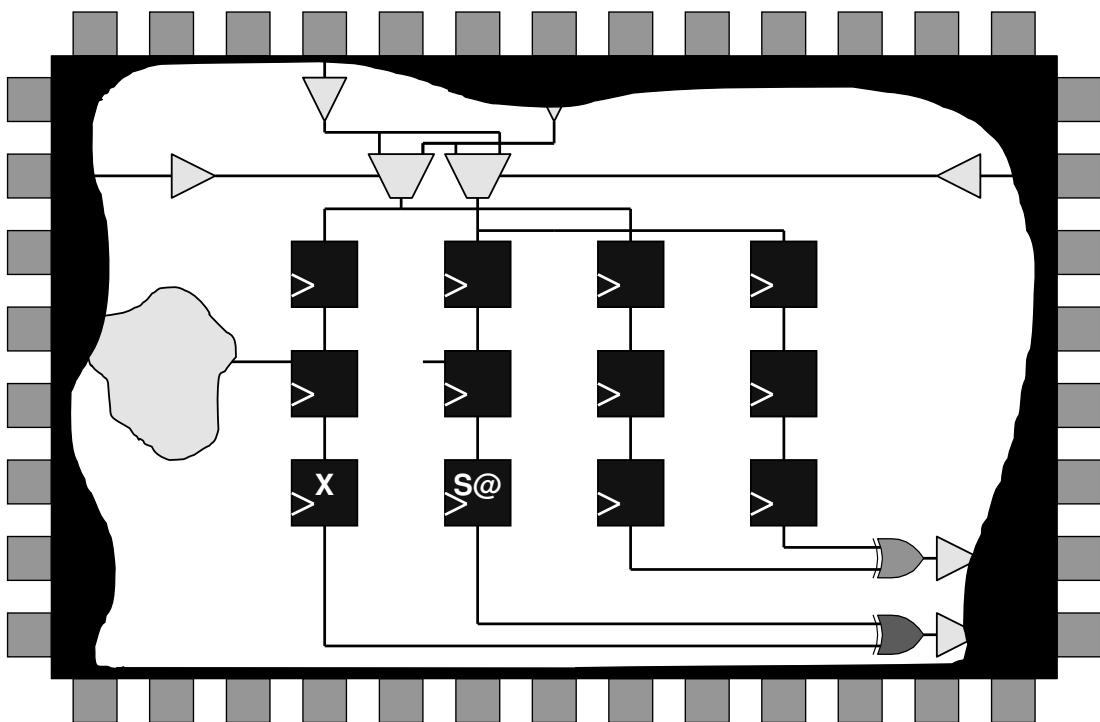
For this picture, the data required for the first shift position requires the 'red' configuration. The next requires the 'green' configuration. The last could use either the 'red' or 'yellow' configuration.

# Built-In (Default) X-Tolerance



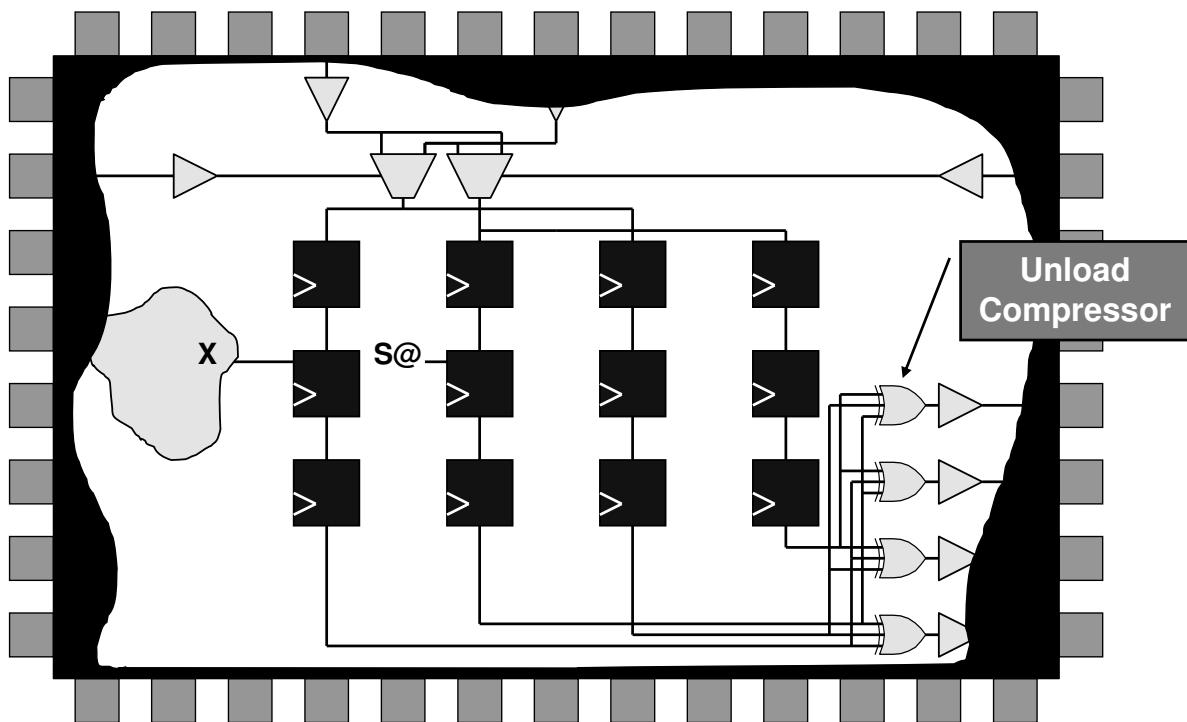
12-13

# Some Chains Still Good, Some Bad



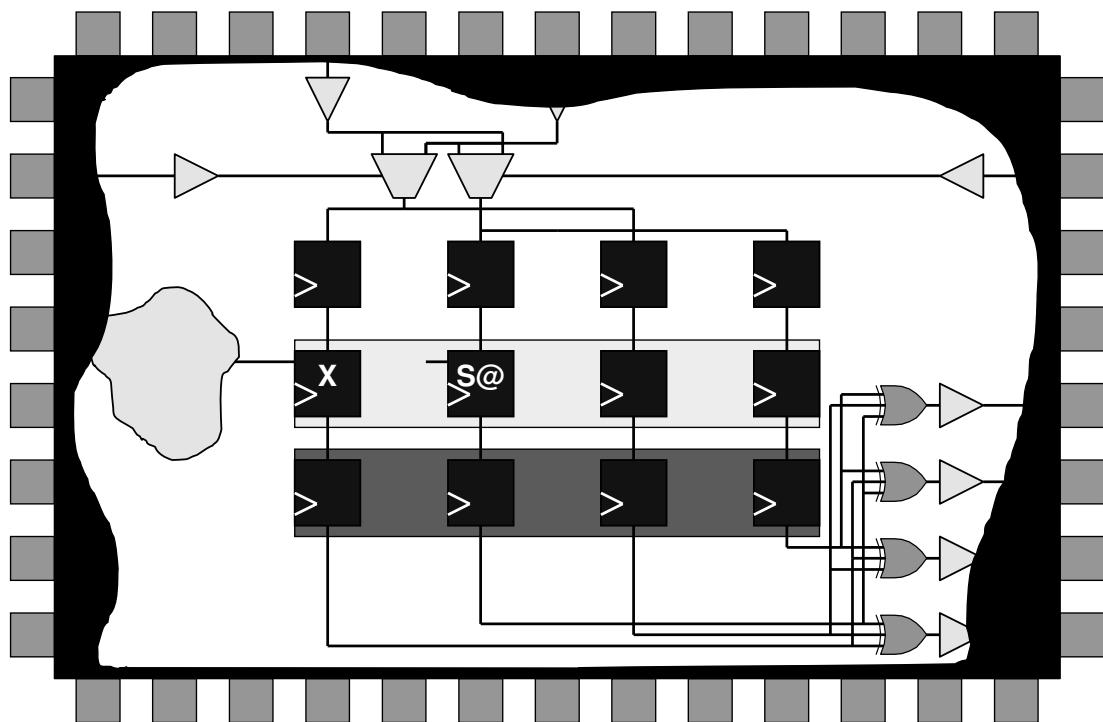
12-14

# Scan Output Redundancy



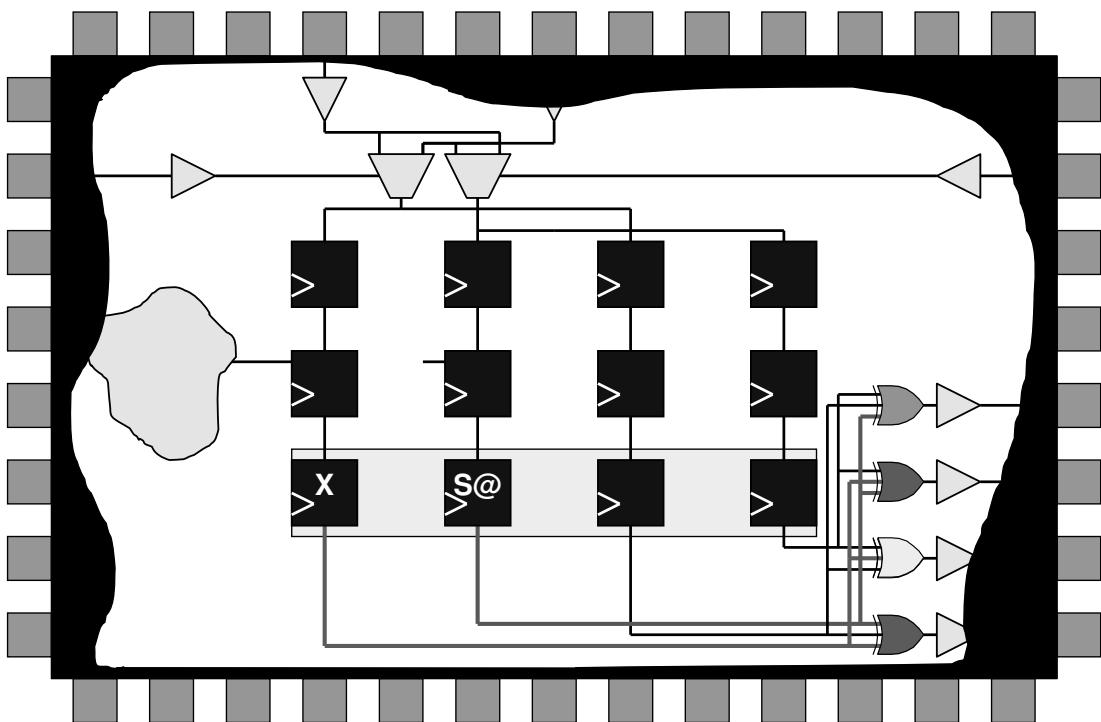
12-15

# This Row Is All Good



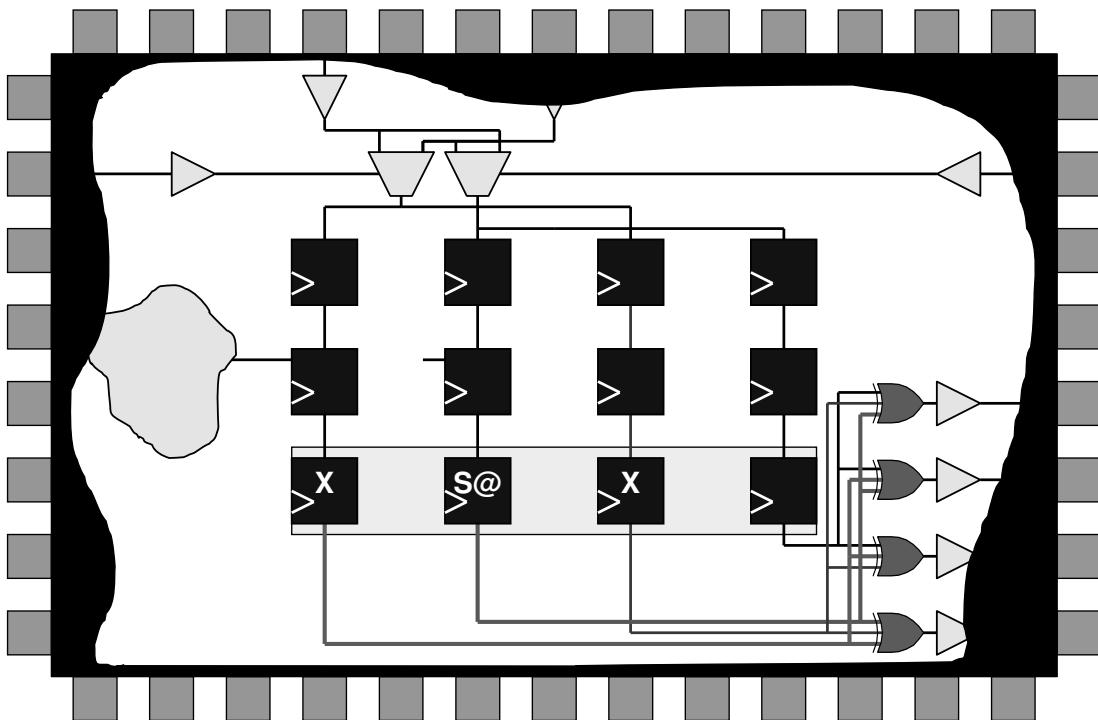
12-16

# Some Still Bad, But Others Good



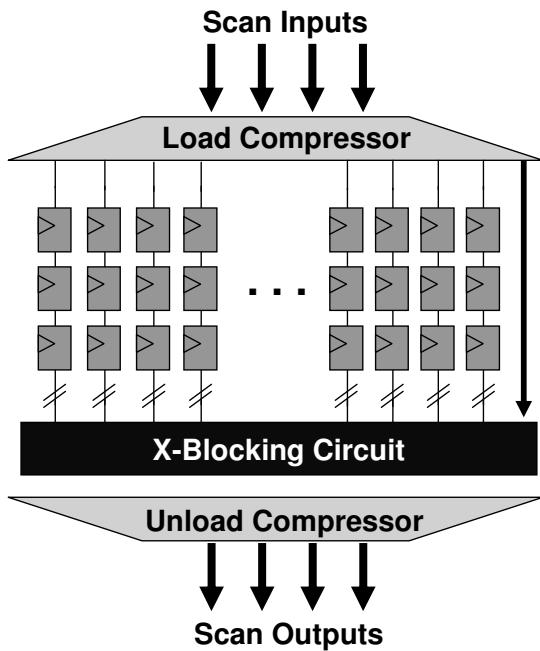
12-17

# With Many X's, All Paths May Be Blocked



12-18

# High X-Tolerance



- Full X-Tolerance
- No Additional Pins
- Works for four (4) or more scan chains
- Ideal for designs with large numbers of timing exceptions
- No drop in coverage

12-19

This slide shows the minimum 9 pin configuration possible for the current version of high X-tolerant DFTMAX.

1 test\_mode  
2 scan\_in  
1 load\_mode control  
1 mask enable  
4 scan\_out

# DFT MAX: Agenda

**Scan Compression Basics**

**Adaptive Scan Commands**

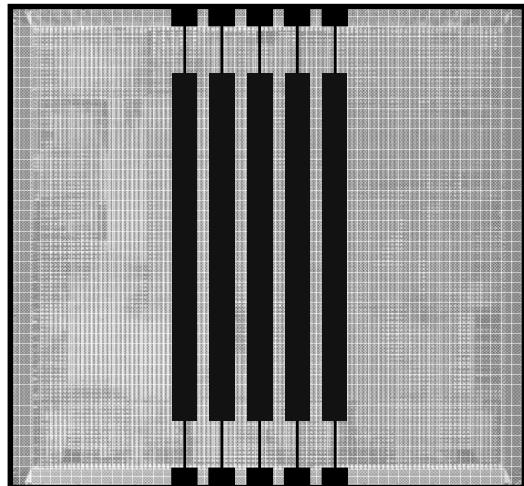
**Hierarchical Adaptive Scan**

**Multi-Mode Adaptive Scan**

**12-20**

# DFT Compiler: Regular Scan Flow

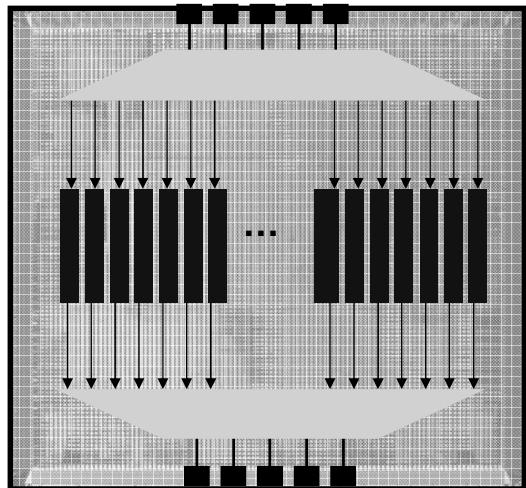
```
set_scan_configuration -chain_count <N>
set_dft_signal -type ScanClock -port clk
create_test_protocol -capture_procedure multi
dft_drc
preview_dft
insert_dft
write -f verilog -hier -o block1.v
write_test_protocol -out scan.spf \
-test_mode Internal_scan
```



12-21

# DFT Compiler: DFT MAX Flow

```
set_dft_configuration -scan_compression enable  
set_scan_configuration -chain_count <N>  
set_dft_signal -type ScanClock -port clk  
create_test_protocol -capture_procedure multi  
dft_drc  
preview_dft  
insert_dft  
write -f verilog -hier -o block1.v  
write_test_protocol -out scan.spf \  
-test_mode Internal_scan  
write_test_protocol -out scancompress.spf \  
-test_mode ScanCompression_mode
```



12-22

# DFT MAX Commands (1/2)

- **Enable Adaptive Scan (Core and Top-level)**

```
set_dft_configuration  
-scan_compression enable | disable
```

- **Control Scan Architecting (“external” chain count)**

```
set_scan_configuration -chain_count <N>
```

12-23

## DFT MAX Commands (2/2)

- Specify the number of internal compressed chains with the `set_scan_compression_configuration` command using one of three command options:
  - Set the `ScanCompression_mode` chain count  
`-chain_count <#>`
  - Set the `ScanCompression_mode` maximum chain length  
`-max_length <#>`
  - Set the Compression Level  
`-minimum_compression <N> (Default: 10)`

- To specify the X-tolerance

```
set_scan_compression_configuration
    -xtolerance default | high
```

12-24

The smallest amount of compression that can be targeted with the `set_scan_compression_configuration -minimum_compression` option is “2”. Must be integer values.

Precedence order for `set_scan_compression_configuration`

- 1) `max_length`
- 2) `chain_count`
- 3) `minimum_compression`

To compensate for a minor pattern increase in scan compression mode compared to scan mode, an “inflation factor” is used to determine the number of internal chains when `-minimum_compression` is used to determine the number of internal chains.

#internal chains = `chain_count * minimum_compression * 1.2`

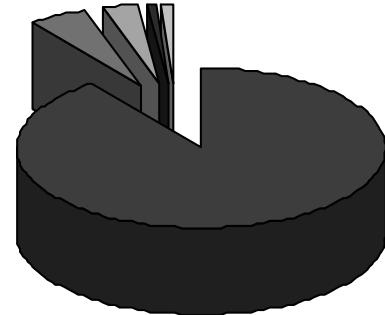
Example: For 8 regular scan chains and 10X compression:

#internal chains =  $8 * 10 * 1.2 = 96$

There's no maximum amount of compression that can be targeted with `-minimum_compression`. However, scan chain compression (not TATR nor TDVR) does increase compression costs, so an optimal amount of scan compression should be chosen rather than an arbitrarily high number.

# Incremental Improvements in TATR

Compression Factor ( $x$ )	TATR/TDVR ( $1 - 1/x$ )	Incremental Improvement
0	0%	0%
10X	$(1 - 1/10) = 90\%$	90%
20X	$(1 - 1/20) = 95\%$	5%
50X	$(1 - 1/50) = 98\%$	3%
100X	$(1 - 1/100) = 99\%$	1%



- 10X = 90% Savings
- 20X = 5% More
- 50X = 3% More
- 100X = 1% More
- ...

Increasing compression from 50X to 100X reduces test time by only 1%

**12-25**

Typical requirements for more compression is to fit on a low cost or minimally upgraded tester.

Recommendation is to only target the amount of compression that you need. With higher levels of compression come higher area overhead, increased risk of routing congestion issues, and only a small incremental improvement in TATR/TDVR.

# DFT MAX & Test Modes

- In a typical DFT MAX run, 2 new test modes are created:
  - ScanCompression\_mode (Compression mode)
  - Internal\_scan (Regular scan mode)
- In the default flow, the Compression and Regular scan modes are created automatically during scan insertion (`insert_dft`)
- Scan Inputs, Scan Outputs, and Scan Enable ports are shared by regular scan and scan compression modes
- A `TestMode` signal is required to differentiate between regular scan and scan compression modes

12-26

# TestMode Signal

- A TestMode signal is required to differentiate between regular scan and scan compression modes
  - Use `set_dft_signal` to identify an existing port
  - Cannot share the TestMode port used by AutoFix
  - Use `set_ autofix_configuration -control_signal` to identify a particular TestMode port to be used by AutoFix
  - DFT Compiler will create a new port (named “`test_mode`”) if an existing port is not identified

- Example

```
set_dft_signal -type TestMode -port tmode1  
set_dft_signal -type TestMode -port tmode2  
set_ autofix_configuration -control_signal tmode1  
• Port “tmode2” will be used as control signal to differentiate  
between Internal_scan and ScanCompression_mode
```

12-27

# DFT MAX: Agenda

Scan Compression Basics

Adaptive Scan Commands

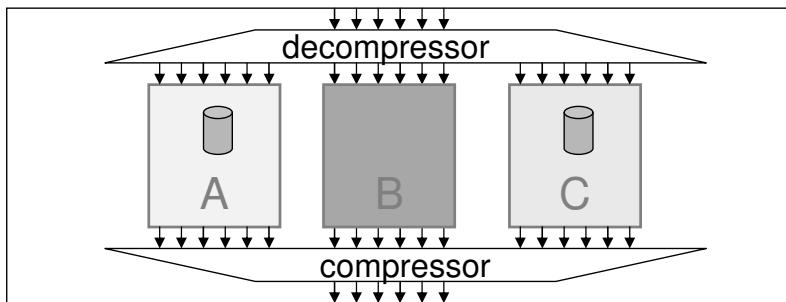
Hierarchical Adaptive Scan

Multi-Mode Adaptive Scan

12-28

# Bottom Up HSS Flow

- Bottom-up HSS flows with Test Models are supported by DFT-MAX
- Compression logic is inserted at the top level
- Make sure you create enough chains in the sub-blocks to enable balanced chains in compression mode



12-29

Review:

HSS = Hierarchical Scan Synthesis

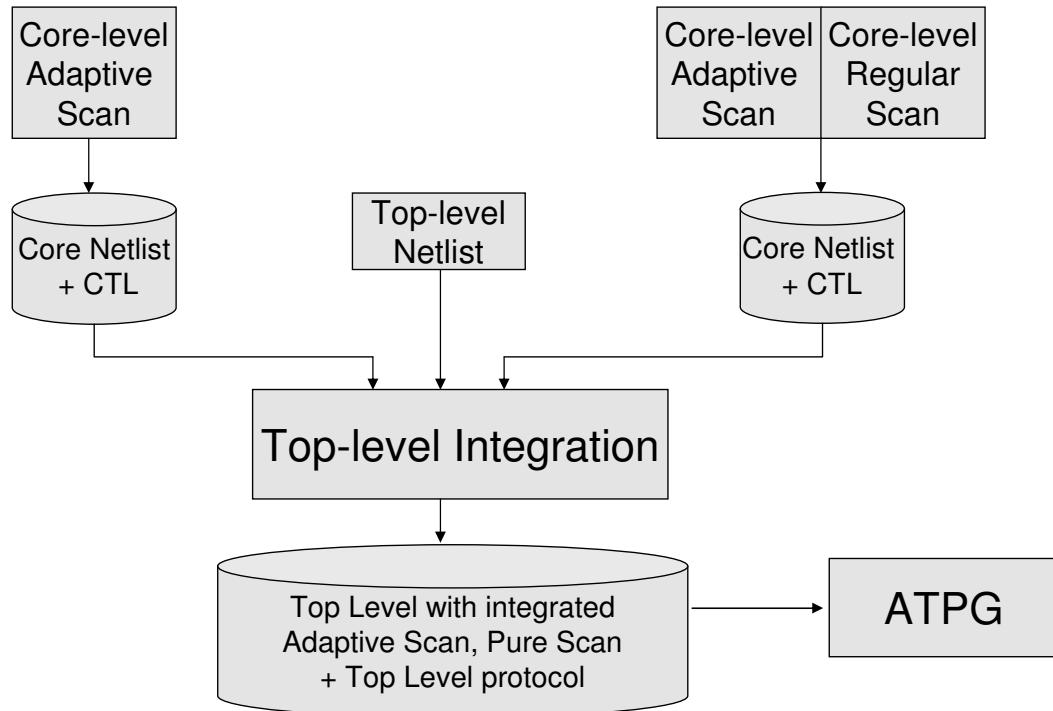
# Hierarchical Adaptive Scan Synthesis (HASS)

- Adaptive Scan cores, regular scan cores, and top level sequential logic are integrated at the top level
- Adaptive Scan logic is placed at the core level
- Helps reduce top level congestion
- Core level command sequence is the same as for a “Top Down” Adaptive Scan flow
- Top-level integration of Adaptive Scan cores is enabled by the following command:

```
set_scan_compression_configuration  
-integration_only true | false
```

12-30

# Typical HASS Flow

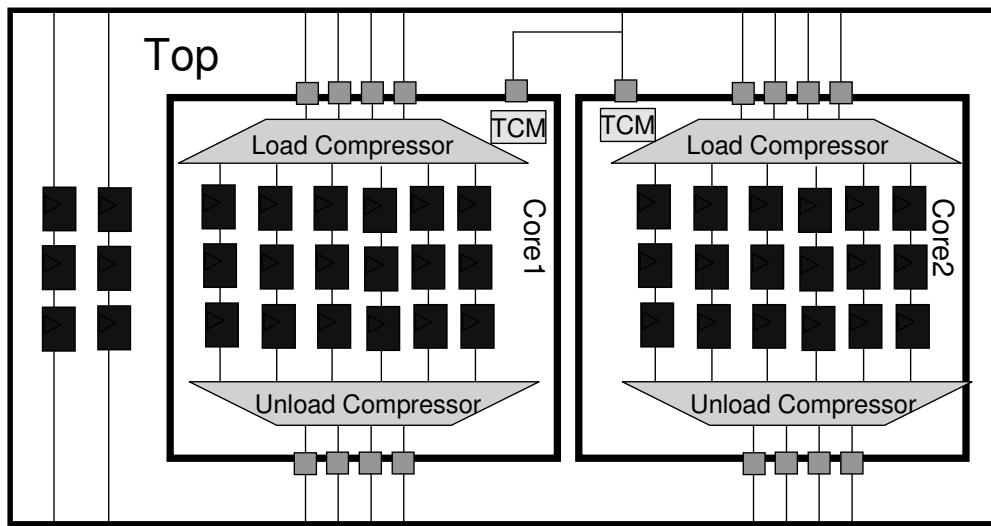


12-31

# HASS Example: Multiple Compressed Cores

## ■ Core-based integration

```
set_dft_configuration -scan_compression enable  
set_scan_compression_configuration -integration_only true
```



12-32

This HASS example has multiple compressed cores.

The Adaptive Scan logic is inserted at the Core level. After core level scan insertion (**insert\_dft**), the details of the Adaptive Scan logic is stored in the Core level Test Model.

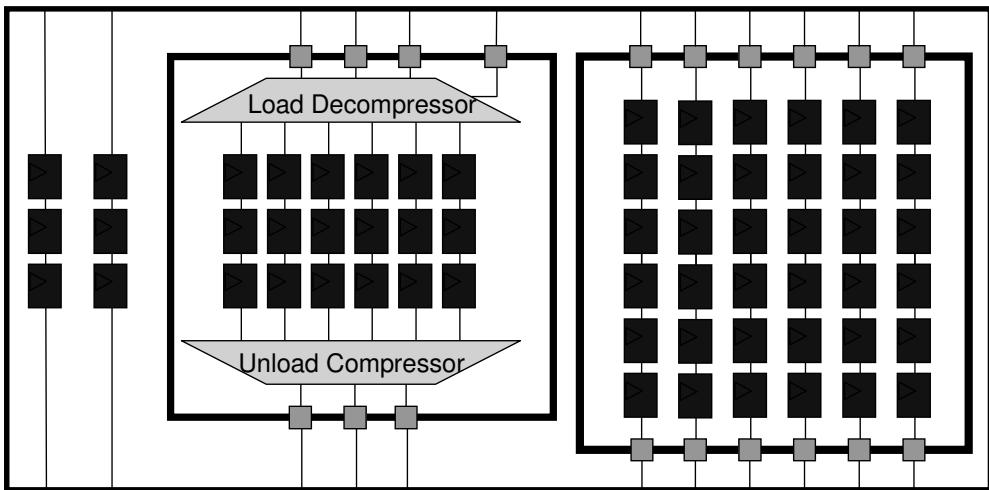
At the top-level an “Integration” step is performed. In this step Core level Adaptive Scan ports are promoted to top-level ports. Any top-level sequential logic is scan stitched at the top-level.

Enough chains need to be allocated at the top-level to account for the Adaptive scan chains being promoted to the top-level as well as any new top-level chains needed.

# HASS Example: Adaptive & Regular Scan Cores

## ■ Core-based integration

```
set_dft_configuration -scan_compression enable  
set_scan_compression_configuration -integration_only true
```



12-33

This HASS example has a compressed core, and an uncompressed core.

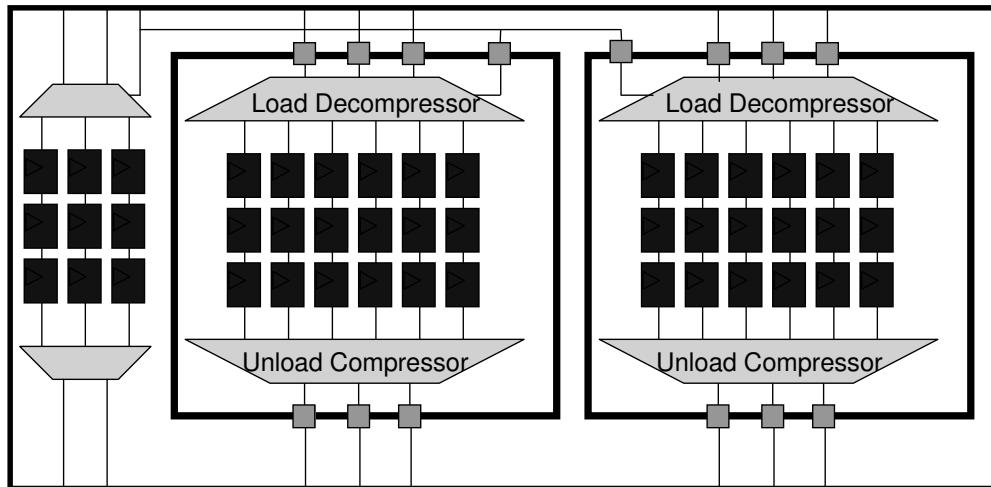
The Adaptive Scan logic is inserted at the Core level. After core level scan insertion (**insert\_dft**), the details of the Adaptive Scan logic is stored in the Core level Test Model. Similarly the uncompressed core contains a Test Model with the details of its core scan chains.

At the top-level an “Integration” step is performed. In this step Core level Adaptive Scan ports are promoted to top-level ports. The scan chains of the uncompressed core are also promoted to the top-level. Any top-level sequential logic is scan stitched at the top-level.

# HASS Example: Hybrid Flow

- Core-based Integration plus Adaptive Scan at the Top-level

```
set_dft_configuration -scan_compression enable  
set_scan_compression_configuration -hybrid true
```



```
set_scan_compression_configuration -minimum_compression X
```

12-34

This HASS example has a compressed cores, and compression logic at the top-level.

The Adaptive Scan logic is inserted at the Core level. After core level scan insertion (**insert\_dft**), the details of the Adaptive Scan logic is stored in the Core level Test Model.

At the top-level a “hybrid” integration step is performed. In this step, Core level Adaptive Scan ports are promoted to top-level ports. Adaptive scan compression logic is inserted at the top-level for any top-level sequential logic and/or regular scan cores.

## HASS Details

- **CTL, CTLDDC, or DDC test models must be used for Adaptive Scan core integration**
  - Test model is how Adaptive Scan blocks are identified
- **At the top level, HASS creates the same number of scan chains as the sum of all core scan chains in Internal\_scan mode**
- **Only one TestMode port required**
  - Shared across all Adaptive Scan cores
  - Selects between ScanCompression\_mode and Internal\_scan

12-35

# HASS ATPG vs. Regular Scan ATPG

- ATPG is run in parallel for ALL adaptive scan and regular scan cores
- The top-level `ScanCompression_mode` active chains are:
  - All Core-level Adaptive Scan chains
  - PLUS the core-level regular scan chains
  - PLUS any additional top-level regular scan chains
- The top-level `Internal_scan` active chains are:
  - All core-level Adaptive Scan reconfigured scan chains
  - PLUS the core-level regular scan chains
  - PLUS any additional top-level regular scan chains
- Top-level regular scan chains and regular scan core-level scan chains will be active in `ScanCompression_mode` AND `Internal_scan` modes

12-36

# Example Script: Top Level Integration

```
# Read in top level with any test_ready sequential logic
read_verilog my_top_test_ready.v

# Read CTL model for Adaptive Scan inserted block
read_test_model core1.ctlldc

# Read CTL model for pure scan block
read_test_model core2.ctlldc

current_design my_top
link

# Enable Adaptive Scan and configure for integration only
set_dft_configuration -scan_compression enable
set_scan_compression_configuration -integration_only true

# Create the test protocol, run DRC, and preview
create_test_protocol
dft_drc
preview_dft -show all

# Integrate the Adaptive Scan and Pure Scan cores
insert_dft

# DFT DRC in Internal_scan mode: not supported in ScanCompression_mode
current_test_mode Internal_scan
dft_drc
```

12-37

# DFT MAX: Agenda

**Scan Compression Basics**

**Adaptive Scan Commands**

**Hierarchical Adaptive Scan**

**Multi-Mode Adaptive Scan**

**12-38**

# Multi-Mode with DFT MAX

- Recall: in a typical DFT MAX run two new test modes are created:
  - `ScanCompression_mode` (Compression mode)
  - `Internal_scan` (Regular scan mode)
- In the default flow, the Compression and Regular scan modes are created automatically during scan insertion (`insert_dft`)
- The Multi-Mode client enables the explicit specification of Compression and Regular scan modes prior to scan insertion
- Using Multi-Mode specifications, various DFT attributes can be set on these modes prior to scan insertion

12-39

# Setting the Base Mode

- When multiple regular scan modes are defined as well as a compression mode, one of the regular scan modes must be associated with the compression mode as a “base mode”

```
set_scan_compression_configuration  
    -base_mode <regular_scan_mode_name>  
    -test_mode <compression_mode_name>
```

- Example:

```
define_test_mode my_base -usage scan  
define_test_mode burn_in -usage scan  
define_test_mode comp_mode -usage scan_compression  
  
set_scan_compression_configuration \  
    -base_mode my_base -chain_count 96 \  
    -test_mode comp_mode
```

12-40

# Controlling Clock Mixing by Mode

- Multi-Mode can be used to control clock mixing in Compression and Regular scan modes
- Example:
  - Clock domains : 20
  - Number of scan chains : 10
  - Number of internal chains : 120
- In this case it is not possible to turn off clock mixing for regular scan, as there are more clock domains than chains

12-41

With clock mixing at “**no\_mix**” requires at least 20 external scan chain in regular scan mode.

# Multi-Mode Adaptive Scan Example

```
# Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port [list tmodel tmode2]

# Define the two modes
define_test_mode Internal_scan -usage scan -view spec
define_test_mode burn_in -usage scan -view spec
define_test_mode ScanCompression_mode -usage scan_compression -view spec

# Specify clocks and asynchs
set_dft_signal -type ScanClock -port clk -timing {45 55} -view existing \
-test_mode all

# Specify chain counts
set_scan_configuration -chain_count 10 -test_mode Internal_scan
set_scan_configuration -chain_count 1 -test_mode burn_in
set_scan_compression_configuration -chain_count 120 \
-base_mode Internal_scan -test_mode ScanCompression_mode

# Specify clock mixing
set_scan_configuration -clock_mixing mix_clocks -test_mode Internal_scan
set_scan_configuration -clock_mixing mix_clocks -test_mode burn_in
set_scan_configuration -clock_mixing no_mix -test_mode ScanCompression_mode
```

12-42

Specify “**mix\_clocks**” in **Internal\_scan** mode to allow optimal balancing of the “external” scan chains.

Specify “**no\_mix**” in **ScanCompression\_mode** mode to keep clock domains separate during scan compression testing. Since the internal chains are much shorter in **ScanCompression\_mode**, scan chain balancing shouldn’t be a problem with “**no\_mix**”.

**Limitation:** HASS and Hybrid flows only support DFT-MAX cores with “native” modes (**ScanCompression\_mode** and **Internal\_scan**)

**Note:** the **define\_test\_mode** command will change the “current test mode” to the mode just defined.

After the **define\_test\_mode** command, to set the “current test mode” to cover all modes, use: **current\_test\_mode all\_dft**.

However, it’s recommended to use **-test\_mode <mode>** to explicitly declare the mode when declaring multi-mode specifications.

# How to Generate Output for TetraMAX

- All of the information needed to run TetraMAX is stored in the DFT MAX protocol file
- This protocol is generated during DFT insertion, and can be written out as follows:

```
write_test_protocol -out scancompress.spf \
    -test_mode ScanCompression_mode
```

- The ATPG flow in TetraMAX for DFT MAX is the same as for regular scan
- Both DFT MAX and regular scan designs use the same fault diagnosis flow in TetraMAX

12-43

# **Unit Summary**

**Having completed this unit, you should now be able to:**

- **State which steps in the DFT flow are involved with Adaptive Scan (DFT MAX)**
- **Explain how DFT MAX maintains test coverage while reducing test application time and test data volume**
- **Modify a Mapped Flow DFT script to include Adaptive Scan**

**12-44**

# Lab 12: DFT MAX



45 minutes

**After completing this lab, you should be able to:**

- Insert Adaptive Scan using top down flow
- Understand needed components for Adaptive Scan
- Run TetraMAX flow on Adaptive Scan design

**12-45**

# Command Summary (Lecture, Lab)

<code>set_dft_configuration</code>	Enables or disables the insertion of scan compression structures
<code>-scan_compression enable</code>	
<code>set_scan_compression_configuration</code>	Specifies the scan compression configuration
<code>-chain_count</code>	Specify the number of scan compression mode chains
<code>-max_length</code>	Specify the maximum allowed length of scan compression mode chains
<code>-minimum_compression</code>	Specifies the compression factor
<code>-xtolerance default   high</code>	Selects either default or fully X tolerant compression
<code>-intergration_only</code>	Turns on scan compression integration
<code>-hybrid</code>	Turns on the hybrid flow
<code>-base_mode</code>	Specifies the <code>base_mode</code> mode which is to be associated with the <code>ScanCompression_mode</code>
<code>-test_mode</code>	Specifies the scan compression mode
<code>set_scan_configuration -chain_count</code>	Specifies the number of chains <code>insert_dft</code> is to build
<code>set_dft_signal</code>	Specifies DFT signal types for DRC and DFT insertion
<code>set_autofix_configuration</code>	Controls automatic fixing of violations
<code>insert_dft</code>	Adds scan circuitry to the current design
<code>write_test_protocol -test_mode</code>	Writes a test protocol file for the given test mode

12-46

# Agenda

**DAY  
3**

**9 Export**



**10 High Capacity DFT Flow**



**11 Multi-Mode DFT**

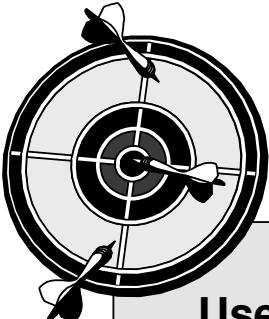


**12 DFT MAX**



**13 Conclusion**

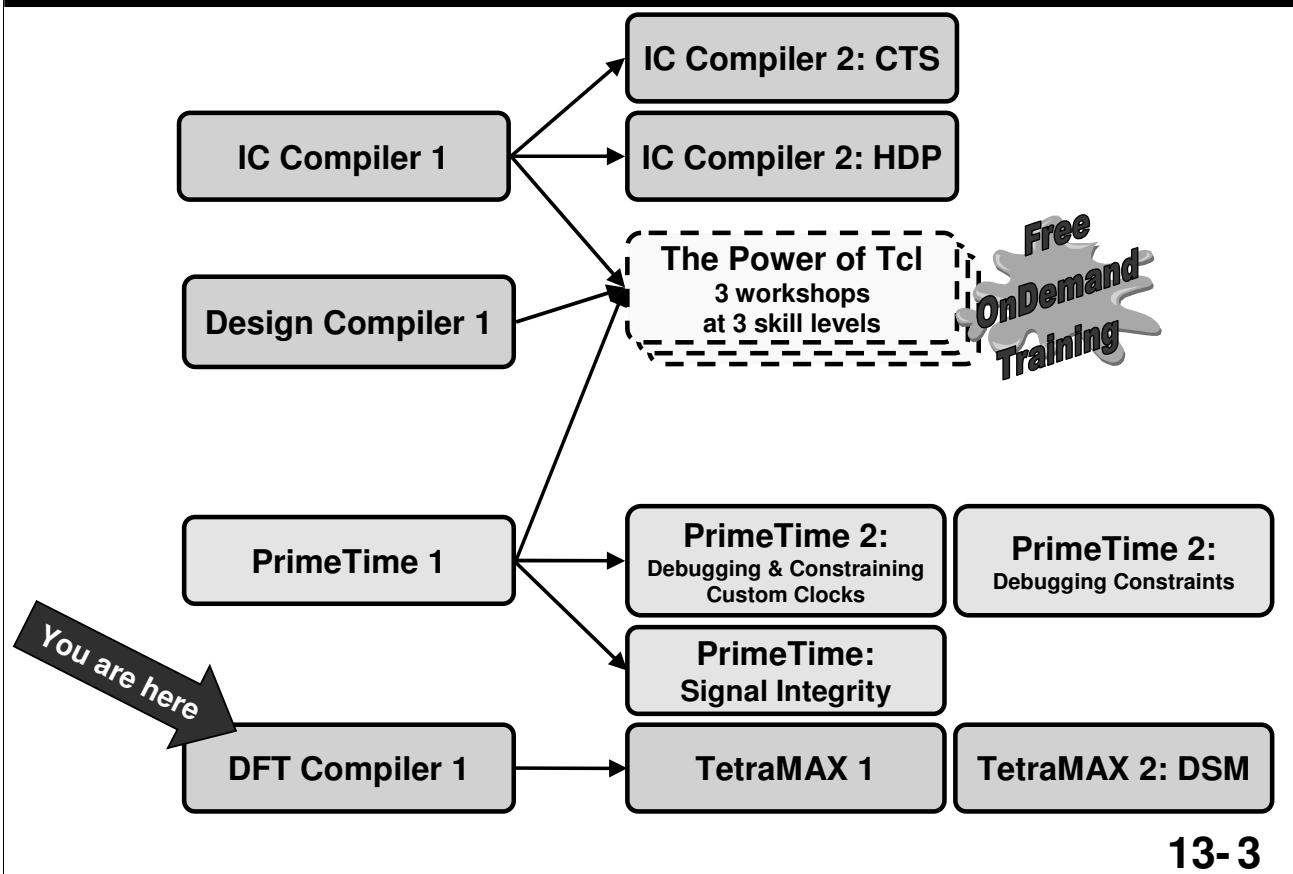
# Workshop Goal



**Use DFT Compiler to check RTL and mapped designs for DFT violations, insert scan chains into very large multi-million gate designs, and export all the required files for downstream tools**

13-2

# Curriculum Flow



The entire Synopsys Customer Education Services course offering can be found at:

<http://training.synopsys.com>

Synopsys Customer Education Services offers workshops in two formats: The “classic” workshops, delivered at one of our centers, and the virtual classes, that are offered conveniently over the web. Both flavors are delivered *live* by expert Synopsys instructors.

In addition, a number of workshops are also offered as OnDemand playback training for FREE! Visit the following link to view the available workshops:

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

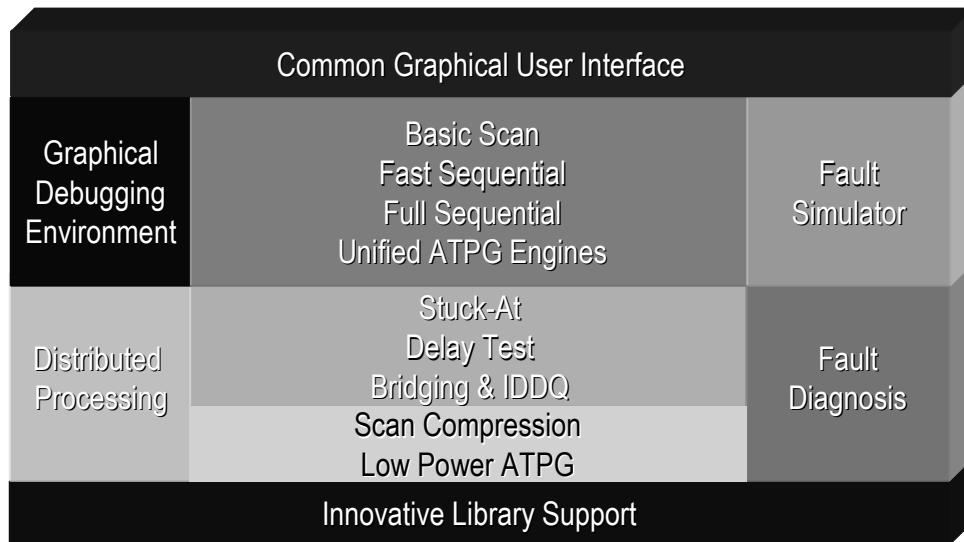
(see under “Tool and Methodology Training”)

# Advanced DFT Compiler Topics

- Some advanced DFT Compiler topics are beyond the scope of this class and will be covered in a future “DFT Compiler 2” class currently under development
- Topics to be covered in DFT Compiler 2 include:
  - On-Chip Clocking (OCC) support in DFT Compiler
  - Low power DFT (Multi-Voltage/Supply, UPF)
  - Advanced DFT MAX options for Delay Testing
  - Physical DFT topics (re-ordering flows in ICC)
- If you would like more information on any of these topics, please consult your local Test AC

13- 4

# What is TetraMAX?



13-5

The TetraMAX ATPG follow-on workshop is two days long.

Intensive labs on debugging of real-world DRC violations.

Tips on library and ATPG modeling issues.

Examples using:

Basic-Scan (D-Algorithm)

Fast-Sequential –D-Algorithm w/2 to 10 capture cycles

Full-Sequential --Sequential Algorithm w/unlimited capture cycles

Note that DFTC (**dft\_drc**) only used the Stuck-At model for testing and only for Basic Scan. If fault coverage is not acceptable in DFTC, TetraMAX may still be able to increase coverage by using Fast-Sequential, Full-Sequential, or Fault Simulation.

# Want More Training?

- There is more training material available to customers on SolvNet in the “Training Center”  
<https://solvnet.synopsys.com/training>
- The Training Center includes:
  - “Product Update Training” modules covering feature updates in recent major releases
  - “On Demand” training modules on various tool and methodology related topics

13- 6

# Helpful SolvNet Articles

- Many useful SolvNet articles were referenced as part of this training. Here's a list of some of them:

Doc ID	Title
020353	TCL Training
900679	NON-END-OF-CYCLE-Measure vs. END-OF-CYCLE-Measure
008685	Understanding the test_simulation_library variable
012388	Using TetraMAX for Interactive Debugging Inside DFT Compiler
015281	Converting Collections into TCL Lists for XG Mode
018658	TCL Script For Generating CTL Test Models
016684	Multimode Scan - some hints and tips on its usage in DFT Compiler & DFT MAX
016862	DFTC in XG mode: TestMode vs Constant
015688	How do I deal with constant value cells in DFT Compiler?
017957	When to use -view spec versus -view existing for set_dft_signal?
020531	Controlling the names of logic inserted by insert_dft
021644	What method should I use to autofix asynchronous resets and sets?
021646	What dft_drc "D rule" violations will prevent scan insertion?
021068	Defining Test Mode signals in DFT-Compiler

13-7

# How to Download Lab Files (1/3)

From the Synopsys home page select “Training & Support”:

The screenshot shows the Synopsys website in a browser window. At the top, there's a navigation bar with links for 'PRODUCTS & SOLUTIONS', 'PROFESSIONAL SERVICES', 'TRAINING & SUPPORT' (which is highlighted in black), 'CORPORATE', and 'PARTNERS'. Below the navigation, a banner reads 'HELPING YOU DESIGN THE CHIP INSIDE' with categories like 'DESIGN IMPLEMENTATION', 'VERIFICATION', 'INTELLECTUAL PROPERTY', 'DFM/TCAD', and 'DESIGN SERVICES'. The main content area is titled 'Training & Support' and includes sections for 'CUSTOMER EDUCATION SERVICES', 'SOLVNET SELF-SERVICE SUPPORT', and 'GLOBAL SUPPORT CENTERS'. A large callout box highlights the 'CUSTOMER EDUCATION SERVICES' link. To the right of the callout, an arrow points from the 'Customer Education Services' link to a detailed description of the service, which includes a 'Learn More >' button.

**Then select Customer Education Services**

**Customer Education Services**  
Synopsys backs its industry-leading products with top-rated service and support available around the clock—and around the world. From Customer Education Services and SolNet online support to the Global Support Centers and Synopsys Professional Services, Synopsys delivers the essential expertise and personal attention required to get the most from your tool investment, help keep your project on schedule, and accelerate your designers' proficiency and productivity with Synopsys technology.

**SolvNet Self-Service Support**  
SolvNet gives you instant access to the information and self-help resources to resolve many support issues. Create a personal profile—including product preferences—to tailor the support information you view and the updates you receive. In addition to documentation, software downloads and a comprehensive, searchable knowledgebase that provides solutions to frequently encountered problems, SolvNet features information on known issues and workarounds and a convenient way to open, update and monitor your support case.

**Global Support Centers**  
Synopsys technical support centers are staffed by experienced design engineers with specialized Synopsys tool knowledge. They are trained to understand, diagnose and resolve problems of all levels of technical complexity. If you can't find a solution on SolvNet, open a support case, and a product specialist will research your issue, follow up with you, and, if necessary, leverage a global team of Synopsys experts, including R&D, to solve your problem. With the ViewConnect secure support environment, you can share your desktop with the Support Center, allowing

13-8

# How to Download Lab Files (2/3)

**Then Select  
Download Labs**

The screenshot shows the Synopsys Customer Education Services page. At the top, there's a navigation bar with links for PRODUCTS & SOLUTIONS, PROFESSIONAL SERVICES, TRAINING & SUPPORT (which is highlighted in black), CORPORATE, and PARTNERS. Below the navigation, there are several tabs: DESIGN IMPLEMENTATION, VERIFICATION, INTELLECTUAL PROPERTY, DFM/TCAD, and DESIGN SERVICES. On the left side, there's a sidebar with a heading 'TRAINING & SUPPORT' and a list of links: OPEN A SUPPORT CASE, SOLVNET, CUSTOMER EDUCATION, GLOBAL SUPPORT CENTERS, SYNOPSYS USERS GROUP, COURSE CATALOGS/REGISTRATION, DOWNLOAD LABS (which is underlined and has a large black arrow pointing to it from the callout box), TRAINING CENTERS, TIPS AND TRICKS, and CONTACT US. The main content area is titled 'Customer Education Services' and contains three sections: 'Private Classes', 'Public Classes', and 'On-Demand Training'. Each section includes a brief description and a thumbnail image. At the bottom of the page, there's a footer with a 'VERIFICATION' section containing links to Formality, HSPICE Essentials, and HSPICE Advanced Topics.

13-9

# How to Download Lab Files (3/3)

Locate Course Title

Workshop Description	Software Version	FTP Product Dir
Astro 1	2005.09	Labs_ASTRO1_2005_09 Download
Astro Rail - Power Integrity Analysis	2004.12	Labs_ASTRO_RAIL_2004_12 Download
Astro: Advanced Clock Tree Synthesis	2004.12	Labs_ASTRO_ACTS_2004_12 Download
DC FPGA Synthesis	2004.06	Labs_DCFPGA_2004_06 Download
Design Compiler 1	2005.09	Labs_DC1_2005_09 Download
Design Compiler 1	2006.06	Labs_DC1_2006_06 Download
Design Compiler Essentials for Place and Route	2005.09_SP3	Labs_DCEPR_2005_09-SP3 Download
<b>DFT Compiler 1</b>	<b>2007.12</b>	<b>Labs_DFTC1_2007.12 Download</b>
Formality	2004.06	Labs_FORMALITY_2004.06 Download
Hercules Beginning Runset	2004.12	Labs_HERCULES_RUNSET_2004.12 Download

Click to Download Labs  
( **ftp> get file.tar.gz** )

A README file will walk you through decompression and untar steps

13-10

The lab files download page can be accessed directly via SolvNet (article # 002471):  
<https://solvnet.synopsys.com/retrieve/002471.html>

# **Course Evaluation**

---

- **Synopsys needs your feedback to improve courses**
- **Please take the time to complete the course evaluation when received by email**

**13-11**

# Thank You!



13-12