Scan and ATPG Process Guide

Software Version V8.6_4

Copyright © Mentor Graphics Corporation 1999. All rights reserved.
This document contains information that is proprietary to Mentor Graphics Corporation and may be duplicated in whole or in part by the original recipient for internal business purposes only, provided that this entire notice appears in all copies. In accepting this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use of this information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

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

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

RESTRICTED RIGHTS LEGEND 03/97

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

A complete list of trademark names appears in a separate “Trademark Information” document.

This is an unpublished work of Mentor Graphics Corporation.

TABLE OF CONTENTS

About This Manual xvii

Related Publications xviii
Mentor Graphics Documentation xviii
General DFT Documentation xx
BIST Documentation xx
IDDQ Documentation xxi
Command Line Syntax Conventions xxii
Acronyms Used in This Manual xxiii

Chapter 1
Overview 1-1

What is Design-for-Test? 1-1
DFT Strategies 1-1
Top-Down Design Flow with DFT 1-2
DFT Design Tasks and Products 1-5
User Interface Overview 1-9
Command Line Window 1-10
Control Panel Window 1-14
Getting Help 1-15
Running Batch Mode Using Dofiles 1-18
Generating a Log File 1-19
Running UNIX Commands 1-20
Interrupting the Session 1-20
Exiting the Session 1-20
DFTAdvisor User Interface 1-21
FastScan User Interface 1-23
FlexTest User Interface 1-25

Chapter 2
Understanding Scan and ATPG Basics 2-1

Understanding Scan Design 2-2
Internal Scan Circuitry 2-2
Scan Design Overview 2-2

TABLE OF CONTENTS [continued]

Understanding Full Scan 2-4
Understanding Partial Scan 2-5
Choosing Between Full or Partial Scan 2-7
Understanding Partition Scan 2-8
Understanding Test Points 2-10
Test Structure Insertion with DFTAdvisor 2-12
Understanding ATPG 2-14
The ATPG Process 2-14
Mentor Graphics ATPG Applications 2-16
Full-Scan and Scan Sequential ATPG with FastScan 2-16
Non- to Full-Scan ATPG with FlexTest 2-17
Understanding Test Types and Fault Models 2-18
Test Types 2-19
Fault Modeling 2-22
Fault Detection 2-31
Fault Classes 2-32
Testability Calculations 2-40

Chapter 3
Understanding Common Tool Terminology and Concepts 3-1

Scan Terminology 3-2
Scan Cells 3-2
Scan Chains 3-6
Scan Groups 3-7
Scan Clocks 3-8
Scan Architectures 3-8
Mux-DFF 3-9
Clocked-Scan 3-9
LSSD 3-10
Test Procedure Files 3-11
Test Procedure File Rules 3-11
Test Procedure Statements 3-12
The Procedures 3-15
Scan Chain Operation Checking 3-28

TABLE OF CONTENTS [continued]

Model Flattening 3-29
Understanding Design Object Naming 3-30
The Flattening Process 3-30
Simulation Primitives of the Flattened Model 3-32
Learning Analysis 3-36
Equivalence Relationships 3-36
Logic Behavior 3-37
Implied Relationships 3-38
Forbidden Relationships 3-38
Dominance Relationships 3-39
ATPG Design Rules Checking 3-40
General Rules Checking 3-40
Procedure Rules Checking 3-40
Bus Mutual Exclusivity Analysis 3-41
Scan Chain Tracing 3-42
Shadow Latch Identification 3-43
Data Rules Checking 3-44
Transparent Latch Identification 3-44
Clock Rules Checking 3-44
RAM Rules Checking 3-44
Bus Keeper Analysis 3-45
Extra Rules Checking 3-46
Scannability Rules Checking 3-46
BIST Rules Checking 3-46
Constrained/Forbidden/Block Value Calculations 3-46

Chapter 4
Understanding Testability Issues 4-1

Synchronous Circuitry 4-2
Synchronous Design Techniques 4-2
Asynchronous Circuitry 4-3
Scannability Checking 4-3
Scannability Checking of Latches 4-4
Support for Special Testability Cases 4-5
Feedback Loops 4-5

TABLE OF CONTENTS [continued]

Structural Combinational Loops and Loop-Cutting Methods 4-5
Structural Sequential Loops and Handling 4-14
Redundant Logic 4-16
Asynchronous Sets and Resets 4-16
Gated Clocks 4-17
Tri-State Devices 4-18
Non-Scan Cell Handling 4-19
Clock Dividers 4-26
Pulse Generators 4-27
JTAG-Based Circuits 4-28
Built-In Self-Test (FastScan Only) 4-28
Testing with RAM and ROM 4-34

Chapter 5
Inserting Internal Scan
and Test Circuitry 5-1

Understanding DFTAdvisor 5-2
The DFTAdvisor Process Flow 5-3
DFTAdvisor Inputs and Outputs 5-5
Test Structures Supported by DFTAdvisor 5-7
Invoking DFTAdvisor 5-10
Preparing for Test Structure Insertion 5-11
Selecting the Scan Methodology 5-11
Defining Scan Cell and Scan Output Mapping 5-11
Enabling Test Logic Insertion 5-12
Specifying Clock Signals 5-15
Specifying Existing Scan Information 5-16
Deleting Existing Scan Circuitry 5-18
Handling Existing Boundary Scan Circuitry 5-18
Changing the System Mode (Running Rules Checking) 5-18
Identifying Test Structures 5-20
Selecting the Type of Test Structure 5-20
Setting Up for Full Scan Identification 5-21
Setting Up for Clocked Sequential Identification 5-21
Setting Up for Sequential Transparent Identification 5-22

TABLE OF CONTENTS [continued]

Setting Up for Partition Scan Identification 5-22
Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification 5-25
Setting Up for Test Point Identification 5-28
Manually Including and Excluding Cells for Scan 5-31
Reporting Scannability Information 5-34
Running the Identification Process 5-36
Reporting Identification Information 5-36
Inserting Test Structures 5-37
Setting Up for Internal Scan Insertion 5-37
Setting Up for Test Point Insertion 5-40
Buffering Test Pins 5-41
Running the Insertion Process 5-41
Saving the New Design and ATPG Setup 5-45
Writing the Netlist 5-45
Writing the Test Procedure File and Dofile for ATPG 5-46
Running Rules Checking on the New Design 5-46
Exiting DFTAdvisor 5-47
Inserting Scan Block-by-Block 5-47
Verilog and EDIF Flow Example 5-48

Chapter 6
Generating Test Patterns 6-1

Understanding FastScan and FlexTest 6-2
FastScan and FlexTest Basic Tool Flow 6-3
FastScan and FlexTest Inputs and Outputs 6-6
Understanding FastScan's ATPG Method 6-8
Understanding FlexTest's ATPG Method 6-14
Performing Basic Operations 6-19
Invoking the Applications 6-19
Setting the System Mode 6-23
Setting Up Design and Tool Behavior 6-24
Setting Up the Circuit Behavior 6-24
Setting Up Tool Behavior 6-28
Setting the Circuit Timing (FlexTest Only) 6-34

TABLE OF CONTENTS [continued]

Defining the Scan Data 6-38
Setting Up for BIST (FastScan Only) 6-42
Checking Rules and Debugging Rules Violations 6-44
Running Good/Fault Simulation on Existing Patterns 6-45
Fault Simulation 6-45
Good Machine Simulation 6-50
Running Random/BIST Pattern Simulation (FastScan) 6-53
Random Pattern Simulation 6-53
BIST Pattern Simulation 6-54
Obtaining Optimum BIST Coverage 6-56
Example ATPG Run on a BIST Circuit 6-59
Setting Up the Fault Information for ATPG 6-62
Changing to the ATPG System Mode 6-62
Setting the Fault Type 6-63
Creating the Faults List 6-63
Adding Faults to an Existing List 6-64
Loading Faults from an External List 6-64
Writing Faults to an External File 6-65
ATPG Library Verification (FlexTest Only) 6-65
Setting Self-Initialized Test Sequences (FlexTest Only) 6-66
Setting the Fault Sampling Percentage (FlexTest Only) 6-66
Setting the Fault Mode 6-67
Setting the Hypertrophic Limit (FlexTest Only) 6-67
Setting the Possible-Detect Credit 6-68
Running ATPG 6-69
Setting Up for ATPG 6-70
Performing a Default ATPG Run 6-76
Compressing Patterns 6-76
Automatic ATPG Compression 6-78
Approaches for Improving ATPG Efficiency 6-78
Saving the Test Patterns 6-84
Creating an IDDQ Test Set 6-85
Creating a Selective IDDQ Test Set 6-86
Generating a Supplemental IDDQ Test Set 6-89
Specifying IDDQ Checks and Constraints 6-90

TABLE OF CONTENTS [continued]

Creating a Path Delay Test Set (FastScan) 6-92
Path Delay Fault Detection 6-92
The Path Definition File 6-96
Path Definition Checking 6-98
Basic Path Delay Test Procedure 6-99
Path Delay Testing Limitations 6-100
Creating a Transition Test Set 6-101
Transition Fault Detection 6-101
Basic Transition Test Procedure 6-103
Generating Patterns for a Boundary Scan Circuit 6-103
Dofile and Explanation 6-104
TAP Controller State Machine 6-105
Test Procedure File and Explanation 6-106
Creating Instruction-Based Test Sets (FlexTest) 6-111
Instruction-Based Fault Detection 6-111
Instruction File Format 6-112
Using FastScan's Macrotest Capability 6-115
When to Use Macrotest 6-115
Defining the Macro Boundary 6-116
Defining Test Values 6-117
Recommendations for Using Macrotest 6-120
A Macrotest Example 6-121
Verifying Design and Test Pattern Timing 6-125
Simulating the Design with Timing 6-125
Checking for Clock-Skew Problems with Mux-DFF Designs 6-129

Chapter 7
Test Pattern Formatting and Timing 7-1

Test Pattern Timing Overview 7-2
Timing Terminology 7-2
Defining Scan-Related Event Timing 7-3
Converting Test Procedures to Test Cycles 7-4
Test Procedure Timing Examples 7-5
Test Procedure Timing Issues 7-12

TABLE OF CONTENTS [continued]

Defining Non-Scan Related Event Timing 7-13
FastScan Non-Scan Event Timing 7-13
FlexTest Non-Scan Event Timing 7-18
Global Timing Issues in the Timing File 7-19
Performing Timing Checks for Tester Formats 7-21
Tester Format Restrictions for FastScan 7-22
Tester Format Restrictions for FlexTest 7-23
Saving the Patterns 7-24
Features of the Formatter 7-25
Pattern Formatting Issues 7-25
Saving Patterns in Basic Test Data Formats 7-28
Saving in ASIC Vendor Data Formats 7-40

Chapter 8
Running Diagnostics 8-1

Understanding FastScan Diagnostic Capabilities 8-1
Understanding Stuck Faults and Defects 8-3
Creating the Failure File 8-4
Failure File Format 8-5
Performing a Diagnosis 8-6

Index

LIST OF FIGURES

Figure 1. DFT Documentation Roadmap xviii
Figure 1-1. Top-Down Design Flow Tasks and Products 1-3
Figure 1-2. ASIC/IC Design-for-Test Tasks 1-6
Figure 1-3. Common Elements of the DFT Graphical User Interfaces 1-9
Figure 1-4. DFTAdvisor Control Panel Window 1-22
Figure 1-5. FastScan Control Panel Window 1-24
Figure 1-6. FlexTest Control Panel Window 1-26
Figure 2-1. DFT Concepts 2-1
Figure 2-2. Design Before and After Adding Scan 2-3
Figure 2-3. Full Scan Representation 2-4
Figure 2-4. Partial Scan Representation 2-6
Figure 2-5. Full, Partial, and Non-Scan Trade-offs 2-7
Figure 2-6. Example of Partitioned Design 2-9
Figure 2-7. Partition Scan Circuitry Added to Partition A 2-10
Figure 2-8. Uncontrollable and Unobservable Circuitry 2-11
Figure 2-9. Testability Benefits from Test Point Circuitry 2-11
Figure 2-10. Manufacturing Defect Space for Design “X 2-19
Figure 2-11. Internal Faulting Example 2-23
Figure 2-12. Single Stuck-At Faults for AND Gate 2-24
Figure 2-13. IDDQ Fault Testing 2-27
Figure 2-14. Transition Fault Detection Process 2-28
Figure 2-15. Fault Detection Process 2-31
Figure 2-16. Path Sensitization Example 2-32
Figure 2-17. Example of “Unused” Fault in Circuitry 2-33
Figure 2-18. Example of “Tied” Fault in Circuitry 2-34
Figure 2-19. Example of “Blocked” Fault in Circuitry 2-34
Figure 2-20. Example of “Redundant” Fault in Circuitry 2-35
Figure 2-21. Fault Class Hierarchy 2-39
Figure 3-1. Common Tool Concepts 3-1
Figure 3-2. Generic Scan Cell 3-2
Figure 3-3. Generic Mux-DFF Scan Cell Implementation 3-3
Figure 3-4. LSSD Master/Slave Element Example 3-4
Figure 3-5. Mux-DFF/Shadow Element Example 3-5
Figure 3-6. Mux-DFF/Copy Element Example 3-5
Figure 3-7. Generic Scan Chain 3-6

LIST OF FIGURES [continued]

Figure 3-8. Generic Scan Group 3-7
Figure 3-9. Scan Clocks Example 3-8
Figure 3-10. Mux-DFF Replacement 3-9
Figure 3-11. Clocked-Scan Replacement 3-10
Figure 3-12. LSSD Replacement 3-10
Figure 3-13. Shift Procedure 3-16
Figure 3-14. Timing Diagram for Shift Procedure 3-18
Figure 3-15. Load_Unload Procedure 3-19
Figure 3-16. Timing Diagram for Load_Unload Procedure 3-20
Figure 3-17. Shadow_Control Procedure 3-22
Figure 3-18. Master_Observe Procedure 3-22
Figure 3-19. Shadow_Observe Procedure 3-23
Figure 3-20. Sequential Transparent Circuitry Example 3-24
Figure 3-21. Skew_Load Procedure 3-27
Figure 3-22. Skew_load applied within Pattern 3-27
Figure 3-23. Design Before Flattening 3-31
Figure 3-24. Design After Flattening 3-31
Figure 3-25. 2x1 MUX Example 3-33
Figure 3-26. LA, DFF Example 3-33
Figure 3-27. TSD, TSH Example 3-34
Figure 3-28. PBUS, SWBUS Example 3-35
Figure 3-29. Equivalence Relationship Example 3-37
Figure 3-30. Example of Learned Logic Behavior 3-37
Figure 3-31. Example of Implied Relationship Learning 3-38
Figure 3-32. Forbidden Relationship Example 3-39
Figure 3-33. Dominance Relationship Example 3-39
Figure 3-34. Bus Contention Example 3-41
Figure 3-35. Bus Contention Analysis 3-42
Figure 3-36. Simulation Model with Bus Keeper 3-45
Figure 3-37. Constrained Values in Circuitry 3-47
Figure 3-38. Forbidden Values in Circuitry 3-47
Figure 3-39. Blocked Values in Circuitry 3-47
Figure 4-1. Testability Issues 4-1
Figure 4-2. Structural Combinational Loop Example 4-5
Figure 4-3. Loop Naturally-Blocked by Constant Value 4-6

LIST OF FIGURES [continued]

Figure 4-4. Cutting Constant Value Loops 4-7
Figure 4-5. Cutting Single Multiple-Fanout Loops 4-7
Figure 4-6. Loop Candidate for Duplication 4-8
Figure 4-7. TIE-X Insertion Simulation Pessimism 4-8
Figure 4-8. Cutting Loops by Gate Duplication 4-9
Figure 4-9. Cutting Coupling Loops 4-10
Figure 4-10. Delay Element Added to Feedback Loop 4-12
Figure 4-11. Sequential Feedback Loop 4-14
Figure 4-12. Fake Sequential Loop 4-15
Figure 4-13. Test Logic Added to Control Asynchronous Reset 4-17
Figure 4-14. Test Logic Added to Control Gated Clock 4-18
Figure 4-15. Tri-state Bus Contention 4-19
Figure 4-16. Requirement for Combinationally Transparent Latches 4-20
Figure 4-17. Example of Sequential Transparency 4-22
Figure 4-18. Clocked Sequential Scan Pattern Events 4-23
Figure 4-19. Clock Divider 4-26
Figure 4-20. Example Pulse Generator Circuitry 4-27
Figure 4-21. LFSR Configuration 4-29
Figure 4-22. Simple BIST Configuration 4-30
Figure 4-23. Design with Embedded RAM 4-35
Figure 4-24. RAM Sequential Example 4-38
Figure 5-1. Internal Scan Insertion Procedure 5-1
Figure 5-2. Basic Scan Insertion Flow with DFTAdvisor 5-3
Figure 5-3. The Inputs and Outputs of DFTAdvisor 5-5
Figure 5-4. DFTAdvisor Supported Test Structures 5-7
Figure 5-5. Test Logic Insertion 5-13
Figure 5-6. Example Report from Report Dft Check Command 5-35
Figure 5-7. Lockup Latch Insertion 5-44
Figure 5-8. Hierarchical Design Prior to Scan 5-47
Figure 5-9. Final Scan-Inserted Design 5-50
Figure 6-1. Test Generation Procedure 6-1
Figure 6-2. Overview of FastScan/FlexTest Usage 6-3
Figure 6-3. FastScan/FlexTest Inputs and Outputs 6-6
Figure 6-4. Clock-PO Circuitry 6-10
Figure 6-5. Cycle-Based Circuit with Single Phase Clock 6-15

LIST OF FIGURES [continued]

Figure 6-6. Cycle-Based Circuit with Two Phase Clock 6-16
Figure 6-7. Example Test Cycle 6-17
Figure 6-8. Data Capture Handling Example 6-32
Figure 6-9. Block Diagram of BIST Example Circuit.................................... 6-60
Figure 6-10. Efficient ATPG Flow 6-69
Figure 6-11. Circuitry with Natural “Select” Functionality 6-72
Figure 6-12. Single Cycle Multiple Events 6-74
Figure 6-13. Path Delay Launch and Capture Events.................................... 6-92
Figure 6-14. Robust Detection Example 6-94
Figure 6-15. Transition Detection Example 6-95
Figure 6-16. Example of Ambiguous Path Definition.................................... 6-98
Figure 6-17. Example of Ambiguous Path Edges 6-99
Figure 6-18. Transition Launch and Capture Events.................................... 6-101
Figure 6-19. Type One Pattern Set 6-101
Figure 6-20. Type Two Pattern Set.. 6-102
Figure 6-21. State Diagram of TAP Controller Circuitry 6-105
Figure 6-22. Example Instruction File.. 6-114
Figure 6-23. Clock-Skew Example.. 6-129
Figure 7-1. Defining Timing Process Flow 7-1
Figure 7-2. Test Cycle Timing for Test_Setup Procedure................................. 7-5
Figure 7-3. Timing for Non-Scan Events 7-19
Figure 8-1. Diagnostics Process Flow 8-6

LIST OF TABLES

Table 1-1. Session Transcript Popup Menu Items 1-11
Table 1-2. Command Transcript Popup Menu Items 1-12
Table 2-1. Test Type/Fault Model Relationship 2-22
Table 4-1. FastScan BIST Commands 4-32
Table 4-2. FastScan and FlexTest RAM/ROM Commands 4-41
Table 5-1. Test Type Interactions 5-9
Table 6-1. ATPG Constraint Conditions 6-72
Table 6-2. Pin Value Requirements for ADD Instruction 6-112

LIST OF TABLES [continued]

About This Manual

The *Scan and ATPG Process Guide* gives an overview of ASIC/IC Design-for-Test (DFT) strategies and shows the use of Mentor Graphics ASIC/IC DFT products as part of typical DFT design processes. This document discusses the following DFT products: DFTAdvisor, FastScan, and FlexTest.

- Chapter 1 discusses the basic concepts behind DFT, establishes the framework in which Mentor Graphic ASIC DFT products are used, and briefly describes each of these products.
- Chapter 2 gives conceptual information necessary for determining what test strategy would work best for you.
- Chapter 3 provides tool methodology information, including common terminology and concepts used by the tools.
- Chapter 4 outlines characteristics of testable designs and explains how to handle special design situations that can affect testability.
- Chapters 5 through 8 discuss the common tasks involved at each step within a typical process flow using Mentor Graphics DFT tools.

The DFT applications use Adobe Acrobat Exchange as their online documentation and help viewer. Online help requires installing the Mentor Graphics-supplied Acrobat Exchange program with Mentor Graphics-specific plugins and also requires setting an environment variable. For more information, refer to the section, “Setting Up Online Manuals and Help” in *Using Mentor Graphics Documentation with Acrobat Exchange*.

Related Publications

This section gives references to both Mentor Graphics product documentation and industry DFT documentation.

Mentor Graphics Documentation

Figure 1 shows the Mentor Graphics DFT manuals and their relationship to each other and is followed by a list of descriptions for these documents.

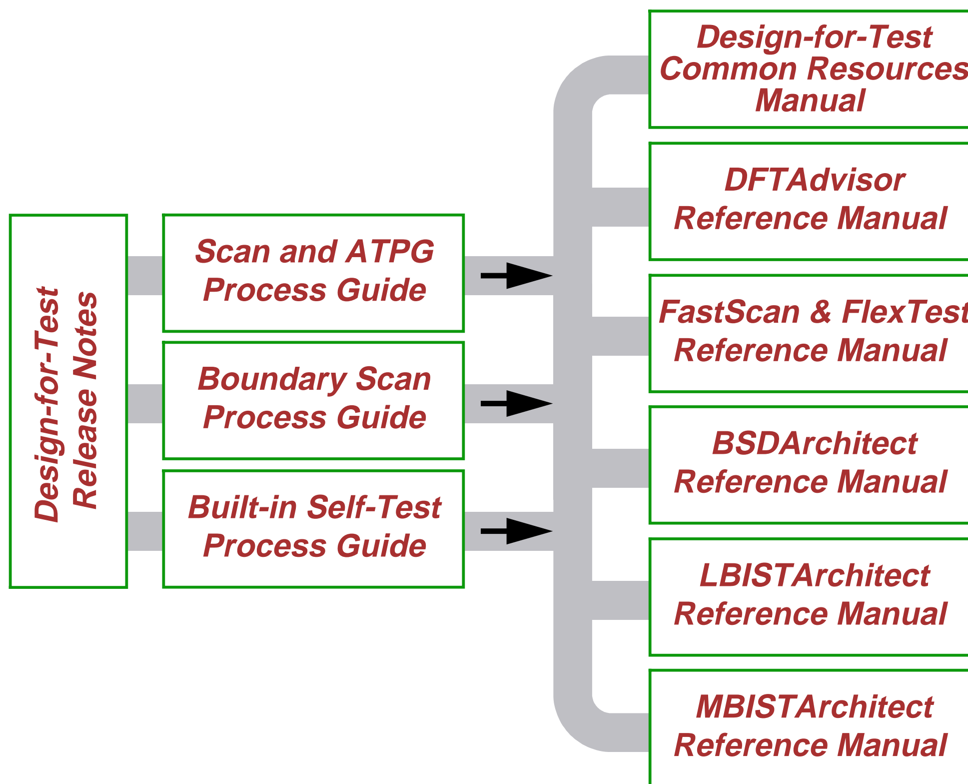


Figure 1. DFT Documentation Roadmap

Boundary Scan Process Guide — provides process, concept, and procedure information for the boundary scan product, BSDArchitect. It also includes information on how to integrate boundary scan with the other DFT technologies.

BSDArchitect Reference Manual — provides reference information for BSDArchitect, the boundary scan product.

Built-in Self-Test Process Guide — provides process, concept, and procedure information for using MBISTArchitect, LBISTArchitect, and other Mentor Graphics tools in the context of your BIST design process.

Design-for-Test Common Resources Manual — contains information common to many of the DFT tools: design rule checks (DRC), DFTInsight (the schematic viewer), library creation, VHDL support, Verilog support, Spice support, and test procedure file format.

Design-for-Test Release Notes — provides release information that reflects changes to the DFT products for the software version release.

DFTAdvisor Reference Manual — provides reference information for DFTAdvisor (internal scan insertion) and DFTInsight (schematic viewer) products.

FastScan and FlexTest Reference Manual — provides reference information for FastScan (full-scan ATPG), FlexTest (non- to partial-scan ATPG), and DFTInsight (schematic viewer) products.

LBISTArchitect Reference Manual — provides reference information for LBISTArchitect, the logic built-in self-test product.

MBISTArchitect Reference Manual — provides reference information for MBISTArchitect, the memory built-in self-test product.

Scan and ATPG Process Guide — provides process, concept, and procedure information for using DFTAdvisor, FastScan, and FlexTest in the context of your DFT design process.

Using Mentor Graphics Documentation with Acrobat Exchange — describes how to set up and use the Mentor Graphics-supplied Acrobat Exchange with enhancement plugins for online viewing of Mentor Graphics PDF-based documentation and help. The manual contains procedures for using Mentor Graphics documentation, including setting up online manuals and help, opening documents, and using full-text searches. Also included are tips on using Exchange.

General DFT Documentation

The *Scan and ATPG Process Guide* gives an overview of a variety of DFT concepts and issues. However, for more detailed information on any of the topics presented in that document, refer to the following:

- Abramovici, Miron, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1990.
- Agarwal, V. D. and S. C. Seth. *Test Generation for VLSI Chips.* Computer Society Press, 1988.
- Fujiwara, Hideo. *Logic Testing and Design for Testability.* Cambridge: The MIT Press, 1985.
- Huber, John P. and Mark W. Rosneck. *Successful ASIC Design the First Time Through.* New York: Van Nostrand Reinhold, 1991.
- IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*. New York: IEEE, 1990.
- McCluskey, Edward J. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Englewood Cliffs: Prentice-Hall, 1986.
- Rajsuman, Rochit, *Digital Hardware Testing: Transistor-Level Fault Modeling and Testing*. Boston: Artech House, 1992.

BIST Documentation

- Decker, Rob and Frans Beenker, (Philips Research Laboratories), “Fault Modeling and Test Algorithm Development for Static Random Access Memories”, Proceedings ITC 1988, pp. 343-351.
- Rajski, Janusz and Jerzy Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems*. New Jersey: Prentice Hall PTR, 1998.
- van de Goor, A.J. *Testing Semiconductor Memories*. John Wiley & Sons, 1991.

IDDQ Documentation

- Aitken, R. C. “Fault Location with current monitoring,” Proceedings ITC-1991, pp. 623-632.
- Chen, Chun-Hung and J. Abraham, “High Quality tests for switch level circuits using current and logic test generation algorithms,” Proceedings ITC-1991, pp. 615-622.
- Ferguson, F. Joel and Tracy Larrabee, “Test Pattern Generation for Realistic Bridge Faults in CMOS ICs,” Proceedings ITC 1991, pp. 492-499.
- Mao, W., R.K. Gulati, D.K. Goel, and M. D. Ciletti, “QUIETEST: A quiescent current testing methodology for detecting leakage faults,” Proceedings ICCAD-90, pp. 280-283.
- Marston, Gregory “Automating IDDQ Test Generation,” Private Communication, November 1993.
- Maxwell, Peter and Robert Aitken, “IDDQ testing as a component of a test suite: The need for several fault coverage metrics,” Journal of Electronic Testing, theory and applications, 3, pp 305-116 (1992).
- Soden, J. M., R. K. Treece, M.R. Taylor, and C.F. Hawkins, “CMOS IC Stuck-open Faults Electrical Effects and Design Considerations,” Proceedings International Test Conference 1989, pp. 423-430.

Command Line Syntax Conventions

The notational elements used in this manual for command line syntax are as follows:

Bold	A bolded font indicates a required argument.
[ ]	Square brackets enclose optional arguments (in command line syntax only). Do not enter the brackets.
UPPercase	Required command letters are in uppercase; you may omit lowercase letters when entering commands or literal arguments and you need not use uppercase. Command names and options are case insensitive. Commands usually follow the 3-2-1 rule: the first three letters of the first word, the first two letters of the second word, and the first letter of the third, fourth, etc. words.
Italic	An italic font indicates a user-supplied argument.
___	An underlined item indicates either the default argument or the default value of an argument.
{ }	Braces enclose arguments to show grouping. Do not enter the braces.
|	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
…	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis in commands.

You should enter literal text (that which is not in italics) exactly as shown.

Acronyms Used in This Manual

Below is an alphabetical listing of the acronyms used in this manual:

ASIC —Application Specific Integrated Circuit

ATE —Automatic Test Equipment

ATPG — Automatic Test Pattern Generation

AVI — ASIC Vector Interfaces

BIST — Built-In Self Test

BSDL — Boundary Scan Design Language

CUT — Circuit Under Test

DFT — Design-for-Test

DRC — Design Rules Checking

DUT — Device Under Test

GUI — Graphical User Interface

HDL — Hardware Description Language

JTAG — Joint Test Action Group

LFSR — Linear Feedback Shift Register

MCM — Multi-Chip Module

MISR — Multiple Input Signature Register

PRPG — Pseudo-Random Pattern Generator

SCOAP — Sandia Controllability Observability Analysis Program

SFP — Single Fault Propagation

TAP — Test Access Port

TCK — Test Clock

TDI — Test Data Input

TDO — Test Data Output

TMS — Test Mode Select

TRST — Test Reset

VHDL — VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

WDB — Waveform DataBase

Chapter 1
Overview

What is Design-for-Test?

Testability is a design attribute that measures how easy it is to create a program to comprehensively test a manufactured design's quality. Traditionally, design and test processes were kept separate, with test considered only at the end of the design cycle. But in contemporary design flows, test merges with design much earlier in the process, creating what is called a *design-for-test (DFT)* process flow. Testable circuitry is both *controllable* and *observable*. In a testable design; setting specific values on the primary inputs results in values on the primary outputs which indicate whether or not the internal circuitry works properly. To ensure maximum design testability, designers must employ special DFT techniques at specific stages in the development process.

DFT Strategies

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

Ad Hoc DFT

Ad hoc DFT implies using good design practices to enhance a design's testability, without making major changes to the design style. Some ad hoc techniques include:

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

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

Structured DFT

Structured DFT provides a more systematic and automatic approach to enhancing design testability. Structured DFT's goal is to increase the controllability and observability of a circuit. Various methods exist for accomplishing this. The most common is the *scan design* technique, which modifies the internal sequential circuitry of the design. You can also use the Built-in Self-Test (BIST) method, which inserts a device's testing function within the device itself. Another method is *boundary scan*, which increases board testability by adding circuitry to a chip. Chapter 2, "Understanding Scan and ATPG Basics," describes these methods in detail.

Top-Down Design Flow with DFT

Figure 1-1 shows the basic steps and the Mentor Graphics tools you would use during a typical ASIC top-down design flow.

This document discusses those steps shown in grey; it also mentions certain aspects of other design steps, where applicable. This flow is just a general description of a top-down design process flow using a structured DFT strategy. The next section, "DFT Design Tasks and Products," gives a more detailed breakdown of the individual DFT tasks involved.

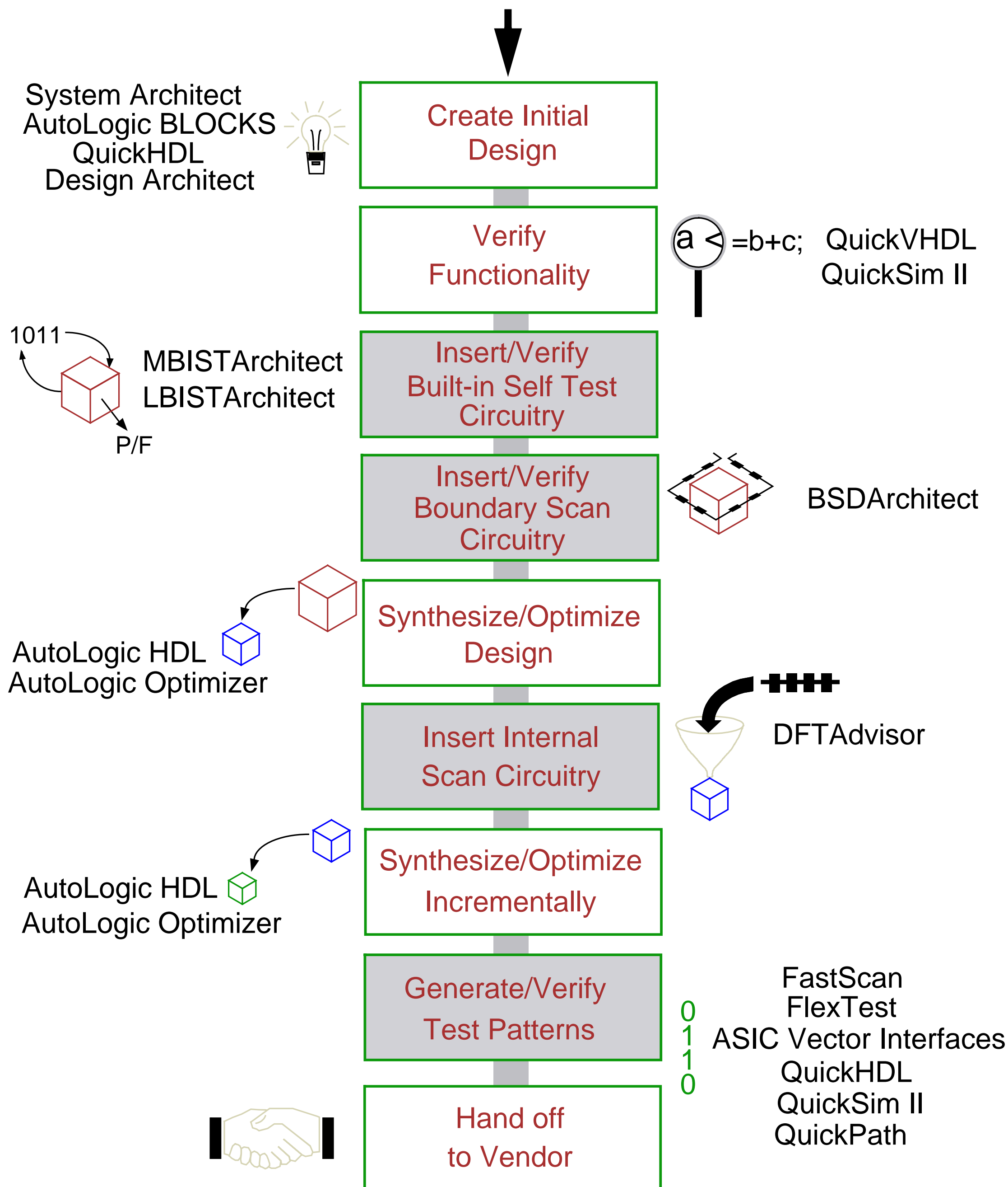


Figure 1-1. Top-Down Design Flow Tasks and Products

As Figure 1-1 shows, the first task in any design flow is creating the initial RTL-level design, through whatever means you choose. In the Mentor Graphics environment, you may choose to create a very-high-level design using System Architect (or AutoLogic BLOCKS), a high-level VHDL or Verilog description

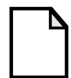
using QuickHDL, or a schematic using Design Architect. You then verify the design’s functionality by performing a functional simulation, using either QuickSim II, QuickHDL, or another vendor's VHDL simulator.

If your design’s format is in VHDL or Verilog format and it contains memory models, at this point you can add built-in self-test (BIST) circuitry. MBISTArchitect creates and inserts RTL-level customized internal testing structures for design memories. Additionally, if your design’s format is in VHDL, you can use LBISTArchitect to synthesizes BIST structures into its random logic design blocks.

Also at the RTL-level, you can insert and verify boundary scan circuitry using BSDArchitect (BSDA). Then you can synthesize and optimize the design using either AutoLogic II or another vendor's synthesis tool.

At this point in the flow you are ready to insert internal scan circuitry into your design using DFTAdvisor. You then perform a timing optimization on the design because you added scan circuitry. Once you are sure the design is functioning as desired, you can generate test patterns. You can use FastScan or FlexTest (depending on your scan strategy) and ASIC Vector Interfaces to generate a test pattern set in the appropriate format.

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


Note

It is important for you to check with your vendor early on in your design process for specific requirements and restrictions that may affect your DFT strategies. For example, the vendor's test equipment may only be able to handle single scan chains (see page 2-2), have memory limitations, or have special timing requirements that affect the way you generate scan circuitry and test patterns.

DFT Design Tasks and Products

Figure 1-2 gives a sequential breakdown of the understanding you should have of DFT, all the major ASIC/IC DFT tasks, and the associated Mentor Graphics DFT tools used for each task. Be aware that the test synthesis and ATPG design flow shown is not necessarily a Mentor Graphics flow, as Mentor Graphics DFT tools *do* work within other EDA vendor's design flows.

The following list briefly describes each of the tasks presented in Figure 1-2.

1. **Understand DFT Basics** —Before you can make intelligent decisions regarding your test strategy, you should have a basic understanding of DFT. Chapter 2, "Understanding Scan and ATPG Basics," prepares you to make decisions about test strategies for your design by presenting information about full scan, partial scan, boundary scan, partition scan, and the variety of options available to you.

2. **Understand Tool Concepts** — The Mentor Graphics DFT tools share some common functionality, as well as terminology and tool concepts. To effectively utilize these tools in your design flow, you should have a basic understanding of what they do and how they operate. Chapter 3, "Understanding Common Tool Terminology and Concepts," discusses this information.

3. **Understand Testability Issues** — Some design features can enhance a design's testability, while other features can hinder it. Chapter 4, "Understanding Testability Issues," discusses synchronous versus asynchronous design practices, and outlines a number of individual situations that require special consideration with regard to design testability.

4. **Insert/Verify Memory BIST Circuitry** — MBISTArchitect is a Mentor Graphics RTL-level tool you use to insert built-in self test (BIST) structures for memory devices. MBISTArchitect lets you specify the testing architecture and algorithms you wish to use, and creates and connects the appropriate BIST models to your VHDL or Verilog memory models. The *Build-in Self-Test Process Guide* discusses how to prepare for, insert, and verify memory BIST circuitry using MBISTArchitect.

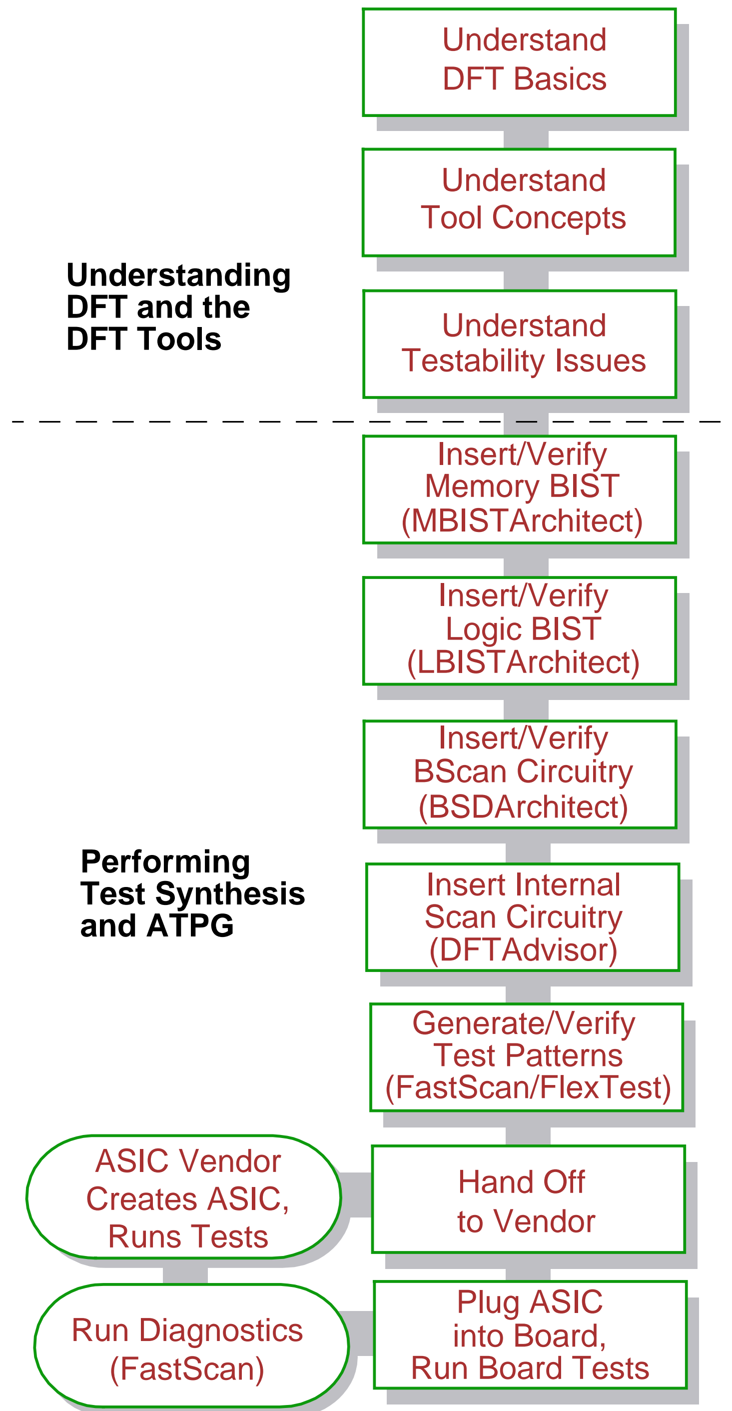


Figure 1-2. ASIC/IC Design-for-Test Tasks

5. **Insert/Verify Logic BIST Circuitry** — LBISTArchitect is a Mentor Graphics RTL-level tool you use to insert built-in self-test (BIST) structures in VHDL or Verilog format. LBISTArchitect lets you specify the testing architecture and algorithms you wish to use, and creates and connects the appropriate BIST models to your HDL models. The *Build-in Self-Test Process Guide* discusses how to prepare for, insert, and verify logic BIST circuitry using LBISTArchitect.

6. **Insert/Verify Boundary Scan Circuitry** —BSDArchitect is a Mentor Graphics IEEE 1149.1 compliant boundary scan insertion tool. BSDA lets you specify the boundary scan architecture you wish to use and inserts it into your RTL-level design. It generates VHDL, Verilog, and BSDL models with IEEE 1149.1 compliant boundary scan circuitry and an HDL test bench for verifying those models. The *Boundary Scan Process Guide* discusses how to prepare for, insert, and verify boundary scan circuitry using BSDA.

7. **Insert Internal Scan Circuitry** — Before you add internal scan or test circuitry to your design, you should analyze your design to ensure that it does not contain problems that may impact test coverage. Identifying and correcting these problems early in the DFT process can minimize design iterations downstream. DFTAdvisor is the Mentor Graphics testability analysis and test synthesis tool. DFTAdvisor can analyze, identify, and help you correct design testability problems early on in the design process. Chapter 5, "Inserting Internal Scan and Test Circuitry," introduces you to DFTAdvisor and discusses preparations and procedures for adding scan circuitry to your design.

8. **Generate/Verify Test Patterns** — FastScan and FlexTest are Mentor Graphics ATPG tools. FastScan is a high performance, full-scan Automatic Test Pattern Generation (ATPG) tool. FastScan quickly and efficiently creates a set of test patterns for your (primarily full scan) scan-based design.

   FlexTest is a high-performance, sequential ATPG tool. FlexTest quickly and efficiently creates a set of test patterns for your full, partial, or non-scan design. FastScan and FlexTest both contain an embedded high-speed fault simulator that can verify a set of properly formatted external test patterns.

ASIC Vector Interfaces (AVI) is the optional ASIC vendor-specific pattern formatter available through FastScan and FlexTest. AVI generates a variety of ASIC vendor test pattern formats. FastScan and FlexTest can also generate patterns in a number of different simulation formats so you can verify the design and test patterns with timing. Within the Mentor Graphics environment, you can use QuickSim II and QuickPath for this verification. Chapter 6, "Generating Test Patterns," discusses the ATPG process and formatting and verifying test patterns.

9. **Vendor Creates ASIC and Runs Tests** — At this point, the manufacture of your device is in the hands of the ASIC vendor. Once the ASIC vendor fabricates your design, it will test the device on automatic test equipment (ATE) using test vectors you provide. This manual does not discuss this process, except to mention how constraints of the testing environment might affect your use of the DFT tools.

10. **Vendor Runs Diagnostics** — The ASIC vendor performs a diagnostic analysis on the full set of manufactured chips. Chapter 8, "Running Diagnostics," discusses how to perform diagnostics using FastScan to acquire information on chip failures.

11. **Plug ASIC into Board and Run Board Tests**—When your ASIC design is complete and you have the actual tested device, you are ready to plug it into the board. After board manufacture, the test engineer can run the board level tests, which may include boundary scan testing. This manual does not discuss these tasks.

User Interface Overview

DFT products use two similar graphical user interfaces (GUI): one for BIST products and one for ATPG products. The BIST graphical user interface supports MBISTArchitect, LBISTArchitect, and BSDArchitect. The ATPG graphical user interface supports DFTAdvisor, FastScan, and FlexTest. Both of these user interfaces share many common elements. This subsection describes these common elements. Later in this chapter are descriptions of the product specific elements.

Figure 1-3 shows a representation of the GUI elements that are common to both user interfaces. Notice that the graphical user interfaces consist of two windows: the Command Line window and the Control Panel window.

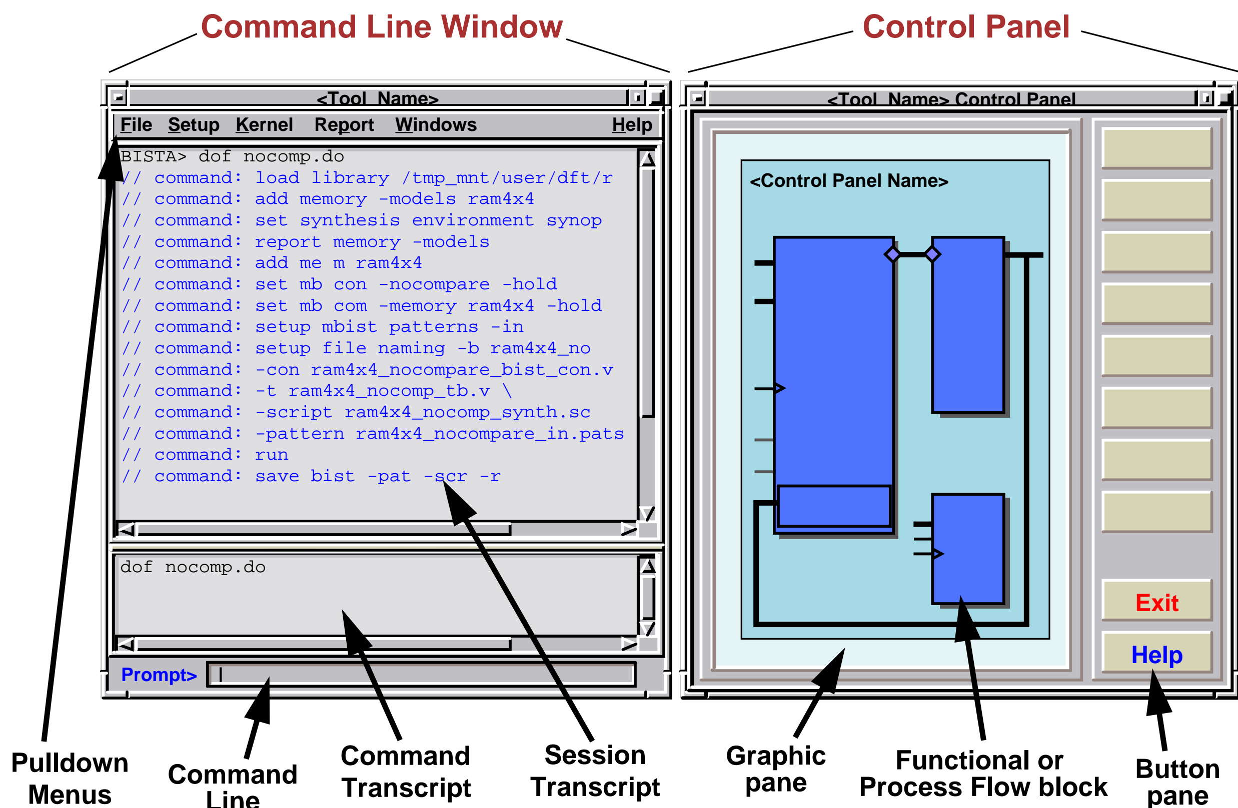


Figure 1-3. Common Elements of the DFT Graphical User Interfaces

When you invoke a DFT product in graphical user interface mode, it opens both the Command Line and Control Panel windows. You can move these two

windows at the same time by pressing the left mouse button in the title bar of the Command Line window and moving the mouse. This is called *window tracking*. If you want to disable window tracking, choose the **Windows > Control Panel > Tracks Main Window** menu item.

The following sections describe each of the user interface common elements shown in Figure 1-3.

Command Line Window

The Command Line window, shown in Figure 1-3 on page 1-9, provides several ways for you to issue commands to your DFT product. For those of you that are mouse oriented, there are pulldown and popup menu items. For those that are more command oriented, there is the command line. In either case, the session and command transcript windows provide a running log of your session.

Pulldown Menus

Pulldown menus are available for all the DFT products. The following lists the pulldown menus that are shared by most of the products and the types of actions typically supported by each menu:

- **File** > menu contains menu items that allow you to load a library or design, read command files, view files or designs, save your session information, and exit your session.
- **Setup** > menu contains menu items that allow you to perform various circuit or session setups. These may include things like setting up your session logfiles or output files.
- **Report** > menu contains menu items that allow you to display various reports regarding your sessions setup or run results.
- **Window** > menu contains menu items that allow you to toggle the visibility and tracking of the Control Panel Window.
- **Help** > menu contains menu items that allow you to directly access the online manual set for the DFT tools. This includes, but is not limited to, the individual command reference pages, the user's manual, and the release

notes. For more information about getting help refer to "Getting Help" on page 1-15.

Within DFTAdvisor, FastScan, and FlexTest, you can add custom menu items. For information on how to add menu items, refer to either "DFTAdvisor User Interface" on page 1-21, "FastScan User Interface" on page 1-23, or "FlexTest User Interface" on page 1-25.

Session Transcript

The session transcript is the largest pane in the Command Line window, as shown in Figure 1-3 on page 1-9. The session transcript lists all commands performed and tool messages in different colors:

- **Black** text - commands issued.
- **Red** text - error messages.
- **Green** text - warning messages.
- **Blue** text - output from the tool other than error and warning messages.

In the session transcript you can re-execute a command by triple-clicking the left mouse button on any portion of the command, then clicking the middle mouse button to execute it. You also have a popup menu available by clicking the right mouse button in the session transcript. The popup menu items are described in Table 1-1.

Table 1-1. Session Transcript Popup Menu Items

Menu Item	Description
Word Wrap	Toggles word wrapping in the window.
Clear Transcript	Clears all text from the transcript.
Save Transcript	Saves the transcript to the specified file.
Font	Adjusts the size of the transcript text.
Exit	Terminates the application tool program.

Command Transcript

The command transcript is located near the bottom of the Command Line window, as shown in Figure 1-3 on page 1-9. The command transcript lists all of the commands executed. You can repeat a command by double-clicking on the command in the command transcript. You can place a command on the command line for editing by clicking once on the command in the command transcript. You also have a popup menu available by clicking the right mouse button in the command transcript. The menu items are described in Table 1-2.

Table 1-2. Command Transcript Popup Menu Items

Menu Item	Description
Clear Command History	Clears all text from the command transcript.
Save Command History	Saves the command transcript to a file you specify.
Previous Command	Copies the previous command to the command line.
Next Command	Copies the next previous command to the command line.
Exit	Terminates the application tool program.

Command Line

The DFT products each support a command set that provide both user information and user-control. You enter these commands on the command line located at the bottom of the Command Line window, as shown in Figure 1-3 on page 1-9. You can also enter commands through a batch file called a dofile. These commands typically fall into one of the following categories:

- **Add commands** - These commands let you specify architectural information, such as clock, memory, scan chain definition.
- **Delete commands** - These commands let you individually “undo” the information you specified with the Add commands. Each Add command has a corresponding Delete command.
- **Report commands** - These commands report on both system and user-specified information.

- **Set and Setup commands** - These commands provide user control over the architecture and outputs.

- **Miscellaneous commands** - The DFT products provides a number of other commands that do not fit neatly into the previous categories. Some of these, such as Help, Dofile, and System, are common to all the DFT/ATPG tools. Others, are specific to the individual products.

Most DFT product commands follow the 3-2-1 minimum typing convention. That is, as a short cut, you need only type the first three characters of the first command word, the first two characters of the second command word, and the first character of the third command word. For example, the DFTAdvisor command Add Nonscan Instance reduces to “add no i” when you use minimum typing.

In cases where the 3-2-1 rule leads to ambiguity between commands, such as Report Scan Cells and Report Scan Chains (both reducing to “rep sc c”), you need to specify the additional characters to alleviate the ambiguity. For example, the DFTAdvisor command Report Scan Chains becomes “rep sc ch” and Report Scan Cells becomes “rep sc ce”.

You should also be aware that when you issue commands with very long argument lists, you can use the “\” line continuation character. For example, in DFTAdvisor you could specify the Add Nonscan Instance command within a dofile (or at the system mode prompt) as follows:

```
add no i\
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$2 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$3 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$5 \
/CBA_SCH/MPI_BLOCK/IDSE$2263/C_A0321H$76/I$8
```

For more information on dofile scripts, refer to “Running Batch Mode Using Dofiles” on page 1-18.

Control Panel Window

The Control Panel window, shown in Figure 1-3 on page 1-9, provides a graphical link to either the functional blocks whose setup you can modify or the flow process from which you can modify your run. The window also present a series of buttons that represent the actions most commonly performed.

Graphic Pane

The graphic pane is located on the left half of the Control Panel window, as shown in Figure 1-3 on page 1-9. The graphic pane can either show the *functional blocks* that represent the typical relationship between a core design and the logic being manipulated by the DFT product or show the *process flow blocks* that represent the groups of tasks that are a part of the DFT product session. Some tools, such as DFTAdvisor or FastScan, have multiple graphic panes that change based on the current step in the process.

When you move the cursor over a functional or process flow block, the block changes color to yellow, which indicates that the block is active. When the block is active, you can click the left mouse button to open a dialog box that lets you perform a task, or click the right mouse button for popup help on that block. For more information on popup help, refer to "Popup Help" on page 1-15.

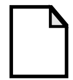
Button Pane

The button pane is located on the right half of the Control Panel window, as shown in Figure 1-3 on page 1-9. The button pane provides a list of buttons that are the actions commonly used while in the tool. You can click the left mouse button a button in the button pane to perform the listed task, or you can click the right mouse button that button for popup help specific to that button. For more information on popup help, refer to "Popup Help" on page 1-15.

Getting Help

There are many different types of online help. These different types include query help, popup help, information messages, Tool Guide help, command usage, online manuals, and the **Help** menu. The following sections describe how to access the different help types.

Query Help


Note

Query help is only supported in the DFTAdvisor, DFTInsight, FastScan, and FlexTest user interfaces.

Query help provides quick text-based messages on the purpose of a button, text field, text area, or drop-down list within a dialog box. If additional information is available in the online PDF manual, a “Go To Manual” button is provided that opens that manual to that information. In dialog boxes that contain multiple pages, query help is also available for each dialog tab.

You activate query help mode by clicking the Turn On Query Help button located at the bottom of the dialog box. The mouse cursor changes to a question mark. You can then click the left mouse button on the different objects in the dialog box to open a help window on that object. You leave query help mode by clicking on the same button, but now named Turn Off Query Help, or by hitting the Escape key.

Popup Help

Popup help is available on all active areas of the Control Panel. You activate this type of help by clicking the right mouse button on a functional block, process block, or button. To remove the help window:

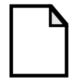
- Click on any other functional block or button in the control panel
- Press any key while the control panel is active

- Click anywhere in the window itself
- Move the mouse outside of the control panel

Information Messages

Information messages are provided in some dialog boxes to help you understand the purpose and use of the dialog box or its options. You do not need to do anything to get these messages to appear.

Tool Guide


Note

The Tool Guide is only available in the DFTAdvisor, FastScan, and FlexTest user interfaces.

The Tool Guide provides quick information on different aspects of the application. You can click on the different topics listed in the upper portion of the window to change the information displayed in the lower portion of the window. You can open the Tool Guide by clicking on the Help button located at the bottom of the Control Panel or from the **Help > Open Tool Guide** menu item.

Command Usage

You can get the command syntax for any command from the command line by using the Help command followed either by a full or partial command name. For example, to list all the "Add" commands in MBISTArchitect, enter:

help add
```
// ADD DAta Backgrounds  ADD MBist Algorithms
// ADD MEmory
```

To see the usage line for a command, enter the Help command followed by the command name. For example, to see the usage for the DFTAdvisor Add Clocks command, enter:

help add clocks
```
//  Add Scan Capture Clocks
//    usage: ADD CLocks <off_state> <primary_pin...>
//    legal system mode: SETUP
```

To open the reference page for a command using the PDF viewer, execute the menu item:

Help > On Commands > Reference Page

Next, select the desired command in the list. The PDF viewer opens to the reference page for the command.

Online Manuals

Application documentation is provided online in PDF format. You can open the manuals using the Help menu (all tools) or the Go To Manual button in query help messages (DFTAdvisor, FastScan, and FlexTest). You can also open a separate shell window and execute *$MGC_HOME/bin/mgc_acro*. In the PDF viewer, you then execute the **MGC > Bookcases > DFT Bookcase** menu item to open the bookcase of DFT documentation.

For information on using the Help menu to open a manual, refer to the following "Help Menu" section.

Help Menu

Many of the menu items use a PDF viewer to display the help text associated with the topic request. To enable the reader's proper behavior you should ensure that you have the proper environment. To do so, select the following menu item:

Help > Set Environment

The Help pulldown menu provides help on the following topics:

- **Open Tool Guide** - Opens the ASCII help tool. For more information, refer to the preceding Tool Guide section. This menu item is only supported in DFTAdvisor, FastScan, and FlexTest user interfaces.
- **On Commands > Open Reference Page** - Displays a window that lists the commands for which help is available. Select or specify a command and click Display. Help opens the PDF viewer and displays the reference page for that command.

- **On Commands > Open Summary Table** - Opens the PDF viewer and displays the Command Summary Table from the current tool's reference manual. You can then click on the command name and jump to the reference page.
- **On Key Bindings** - Displays the key binding definitions for the text entry boxes.
- **Open Bookcase** - Opens the PDF viewer and displays a list of the manuals that apply to the current tool.
- **Open User's Manual** - Opens the PDF viewer and displays the user's manual that applies to the current tool.
- **Open Reference Manual** - Opens the PDF viewer and displays the reference manual that applies to the current tool.
- **Open Release Notes** - Opens the PDF viewer and displays the release note information for this release of the current tool.
- **Customer Support** - Displays helpful information regarding the Mentor Graphics Customer Support organization.
- **How to use Help** - Displays text on how to use help.
- **Setup Environment** - Displays a dialog box that assists you in setting up your Online Help environment and PDF viewer.

Running Batch Mode Using Dofiles

You can run your DFT application in batch mode by using a *dofile* to pipe commands into the application. Dofiles let you automatically control the operations of the tool. The dofile is a text file that you create that contains a list of application commands that you want to run, but without entering them individually. If you have a large number of commands, or a common set of commands that you use frequently, you can save time by placing these commands in a dofile.

You can specify a dofile at invocation by using the -Dofile switch. You can also execute the **File > Command File** menu item, the Dofile command, or click on the Dofile button to execute a dofile at any time during a DFT application session.

If you place all commands, including the Exit command, in a dofile, you can run the entire session as a batch process. Once you generate a dofile, you can run it at invocation. For example, to run MBISTArchitect as a batch process using the commands contained in *my_dofile.do*, enter:

```
shell> $MGC_HOME/bin/bista -m -dofile my_dofile.do
```

The following shows an example MBISTArchitect dofile:

```
load library dft.lib
add memory -models ram16X16
add mbist algorithms 1 march1
add mbist algorithms 2 unique
report mbist algorithms
set file naming -bist_model ram16X16.vhd
run
save bist -VHDL
exit
```

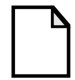
By default, if an ATPG application encounters an error when running one of the commands in the dofile, it stops dofile execution. However, you can turn this setting off by using the Set Dofile Abort command

Generating a Log File

Log files provide a useful way to examine the operation of the tool, especially when you run the tool in batch mode using a dofile. If errors occur, you can examine the log file to see exactly what happened. The log file contains all DFT application operations and any notes, warning, or error messages that occur during the session.

You can generate log files in one of three ways: by using the -Logfile switch when you invoke the tool, by executing the **Setup > Logfile** menu item, or by issuing either the Set Logfile Handling command for ATPG products or the Set Message Handling command for BIST products. When setting up a log file, you can use

instruct the DFT product to generate a new log file, replace an existing log file, or append information to a log file that already exists.


Note

If you create a log file during a DFT product session, the log file will only contain notes, warning, or error messages that occur after you issue the command. Therefore, it should be entered as one of the first commands in the session.

Running UNIX Commands

You can run UNIX operating system commands within DFT applications by using the System command. For example, the following command executes the UNIX operating system command **ls** within a DFT application session:

prompt> **system ls**

Interrupting the Session

To interrupt the invocation of a DFT product and return to the operating system, enter Control-C. You can also use the Control-C key sequence to interrupt the current operation and return control to the tool.

Exiting the Session

To exit a DFT application and return to the operating system, you can execute the **File > Exit** menu item, click on the Exit button in the Control Panel, or enter Exit at the command line:

prompt> **exit**

For information on an individual tool user interface, refer to the following sections:

- "DFTAdvisor User Interface" on page 1-21
- "FastScan User Interface" on page 1-23
- "FlexTest User Interface" on page 1-25

DFTAdvisor User Interface

DFTAdvisor functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by DFTAdvisor has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-9. The remainder of this section describes features unique to DFTAdvisor.

When you invoke DFTAdvisor in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-9. The DFTAdvisor Control Panel window, shown in Figure 1-4, lets you easily set up the different aspects of your design in order to identify and insert test structures. The DFTAdvisor Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. You can get information on each of the these tasks by clicking the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-4, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, simply click on the "Done with" button in the graphic pane or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, DFTAdvisor asks you if you really want to move to the next step.

Within DFTAdvisor, you can add custom pulldown menus in the Command Line window and help topics to the DFTAdvisor Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic "How can I add custom menus and help topics?".

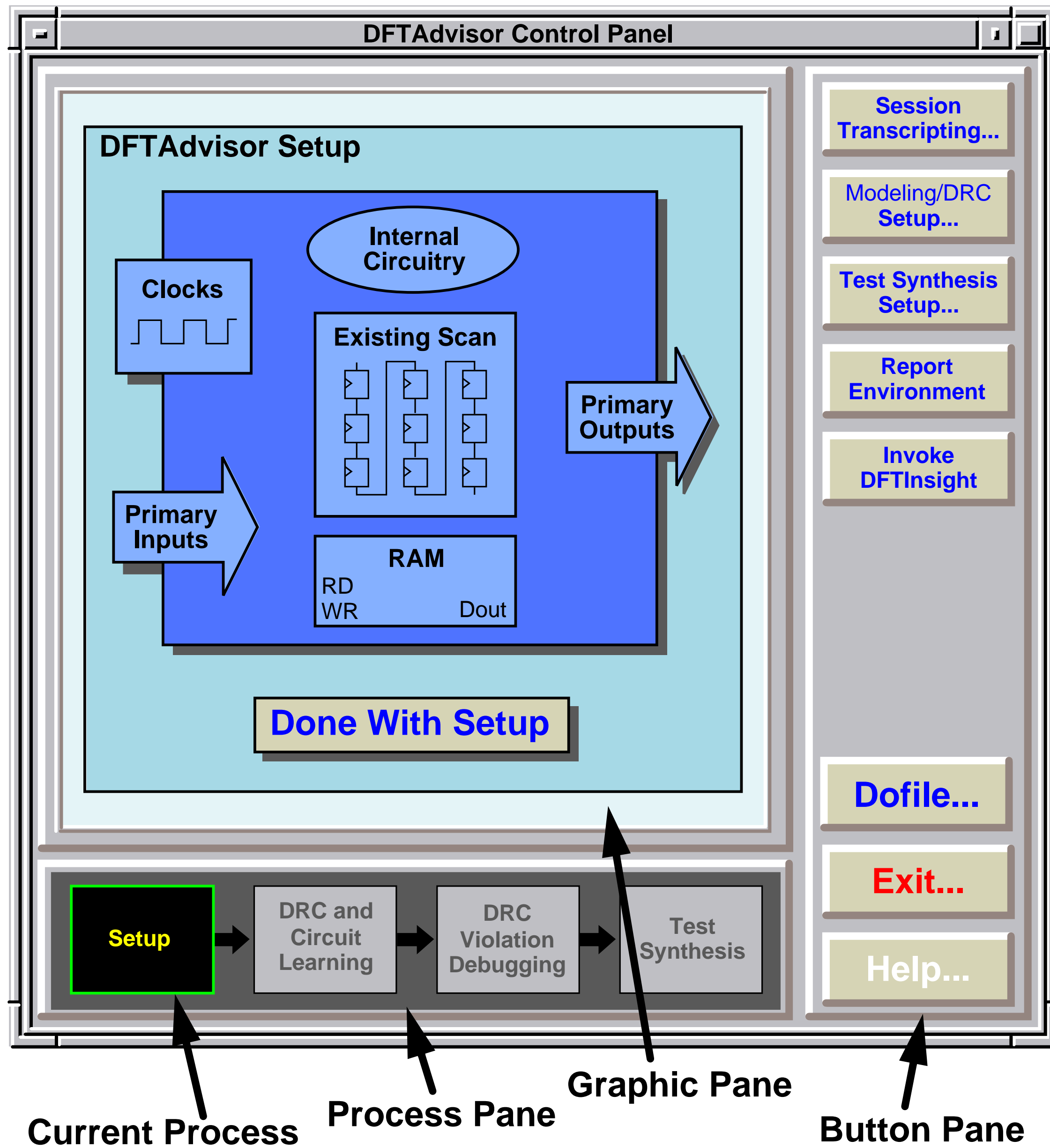


Figure 1-4. DFTAdvisor Control Panel Window

FastScan User Interface

FastScan functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by FastScan has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-9. The remainder of this section describes features unique to DFTAdvisor.

When you invoke FastScan in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-9. The FastScan Control Panel window, shown in Figure 1-5, lets you easily set up the different aspects of your design in order to identify and insert full-scan test structures. The FastScan Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. You can get information on each of the these tasks by clicking the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-5, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, simply click on the "Done with" button in the graphic pane or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, FastScan asks you if you really want to move to the next step.

Within FastScan, you can add custom pulldown menus in the Command Line window and help topics to the FastScan Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic "How can I add custom menus and help topics?".

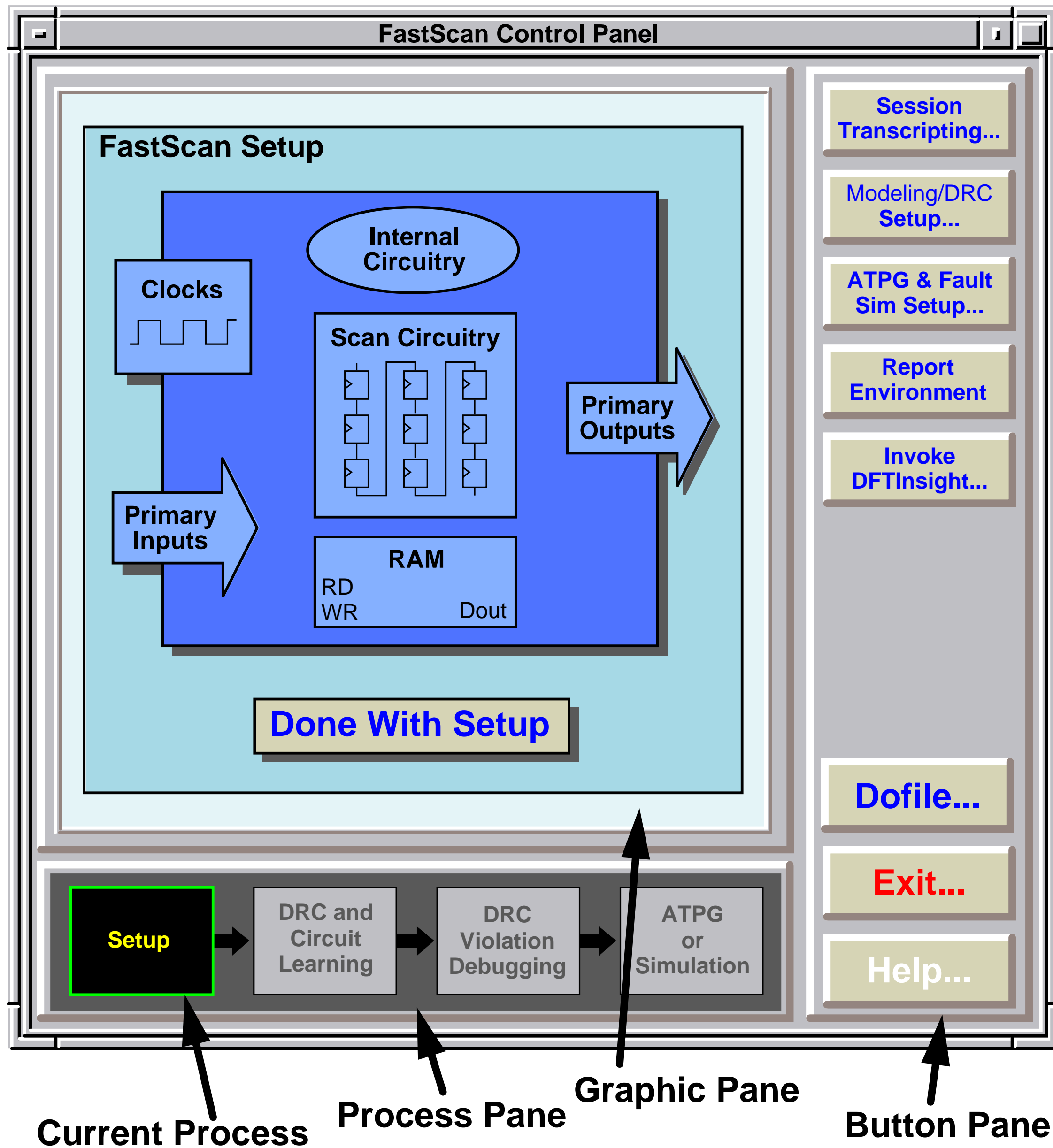


Figure 1-5. FastScan Control Panel Window

FlexTest User Interface

FlexTest functionality is available in two modes: graphical user interface or command-line user interface. The graphical mode employed by FlexTest has many features shared by all DFT products. These shared features are described in "User Interface Overview" on page 1-9. The remainder of this section describes features unique to DFTAdvisor.

When you invoke FlexTest in graphical mode, the Command Line and Control Panel windows are opened. An example of these two windows is shown in Figure 1-3 on page 1-9. The FlexTest Control Panel window, shown in Figure 1-6, lets you easily set up the different aspects of your design in order to identify and insert partial-scan test structures. The FlexTest Control Panel contains three panes: a graphic pane, a button pane, and a process pane. These panes are available in each of the process steps identified in the process pane at the bottom of the Control Panel window.

You use the process pane to step through the major tasks in the process. Each of the process steps has a different graphic pane and a different set of buttons in the button pane. The current process step is highlighted in green. Within the process step, you have sub-tasks that are shown as functional or process flow blocks in the graphic pane. You can get information on each of the these tasks by clicking the right mouse button on the block. For example, to get help on the Clocks functional block in Figure 1-6, click the right mouse button on it.

When you have completed the sub-tasks within a major task and are ready to move on to the next process step, simply click on the "Done with" button in the graphic pane or on the process button in the process pane. If you have not completed all of the required sub-tasks associated with that process step, FlexTest asks you if you really want to move to the next step.

Within FlexTest, you can add custom pulldown menus in the Command Line window and help topics to the FlexTest Tool Guide window. This gives you the ability to automate common tasks and create notes on tool usage. For more information on creating these custom menus and help topics, click on the Help button in the button pane and then choose the help topic "How can I add custom menus and help topics?".

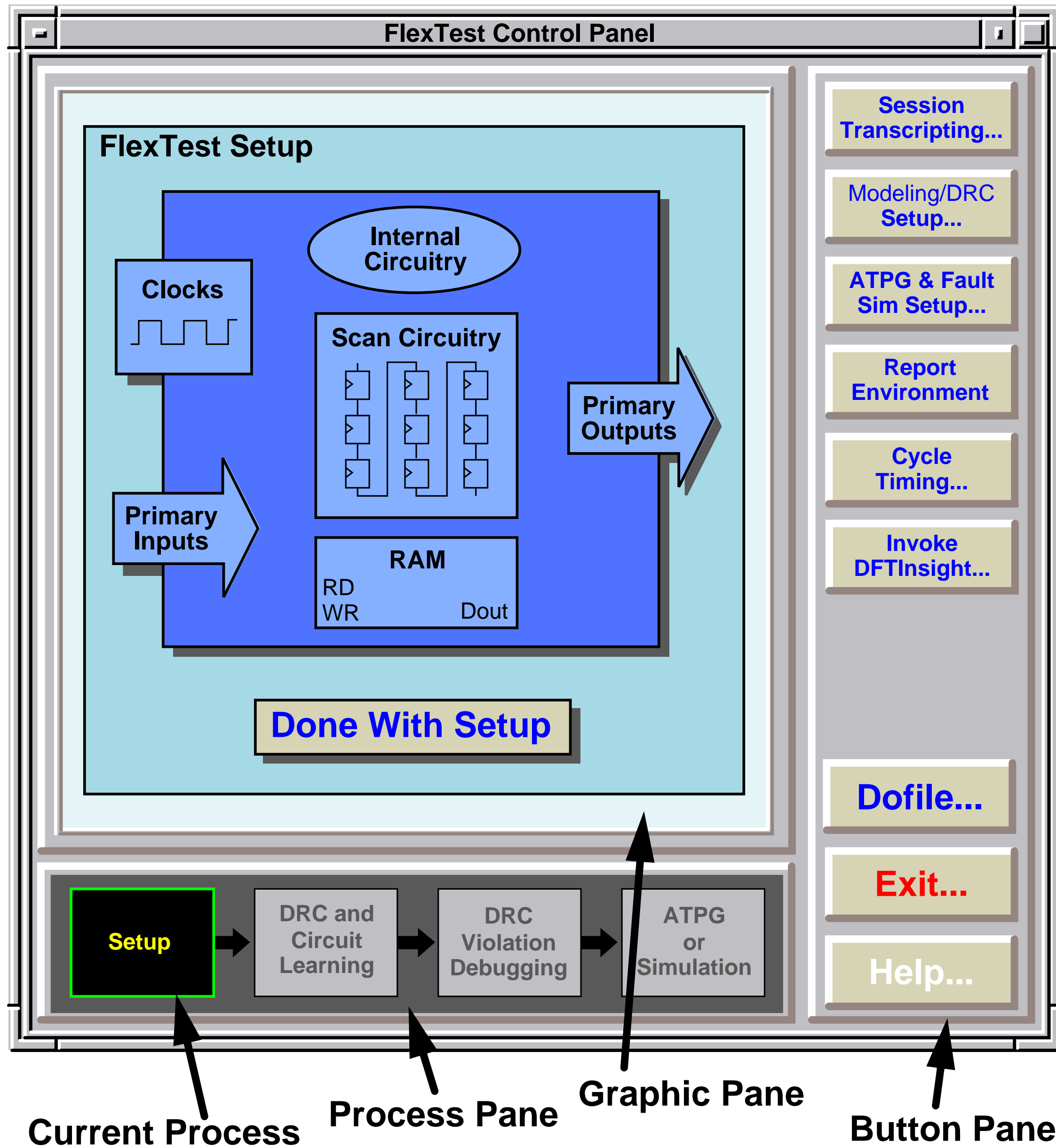


Figure 1-6. FlexTest Control Panel Window

Chapter 2
Understanding Scan and ATPG Basics

Before you begin the DFT process, you must first have an understanding of certain DFT concepts. Once you understand these concepts, you can determine the best test strategy for your particular design. Figure 2-1 shows the concepts this section discusses.

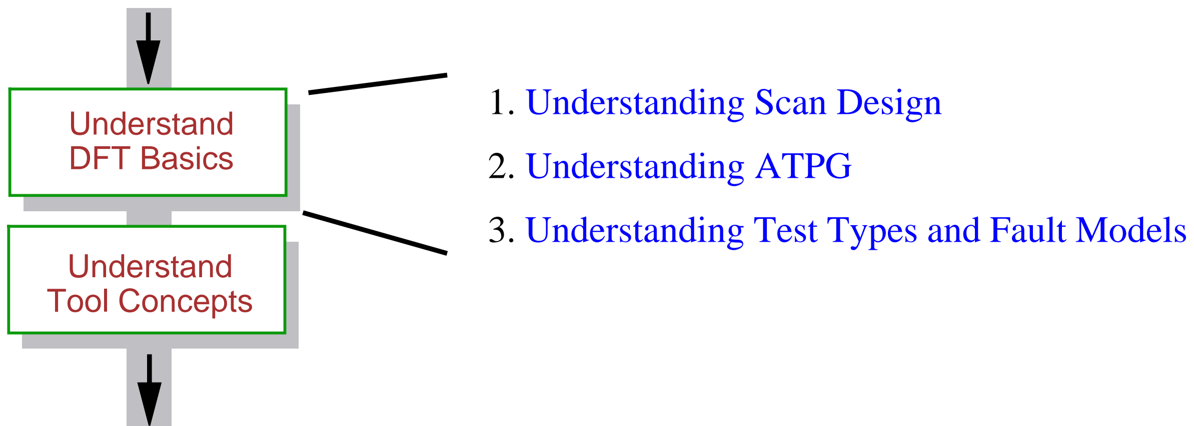


Figure 2-1. DFT Concepts

Built-in self-test (BIST) circuitry, along with scan circuitry, greatly enhances a design's testability. BIST leaves the job of testing up to the device itself, eliminating or minimizing the need for external test equipment. A discussion of BIST and the BIST process is provided in the *Built-in Self-Test Process Guide*.

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

design's circuitry to increase its testability. A detailed discussion of internal scan begins on page 2-2.

While scan design modifies circuitry within the original design, *boundary scan* adds scan circuitry around the periphery of the design to make internal circuitry on a chip accessible via a standard board interface. The added circuitry enhances board testability of the chip, the chip I/O pads, and the interconnections of the chip to other board circuitry. A discussion of boundary scan and the boundary scan process is available in the *Boundary Scan Process Guide*.

Understanding Scan Design

This section gives you an overview of scan design and how it works. For more detailed information on the concepts presented in this section, refer to the documentation references cited on page xx.

Internal Scan Circuitry

As previously discussed, *internal scan* (or *scan design*) is the internal modification of your design's circuitry to increase its testability. Scan design uses either full or partial scan techniques, depending on design criteria. Full scan techniques are discussed on page 2-4. Partial scan techniques are discussed on page 2-5.

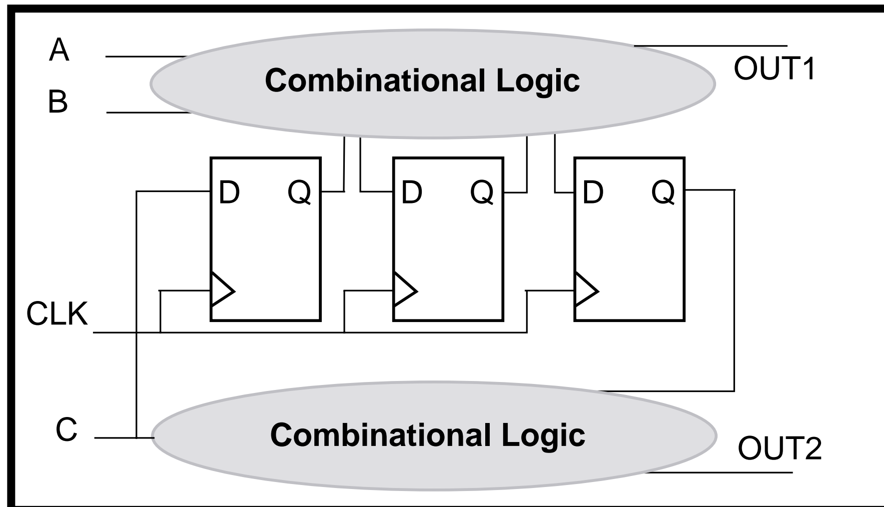
Scan Design Overview

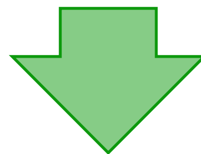
The goal of scan design is to make a difficult-to-test sequential circuit behave (during the testing process) like an easier-to-test combinational circuit. Achieving this goal involves replacing sequential elements with scannable sequential elements (scan cells) and then stitching the scan cells together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

The design shown in Figure 2-2 contains both combinational and sequential portions. Before adding scan, the design had three inputs, A, B, and C, and two outputs, OUT1 and OUT2. This "Before Scan" version is difficult to initialize to a

known state, making it difficult to both control the internal circuitry and observe its behavior using the primary inputs and outputs of the design.

Before Scan





After Scan

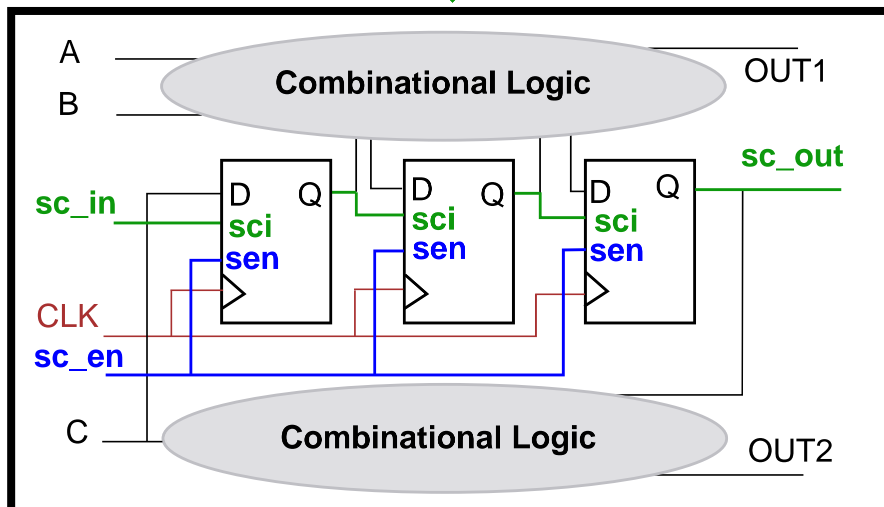


Figure 2-2. Design Before and After Adding Scan

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

The operating procedure of the scan circuitry is as follows:

1. Enable the scan operation to allow shifting (to initialize scan cells).
2. After loading the scan cells, hold the scan clocks off and then apply stimulus to the primary inputs.
3. Measure the outputs.
4. Pulse the clock to capture new values into scan cells.
5. Enable the scan operation to unload and measure the captured values while simultaneously loading in new values via the shifting procedure (as in step 1).

Understanding Full Scan

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

Figure 2-3 gives a symbolic representation of a full scan design.

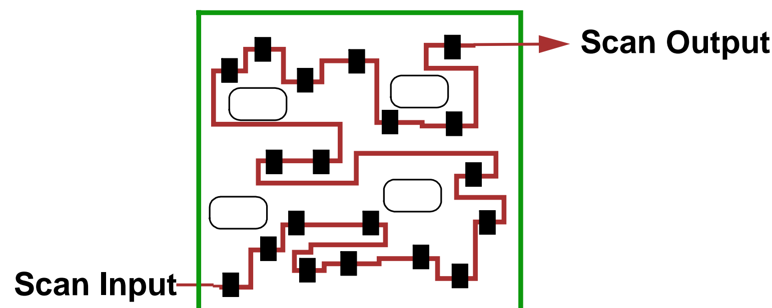


Figure 2-3. Full Scan Representation

The black rectangles in Figure 2-4 represent scan elements. The line connecting them is the scan path. Because this is a full scan design, all storage elements were converted and connected in the scan path. The rounded boxes represent combinational portions of the circuit.

For information on implementing a full scan strategy for your design, refer to "Test Structures Supported by DFTAdvisor" on page 5-7.

Full Scan Benefits

The following are benefits of employing a full scan strategy:

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

- **Highly-effective, predictable method**.
  Full scan design is a highly-effective, well-understood, and well-accepted method for generating high test coverage for your design.

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

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

Understanding Partial Scan

Because full scan design makes all storage elements scannable, it may not be acceptable for all your designs because of area and timing constraints. *Partial scan* is a scan design methodology where only a percentage of the storage elements in the design are replaced by their scannable equivalents and stitched into scan chains. Using the partial scan method, you can increase the testability of your design with minimal impact on the design's area or timing. In general, the amount of scan required to get an acceptable fault coverage varies from design to design.

Figure 2-4 gives a symbolic representation of a partial scan design.

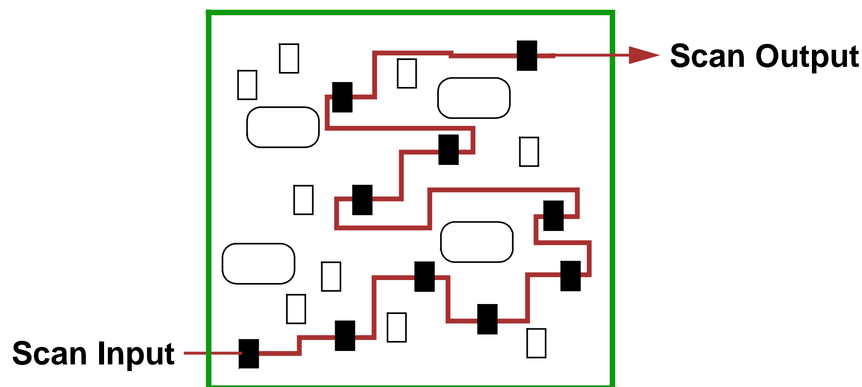


Figure 2-4. Partial Scan Representation

The rectangles in Figure 2-4 represent sequential elements of the design. The black rectangles are storage elements that have been converted to scan elements. The line connecting them is the scan path. The white rectangles are elements that have *not* been converted to scan elements and thus, are not part of the scan chain. The rounded boxes represent combinational portions of the circuit.

In the partial scan methodology, the test engineer, designer, or scan insertion tool selects the desired flip-flops for the scan chain. For information on implementing a partial scan strategy for your design, refer to "Test Structures Supported by DFTAdvisor" on page 5-7.

Partial Scan Benefits

- **Reduced impact on area**.
  If your design cannot tolerate full scan's extra area overhead, you can instead employ partial scan to improve testability to the degree that you can afford.

- **Reduced impact on timing**.
  If you cannot tolerate the extra delay added to your critical path (due to added scan component delay), you can exclude those critical flip-flops from the scan chain using partial scan.

- **More flexibility between overhead and fault coverage**.
  You can make trade-offs between area/timing overhead and acceptable testability improvements.

- **Re-use of non-scan macros**.
  You can include an existing design block, or *macro*, that you want to use within your design "as-is" (with absolutely no changes). You can then employ whatever scan strategy you want within the rest of the design. This would be considered a partial scan strategy.

Choosing Between Full or Partial Scan

The decision to use a full scan or partial scan methodology has a significant impact on which ATPG tool you use. Full scan designs allow combinational ATPG methods, which require minimal test generation effort, but carry a significant amount of area overhead. On the other hand, partial to non-scan designs consume far less area overhead, but require sequential ATPG techniques, which demand significantly more test generation effort. Figure 2-5 gives a pictorial representation of these trade-offs.

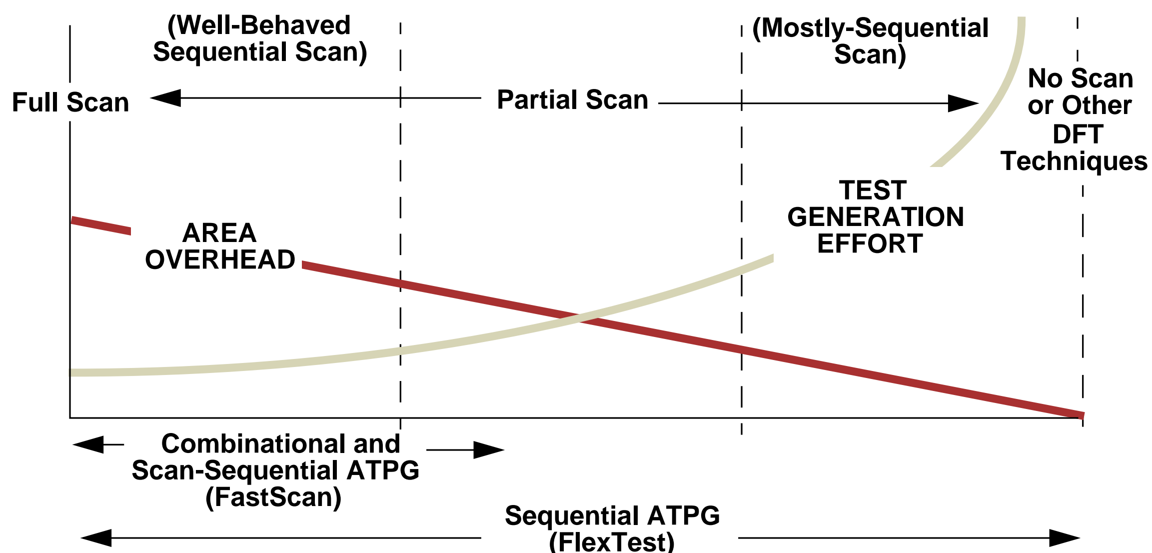


Figure 2-5. Full, Partial, and Non-Scan Trade-offs

Mentor Graphics provides two ATPG tools, FastScan and FlexTest. FastScan uses both combinational (for full scan) and scan-sequential ATPG algorithms. Well-

behaved sequential scan designs can use scan-sequential ATPG. Such designs normally contain a high percentage of scan but can also contain "well-behaved" sequential logic, such as non-scan latches, sequential memories, and limited sequential depth. Although you can use FastScan on other design types, its ATPG algorithms work most efficiently on full scan and scan-sequential designs.

FlexTest uses sequential ATPG algorithms and is thus effective over a wider range of design styles. However, FlexTest works most effectively on primarily sequential designs; that is, those containing a lower percentage of scan circuitry. Because the ATPG algorithms of the two tools differ, you *can* use both FastScan and FlexTest together to create an optimal test set on nearly any type of design.

"Understanding ATPG" on page 2-14 covers ATPG, FastScan, and FlexTest in more detail.

Understanding Partition Scan

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

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

The architecture of partition scan is illustrated in the following two figures. Figure 2-6 shows a design with three partitions, A, B, and C.

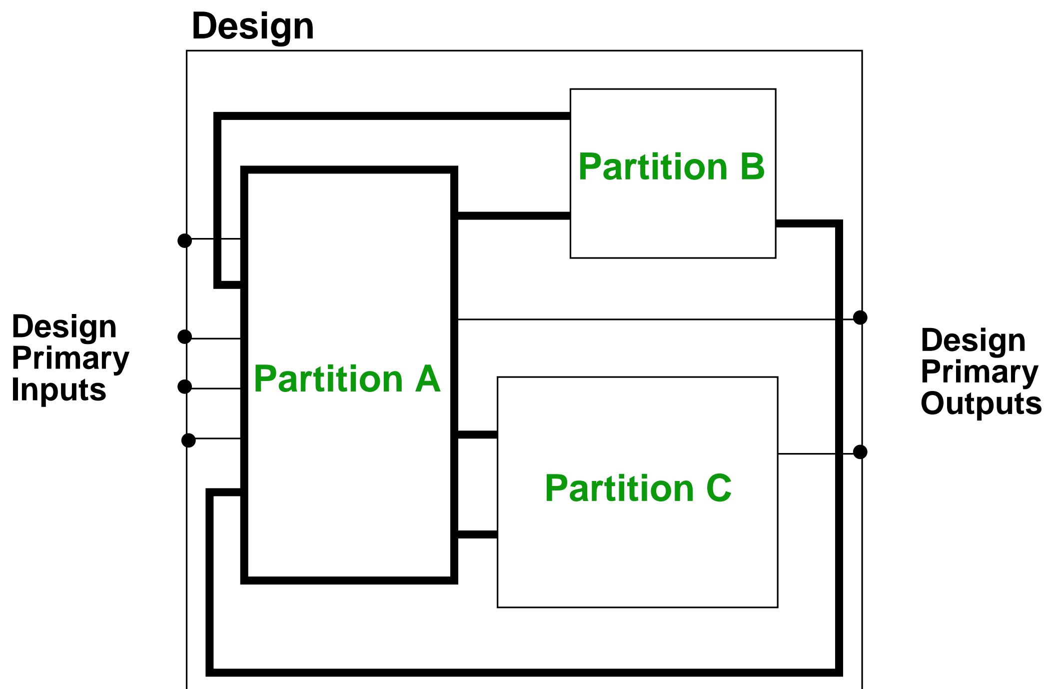
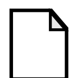


Figure 2-6. Example of Partitioned Design

The bold lines in Figure 2-6 indicate inputs and outputs of partition A that are not directly controllable or observable from the design level. Because these lines are not directly accessible at the design level, the circuitry controlled by these pins can cause testability problems for the design.

Figure 2-7 shows how adding partition scan structures to partition A increases the controllability and observability (testability) of partition A from the design level.


Note

Only the first elements directly connected to the uncontrollable (unobservable) primary inputs (primary outputs) become part of the partition scan chain.

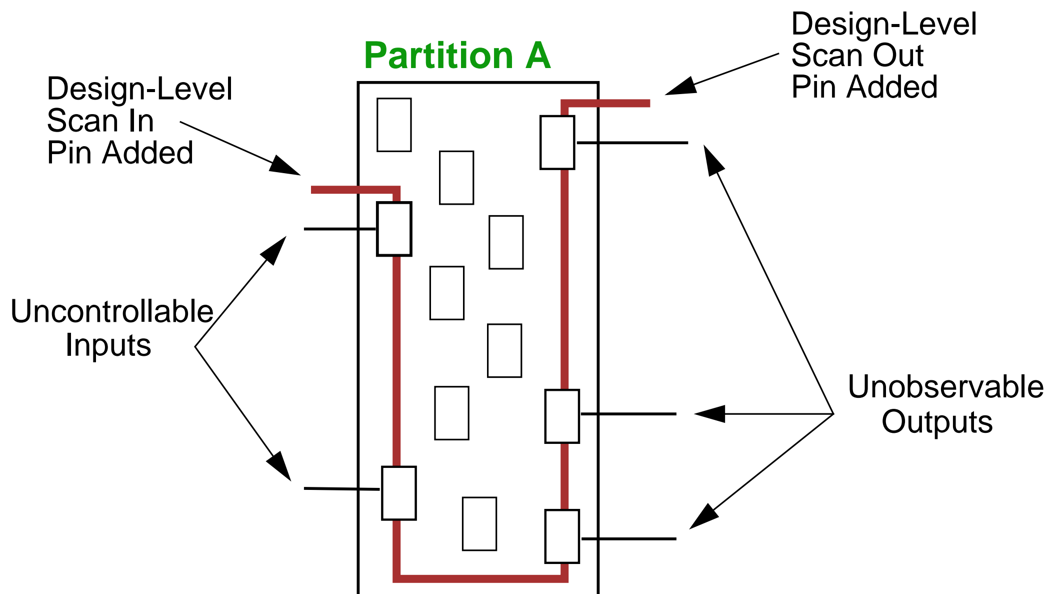


Figure 2-7. Partition Scan Circuitry Added to Partition A

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

You can also use partition scan in conjunction with either full or partial scan structures. Sequential elements not eligible for partition scan become candidates for internal scan.

For information on implementing a partition scan strategy for your design, refer to "Setting Up for Partition Scan Identification" on page 5-22.

Understanding Test Points

A design can contain a number of points that are difficult to control or observe. Sometimes this is true even in designs containing scan. By adding special circuitry at certain locations called test points, you can increase the testability of the design. For example, Figure 2-8 shows a portion of circuitry with a controllability and observability problem.

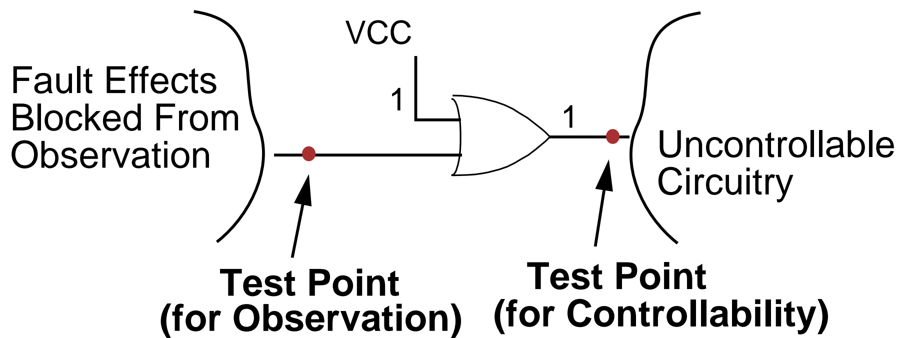


Figure 2-8. Uncontrollable and Unobservable Circuitry

In this example, one input of an OR gate is tied to a 1. This blocks the ability to propagate through this path any fault effects in circuitry feeding the other input. Thus, the other input must become a test point to improve observation. The tied input also causes a constant 1 at the output of the OR gate. This means any circuitry downstream from that output is uncontrollable. The pin at the output of the gate becomes a test point to improve controllability. Once identification of these points occurs, added circuitry can improve the controllability and observability problems.

Figure 2-9 shows circuitry added at these test points.

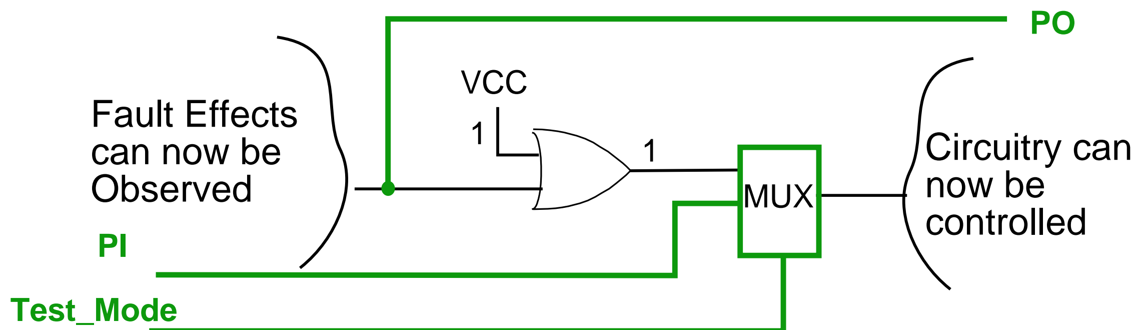


Figure 2-9. Testability Benefits from Test Point Circuitry

At the observability test point, an added primary output provides direct observation of the signal value. At the controllability test point, an added MUX controlled by a test_mode signal and primary input controls the value fed to the associated circuitry.

This is just one example of how test point circuitry can increase design testability. Refer to "Setting Up for Test Point Identification" on page 5-28 for information on identifying test points and inserting test point circuitry.

Test point circuitry is similar to test logic circuitry. For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-12.

Test Structure Insertion with DFTAdvisor

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

DFTAdvisor contains the following features:

- **Multiple formats**.
  Reads and writes the following design data formats: GENIE, EDIF (2.0.0), TDL, VHDL, or Verilog.

- **Multiple scan types**.
  Supports insertion of three different scan types, or methodologies: mux-DFF, clocked-scan, and LSSD.

- **Multiple test structures**.
  Supports identification and insertion of full scan, partial scan (both sequential ATPG-based and scan sequential procedure-based), partition scan, and test points.

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

- **Design rules checking**.
  Performs design rules checking to ensure scan setup and operation are correct--before scan is actually inserted. This rules checking also guarantees that the scan insertion done by DFTAdvisor produces results that function properly in the ATPG tools, FastScan and FlexTest.

- **Interface to ATPG tools**.
  Automatically generates information for the ATPG tools on how to operate the scan circuitry DFTAdvisor creates.

- **Optimal partial scan selection**.
  Provides optimal partial scan analysis and insertion capabilities.

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

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

- **User specified pins**.
  Allows user-specified pin names for test and other I/O pins.

- **Multiple model levels**.
  Handles gate-level, as well as gate/transistor-level models.

- **Online help**.
  Provides online help for every command along with online manuals.

For information on using DFTAdvisor to insert scan circuitry into your design, refer to "Inserting Internal Scan and Test Circuitry" on page 5-1.

Understanding ATPG

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

The ATPG Process

The goal of ATPG is to create a set of patterns that achieves a given test coverage, where test coverage is the total percentage of testable faults the pattern set actually detects (For a more precise definition of test coverage, see page 2-40.) The ATPG run itself consists of two main steps: 1) generating patterns and, 2) performing fault simulation to determine which faults the patterns detect. This section discusses only the generation of test patterns. “Fault Classes” on page 2-32 discusses the fault simulation process.

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

Random Pattern Test Generation

An ATPG tool uses *random pattern test generation* when it produces a number of random patterns and identifies only those patterns necessary to detect faults. It then stores only those patterns in the test pattern set. The type of fault simulation used in random pattern test generation cannot replace deterministic test generation because it can never identify redundant faults. Nor can it create test patterns for faults that have a very low probability of detection. However, it can be useful on testable faults aborted by deterministic test generation. Using a small number of random patterns as the initial ATPG step can improve ATPG performance.

Deterministic Test Pattern Generation

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

More specifically, the tool assigns a set of values to control points that force the fault site to the state opposite the fault-free state, so there is a detectable difference between the fault value and the fault-free value. The tool must then find a way to propagate this difference to a point where it can observe the fault effect. To satisfy the conditions necessary to create a test pattern, the test generation process makes intelligent decisions on how best to place a desired value on a gate. If a conflict prevents the placing of those values on the gate, the tool refines those decisions as it attempts to find a successful test pattern.

If the tool exhausts all possible choices without finding a successful test pattern, it must perform further analysis before classifying the fault. Faults requiring this analysis include redundant, ATPG-untestable, and possible-detected-untestable categories (see page 2-32 for more information on fault classes). Identifying these fault types is an important by-product of deterministic test generation and is critical to achieving high test coverage. For example, if a fault is proven redundant, the tool may safely mark it as untestable. Otherwise, it is classified as a potentially detectable fault and counts as an untested fault when calculating test coverage.

External Pattern Test Generation

An ATPG tool uses *external pattern test generation* when the preliminary source of ATPG is a pre-existing set of external patterns that already exists. The tool analyzes this external pattern set to determine which patterns detect faults from the active fault list. It then places these effective patterns into an internal test pattern set. The "generated patterns", in this case, include the patterns (selected from the external set) that can efficiently obtain the highest test coverage for the design.

Mentor Graphics ATPG Applications

Mentor Graphics provides two ATPG applications: FastScan and FlexTest. FastScan is Mentor Graphics full-scan and scan sequential ATPG solution. FlexTest is Mentor Graphics non-scan to full-scan ATPG solution. The following subsections introduce the features of these two tools. Chapter 6, "Generating Test Patterns," discusses FastScan and FlexTest in greater detail.

Full-Scan and Scan Sequential ATPG with FastScan

FastScan has many features, including:

- **Very high performance and capacity**.
  In benchmarks, FastScan produced 99.9% fault coverage on a 100k gate design in less than 1/2 hour. In addition, FastScan has successfully benchmarked designs exceeding 1 million gates.
- **Reduced size pattern sets**.
  FastScan produces an efficient, compact pattern set.
- **The ability to support a wide range of DFT structures**.
  FastScan supports stuck-at, IDDQ, transition, toggle, and path delay fault models. FastScan also supports all scan styles, multiple scan chains, multiple scan clocks, plus gated clocks, set, and reset lines. Additionally, FastScan has some sequential testing capabilities for your design's non-scan circuitry.
- **Additions to scan ATPG**.
  FastScan provides easy and flexible scan setup using a test procedure file. FastScan also provides DFT rules checking (before you can generate test patterns) to ensure proper scan operation. FastScan's pattern compression abilities ensure that you have a small, yet efficient, set of test patterns. FastScan also provides diagnostic capabilities, so you not only know if a chip is good or faulty, but you also have some information to pinpoint problems. FastScan also supports built-in self-test (BIST) functionality, and supports both RAM/ROM components and transparent latches.

- **Tight integration in Mentor Graphics top-down design flow**.
  FastScan is tightly coupled with DFTAdvisor and AutoLogic in the Mentor Graphics top-down design flow.

- **Support for use in external tool environments**.
  You can use FastScan in many non-Mentor Graphics design flows, including Verilog and Synopsys.

- **Flexible packaging**.
  FastScan is available in a variety of packages. The standard package, *fastscan*, runs under Falcon Framework and operates in both graphical and non-graphical modes. The non-Falcon product, *fastscan_pt,* is a much smaller package intended for use as a point tool in non-Mentor Graphics design flows. This package has the same licensing requirements and capabilities as the standard *fastscan* package, except for the exclusion of the SimView graphical user interface, EDDM input, and WDB output.

  FastScan also has a diagnostic-only package, which you install normally but which licenses only the setup and diagnostic capabilities of the tool; that is, you cannot run ATPG.

Refer to the *FastScan and FlexTest Reference Manual* for the full set of FastScan functions.

Non- to Full-Scan ATPG with FlexTest

FlexTest has many features, including:

- **Flexibility of design styles**.
  You can use FlexTest on designs with a wide-range of scan circuitry--from no internal scan to full scan.

- **Tight integration in the Mentor Graphics top-down design flow**.
  FlexTest is tightly coupled with QuickSim II, DFTAdvisor, and AutoLogic in the Mentor Graphics top-down design flow.

- **Additions to scan ATPG**.
  FlexTest provides easy and flexible scan setup using a test procedure file.

FlexTest also provides DFT rules checking (before you generate test patterns) to ensure proper scan operation.

- **Support for use in external tool environments**.
  You can also use FlexTest as a point tool in many non-Mentor Graphics design flows, including Verilog and Synopsys.

- **Versatile DFT structure support**.
  FlexTest supports a wide range of DFT structures.

- **Flexible packaging**.
  FlexTest is available in a variety of packages. The standard Falcon-Framework package, *flextest*, operates in both graphical and non-graphical modes. The non-Falcon product, *flextest_pt,* is a much smaller package intended for use as a point tool in non-Mentor Graphics design flows. This package has the same capabilities and licensing requirements as the standard *flextest* package, excluding the SimView graphical user interface, EDDM input, WDB output, and SVDM. FlexTest also has a fault simulation-only package, which you install normally but which licenses only the setup, good, and fault simulation capabilities of the tool; that is, you cannot run ATPG and scan identification.

Refer to the *FastScan and FlexTest Reference Manual* for the full set of FlexTest functions.

Understanding Test Types and Fault Models

A *manufacturing defect* is a physical problem that occurs during the manufacturing process, causing device malfunctions of some kind. The purpose of test generation is to create a set of test patterns that detects as many manufacturing defects as possible. Figure 2-10 gives an example of possible device defect types.

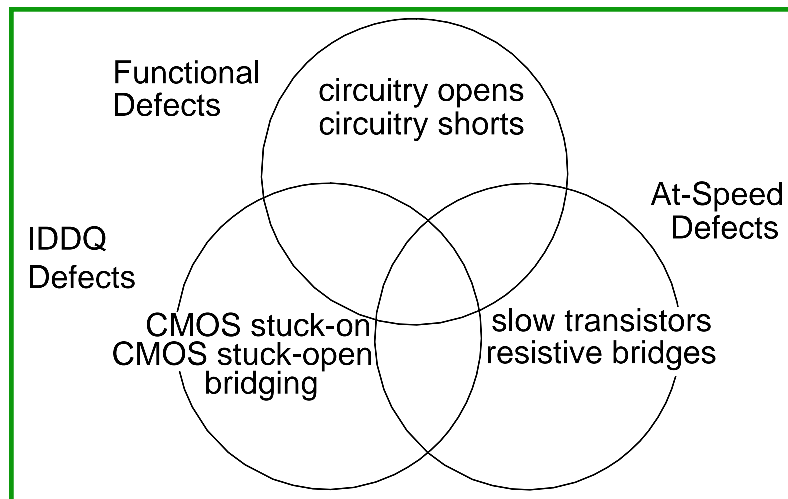


Figure 2-10. Manufacturing Defect Space for Design “X

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

Test Types

Figure 2-10 shows three main categories of defects and their associated test types: *functional*, *IDDQ*, and *at-speed*. Functional testing checks the logic levels of output pins for a “0” and “1” response. IDDQ testing measures the current going through the circuit devices. At-speed testing checks the amount of time it takes for a device to change logic states. The following subsections discuss each of these test types in more detail.

Functional Test

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

Functional testing uses logic levels at the device input pins to detect the most common manufacturing process-caused problem, static defects (open, short, stuck-on, and stuck-open conditions). Functional testing applies a pattern of 1s and 0s to the input pins of a circuit and then measures the logical results at the

output pins. In general, a defect produces a logical value at the outputs different from the expected output value.

IDDQ Test

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

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

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

IDDQ test generation methodologies break down into three categories:

- **Every-vector**
  This methodology monitors the power-supply current for every vector in a functional or stuck-at fault test set. Unfortunately, this method is relatively slow--on the order of 10-100 milliseconds per measurement--making it impractical in a manufacturing environment.

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

- **Selective**
  This methodology intelligently chooses a small set of test vectors from the existing sequence of test vectors to measure current.

Fastscan and Flextest support both supplemental and selective IDDQ test methodologies.

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

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

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

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

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

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

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

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

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

Fastscan and Flextest currently support only the ideal IDDQ test methodology for fully static, resistive, and some dynamic CMOS circuits. The tools can also perform IDDQ checks during ATPG to ensure the vectors they produce meet the ideal requirements. For information on creating IDDQ test sets, refer to"Creating an IDDQ Test Set" on page 6-85.

At-Speed Test

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

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

Fault Modeling

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

Test Types and Associated Fault Models

Table 2-1 associates test types, fault models, and the types of manufacturing defects targeted for detection.

Table 2-1. Test Type/Fault Model Relationship

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

Fault Locations

By default, faults reside at the inputs and outputs of gates within library cells. This is called *internal faulting*. However, faults can instead reside at the inputs and outputs of the library cell if you turn internal faulting off. Figure 2-11 shows the fault sites for both cases.

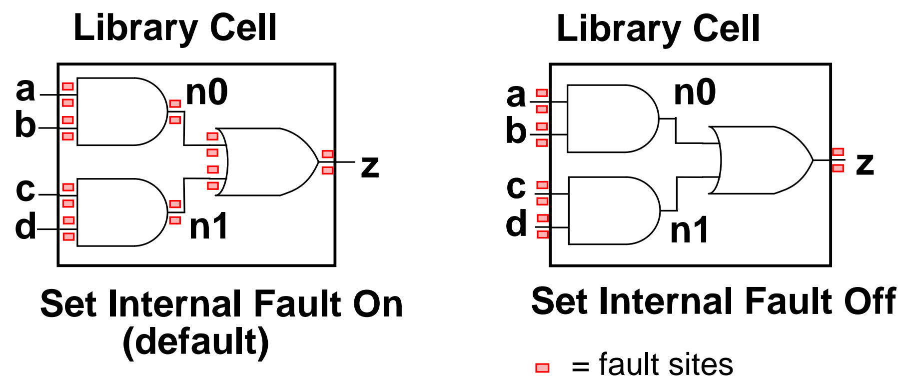


Figure 2-11. Internal Faulting Example

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

Fault Collapsing

A circuit can contain a significant number of faults that behave identically to other faults. That is, the test may identify a fault, but may not be able to distinguish it from another fault. In this case, the faults are said to be equivalent, and the fault identification process reduces the faults to one equivalent fault in a process known as *fault collapsing*. For performance reasons, FastScan and FlexTest evaluate only the one equivalent fault, or *collapsed fault*, during fault simulation and test pattern generation. However, these applications retain information on both collapsed and uncollapsed faults so they can still make fault reports and test coverage calculations.

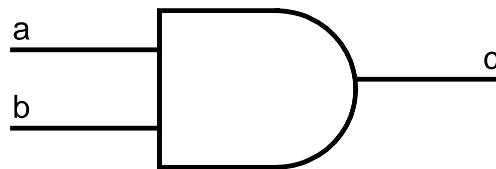
Supported Fault Model Types

FastScan and FlexTest support stuck-at, pseudo stuck-at, toggle, and transition fault models. In addition to these, FastScan supports the path delay fault model.

The following subsections discuss these supported fault models, along with their fault collapsing rules.

Functional Testing and the Stuck-At Fault Model

Functional testing uses the *single stuck-at model*, the most common fault model used in fault simulation, because of its effectiveness in finding many common defect types. The stuck-at fault models the behavior that occurs if the terminals of a gate are stuck at either a high (stuck-at-1) or low (stuck-at-0) voltage. The fault sites for this fault model include the pins of primitive instances. Figure 2-12 shows the possible stuck-at faults that could occur on a single AND gate.



Possible Errors: 6
"a" s-a-1, "a" s-a-0
"b" s-a-1, "b" s-a-0
"c" s-a-1, "c" s-a-0

Figure 2-12. Single Stuck-At Faults for AND Gate

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

FastScan and FlexTest use the following fault collapsing rules for the single stuck-at model:

- **Buffer** - input stuck-at-0 is equivalent to output stuck-at-0. Input stuck-at-1 is equivalent to output stuck-at-1.
- **Inverter** - input stuck-at-0 is equivalent to output stuck-at-1. Input stuck-at-1 is equivalent to output stuck-at-0.
- **AND** - output stuck-at-0 is equivalent to any input stuck-at-0.
- **NAND** - output stuck-at-1 is equivalent to any input stuck-at-0.

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

Functional Testing and the Toggle Fault Model

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

FastScan and FlexTest use the following fault collapsing rules for the toggle fault model:

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

IDDQ Testing and the Pseudo Stuck-At Fault Model

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

FastScan and FlexTest support the *pseudo stuck-at* fault model for IDDQ testing. Testing detects a pseudo stuck-at model at a node if the fault is excited and propagated to the output of a cell (library model instance or primitive). Because

FastScan and FlexTest library models can be hierarchical, fault modeling occurs at different levels of detail.

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

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

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

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

Figure 2-13 shows the IDDQ testing process using the pseudo stuck-at fault model.

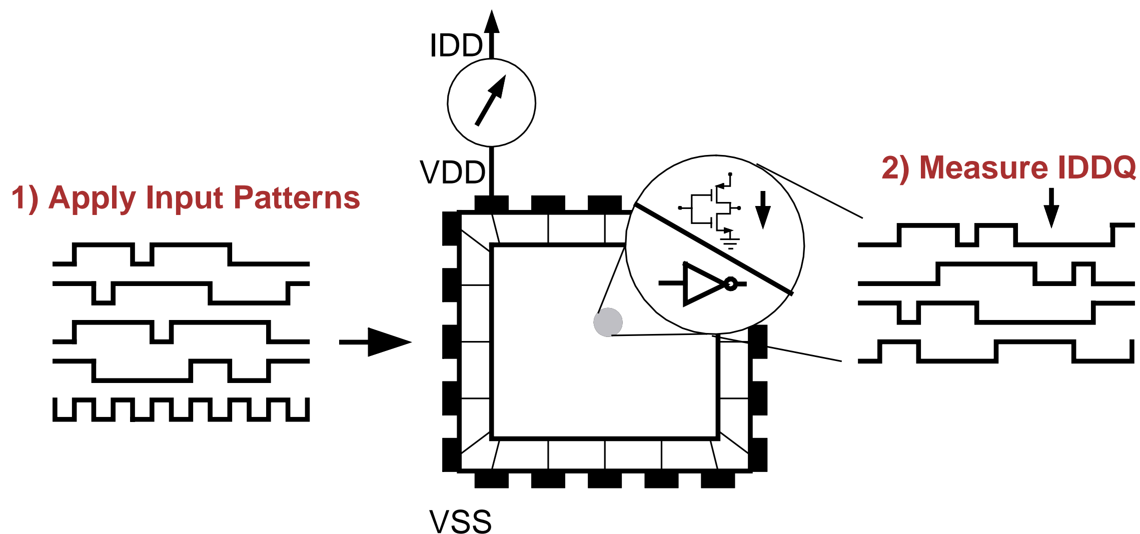


Figure 2-13. IDDQ Fault Testing

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

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

By combining IDDQ testing with traditional stuck-at fault testing, you can greatly improve the overall test coverage of your design. However, because it is costly and impractical to monitor current for every vector in the test set, you can supplement an existing stuck-at test set with a compact set of test vectors for measuring IDDQ. This set of IDDQ vectors can either be generated automatically or intelligently chosen from an existing set of test vectors. Refer to section "Creating an IDDQ Test Set" on page 6-85 for information.

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

Related Commands

Set Transition Holdpi - freezes all primary inputs values other than clocks and RAM controls during multiple cycles of pattern generation.

At-Speed Testing and the Transition Fault Model

Transition faults model large delay defects in the circuit under test. The transition fault model, which is supported by both FastScan and FlexTest, behaves as a stuck-at fault for a temporary period of time for FastScan and one test cycle for FlexTest. The *slow-to-rise* transition fault models a device pin that is defective because its value is slow to change from a 0 to a 1. The *slow-to-fall* transition fault models a device pin that is defective because its value is slow to change from a 1 to a 0.

Figure 2-14 demonstrates the at-speed testing process using the transition fault model. In this example, the process could be testing for a slow-to-rise or slow-to-fall fault on any of the pins of the AND gate.

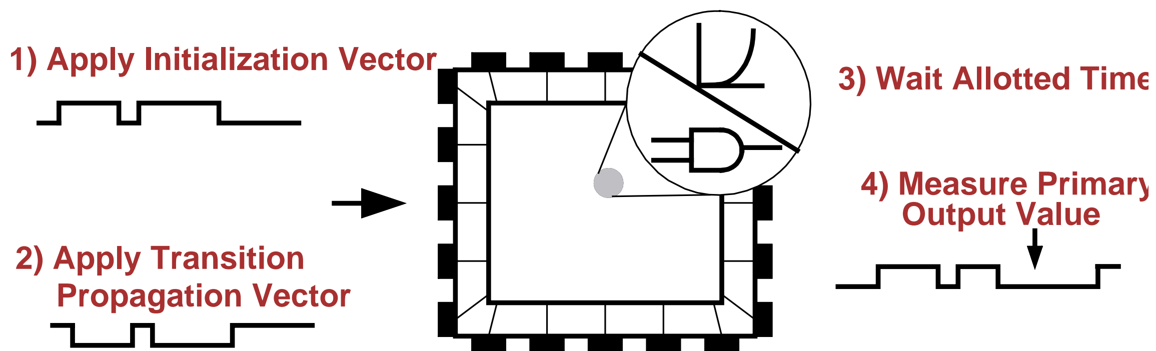


Figure 2-14. Transition Fault Detection Process

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

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

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

FlexTest Only - In FlexTest, a transition fault is modeled as a fault which causes a 1 cycle delay of rising or falling. In comparison, a stuck-at fault is modeled as a fault which causes infinite delay of rising or falling. The main difference between the transition fault model and the stuck-at fault model is their fault site behavior. Also, since it is more difficult to detect a transition fault than a stuck-at fault, the run time for a typical circuit may be slightly worse.

Related Commands

Set Fault Type - Specifies the fault model for which the tool develops or selects ATPG patterns. The -transition option for this command specifies the tool to develop or select ATPG patterns for the transition fault model.

At-Speed Testing and the Path Delay Fault Model

Path delay faults (supported only by FastScan) model defects in circuit paths. Unlike the other fault types, path delay faults do not have localized fault sites. Rather, they are associated with testing AC performance of specific paths (typically critical paths).

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

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

FastScan targets only a single path delay fault for each pattern it generates. Within the (ASCII) test pattern set, patterns that detect path delay faults include comments after the pattern statement identifying the path fault, type of detection, time and point of launch event, time and point of capture event, and the observation point.

For more information on generating path delay test sets “Creating a Path Delay Test Set (FastScan)” on page 6-92.

Fault Detection

Figure 2-15 shows the basic fault detection process.

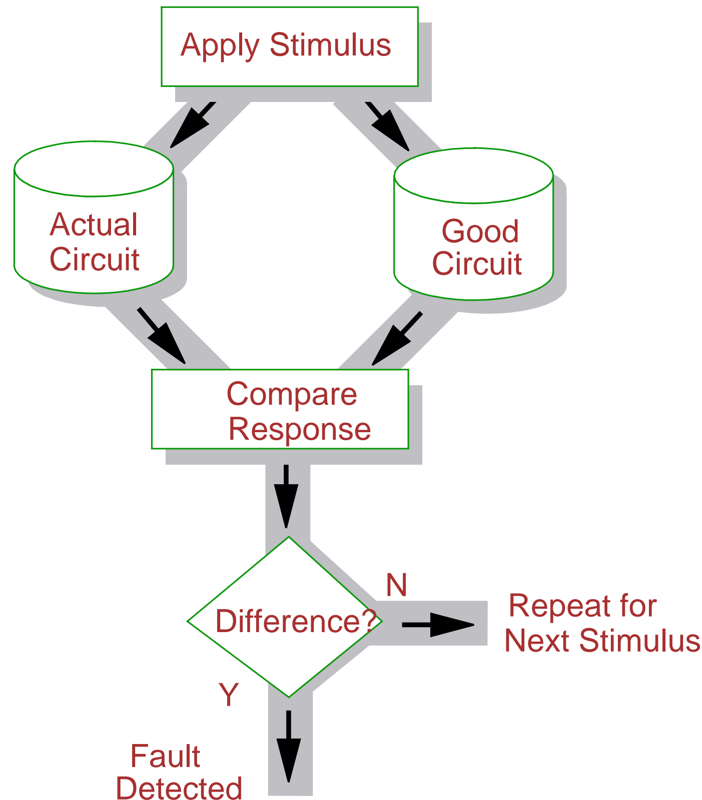


Figure 2-15. Fault Detection Process

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

The actual fault detection methods vary. One common approach is *path sensitization*. The path sensitization method, which is used by FastScan and FlexTest to detect stuck-at faults, starts at the fault site and tries to construct a vector to propagate the fault effect to a primary output. When successful, the tools create a stimulus set (a test pattern) to detect the fault. They attempt to do this for each fault in the circuit's fault universe. Figure 2-16 shows an example circuit for which path sensitization is appropriate.

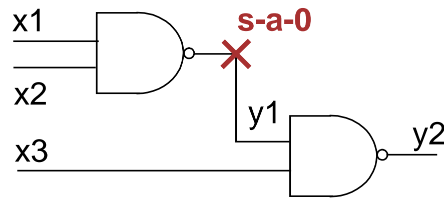


Figure 2-16. Path Sensitization Example

Figure 2-16 has a stuck-at-0 on line y1 as the target fault. The x1, x2, and x3 signals are the primary inputs, and y2 is the primary output. The path sensitization procedure for this example follows:

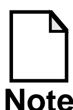
1. Find an input value that sets the fault site to the opposite of the desired value. In this case, the process needs to determine the input values necessary at x1 and/or x2 that set y1 to a 1, since the target fault is s-a-0. Setting x1 (or x2) to a 0 properly sets y1 to a 1.

2. Select a path to propagate the response of the fault site to a primary output. In this case, the fault response propagates to primary output y2.

3. Specify the input values (in addition to those specified in step 1) to enable detection at the primary output. In this case, in order to detect the fault at y1, the x3 input must be set to a 1.

Fault Classes

FastScan and FlexTest categorize faults into *fault classes*, based on how the faults were detected or why they could not be detected. Each fault class has a unique name and two character class code. When reporting faults, FastScan and FlexTest use either the class name or the class code to identify the fault class to which the fault belongs.


Note

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

Untestable

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

The following list discusses each of the untestable fault classes.

- **Unused (UU)**
  The unused fault class includes all faults on circuitry unconnected to any circuit observation point. Figure 2-17 shows the site of an unused fault.

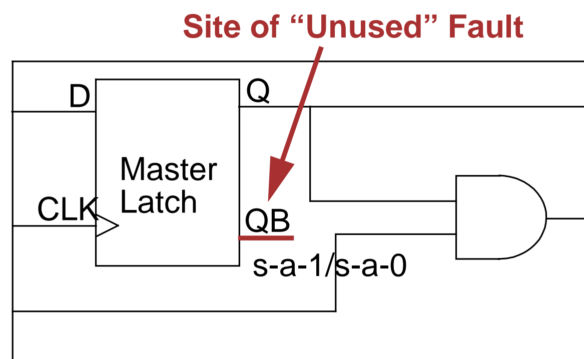


Figure 2-17. Example of "Unused" Fault in Circuitry

- **Tied (TI)**
  The tied fault class includes faults on gates where the point of the fault is tied to a value identical to the fault stuck value. The tied circuitry could be due to tied signals, or AND and OR gates with complementary inputs. Another possibility is exclusive-OR gates with common inputs. The tools will not use line holds (pins held at a constant logic value during test and set by the FastScan and FlexTest Add Pin Constraints command) to determine

tied circuitry. Line holds, or pin constraints, do result in ATPG_untestable faults. Figure 2-18 shows the site of a tied fault.

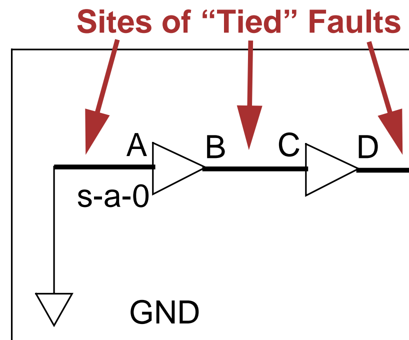


Figure 2-18. Example of "Tied" Fault in Circuitry

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

- **Blocked (BL)**
  The blocked fault class includes faults on circuitry for which tied logic blocks all paths to an observable point. This class also includes faults on selector lines of multiplexers that have identical data lines. Figure 2-19 shows the site of a blocked fault.

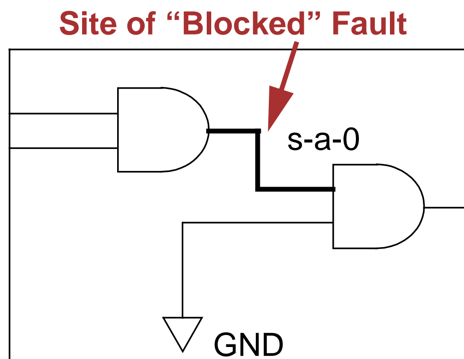
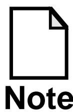


Figure 2-19. Example of "Blocked" Fault in Circuitry


Note

Tied faults and blocked faults can be equivalent faults.

- **Redundant (RE)**
  The redundant fault class includes faults the test generator considers undetectable. After the test pattern generator exhausts all patterns, it performs a special analysis to verify that the fault is undetectable under any conditions. Figure 2-20 shows the site of a redundant fault.

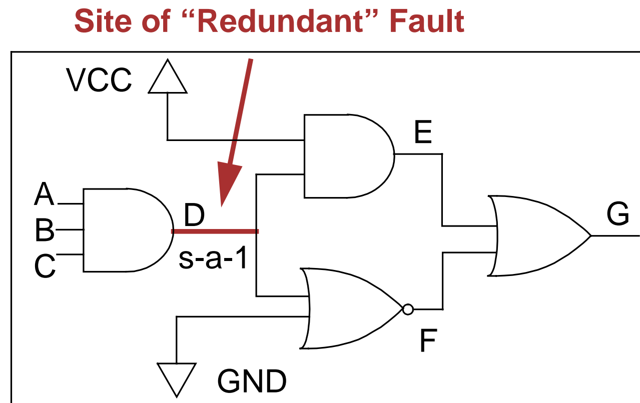


Figure 2-20. Example of “Redundant” Fault in Circuitry

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

Testable

Testable (TE) faults are all those faults that cannot be proven untestable. The testable fault classes include:

- **Detected (DT)**
  The detected fault class includes all faults that the ATPG process identifies as detected. The detected fault class contains two subclasses:
  - det_simulation (DS) - faults detected when the tool performs fault simulation.
  - det_implication (DI) - faults detected when the tool performs learning analysis.

  The det_implication class normally includes faults in the scan path circuitry, as well as faults that propagate ungated to the shift clock input of

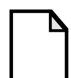
scan cells. The scan chain functional test, which detects a binary difference at an observation point, guarantees detection of these faults. FastScan and FlexTest both provide the Update Implication Detections command, which lets you specify additional types of faults for this category. Refer to the Update Implication Detections command description in the *FastScan and FlexTest Reference Manual*.

For path delay testing, detected faults include another category, detected_robust (DR), to categorize robust detected faults. "Path Delay Fault Detection" on page 6-92 describes this fault class in more detail.

- **Posdet (PD)**
  The posdet, or possible-detected, fault class includes all faults that fault simulation identifies as possible-detected but not hard detected. The posdet class contains two subclasses:

  - posdet_testable (PT) - potentially detectable posdet faults.
  - posdet_untestable (PU) - proven ATPG_untestable and hard undetectable posdet faults.

  A possible-detected fault results in a 0-X or 1-X difference at an observation point. By default, the calculations give 50% credit for Posdet faults.

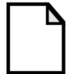

Note

If you use FlexTest and change the posdet credit to 0, the tool does not place any faults in this category.

- **Oscillatory (OS) -- FlexTest Only**
  The oscillatory fault class includes all faults with unstable circuit status for at least one test pattern. Oscillatory faults require a great deal of CPU time to calculate their circuit status. To maintain fault simulation performance, the tool drops oscillatory faults from the simulation. The tool calculates test coverage by classifying oscillatory faults as posdet faults.

The oscillatory fault class contains two subclasses:

- osc_untestable (OU) - ATPG_untestable oscillatory faults
- osc_testable (OT) - all other oscillatory faults.


Note

These faults may stabilize after a long simulation time.

- **Hypertrophic (HY) -- FlexTest Only**
  The hypertrophic fault class includes all faults whose effects spread extensively throughout the design, causing divergence from good state machine status for a large percentage of the design. Hypertrophic faults require a large amount of memory and CPU time to calculate their circuit status. To maintain fault simulation performance, the tool drops hypertrophic faults from the simulation. The tool calculates fault coverage, test coverage, and ATPG effectiveness by treating hypertrophic faults as posdet faults. The hypertrophic fault class contains two subclasses:

  - hyp_untestable (HU) - ATPG_untestable hypertrophic faults.
  - hyp_testable (HT) - all other hypertrophic faults.

  FlexTest defines hypertrophic faults with the internal state difference between each faulty machine and good machine. You can use the Set Hypertrophic Limit command to specify the percentage of internal state difference required to classify a fault as hypertrophic. The default difference is 30%.

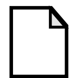
- **Uninitialized (UI) -- FlexTest Only**
  The uninitialized fault class includes faults for which the test generator is unable to:

  - find an initialization pattern that creates the opposite value of the faulty value at the fault pin.
  - prove the fault is tied.

In sequential circuits, these faults indicate that the tool cannot initialize portions of the circuit.

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

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


Note

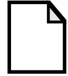
FastScan and FlexTest place AU faults in the testable category, counting the AU faults in the test coverage metrics. You should be aware that most other ATPG tools drop these faults from the calculations, and thus may inaccurately report higher test coverage.

- **Undetected (UD)**
  The undetected fault class includes undetected faults that cannot be proven untestable or ATPG_untestable. The undetected class contains two subclasses:

  - uncontrolled (UC) - undetected faults, which during pattern simulation, never achieve the value at the point of the fault required for fault detection--that is, they are uncontrollable.

  - unobserved (UO) - faults whose effects do not propagate to an observable point.

There is no guarantee the ATPG process will retain patterns that make a fault controllable.


Note

Uncontrolled and unobserved faults can be equivalent faults.

Fault Class Hierarchy

Fault classes are hierarchical. The highest level, Full, includes all faults in the fault list. Within Full, faults are classified into untestable and testable fault classes, and so on, in the manner shown in Figure 2-21.

Figure 2-21. Fault Class Hierarchy

```
1. Full (FU)
   1.1 TEstable (TE)
       a. DETEcted (DT)
          i.   DET_Simulation (DS)
          ii.  DET_Implication (DI)
          iii. DET_Robust (DR)--Path Delay Testing Only

       b. POSDET (PD)
          i.  POSDET_Untestable (PU)
          ii. POSDET_Testable (PT)
       c. OSCIllatory (OS)--FlexTest Only
          i.  OSC_Untestable (OU)
          ii. OSC_Testable (OT)
       d. HYPErtrophic (HY)--FlexTest Only
          i.  HYP_Untestable (HU)
          ii. HYP_Testable (HT)
       e. Uninitializable (UI)--FlexTest Only
       f. Atpg_untestable (AU)
       g. UNDetected (UD)
          i.  UNControlled (UC)
          ii. UNObserved (UO)
   1.2 UNTestable (UT)
       a. UNUsed (UU)
       b. TIed (TI)
       c. Blocked (BL)
       d. Redundant (RE)
```

For any given level of the hierarchy, FastScan and FlexTest assign a fault to one--and only one--class. If the tools can place a fault in more than one class of the same level, they place it in the class that occurs first in the list of fault classes.

Fault Reporting

When reporting faults, FastScan and FlexTest identify each fault by three ordered fields: the stuck value (0 or 1), the 2 character fault class code, and the pin pathname of the fault site. If the tools report uncollapsed faults, they display faults of a collapsed fault group together, with the representative fault first followed by the other members (with EQ fault codes).

Testability Calculations

Given the fault classes explained in the previous sections, FastScan and FlexTest make the following calculations:

- **Test Coverage**
  Test coverage, which is a measure of test quality, consists of the percentage of all testable faults that the test pattern set tests. Typically, this is the number of most concern when you consider the testability of your design. FastScan calculates test coverage using the formula:

  ```
       #DT + (#PD * posdet_credit)
  ---------------------------------------
                #testable
  ```

  FlexTest calculates it using the formula:

  ```
  #DT + (#PD + #OS + #HY) * posdet_credit)
  ---------------------------------------
                #testable
  ```

  In these formulas, posdet_credit is the user-selectable detection credit (the default is 50%) given to possible detected faults with the Set Possible Credit command.

- **Fault Coverage**
  Fault coverage consists of the percentage of all faults that the test pattern set tests--treating untestable faults the same as undetected faults. FastScan calculates fault coverage using the formula:

```
     #DT + (#PD * posdet_credit)
--------------------------------------
                #full
```

  FlexTest calculates it using the formula:

```
#DT + (#PD + #OS + #HY) * posdet_credit)
--------------------------------------
                #full
```

- **ATPG Effectiveness**
  ATPG effectiveness measures the ATPG tool's ability to either create a test for a fault, or prove that a test cannot be created for the fault under the restrictions placed on the tool. FastScan calculates ATPG effectiveness using the formula:

```
#DT + #UT + #AU + #PU +(#PT *posdet_credit)
-------------------------------------------
               #full
```

  FlexTest calculates it using the formula:

```
#DT+#UT+#AU+#UI+#PU+#OU+#HU+ ((#PT+#OT+#HT)*posdet_credit)
-----------------------------------------------------------
                    #full
```

Chapter 3
Understanding Common Tool Terminology and Concepts

Now that you understand the basic ideas behind DFT, scan design and ATPG, you can concentrate on the Mentor Graphics DFT tools and how they operate. DFTAdvisor, FastScan, and FlexTest not only work toward a common goal (to improve test coverage), they also share common terminology, internal processes, and other tool concepts, such as how to view the design and the scan circuitry. Figure 3-1 shows the range of subjects common to the three tools.

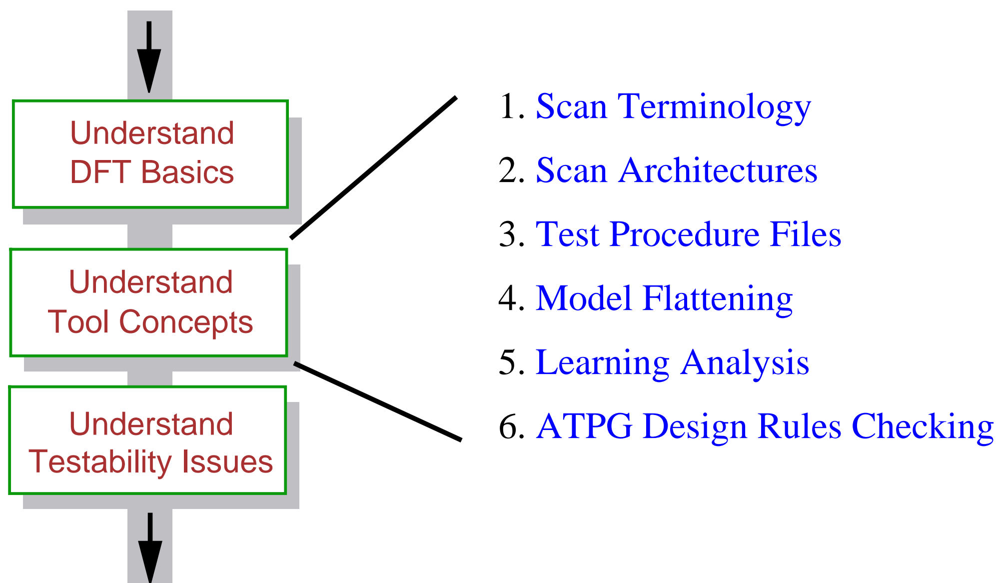


Figure 3-1. Common Tool Concepts

The following subsections discuss common terminology and concepts associated with scan insertion and ATPG using DFTAdvisor, FastScan, and FlexTest.

Scan Terminology

This section introduces the scan terminology common to DFTAdvisor, FastScan, and FlexTest.

Scan Cells

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

Because scan cell operation depends on an external procedure, scan cells are tightly linked to the notion of test procedure files. "Test Procedure Files" on page 3-11 discusses test procedure files in detail. Figure 3-2 illustrates the black box concept of a scan cell and its reliance on a test procedure.

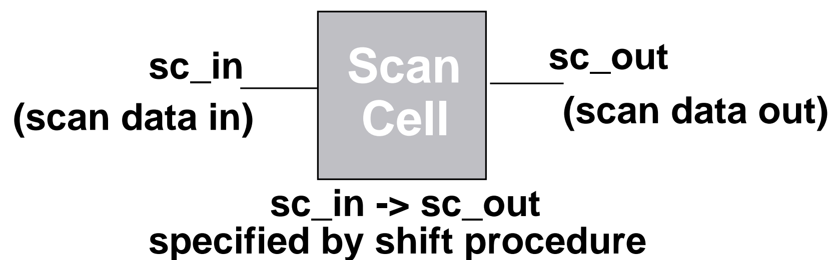


Figure 3-2. Generic Scan Cell

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

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

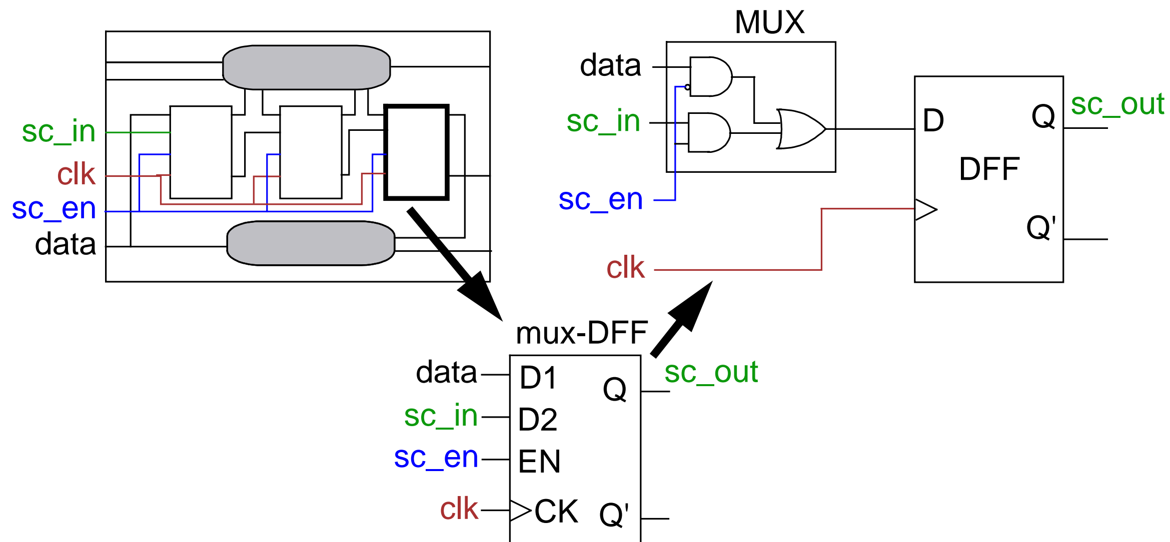


Figure 3-3. Generic Mux-DFF Scan Cell Implementation

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

Master Element

The *master element*, the primary memory element of a scan cell, captures data directly from the output of the previous scan cell. Each scan cell must contain one and only one master element. For example, Figure 3-3 shows a mux-DFF scan cell, which contains only a master element. However, scan cells can contain memory elements in addition to the master. Figures 3-4, 3-5, and 3-6 illustrate examples of master elements in a variety of other scan cells.

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

Slave Element

The *slave element*, an independently-clocked scan cell memory element, resides in the scan chain path. It cannot capture data directly from the previous scan cell. When used, it stores the output of the scan cell. The **shift** procedure both controls and observes the slave element. The value of the slave may be inverted relative to the master element. Figure 3-4 shows a slave element within a scan cell.

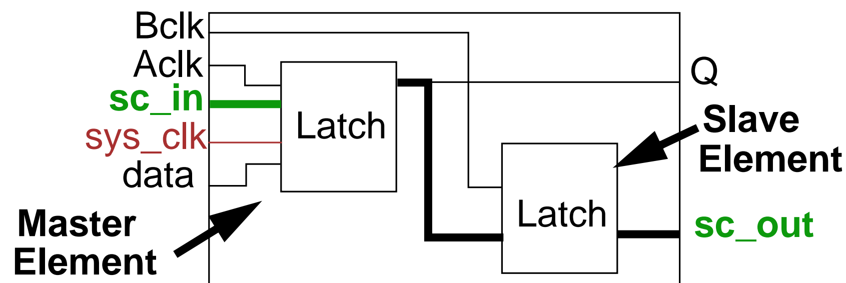


Figure 3-4. LSSD Master/Slave Element Example

In the example of Figure 3-4, Aclk controls scan data input. Activating Aclk, with sys_clk (which controls system data) held off, shifts scan data into the scan cell. Activating Bclk propagates scan data to the output.

Shadow Element

The *shadow element*, either dependently- or independently-clocked, resides outside the scan chain path. Figure 3-5 gives an example of a scan cell with an independently-clocked, non-observable shadow element with a non-inverted value.

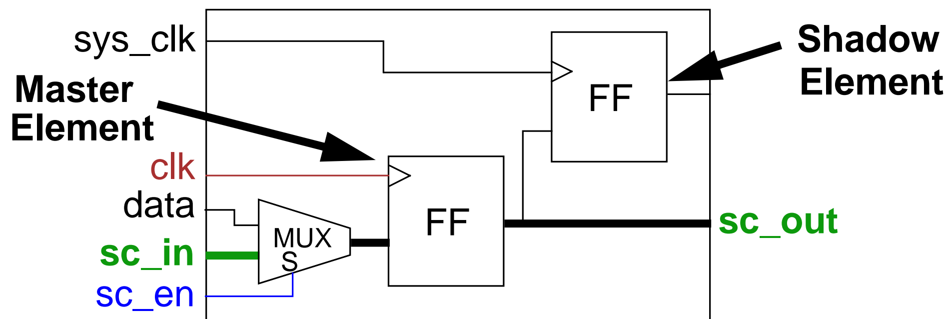


Figure 3-5. Mux-DFF/Shadow Element Example

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

Copy Element

The *copy element* is a memory element that lies in the scan chain path and can contain the same (or inverted) data as any associated independent memory element in the scan cell. Figure 3-6 gives an example of a copy element within a scan cell in which the master is the independent state element.

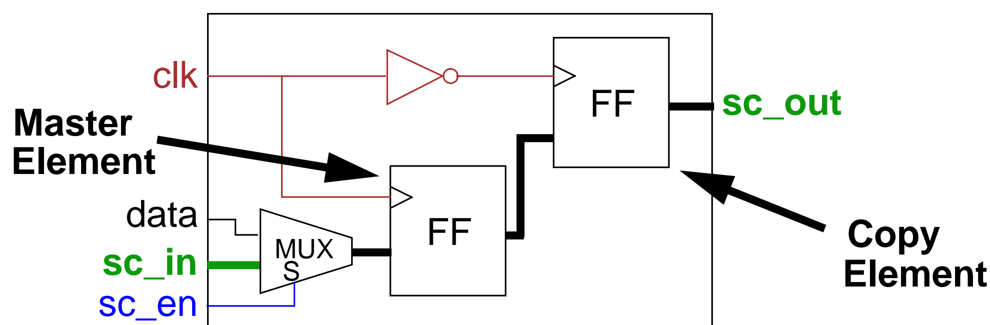


Figure 3-6. Mux-DFF/Copy Element Example

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

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

Extra Element

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

Scan Chains

A *scan chain* is a set of serially linked scan cells. Each scan chain contains an external input pin and an external output pin that provide access to the scan cells. Figure 3-7 shows a scan chain, with scan input "sc_in" and scan output "sc_out".

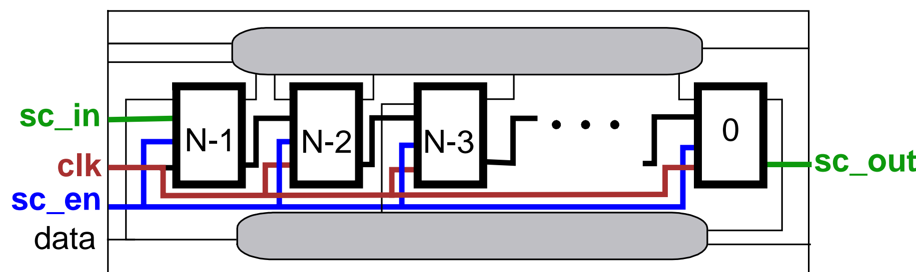


Figure 3-7. Generic Scan Chain

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

Scan Groups

A *scan chain group* is a set of scan chains that operate in parallel and share a common test procedure file. The test procedure file defines how to access the scan cells in all of the scan chains of the group. Normally, all of a circuit's scan chains operate in parallel and are thus in a single scan chain group. Scan chains in a scan group can also share a common scan input pin.

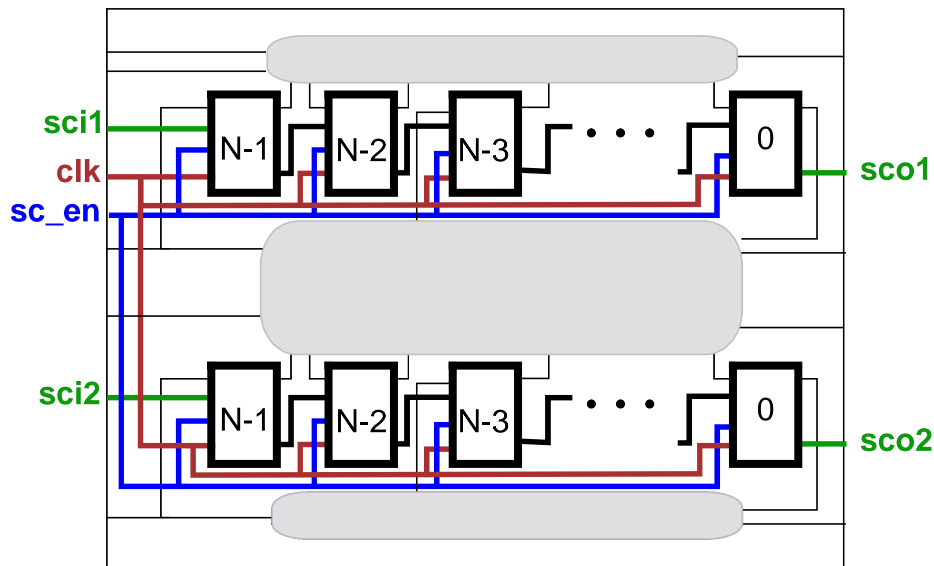


Figure 3-8. Generic Scan Group

You may have two clocks, A and B, each of which clock different scan chains. You can often clock, and therefore operate, the A and B chains concurrently, as shown in Figure 3-8. However, if two chains share a single scan_in pin, these chains cannot be operated in parallel. Regardless of operation, all defined scan chains in a circuit must be associated with a scan group. A scan group is a concept used by MGC DFT and ATPG tools.

Scan groups are a way to group scan chains based on operation. All scan chains in a group must be able to operate in parallel, which is normal for scan chains in a circuit. However when scan chains cannot operate in parallel, such as in the example above (sharing a common input pin), the operation of each must be specified separately. This means the scan chains belong to different scan groups

Scan Clocks

Scan clocks are external pins capable of capturing values into scan cell elements. Scan clocks include set and reset lines, as well as traditional clocks. Any pin defined as a clock can act as a capture clock during ATPG. Figure 3-9 shows a scan cell whose scan clock signals are shown in bold.

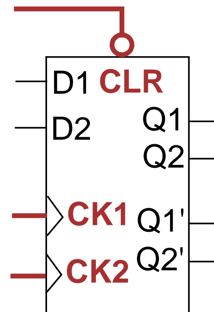


Figure 3-9. Scan Clocks Example

In addition to capturing data into scan cells, scan clocks, in their off state, ensure that the cells hold their data. Design rule checks ensure that clocks perform both functions. A clock's *off-state* is the primary input value that results in a scan element's clock input being at its inactive state (for latches) or state prior to a capturing transition (for edge-triggered devices). In the case of Figure 3-9, the off-state for the CLR signal is 1, and the off-states for CK1 and CK2 are both 0.

Scan Architectures

You can choose from a number of different scan types, or *scan architectures*. DFTAdvisor, the Mentor Graphics internal scan synthesis tool, supports the insertion of mux-DFF (mux-scan), clocked-scan, and LSSD architectures. Additionally, DFTAdvisor supports all standard scan types, or combinations thereof, in designs containing pre-existing scan circuitry. You can use the Set Scan Type command (see page 5-11) to specify the type of scan architecture you want inserted in your design.

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

clocked-scan ensures data hold for non-scan cells during scan loading. LSSD is most effective on latch-based designs.

The following subsections detail the mux-DFF, clocked-scan, and LSSD architectures.

Mux-DFF

A mux-DFF cell contains a single D flip-flop with a multiplexed input line that allows selection of either normal system data or scan data. Figure 3-10 shows the replacement of an original design flip-flop with mux-DFF circuitry.

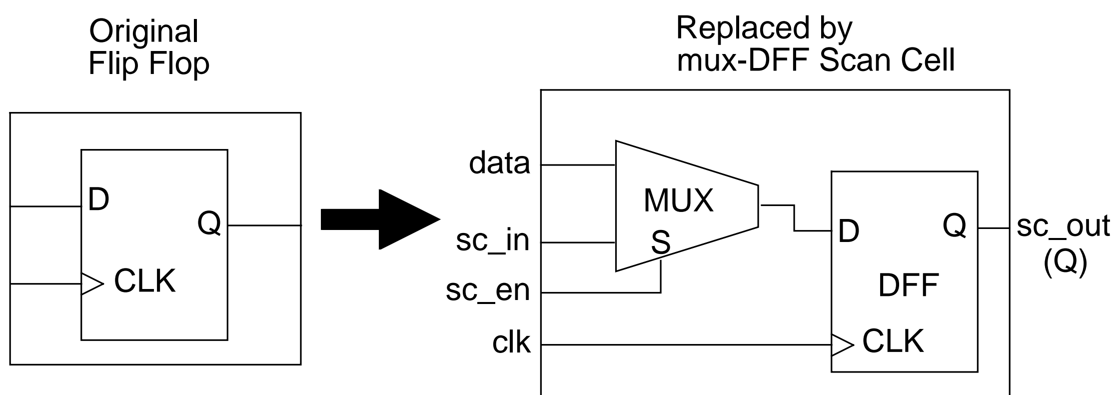


Figure 3-10. Mux-DFF Replacement

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

Clocked-Scan

The clocked-scan architecture is very similar to the mux-DFF architecture, but uses a dedicated test clock to shift in scan data instead of a multiplexer. Figure 3-11 shows an original design flip-flop replaced with clocked-scan circuitry.

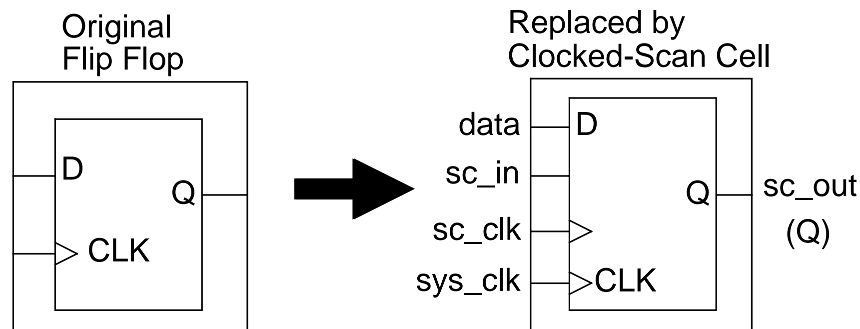


Figure 3-11. Clocked-Scan Replacement

In normal operation, the system clock (sys_clk) clocks system data (data) into the circuit and through to the output (Q). In scan mode, the scan clock (s_clk) clocks scan input data (sc_in) into the circuit and through to the output (sc_out).

LSSD

LSSD, or Level-Sensitive Scan Design, uses three independent clocks to capture data into the two polarity hold latches contained within the cell. Figure 3-12 shows the replacement of an original design latch with LSSD circuitry.

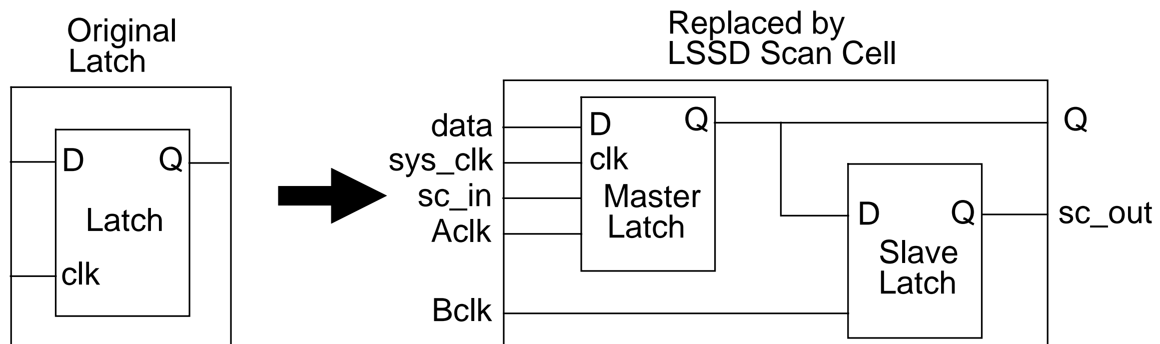


Figure 3-12. LSSD Replacement

In normal mode, the master latch captures system data (data) using the system clock (sys_clk) and sends it to the normal system output (Q). In test mode, the two clocks (Aclk and Bclk) trigger the shifting of test data through both master and slave latches to the scan output (sc_out).

There are several varieties of the LSSD architecture, including single latch, double latch, and clocked LSSD.

Test Procedure Files

Test procedure files contain event-based procedures that tell FastScan or FlexTest how to operate the scan structures within a design. You specify scan circuitry operation using previously defined scan clocks and other control signals. Thus, in order to utilize the scan circuitry in your design, you must define the scan circuitry to the tool and provide a test procedure file to describe its operation. The design rules checking (DRC) process, which occurs when you exit from Setup mode, performs extensive checking to ensure the scan circuitry operates correctly. Once the scan circuitry operation (specified by the test procedure file) passes DRC, other processes of FastScan and FlexTest assume the scan circuitry works properly.

After it inserts scan circuitry, DFTAdvisor can create test procedure files that you can use with FastScan or FlexTest. If your design contains scan circuitry, and if you have not already created a test procedure file, either by hand or by using DFTAdvisor, you must do so before running ATPG with FastScan and FlexTest. The following subsections describe the syntax and rules of test procedure files, give examples for the various types of scan architectures, and outline the checking that determines whether the circuitry is operating correctly.

For more information on the new test procedure file format, see the "Enhanced Procedure File" chapter of the *Design-for-Test: Common Resources Manual*.

Test Procedure File Rules

The test procedure file must conform to the following rules:

- Each scan group needs a unique test procedure file. You associate the test procedure file with the scan group when you specify the Add Scan Group command.
- Each statement must be on a single line.
- Text following // is a comment and is ignored.
- You can include blank lines.

- All statements must be within the **procedure** and **end** statements.
- You define a procedure type (with the exception of the **seq_transparent** procedure) only once in a test procedure file.
- You can only have a single **test_setup** procedure, even if you define multiple scan groups for your design.
- For each procedure, time begins at 0, and you must list all statements in chronological order; that is, the time in one statement cannot be less than the time in a previous statement. Statements with identical times execute simultaneously. Events must stabilize before the next time period.
- For all test procedures, a time period with any clock pin forced on may only contain clock pins forced on. The time periods before and after this “on” state, must contain clock pins in their off states.

Test Procedure Statements

The following sections describes the statements you can use in a test procedure.

procedure <procedure_type> =

This statement marks the beginning of any procedure definition. The procedure_type is a keyword defining the type of procedure, such as **shift**, **load_unload**, and so on.

You can specify multiple **seq_transparent** and **clock** procedures in a test procedure file. Thus, these procedure types require explicit procedure names for each procedure you define. The syntax for these procedure statements is as follows:

procedure <procedure_type> <procedure_name> =

end;

This statement indicates the end of a procedure definition.

force <pin_pathname> <value> <time>;

This statement forces a value of 0, 1, X, or Z on the specified pin at the given time. The pin names you specify must be valid pin pathnames for primary inputs, and may optionally begin with a "/" or be contained in double-quotes.

apply <shift|shadow_control> <#times> <time>;

This statement tells the tool to apply the selected procedure the selected number of times starting at the specified time. You must use the **apply shift** statement at least once in the **load_unload** procedure. For the **apply shift** statement, you should enter a proper #times parameter, otherwise you will get a warning message. You must enter the **apply shadow_control** statement, if required, immediately after the **apply shift** procedure statement, and you must set the #times argument to 1.

force_sci <time>;

This statement indicates the time in the **shift** procedure at which the tool places values on the scan chain inputs. This statement implements scan cell controllability.

force_sci_equiv <time>;

This statement acts the same as the **force_sci** statement, except that it also forces all pins equivalent to the scan input pins. Using this statement places the complement value on the associated differential pin of a scan input during scan loading. This statement is necessary because the test procedures do not consider pin equivalence relationships (those specified with Add Pin Equivalence).

measure_sco <time>;

This statement indicates when in the **shift** procedure to measure scan output values, thus implementing scan cell observability.

initialize <instance_name> [0|1];

This statement lets you initialize a memory element. This statement is particularly useful for initializing the finite state machine in the TAP controller of boundary

scan circuitry, when the TAP does not contain the TRST signal. Once set to a binary state, the TCK and TMS pins can place the finite state machine in a desired state. If not set, these pins remain at X.

You are restricted to specifying this statement only at time 0 of the **test_setup** procedure. A rules violation occurs if you use this command at any time other than 0, or if no instance is found with the specified name. If you do not specify a value, the tool chooses a random value to assign to all latches and flip-flops with the specified instance name.

condition <pin_pathname> <value>;

You use this statement at the beginning of a **seq_transparent** procedure to identify the necessary scan cell states (conditions) to establish transparency in non-scan cells. For more information on transparency, refer to "FastScan Handling of Non-Scan Cells" in the *Scan and ATPG Process Guide*. You identify the scan cell by the pin pathname associated with the output of its state element. The path from the defined pin to the scan cell must only contain buffers and inverters. The value argument sets the value at the specified pin_pathname, which may be inverted relative to the associated scan cell value

restore_pis <time>;

You use the **restore_pis** statement at the end of a **seq_transparent** procedure to return primary inputs to their original states (prior to this procedure's execution).

restore_bidis <time>;

You use the **restore_bidis** statement at the end of a **clock** procedure to return bidirectional pins to their original states (prior to this procedure's execution).

break <time>;

You use the **break** statement to explicitly initiate a new test cycle at the specified time. The test pattern data formatter must convert the event-based test procedures to cycles before it can write out patterns. By default, it uses an algorithm that places as many events as possible in each test cycle. The **break** statement gives you some control over how the formatter maps test procedure events into test

cycles. For more information on the event-to-cycle mapping algorithm, refer to "Converting Test Procedures to Test Cycles" on page 7-4.

break_repeat <time>;

The **break_repeat** statement is identical to the **break** statement, except that it specifies to start a new test cycle at each multiple of the specified time.

The Procedures

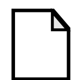
The following list describes the test procedures that can comprise a test procedure file:

Test_Setup (optional)

This procedure, which may only contain **force**, **period**, **break**, and **break_repeat** statements, sets non-scan elements to the desired states for the **load_unload** procedure. You may use this procedure only once for all scan groups, and it appears only once at the beginning of the test pattern set.

This procedure is particularly useful for initializing boundary scan circuitry. For an example using this procedure to set up boundary scan circuitry, refer to "Generating Patterns for a Boundary Scan Circuit" on page 6-103.

If a scan out pin is bidirectional, you must force its value to the Z state (indicating it is operating in "output" mode) to properly sensitize the scan chain.


Note

If you run ATPG after setting pin constraints, you should also constrain these pins within the **test_setup** procedure. If you do not properly constrain the pins prior to the end of the **test_setup** procedure the tools will automatically do this for you. However, as a result of the tools automatically handling this, you may encounter timing violations later on in the process.

Shift (required)

This procedure describes how to shift data one position down the scan chain, by toggling the clock(s), forcing the scan input, and strobing the scan output. Figure 3-13 shows the data flow process for the **shift** procedure.

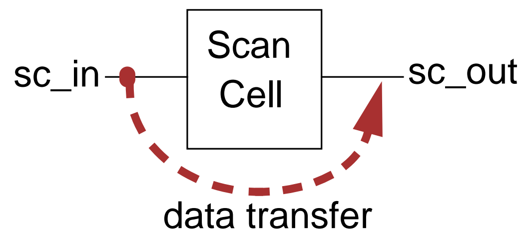


Figure 3-13. Shift Procedure

Within this procedure, you must include **force** commands, the **force_sci** or **force_sci_equiv** command, and the **measure_sco** command. The times at which you apply the **force_sci** and **measure_sco** commands must allow proper operation of the **load_unload** process.

The following list shows examples of the **shift** procedure for both the mux-DFF and LSSD architectures:

- **Mux-DFF**

```
procedure shift =
   // force scan chain input at time 0
   force_sci                 0;
   // measure scan chain output at time 0
   measure_sco               0;
   // pulse the clock
   force scan_clk      1  1;
   force scan_clk      0  2;
   // a unit of dead time for stability
   period                    3;
end;
```

- **LSSD**

```
procedure shift =
    // force scan chain input at time 0
    force_sci            0;
    // measure scan chain output at time 0
    measure_sco          0;
    // pulse master clock
    force  scan_mclk  1  1;
    force  scan_mclk  0  2;
    // pulse slave clock
    force  scan_sclk  1  3;
    force  scan_sclk  0  4;
    // add one dead time period for signal stability
    period               5;
end;
```

The following example shows a **shift** procedure, which specifies the events required to shift scan data into and out of the scan chain:

```
procedure shift =
  force_sci             20;
  measure_sco           40;
  force cp.0     1     100;
  force cp.0     0     200;
  period               400;
end;
```

Figure 3-14 graphically displays the waveforms for the clock pin, the scan-in pin, and the scan-out pin derived from the defined **shift** procedure timing information. This timing diagram shows one scan chain shift cycle, assuming the time unit is 1ns.

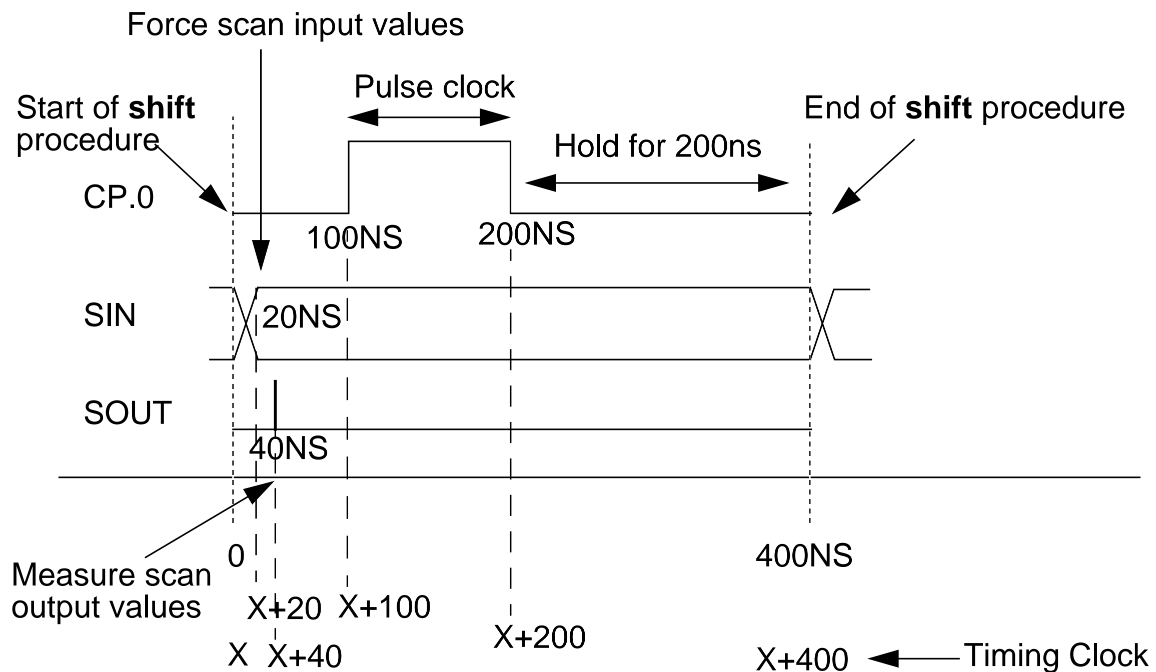


Figure 3-14. Timing Diagram for Shift Procedure

The procedure contains four scan events: it forces scan input values at 20ns, strobes (or measures) scan output values at 40ns, pulses the capture clock cp.0 (turning it on at 100ns and off at 200ns), and holds the state of the last event until the procedure finishes at 400ns.

A timing clock monitors when each significant event occurs. If the timing clock is at X when the **shift** procedure begins, the timing clock assigns those four events with time values X+20, X+40, X+100, and X+200. When the **shift** procedure finishes, the timing clock advances to X+400. The shift cycle ending time becomes the starting time for the next shift cycle.

Load_Unload (required)

This key procedure describes how to load and unload the scan chains in the scan group. To load the scan chain, you must force the circuit into the appropriate state for the start of the shift sequence. This includes forcing clocks, resets, RAM write control signals, and any other signals that need to be at their off states for scan chain loading. Figure 3-15 shows the data flow for the **load_unload** procedure.

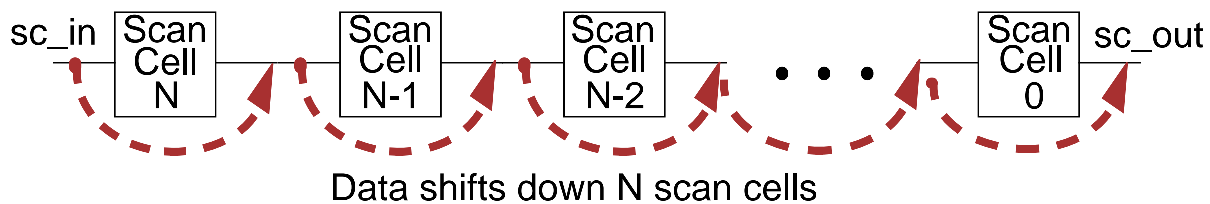


Figure 3-15. Load_Unload Procedure

If the scan out pin is bidirectional, you must force its value to the Z state (indicating it is operating in "output" mode) to properly sensitize the scan chain. If there is a scan enable signal, you must force it on to enable the scan chain prior to the shift. You then use the **apply shift** statement to specify the number of shift cycles (which equals the number of scan elements in the chain). You must also include the **apply** command if you have optionally included the **shadow_control** procedure (which if used, immediately follows the **shift** procedure).

The following list includes the basic statements in the **load_unload** procedure:

- **Mux-DFF**

```
procedure load_unload =
         //force clocks off at time 1
         force  RST  0  0;
         force  CLK  0  0;
         //activate scanning mode
         force  scan_en  1  0;
         //shift data thru each of 7 cells
         apply shift  7  1;
end;
```

- **LSSD**

```
procedure load_unload =
    // force all clocks off at time 0
    force  rst        0  0;
    force  clk        0  0;
    force  scan_sclk  0  0;
    force  scan_mclk  0  0;
    // apply shift procedure 7 times starting at time 1
    apply shift       7  1;
end;
```

The timing for the **shift** procedure is generally straightforward. The timing for the **load_unload** procedure, however, is slightly more complex. The **load_unload** procedure contains the **apply** statement. The time specified for an **apply** statement is only relative to the procedure in which it resides. Therefore, the total time specified for a **load_unload** procedure does not include the time required to execute the embedded **apply** commands.

For example, examine the following **load_unload** procedure.

```
procedure load_unload =
   force m0.0      0      0;
   force m1.0      0      0;
   force cp.0      0      0;
   force cp.1      0      0;
   apply shift     1    100;
   period               300;
end;
```

The **load_unload** procedure specifies the period is 300ns. However, the **load_unload** procedure includes an **apply** statement that executes one **shift** procedure. The **shift** procedure requires an additional 400ns. Thus, the **load_unload** procedure actually requires a total time of 700ns, as shown in Figure 3-16.

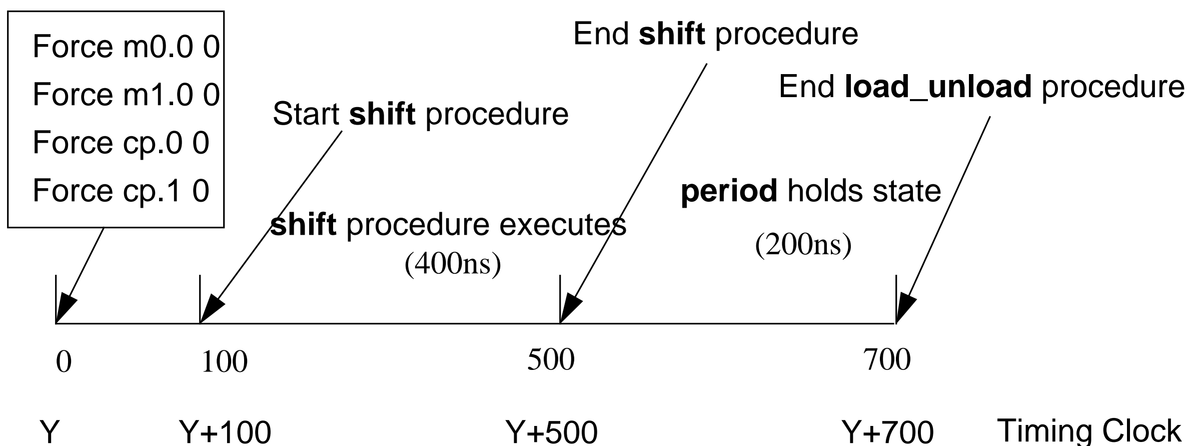


Figure 3-16. Timing Diagram for Load_Unload Procedure

Within the **load_unload** procedure, the **shift** procedure starts at 100ns, executes for 400ns, and ends at 500ns. The **load_unload** procedure then waits another 200ns before finishing.

As with the **shift** procedure, the timing clock determines the event times for the **load_unload** procedure. If the timing clock is at Y when the **load_unload** procedure begins, the first four events happen at time Y. When the **apply** event executes, the timing clock advances to Y+100, which is when the **shift** procedure begins. As mentioned previously, the **shift** procedure requires 400 time units. Therefore, when the **apply** event finishes the timing clock reads Y+500.

Because it is the last event in the **load_unload** procedure, the **apply** event determines how long the state should hold before the next event. The state must hold for the difference between the total time (300) and the start time for the **apply** event (100). Thus, the hold time after finishing the apply event is equal to 200 (=300-100). Thus, Y+700 becomes the real ending time for the **load_unload** procedure.

Shadow_Control (optional)

This procedure, which may only contain **force** commands and the **period** statement, describes how to load the contents of a scan cell into the associated shadow. If you use this procedure, you must also apply the **shadow_control** command in the **load_unload** procedure. This procedure must not disturb the contents of any of the scan cells. Figure 3-17 shows the data flow for the **shadow_control** procedure.

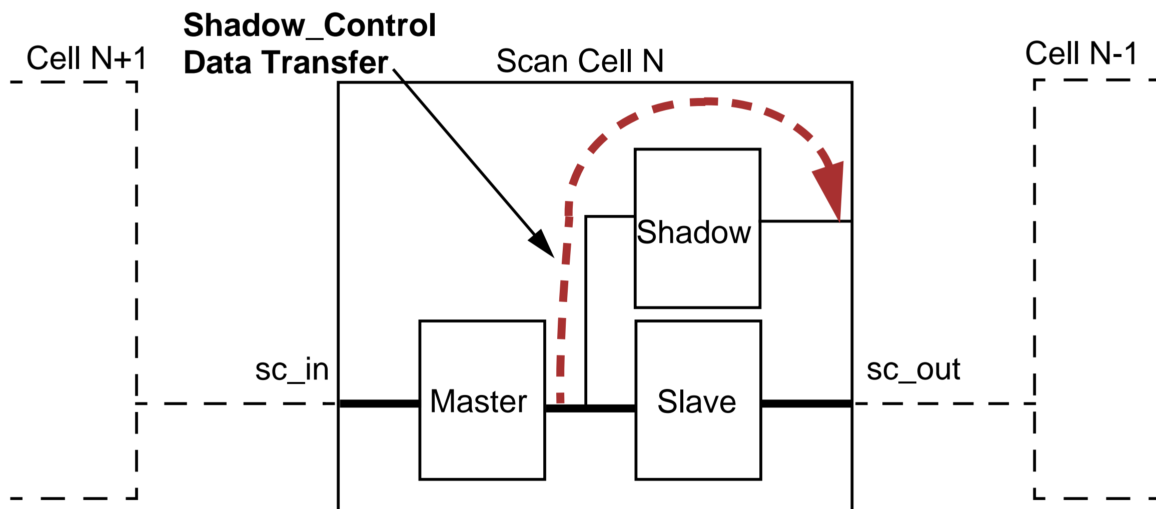


Figure 3-17. Shadow_Control Procedure

Master_Observe (sometimes required)

This procedure, which may only contain **force** commands and the **period** statement, describes how to place the contents of a master into the output of its scan cell, where you can observe it by using the unload operation. Figure 3-18 shows the data flow for the **master_observe** procedure.

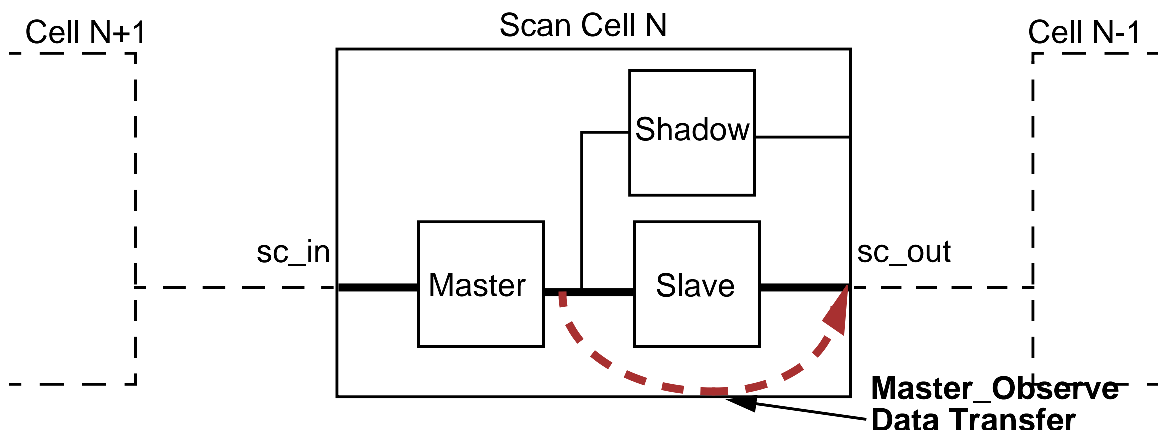


Figure 3-18. Master_Observe Procedure

You do not need to use this procedure if the master element's output *is* the output of the scan cell. The D1 rules ensures this procedure does not disturb master memory element's contents. You can override this requirement by changing the

D1 rule handling. The following example shows a **master_observe** procedure for the LSSD architecture:

```
//LSSD architecture example
procedure master_observe =
    // force all clocks off at time 0
    force  scan_sclk  0  0;
    force  scan_mclk  0  0;
    force  rst        0  0;
    force  clk        0  0;
    // force slave clock on at time 1
    force  scan_sclk  1  1;
    // force slave clock off at time 2
    force  scan_sclk  0  2;
    // add some time for stability
    period               3;
end;
```

Shadow_Observe (optional)

This procedure, which may only contain **force** commands and the **period** statement, describes how to place the contents of a shadow into the output of its scan cell, assuming the circuitry of the scan cell allows the transfer of data in this way. Once the data is at the scan cell output, you can observe it by applying the unload command. This procedure lets the shadow be used as an observation point in the design. Figure 3-19 shows the data flow of the **shadow_observe** procedure.

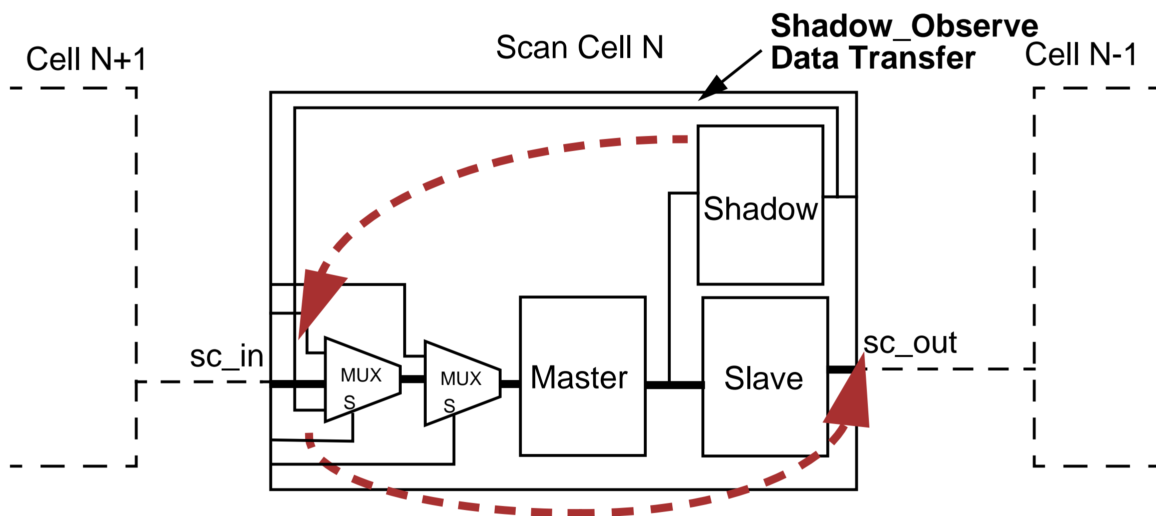


Figure 3-19. Shadow_Observe Procedure

Seq_Transparent (FastScan-only, optional)

This procedure identifies how to make non-scan cells and RAM read ports functionally behave transparently. This procedure activates the clock inputs of non-scan cell inputs, thus pulsing data through the cells “transparently”. All clocks must be at their off-states and constrained pins at their constrained states before applying the **seq_transparent** procedure, and the procedure must immediately follow a force of all the primary inputs. For more information on the sequential transparent operation, refer to “Sequential Transparent Patterns” on page 6-13.

You can use multiple clock cycles to create the sequential transparent conditions. You may define up to 32 different **seq_transparent** procedures within a test procedure file. When simulation mode is set to RAM_sequential, each **force_all** statement in the pattern file can use any of the possible **seq_transparent** procedure choices. FastScan treats non-scan state elements that cannot utilize the sequential transparent procedures as tie-X gates.

There may be occasions when you would want to use **seq_transparent** procedures when the design contains no scan chains. In this case, you would use the Add Scan Group command, specifying the name “dummy” for the chain name and the test procedure filename (which contains only the **seq_transparent** procedure). Refer to the Add Scan Groups command reference page in the *FastScan and FlexTest Reference Manual* for more details. Figure 3-20 shows some circuitry that could benefit from a **seq_transparent** procedure.

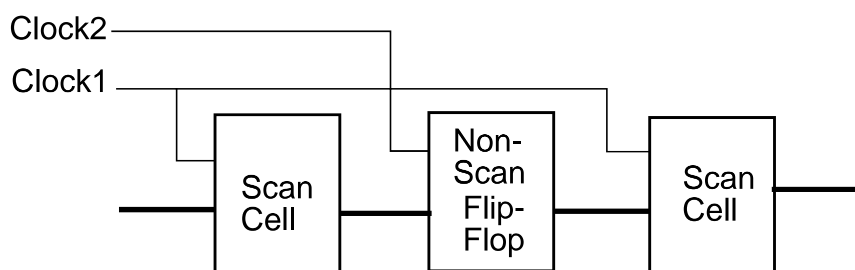


Figure 3-20. Sequential Transparent Circuitry Example

The basic stimuli necessary to create transparent behavior for the non-scan flip-flop shown in Figure 3-20 is:

force all clocks off
force non-scan cell clock Clock2 on
force non-scan cell clock Clock2 off
restore primary inputs to original values

In more complex situations, you may need to set primary inputs to certain values, place conditions on scan cells, pulse multiple clocks, and so on.

You can use the Report Seq_transparent Procedures command to display data defined by the **seq_transparent** procedures. Refer to the Report Seq_transparent Procedures command reference page in the *FastScan and FlexTest Reference Manual* for more details.

Clock (FastScan-only, optional)

This procedure provides flexible clock handling during the test procedures. Using **clock** procedures, instead of pulsing a single clock during a capture cycle, you can serially exercise multiple clocks and force non-clock pins that do not affect captured data.

The following example shows a clock procedure used to operate two clocks in sequence:

```
procedure clock clock_proc1 =
   force clk1 1 1;  //pulse first clock
   force clk1 0 2;
   force clk2 1 3;  //pulse second clock
   force clk2 0 4;
end;
```

Clock procedures must abide by the following rules:

- The procedure must activate at least one clock.
- If you define multiple clock procedures, only one of these procedures can activate a specific clock.
- The procedure events cannot violate pin constraints or equivalence conditions.

- The procedure can only force non-clock pins if they do not affect data captured into state elements whose clocks may activate later in the procedure.
- Multiple clocks that activate serially cannot logically interact.
- The procedure must follow all standard rules for both clock and non-clock pin usage.
- Each clock procedure must have a unique name.
- If a state element can change state during the procedure, the element must be stable when all clocks are off and pins are constrained.
- *Transparent_capture* cells are stable state elements that can capture data during the procedure and whose new data can affect other state elements later in the procedure. Design rules D10 and D11 ensure that these cells do not connect to state elements that capture old data or propagate data to primary outputs. Refer to "Scan Cell Data Rules" in the *Design-for-Test Common Resources Manual* for more information on these checks.
- The procedure must set all bidirectional pins to their input mode prior to executing the **restore_bidis** statement.

Skew_Load (optional)

This optional procedure propagates the output value of the preceding scan cell into the master memory element of the current cell (without changing the slave), for all scan cells. Using only **force** and **period** commands, this procedure defines how to apply an additional pulse of the master shift clock after the scan chains are loaded. Figure 3-21 shows the data flow of the **skew_load** procedure.

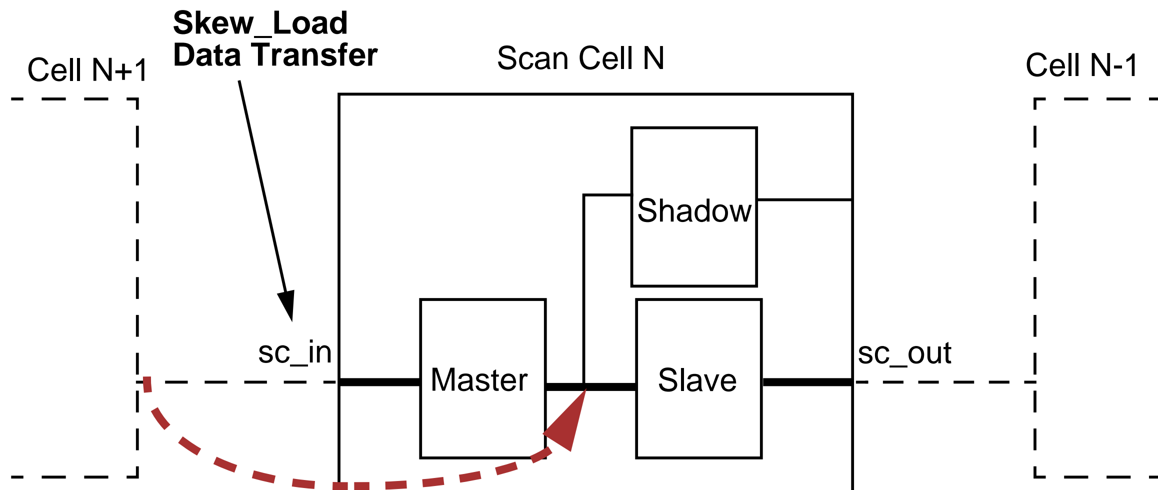


Figure 3-21. Skew_Load Procedure

Figure 3-22 shows where you apply the **skew_load** procedure and the **master_observe** procedure within the basic scan pattern events.

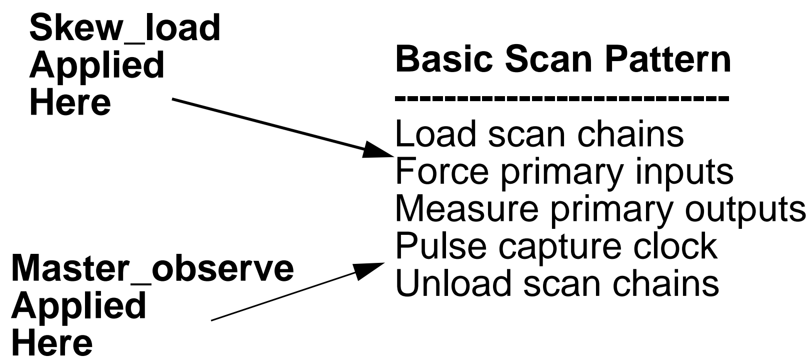


Figure 3-22. Skew_load applied within Pattern

Scan Chain Operation Checking

As mentioned previously, FastScan and FlexTest do not care what the scan architecture of the design looks like. What matters is that the operations specified in the test procedure file work properly to transfer data to and from the scan chains. Thus, test procedure files are put through a variety of checks during design rules checking. Besides checking the test procedure file for syntax and basic rules violations, the design rules checker specifically:

- Simulates the **test_setup** patterns, initializing the memory elements to the values necessary for simulation of the **load_unload** procedure.
- Simulates, for each scan group, a portion of the **load_unload** procedure, which includes a single application of the **shift** procedure.
- Performs a backtrace for each scan chain to identify all the scan cells in the scan chain. The trace begins at the scan chain output and continues tracing through gates that, during the shift procedure's time period, have a single propagable input. You cannot specify a memory element in more than one scan chain. The trace of a scan chain must end at the defined scan chain input pin. During this trace, the rules checker identifies and classifies the scan cell memory elements in the scan path, taking inversion into consideration.
- Checks each scan cell copy element to ensure it captures the value of its associated memory element.
- Checks the **force_sci** and **measure_sco** statements to ensure they occur at the proper time.
- Performs a backward trace (traceback) on all memory elements not in a scan path to determine if they capture data from a scan cell during the **shift** or **shadow_control** procedures. The checker classifies those that capture scan cell data as shadows and includes them as part of their corresponding scan cell.
- If a **master_observe** procedure is present, checks to ensure that all master values successfully propagate to the output of their scan cells. If no

master_observe procedure is present, the rules checker checks the observability of all master elements.

- If a **shadow_observe** procedure is present, simulates the procedure to identify observable shadow elements.
- Checks the **load_unload**, **master_observe**, and **shadow_observe** procedures to ensure they do not disturb the contents of scan cells except during the **shift** procedure.
- Checks the test procedures of each scan group to ensure they do not disturb the scan cell contents of other scan chain groups.
- Analyzes the remaining non-scan memory elements using the final simulated values of the last **load_unload** procedure. FastScan and FlexTest handle non-scan cells differently. Refer to "Non-Scan Cell Handling" on page 4-19 for details.
- Issues a warning message if a bus contention occurs during any application of any test procedure, identifying the location of the bus contention.

Model Flattening

To work properly, FastScan, FlexTest, and DFTAdvisor must use their own internal representations of the design. The tools create these internal design models by flattening the model and replacing the design cells in the netlist (described in the library) with their own primitives. The tools flatten the model when you initially attempt to exit the Setup mode, just prior to design rules checking. FastScan and FlexTest also provide the Flatten Model command, which allows flattening of the design model while still in Setup mode.

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

For a list of the specific DFTAdvisor commands that cause flattening, refer to the Set System Mode command page in the *DFTAdvisor Reference Manual*. For FastScan and FlexTest related commands, see below:

Related Commands

Flatten Model - creates a primitive gate simulation representation of the design.

Report Flatten Rules - displays either a summary of all the flattening rule violations or the data for a specific violation.

Set Flatten Handling - specifies how the tool handles flattening violations.

Understanding Design Object Naming

DFTAdvisor, FastScan, and FlexTest use special terminology to describe different objects in the design hierarchy. The following list describes the most common:

Instance — a specific occurrence of a library model or functional block in the design.

Hierarchical instance — an instance that contains additional instances and/or gates underneath it.

Module — a VHDL or Verilog functional block (module) that can be repeated multiple times. Each occurrence of the module is a hierarchical instance.

The Flattening Process

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

ATPG. Figure 3-23 shows an example of circuitry containing and AND-OR-Invert cell and an AND gate, before flattening.

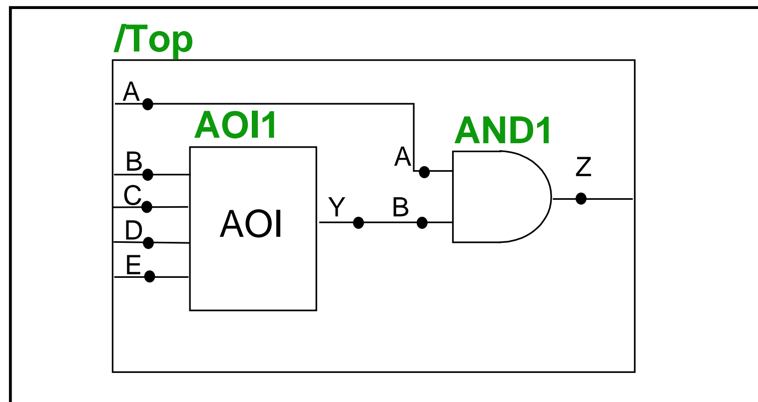


Figure 3-23. Design Before Flattening

Figure 3-24 shows this same design once it has been flattened.

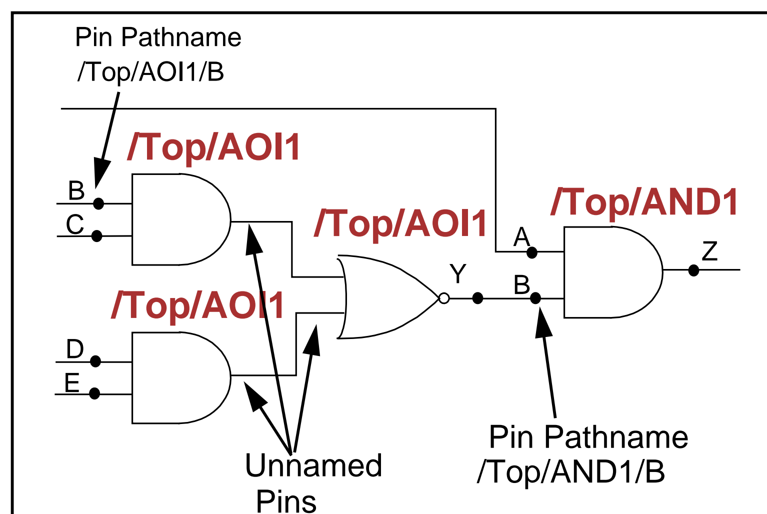


Figure 3-24. Design After Flattening

After flattening, only naming preserves the design hierarchy; that is, the flattened netlist maintains the hierarchy through instance naming. Figures 3-23 and 3-24 show this hierarchy preservation. */Top* is the name of the hierarchy's top level. The simulation primitives (two AND gates and a NOR gate) represent the flattened instance *AOI1* within */Top*. Each of these flattened gates retains the original design hierarchy in its naming--in this case, */Top/AOI1*.

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

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

Simulation Primitives of the Flattened Model

DFTAdvisor, FastScan, and FlexTest select from a number of simulation primitives when they create the flattened circuitry. The simulation primitives are multiple-input (zero to four), single-output gates, except for the RAM, ROM, LA, and DFF primitives. The following list describes these simulation primitives:

- **PI, PO** - primary inputs are gates with no inputs and a single output, while primary outputs are gates with a single input and no fanout.
- **BUF** - a single-input gate that passes the values 0, 1, or X through to the output.
- **ZVAL** - a single-input gate that acts as a buffer unless Z is the input value. When a Z is the input value, the output is an X. You can modify this behavior with the Set Z Handling command.
- **INV** - a single-input gate whose output value is the opposite of the input value. The INV gate cannot accept a Z input value.

- **AND, NAND** - multiple-input gates (two to four) that act as standard AND and NAND gates.
- **OR, NOR** - multiple-input (two to four) gates that act as standard OR and NOR gates.
- **XOR, XNOR** - 2-input gates that act as XOR and XNOR gates, except that when either input is an X, the output is an X.
- **MUX** - a 2x1 mux gate whose pins are order dependent, as shown in Figure 3-25.

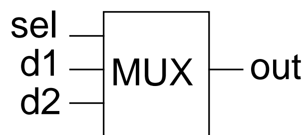
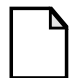


Figure 3-25. 2x1 MUX Example

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


Note

FlexTest uses a different pin naming and ordering scheme, which is the same ordering as the _mux library primitive; that is, in0, in1, and cnt. In this scheme, cnt=0 selects in0 data and cnt=1 selects in1 data.

- **LA, DFF** - state elements, whose order dependent inputs include set, reset, and clock/data pairs, as shown in Figure 3-26.

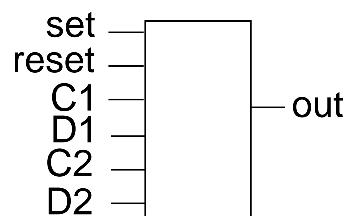


Figure 3-26. LA, DFF Example

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

- **TLA, STLA, STFF** - special types of learned gates that act as, and pass the design rule checks for, transparent latch, sequential transparent latch, or sequential transparent flip-flop. These gates propagate values without holding state.
- **TIE0, TIE1, TIEX, TIEZ** - zero-input, single-output gates that represent the effect of a signal tied to ground or power, or a pin or state element constrained to a specific value (0,1,X, or Z). The rules checker may also determine that state elements exhibit tied behavior and will then replace them with the appropriate tie gates.
- **TSD, TSH** - a 2-input gate that acts as a tri-state™ driver, as shown in Figure 3-27.

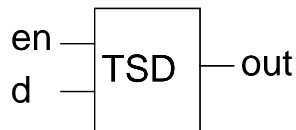


Figure 3-27. TSD, TSH Example

When en=1, out=d. When en=0, out=Z. The data line, d, cannot be a Z. FastScan uses the TSD gate, while FlexTest uses the TSH gate for the same purpose.

- **SW, NMOS** - a 2-input gate that acts like a tri-state driver but can also propagate a Z from input to output. FastScan uses the SW gate, while FlexTest uses the NMOS gate for the same purpose.
- **BUS** - a multiple-input (up to four) gate whose drivers must include at least one TSD or SW gate. If you bus more than four tri-state drivers together, the tool creates cascaded BUS gates. The last bus gate in the cascade is considered the dominant bus gate.

- **WIRE** - a multiple-input gate that differs from a bus in that none of its drivers are tri-statable.

- **PBUS, SWBUS** - a 2-input pull bus gate, for use when you combine strong bus and weak bus signals together, as shown in Figure 3-28.

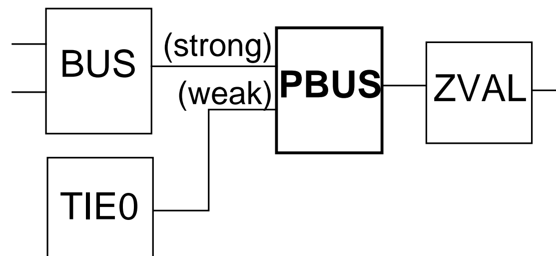


Figure 3-28. PBUS, SWBUS Example

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

- **ZHOLD** - a single-input buskeeper gate (see page 3-45 for more information on buskeepers) associated with a tri-state network that exhibits sequential behavior. If the input is a binary value, the gate acts as a buffer. If the input value is a Z, the output depends on the gate's hold capability. There are three ZHOLD gate types, each with a different hold capability:

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

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

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

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

- **XDET, ZDET** - a single-input gate used to translate EDDM QuickPart Tables to model certain types of behavior. For the XDET gate, an X on the

input results in a 1 on the output. Any other input value results in a 0 on the output. For the ZDET gate, a Z on the input results in a 1 on the output. Any other input value results in a 0 on the output.

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

Learning Analysis

After design flattening, FastScan and FlexTest perform extensive analysis on the design to learn behavior that may be useful for intelligent decision making in later processes, such as fault simulation and ATPG. You have the ability to turn learning analysis off, which may be desirable if you do not want to perform ATPG during the session. For more information on turning learning analysis off, refer to the Set Static Learning command or the Set Sequential Learning command reference pages in the *FastScan and FlexTest Reference Manual.*

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

Equivalence Relationships

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

have the same values in the good machine. The example in Figure 3-29 shows an example of two of these equivalence points within some circuitry.

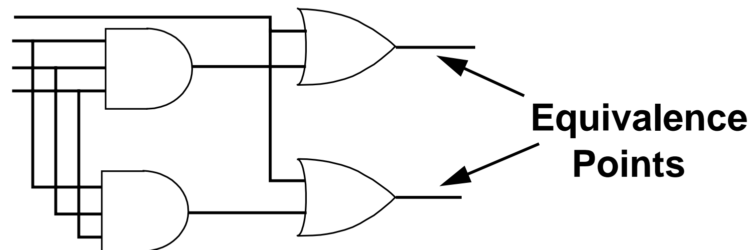


Figure 3-29. Equivalence Relationship Example

Logic Behavior

During logic behavior analysis, simulation determines a circuit's functional behavior. For example, Figure 3-30 shows some circuitry that, according to the analysis, acts as an inverter.

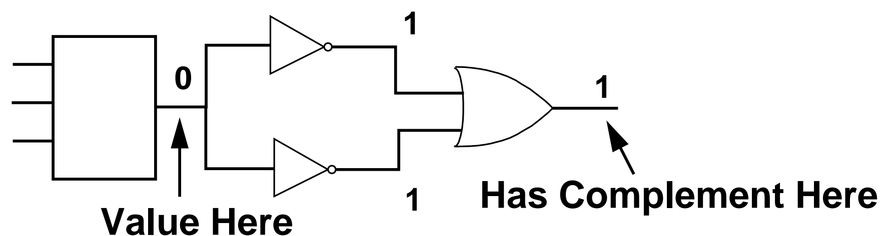


Figure 3-30. Example of Learned Logic Behavior

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

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

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

Implied Relationships

This type of analysis consists of contrapositive relation learning, or learning implications, to determine that one value implies another. This learning analysis simulates nearly every gate in the design, attempting to learn every relationship possible. Figure 3-31 shows the implied learning the analysis derives from a piece of circuitry.

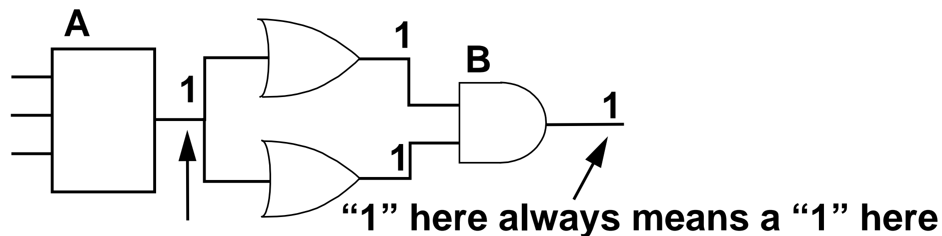


Figure 3-31. Example of Implied Relationship Learning

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

Forbidden Relationships

During forbidden relationship analysis, which is restricted to bus gates, simulation determines that one gate cannot be at a certain value if another gate is at a certain value. Figure 3-32 shows an example of such behavior.

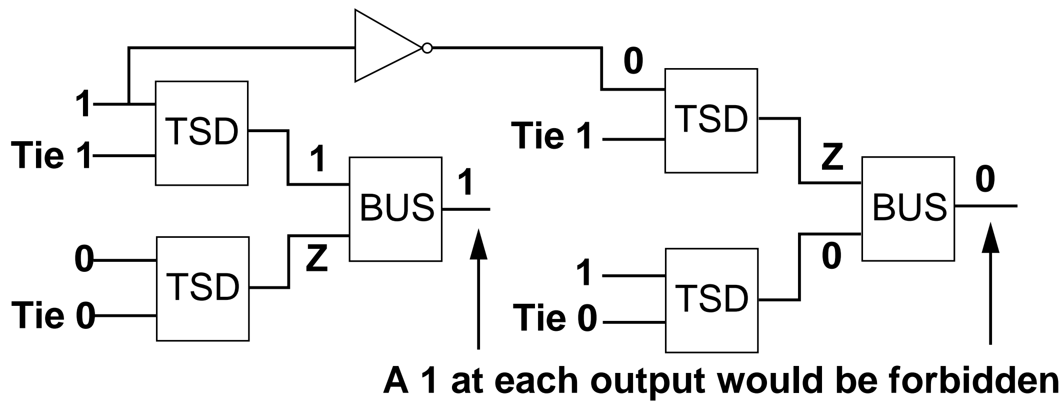


Figure 3-32. Forbidden Relationship Example

Dominance Relationships

During dominance relationship analysis, simulation determines which gates are dominators. If all the fanouts of a gate go to a second gate, the second gate is the dominator of the first. Figure 3-33 shows an example of this relationship.

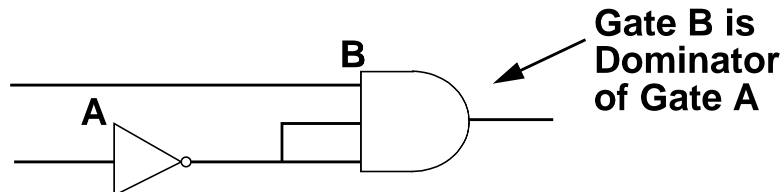


Figure 3-33. Dominance Relationship Example

ATPG Design Rules Checking

DFTAdvisor, FastScan, and FlexTest perform design rules checking after design flattening. While not all of the tools perform the exact same checks, design rules checking generally consists of the following processes, done in the order shown:

1. General Rules Checking
2. Procedure Rules Checking
3. Bus Mutual Exclusivity Analysis
4. Scan Chain Tracing
5. Shadow Latch Identification
6. Data Rules Checking
7. Transparent Latch Identification
8. Clock Rules Checking
9. RAM Rules Checking
10. Bus Keeper Analysis
11. Extra Rules Checking
12. Scannability Rules Checking
13. BIST Rules Checking
14. Constrained/Forbidden/Block Value Calculations

General Rules Checking

General rules checking searches for very-high-level problems in the information defined for the design. For example, it checks to ensure the scan circuitry, clock, and RAM definitions all make sense. General rules violations are errors and you cannot change their handling. The "General Rules" section in the *Design-for-Test Common Resources Manual* describes the general rules in detail.

Procedure Rules Checking

Procedure rules checking examines the test procedure file. These checks look for parsing or syntax errors and ensure adherence to each procedure's rules.

Procedure rules violations are errors and you cannot change their handling. The "Procedure Rules" section in the *Design-for-Test Common Resources Manual* describes the procedure rules in detail.

Bus Mutual Exclusivity Analysis

Buses in circuitry can cause two main problems for ATPG: 1) bus contention during ATPG, and 2) testing stuck-at faults on tri-state drivers of buses. This section addresses the first concern, that ATPG must place buses in a non-contending state. For information on how to handle testing of tri-state devices, see "Tri-State Devices" on page 4-18.

Figure 3-34 shows a bus system that can have contention.

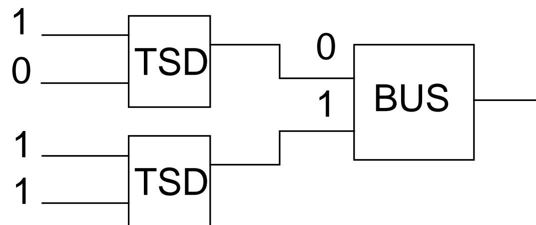


Figure 3-34. Bus Contention Example

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

Buses in a Z-state network can be classified as dominant or non-dominant and strong or weak. Weak buses and pull buses are allowed to have contention. Thus the process only analyzes strong, dominant buses, examining all drivers of these gates and performing full ATPG analysis of all combinations of two drivers being

forced to opposite values. Figure 3-35 demonstrates this process on a simple bus system.

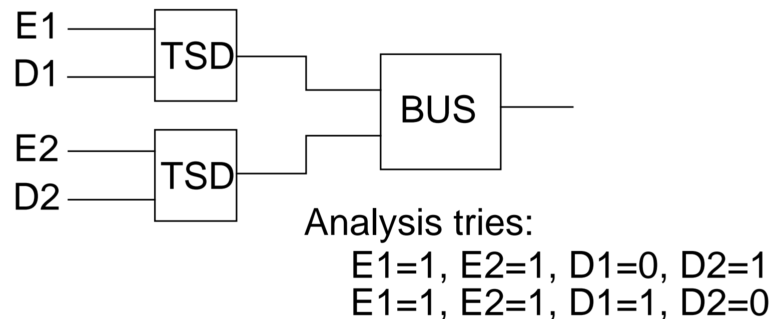


Figure 3-35. Bus Contention Analysis

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

You should resolve bus mutual-exclusivity before ATPG. Extra rules E4, E7, E9, E10, E11, E12, and E13 perform bus analysis and contention checking. Refer to "Extra Rules" in the *Design-for-Test Common Resources Manual* for more information on these bus checking rules.

Scan Chain Tracing

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

Trace rules violations are either errors or warnings, and for most rules you cannot change the handling. The "Scan Chain Trace Rules" section in the *Design-for-Test Common Resources Manual* describes the trace rules in detail.

Shadow Latch Identification

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

If the circuitry allows, you can also make a shadow an observation point by writing a **shadow_observe** test procedure. The section entitled "Shadow Element" on page 3-4 discusses shadows in more detail.

The DRC process identifies shadow latches under the following conditions:

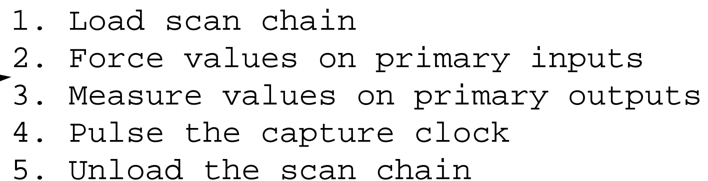
1. The element must not be part of an already identified scan cell.
2. Plus any one of the following:
    - At the time the clock to the shadow latch is active, there must be a single sensitized path from the data input of the shadow latch up to the output of a scan latch. Additionally the final shift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

Data Rules Checking

Data rules checking ensures the proper transfer of data within the scan chain. Data rules violations are either errors or warnings, however, you can change the handling. The "Scan Cell Data Rules" section in the *Design-for-Test Common Resources Manual* describes the data rules in detail.

Transparent Latch Identification

Transparent latches are latches that can propagate values but do not hold state. A basic scan pattern contains the following events:



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

Clock Rules Checking

After the scan chain trace, clock rules checking is the next most important analysis. Clock rules checks ensure data stability and capturability in the chain. Clock rules violations are either errors or warnings, however, you can change the handling. The "Clock Rules" section in the *Design-for-Test Common Resources Manual* describes the clock rules in detail.

RAM Rules Checking

RAM rules checking ensures consistency with the defined RAM information and the chosen testing mode. RAM rules violations are all warnings, however, you can change their handling. The "RAM Rules" section in the *Design-for-Test Common Resources Manual* describes the RAM rules in detail.

Bus Keeper Analysis

Bus keepers model the ability of an undriven bus to retain its previous binary state. You specify bus keeper modeling with a **bus_keeper** attribute in the model definition. When you use the **bus_keeper** attribute, the tool uses a ZHOLD gate to model the bus keeper behavior during design flattening. In this situation, the design's simulation model becomes that shown in Figure 3-36:

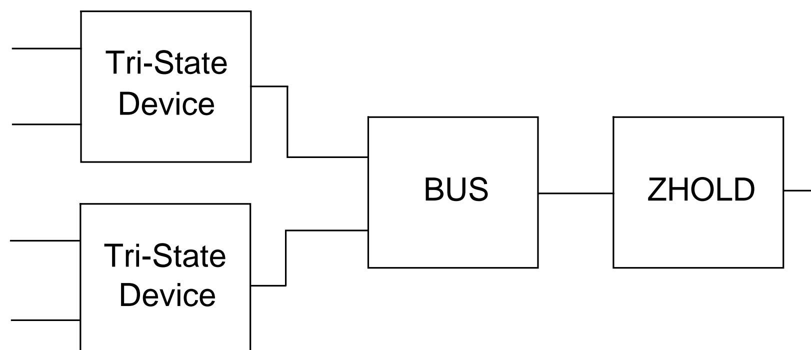
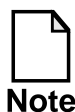


Figure 3-36. Simulation Model with Bus Keeper

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

During rules checking, if a design contains ZHOLD gates, messages indicate when ZHOLD checking begins, the number and type of ZHOLD gates, the number of ZHOLD gates connected to clock, write, and read lines, and the number of ZHOLD gates set to a binary value during the clock off-state condition.


Note

Only FastScan requires this type of analysis, because of the way it "flattens" or simulates a number of events in a single operation.

For information on the bus_keeper model attribute, refer to "Inout and Output Attributes" in the *Design-for-Test Common Resources Manual*.

Extra Rules Checking

Excluding rule E10, which performs bus mutual-exclusivity checking, most extra rules checks do not have an impact on DFTAdvisor, FastScan, or FlexTest processes. However, they may be useful for enforcing certain design rules. By default, most extra rules violations are set to ignore, which means they are not even checked during DRC. However, you may change the handling. For more information, refer to "Extra Rules" in the *Design-for-Test Common Resources Manual* for more information.

Scannability Rules Checking

Each design contains a certain number of memory elements. DFTAdvisor examines all these elements and performs scannability checking on them, which consists mainly of the audits performed by rules S1, S2, and S3. Scannability rules are all warnings, and you cannot change their handling. For more information, refer to "Scannability Rules" in the *Design-for-Test Common Resources Manual*.

BIST Rules Checking

BIST rules checking, a FastScan-only check, ensures that defined BIST circuitry information is correct and that the tool can apply the BIST patterns to the circuit. BIST rules violations are all warnings or errors, and you cannot change their handling. The "BIST Rules" section in the *Design-for-Test Common Resources Manual* describes the BIST rules in detail.

Constrained/Forbidden/Block Value Calculations

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

Figure 3-37 gives an example of a tie value gate that constrains some surrounding circuitry.

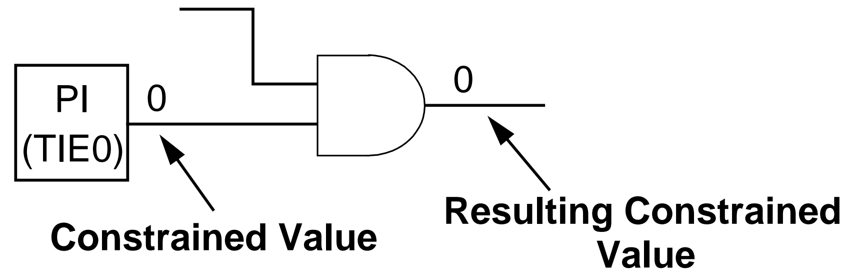


Figure 3-37. Constrained Values in Circuitry

Figure 3-38 gives an example of a tied gate, and the resulting forbidden values of the surrounding circuitry.

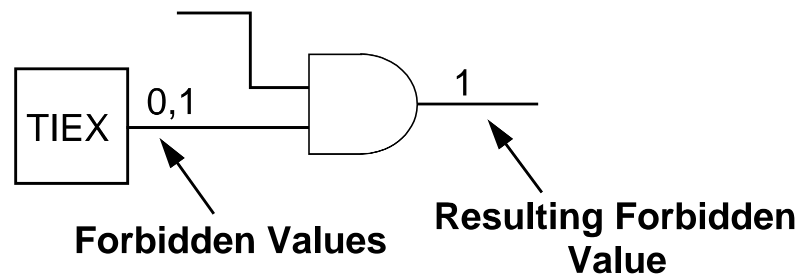


Figure 3-38. Forbidden Values in Circuitry

Figure 3-39 gives an example of a tied gate that blocks fault effects in the surrounding circuitry.

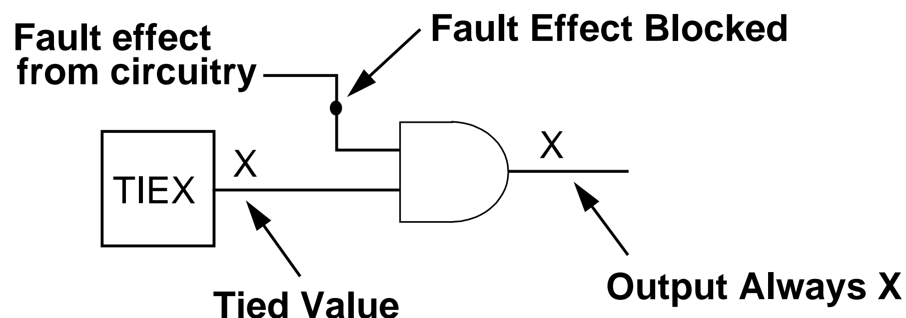


Figure 3-39. Blocked Values in Circuitry

Chapter 4
Understanding Testability Issues

Testability naturally varies from design to design. Some features and design styles make a design difficult, if not impossible, to test, while others enhance a design's testability. Figure 4-1 shows the testability issues this section discusses.

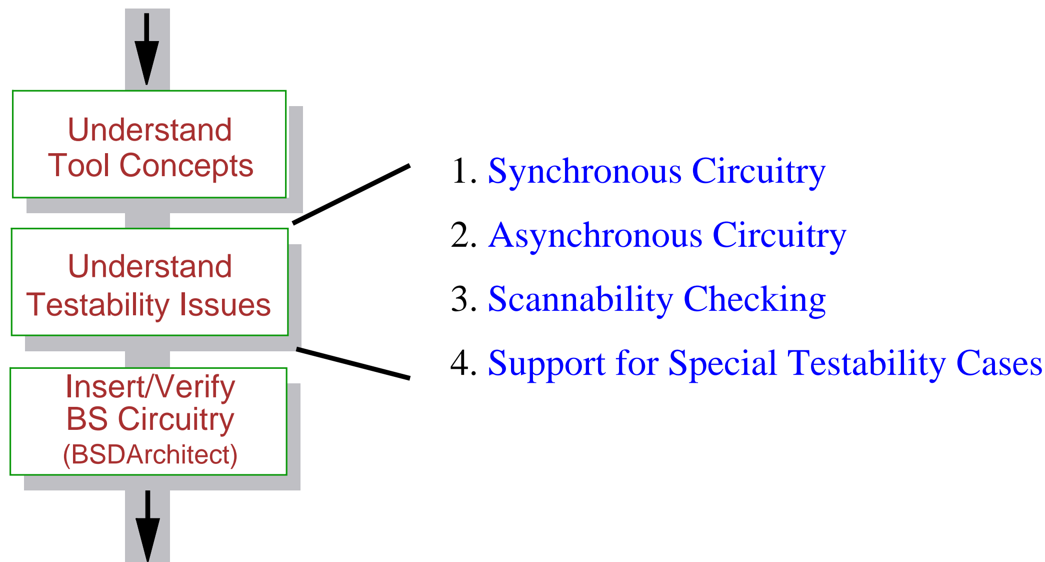


Figure 4-1. Testability Issues

The following subsections discuss these design features and describe their effect on the design's testability.

Synchronous Circuitry

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

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

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

Synchronous Design Techniques

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

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

- Do not use logic to delay signals.
- Do not assume logic delays are longer than routing delays.

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

Asynchronous Circuitry

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

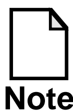
Scannability Checking

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

1. When all clocks are off, all clock inputs (including set and reset inputs) of the sequential element must be in their inactive state (initial state of a capturing transition). This prevents disturbance of the scan chain data before application of the test pattern at the primary input. If the sequential element does not pass this check, its scan values could become unstable when the test tool applies primary input values. This checking is a modification of rule C1. For more information on this rule, refer to "C1 (Clock Rule #1)" in the *Design-for-Test Common Resources Manual*.

2. Each clock input (*not* including set and reset inputs) of the sequential element must be capable of capturing data when a single clock primary input goes active while all other clocks are inactive. This rule ensures that this particular storage element can capture system data. If the sequential element does not meet this rule, some loss of test coverage could result. This checking is a modification of rule C7. For more information on this rule, refer to "C7 (Clock Rule #7)" in the *Design-for-Test Common Resources Manual*.

When a sequential element passes these checks, it becomes a *scan candidate*, meaning that DFTAdvisor can insert its scan equivalent into the scan chain. However, even if the element fails to pass one of these checks, it may still be possible to convert the element to scan. In many cases, you can add additional logic, called *test logic*, to the design to remedy the situation. For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-12.


Note

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

Scannability Checking of Latches

By default, DFTAdvisor performs scannability checking on all flip-flops and latches. When latches do not pass scannability checks, DFTAdvisor considers them non-scan elements and then classifies them into one of the categories explained in "Non-Scan Cell Handling" on page 4-19. However, if you want DFTAdvisor to perform transparency checking on the non-scan latches, you must turn off checking of rule D6 prior to scannability checking. For more information on this rule, refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual*.

Support for Special Testability Cases

The following subsections explain certain design features that can pose design testability problems and describe how Mentor Graphics DFT tools handle these situations.

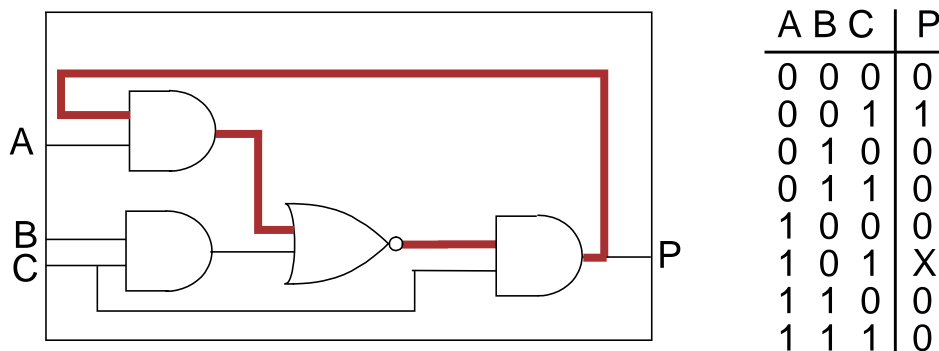
Feedback Loops

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

The tools, FastScan, FlexTest, and DFTAdvisor, all provide some common loop analysis and handling. However, loop treatment can vary depending on the tool. The following subsections discuss the treatment of structural combinational and structural sequential loops.

Structural Combinational Loops and Loop-Cutting Methods

Figure 4-2 shows an example of a structural combinational loop. Notice that the A=1, B=0, C=1 state causes unknown (oscillatory) behavior, which poses a testability problem.



A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	X
1	1	0	0
1	1	1	0

Figure 4-2. Structural Combinational Loop Example

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

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

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

   Figure 4-3 shows a loop with a constrained primary input value that blocks the loop's feedback effects.

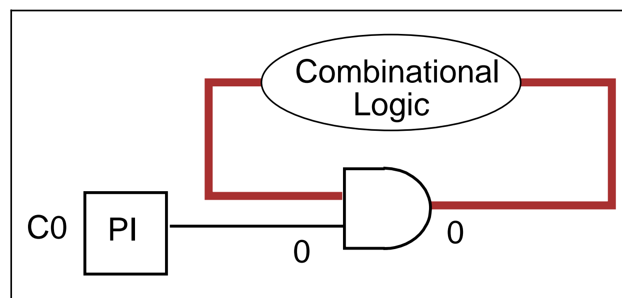


Figure 4-3. Loop Naturally-Blocked by Constant Value

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

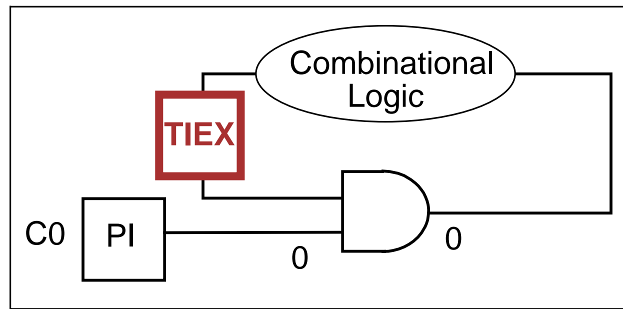
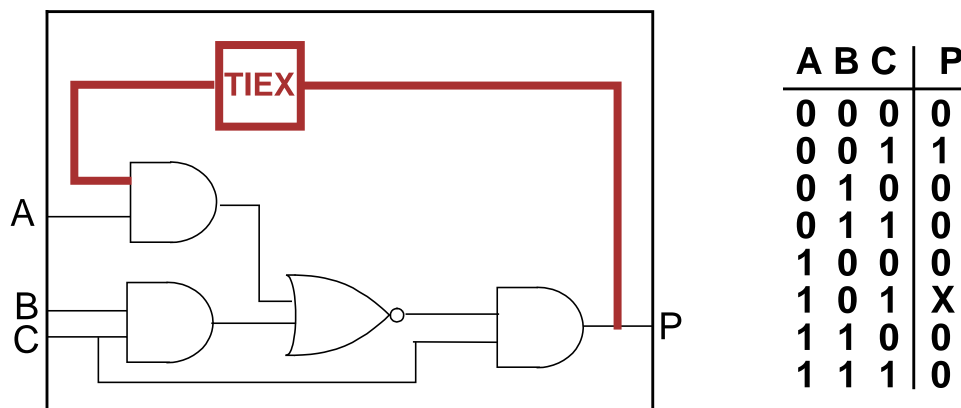


Figure 4-4. Cutting Constant Value Loops

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

2. **Single gate with "multiple fanout"**
   This loop cutting method involves loops containing only a single gate with multiple fanout.

   Figure 4-2 on page 4-5 shows the circuitry and truth table for a single multiple fanout loop. For this class of loops, the tool cuts the loop by inserting a TIE-X gate at one of the fanouts of this "multiple fanout gate" that lie in the loop path, as Figure 4-5 shows.



A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	X
1	1	0	0
1	1	1	0

Figure 4-5. Cutting Single Multiple-Fanout Loops

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

   Figure 4-6 shows a simple loop that the tools would target for gate duplication.

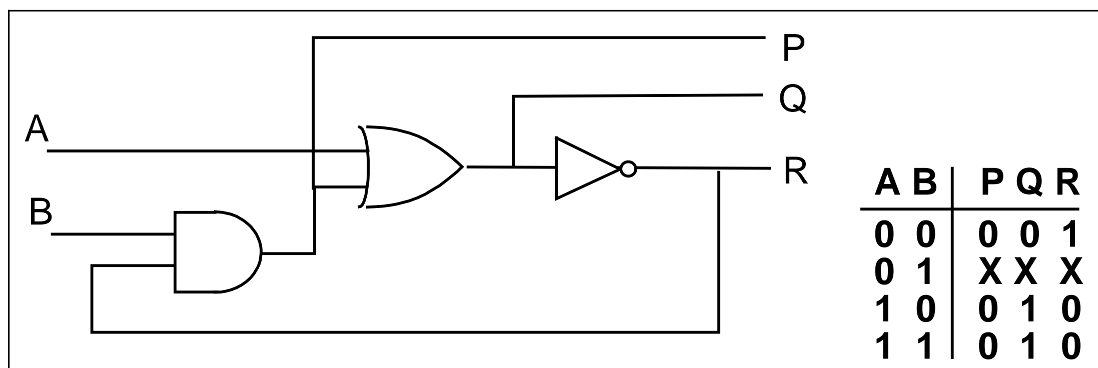


Figure 4-6. Loop Candidate for Duplication

Figure 4-7 shows how TIE-X insertion would add some pessimism to the simulation at output P.

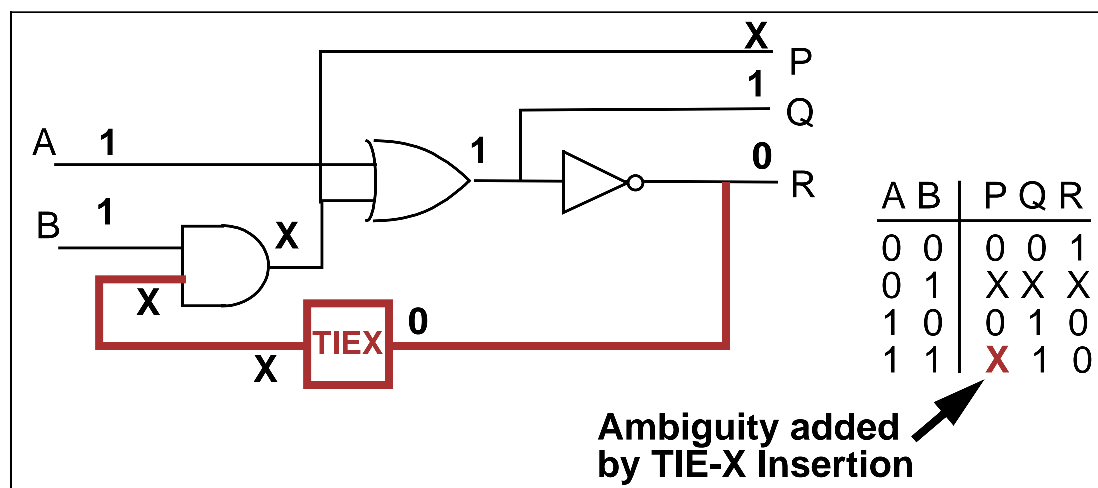


Figure 4-7. TIE-X Insertion Simulation Pessimism

The loop breaking technique proves beneficial in many cases. In the Figure 4-8 example, it provides a more accurate simulation model than the direct TIE-X insertion approach.

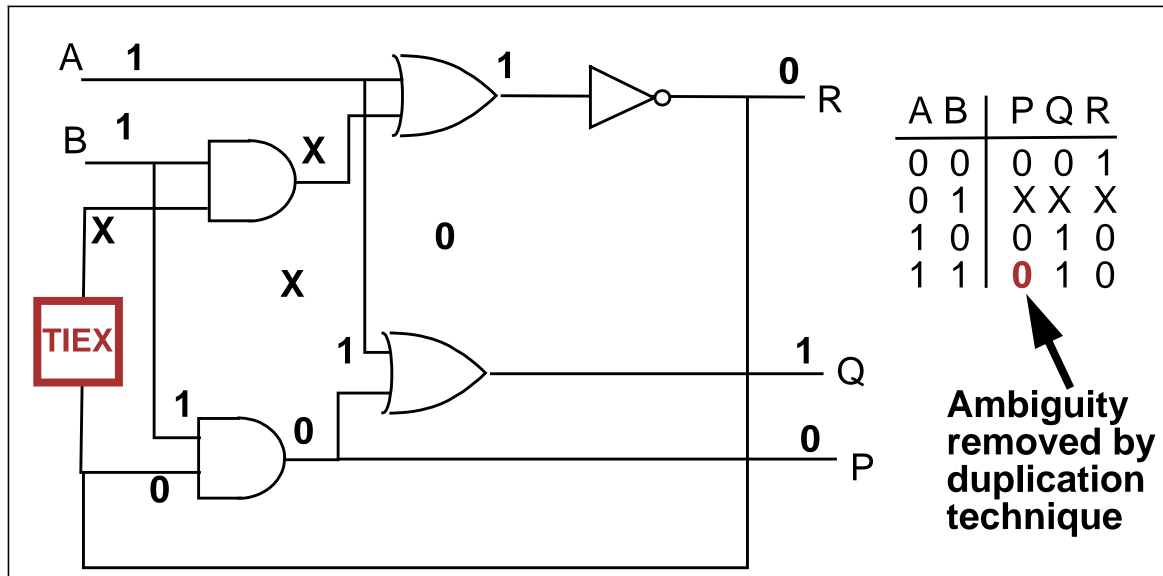


Figure 4-8. Cutting Loops by Gate Duplication

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

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

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

For example, assume the SR latch shown in Figure 4-6 was part of a larger, more complex, loop coupling network. In this case, loop circuitry

duplication would turn into an iterative process that would never converge. So, the tools would have to cut the loop as shown in Figure 4-9.

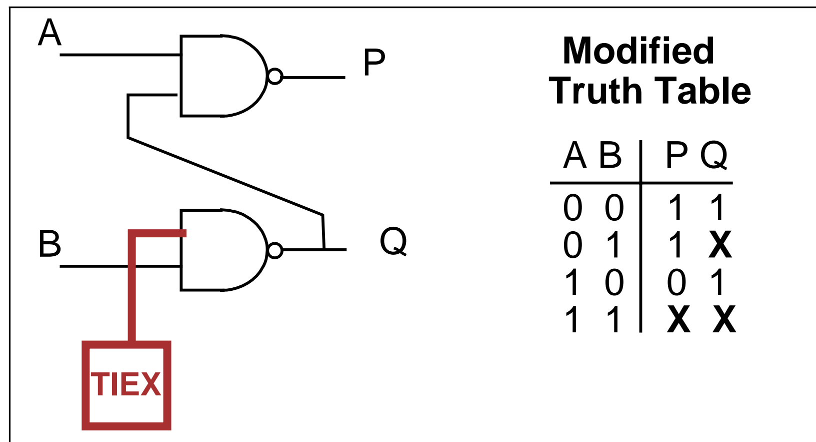


Figure 4-9. Cutting Coupling Loops

The modified truth table shown in Figure 4-9 demonstrates that this method yields the most pessimistic simulation results of all the loop-cutting methods. Because this is the most pessimistic solution to the loop cutting problem, the tools use this technique only when they cannot use any of the previous methods.

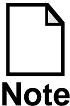
FastScan-Specific Combinational Loop Handling Issues

By default, FastScan performs parallel pattern simulation of circuits containing combinational feedback networks. This is controlled by using the Set Loop Handling Command.

SET LOop Handling {**Tiex** [-Duplication {ON | OFf}]} **|** {**Simulation**
[-Iterations *n*]}

A learning process identifies feedback networks after flattening, and an iterative simulation is used in the feedback network. Although you can define the number of iterations used to stabilize values in the feedback networks, excessive values will have an impact on both performance and memory usage.

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


Note

The Set Loop Handling command replaces functionality previously available by the Set Loop Duplication command.

FlexTest-Specific Combinational Loop Handling Issues

FlexTest provides three options for handling combinational feedback loops. These options are controlled by using the Set Loop Handling command.

SET LOop Handling {{**Tiex | Delay**} [-Duplication {ON | OFf}]} | **Simulation**

The following list itemizes and describes some of the issues specific to FlexTest concerning combinational loop handling:

- **Simulation Method**
  In some cases, using TIEX gates decreases test coverage, and causes DRC failures and bus contentions. Also, using delay elements can cause too optimistic test coverage and create output mismatch and bus contentions. Therefore, by default, FlexTest uses a simulation process to stabilize values in the combinational loop.

  FLexTest has the ability to perform DRC simulation of circuits containing combinational feedback networks by using a learning process to identify feedback networks after flattening, and an iterative simulation process is used in the feedback network. The state is not maintained in a feedback network from one cycle of a sequential pattern to the next.

  Using TIEX gates decreases test coverage, and causes DRC failures and bus contentions. Using delay elements can cause too optimistic test coverage and can create output mismatching and bus contentions.

  Some loop structures may not contain loop behavior. The FlexTest loop cutting point has buffer behavior. However, if loop behavior exists, this buffer has an unknown output. Essentially, during good simulation, this buffer is always initialized to have an unknown output value at each time frame. Its value stays unknown until a dominate value is generated from outside the loop.

To improve performance, for each faulty machine during fault simulation, this loop cutting buffer does not start with an unknown value. Instead, the good machine value is the initial value. However, if the value is changed to the opposite value, an unknown value is then used the first time to ensure loop behavior is properly simulated.

During test generation, this loop cutting buffer has a large SCOAP controllability number for each simulation value.

- **TIEX or DELAY gate insertion**
  Because of its sequential nature, FlexTest can insert a DELAY element, instead of a TIE-X gate, as a means to break loops. The DELAY gate retains the new data for one timeframe before propagating it to the next element in the path. Figure 4-10 shows a DELAY element inserted to break a feedback path.

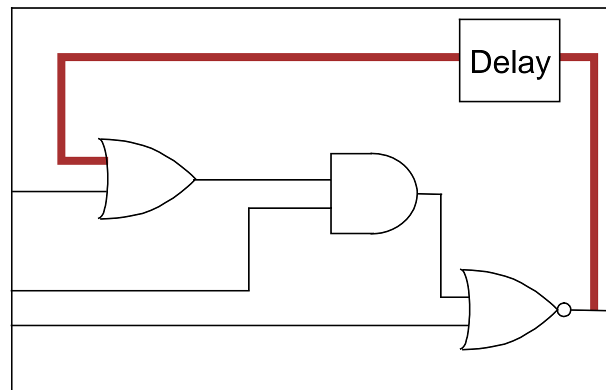
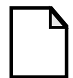


Figure 4-10. Delay Element Added to Feedback Loop

Because FlexTest simulates multiple timeframes per test cycle, DELAY elements often provide a less pessimistic solution for loop breaking as they do not introduce additional X states into the good circuit simulation.


Note

In some cases, inserted DELAY elements can cause mismatches between FlexTest simulation and a full-timing logic simulator. If you experience either of these problems, use TIE-X gates instead of DELAY gates for loop cutting.

- **Turning gate duplication on**
  Gate duplication reduces the impact of the TIE-X or DELAY gates that the tool places to break combinational loops. You can turn this option on only when using the **Tiex** or **Delay** settings. By default, the gate duplication option is off because FlexTest performs the simulation method upon invocation of the tool.

DFTAdvisor-Specific Combinational Loop Handling Issues

DFTAdvisor identifies combinational loops during flattening. By default, it performs TIE-X insertion using the methods specified in "Structural Combinational Loops and Loop-Cutting Methods" on page 4-5 to break all loops detected by the initial loop analysis. You can turn loop duplication off using the Set Loop Duplication command.

You can report on loops using the Report Loops or the Report Feedback Paths commands. While both involved with loop reporting, these commands behave somewhat differently. Refer to the *DFTAdvisor Reference Manual* for details. You can write all identified structural combinational loops to a file using the Write Loops command.

You can use the loop information DFTAdvisor provides to handle each loop in the most desirable way. For example, assuming you wanted to improve the test coverage for a coupling loop, you could use the Add Test Points command within DFTAdvisor to insert a test point to control or observe values at a certain location within the loop.

Structural Sequential Loops and Handling

Sequential feedback loops occur when the output of a latch or flip-flop feeds back to one of its inputs, either directly or through some other logic. Figure 4-11 shows an example of a structural sequential feedback loop.

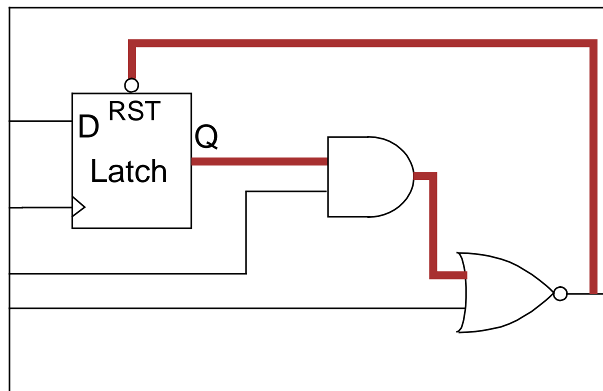
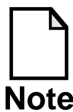


Figure 4-11. Sequential Feedback Loop


Note

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

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

FastScan-Specific Sequential Loop Handling

While FastScan can suffer some loss of test coverage due to sequential loops, these loops do not cause FastScan the extensive problems that combinational loops do. By its very nature, FastScan re-models the non-scan sequential elements in the design using the simulation primitives described in "FastScan Handling of Non-Scan Cells" on page 4-20. Each of these primitives, when inserted, automatically breaks the loops in some manner.

Within FastScan, sequential loops typically trigger C3 and C4 design rules violations. When one sequential element (a source gate) feeds a value to another sequential element (a sink gate), FastScan simulates old data at the sink. You can change this simulation method using the Set Capture Handling command. For

more information on the C3 and C4 rules, refer to "Clock Rules" in the *Design-for-Test Common Resources Manual*. For more information on the Set Capture Handling command refer to its reference page in the *FastScan and FlexTest Reference Manual*.

FlexTest-Specific Sequential Loop Handling

FlexTest identifies sequential loops after both combinational loop analysis and design rules checking. As part of the design rules checking and sequential loop analysis, FlexTest determines both the real and fake sequential loops.

Similar to fake combinational loops, fake sequential loops do not exhibit loop behavior. For example, Figure 4-12 shows a fake sequential loop.

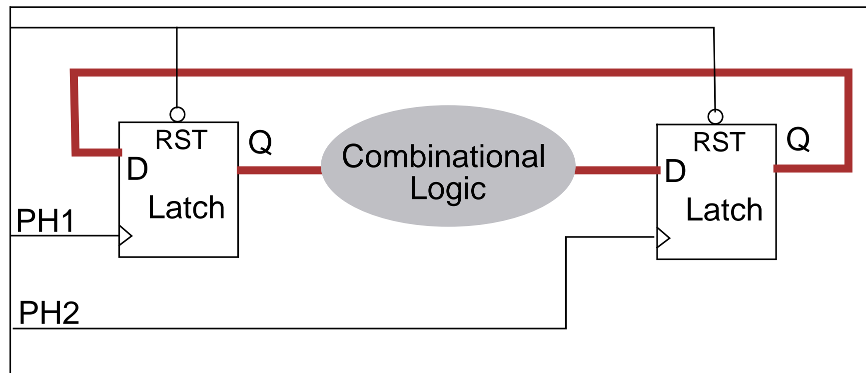


Figure 4-12. Fake Sequential Loop

While this circuitry involves latches that form a structural loop, the two-phase clocking scheme (assuming properly-defined clock constraints) ensures clocking of the two latches at different times. Thus, FlexTest does not treat this situation as a loop.

Only the timeframe considerations vary between the two loop cutting methods. Different timeframes may require different loop cuts. FlexTest additively keeps track of the loop cuts needed, and inserts them at the end of the analysis process.

You set whether FlexTest uses a TIE-X gate or DELAY element for sequential loop cutting with the Set Loop Handling command. By default, FlexTest inserts DELAY elements to cut loops.

DFTAdvisor-Specific Sequential Loop Handling

If you have selected one of the partial scan identification types, DFTAdvisor may perform some sequential loop analysis during the scan cell identification process. If you have set the type to atpg-based scan cell identification (Setup Scan Identification sequential atpg), DFTAdvisor performs the same sequential loop analysis and cutting as FlexTest. If you have set the type to sequential transparent (Setup Scan Identification seq_transparent), DFTAdvisor cuts sequential loops by inserting a scan cell in place of one the latches in the loop. This sets up the design so it can take advantage of the scan-sequential capabilities of FastScan.

Redundant Logic

In most cases, you should avoid using redundant logic because a circuit with redundant logic poses testability problems. First, classifying redundant faults take a great deal of analysis effort. Additionally, redundant faults, by their nature, are untestable and therefore lower your fault coverage. Figure 2-20 on page 2-35 gives an example of redundant circuitry.

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

Asynchronous Sets and Resets

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

Figure 4-13 shows a situation with an asynchronous reset line and the test logic added to control the asynchronous reset line.

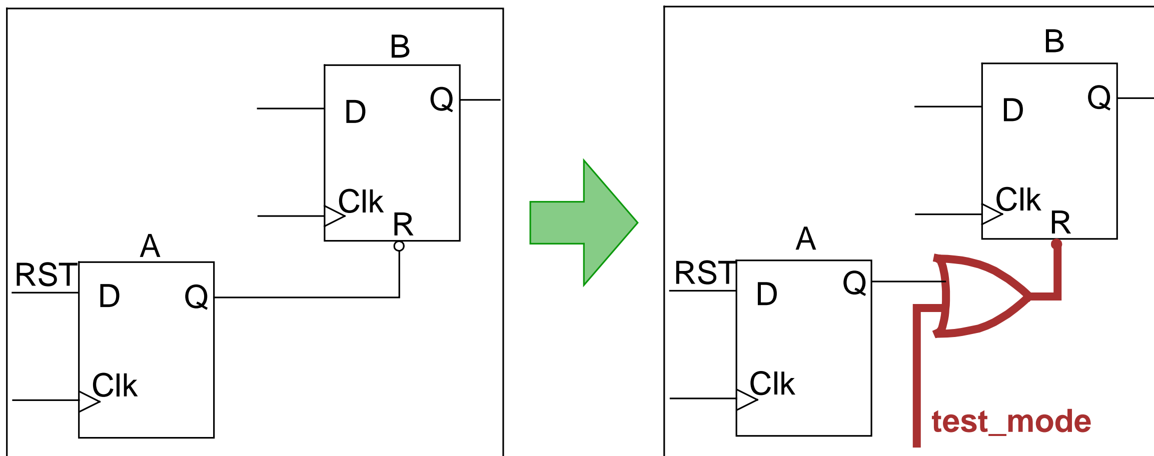


Figure 4-13. Test Logic Added to Control Asynchronous Reset

In this example, DFTAdvisor adds an OR gate that uses the test_mode (not scan_enable) signal to keep the reset of flip-flop B inactive during the testing process. You would then constrain the test_mode signal to be a 1, so flip-flop B could never be reset during testing. To insert this type of test logic, you can use the DFTAdvisor command Set Test Logic (see page 5-12 for more information).

DFTAdvisor also allows you to specify an initialization sequence in the test procedure file to avoid the use of this additional test logic. For additional information, refer to the Add Scan Groups command in the *DFTAdvisor Reference Manual*.

Gated Clocks

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

For example, Figure 4-14 shows an example of a clock that requires some test logic to control it during test mode.

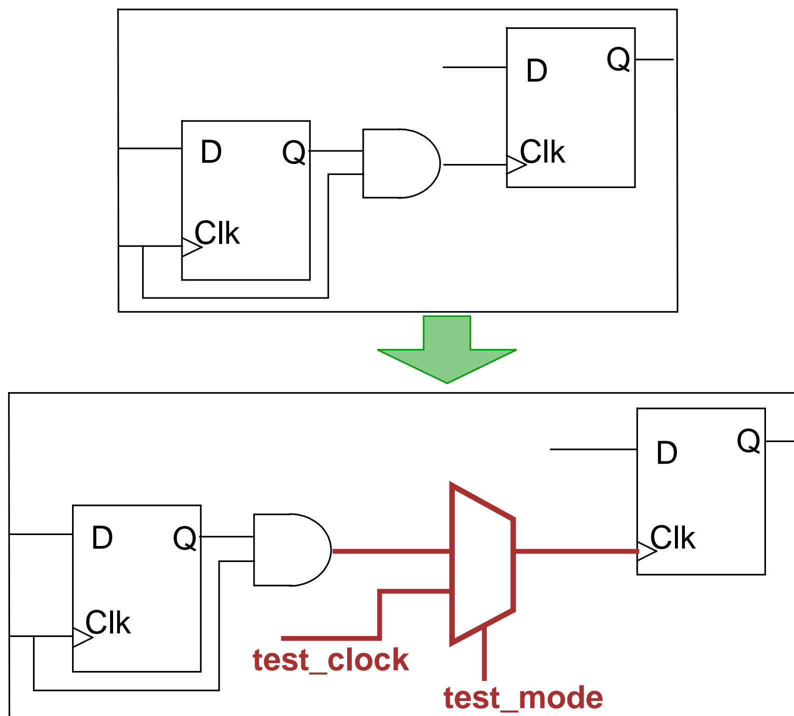


Figure 4-14. Test Logic Added to Control Gated Clock

In this example, DFTAdvisor makes the element scannable by adding a test clock, for both scan loading/unloading and data capture, and multiplexing it with the original clock signal. It also adds a signal called test_mode to control the added multiplexer. The test_mode signal differs from the scan_mode or scan_enable signals in that it is active during the entire duration of the test--not just during scan chain loading/unloading. To add this type of test logic into your design, you can use the Set Test Logic and Setup Scan Insertion commands. For more information on these commands, refer to pages 5-12 and 5-38, respectively.

Tri-State Devices

Tri-state buses are another testability challenge. Faults on tri-state bus enables can cause one of two problems: *bus contention*, which means there is more than one active driver, or *bus float*, which means there is no active driver. Either of these conditions can cause unpredictable logic values on the bus, which allows the enable line fault to go undetected. Figure 4-15 shows a tri-state bus with bus contention caused by a stuck-at-1 fault.

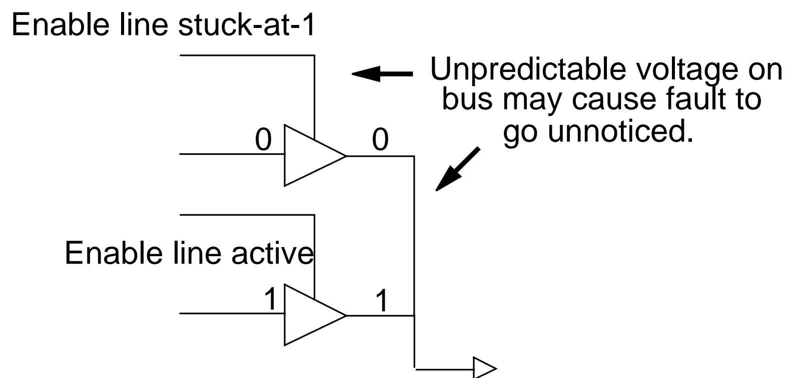


Figure 4-15. Tri-state Bus Contention

DFTAdvisor can add gating logic that turns off the tri-state devices during scan chain shifting. The tool gates the tri-state device enable lines with the scan_enable signal so they are inactive and thus prevent bus contention during scan data shifting. To insert this type of gating logic, you can use the DFTAdvisor command Set Test Logic (see page 5-12 for more information).

In addition, FastScan and FlexTest let you specify the fault effect of bus contention on tri-state nets. This capability increases the testability of the enable line of the tri-state drivers. Refer to the Set Net Dominance command in the *FastScan and FlexTest Reference Manual* for details.

Non-Scan Cell Handling

During rules checking and learning analysis, FastScan and FlexTest learn the behavior of all state elements that are not part of the scan circuitry. This learning involves how the non-scan element behaves after the scan loading operation. As a result of the learning analysis, FastScan and FlexTest categorize each of the non-scan cells. This categorization differs depending on the tool, as shown in the following subsections.

FastScan Handling of Non-Scan Cells

FastScan places non-scan cells in one of the following categories:

- **TIEX -** In this category, FastScan considers the output of a flip-flop or latch to always be an X value during test. This condition may prevent the detection of a number of faults.
- **TIE0 -** In this category, FastScan considers the output of a flip-flop or latch to always be a 0 value during test. This condition may prevent the detection of a number of faults.
- **TIE1 -** In this category, FastScan considers the output of a flip-flop or latch to always be a 1 value during test. This condition may prevent the detection of a number of faults.
- **Transparent (combinational) -**In this category, the non-scan cell is a latch, and the latch behaves transparently. When a latch behaves transparently, it acts, in effect, as a buffer--passing the data input value to the data output. The TLA simulation gate models this behavior. Figure 4-16 shows the point at which the latch must exhibit transparent behavior.

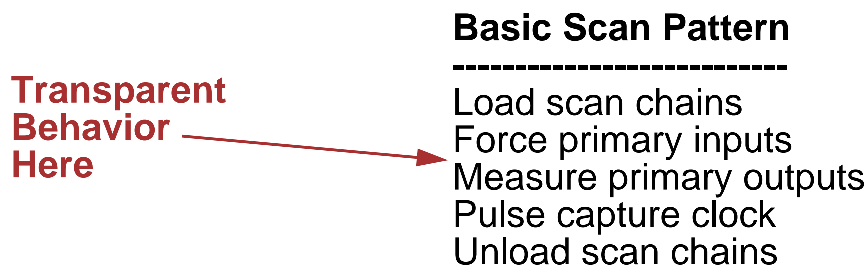


Figure 4-16. Requirement for Combinationally Transparent Latches

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

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

For more information on the transparent latch checking procedure, refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual*.

- **Sequential transparent -** Sequential transparency extends the notion of transparency to include non-scan elements that can be forced to behave transparently at the same point in which natural transparency occurs. In this case, the non-scan element can be either a flip-flop, a latch, or a RAM read port. A non-scan cell behaves as sequentially transparent if, given a sequence of events, it can capture a value and pass this value to its output, without disturbing critical scan cells.

  Sequential transparent handling of non-scan cells lets *you* describe the events that place the non-scan cell in transparent mode. You do this by specifying a procedure, called **seq_transparent**, in your test procedure file. This procedure contains the events necessary to create transparent behavior of the non-scan cell(s). After the tool loads the scan chain, forces the primary inputs, and forces all clocks off, the **seq_transparent** procedure pulses the clocks of all the non-scan cells or performs other specified events to pass data through the cell "transparently" (see "The Procedures" on page 3-15 for details).

  Figure 4-17 shows an example of a scan design with a non-scan element that is a candidate for sequential transparency.

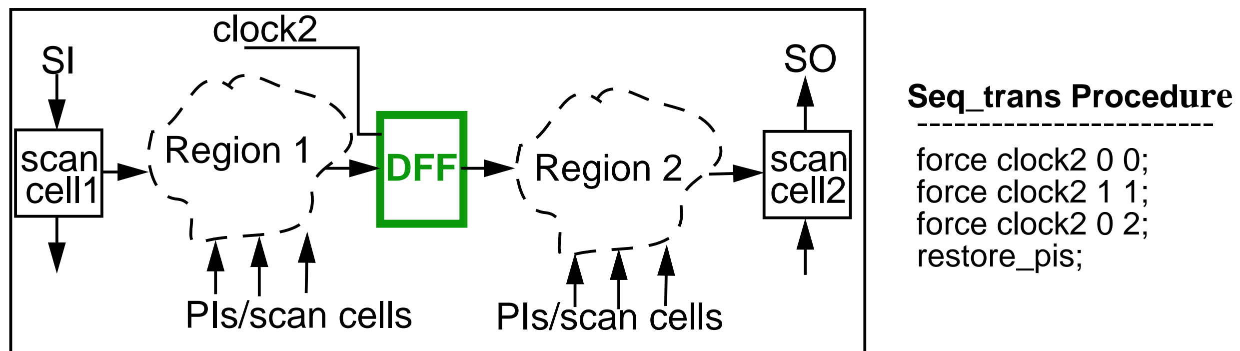
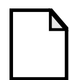


Figure 4-17. Example of Sequential Transparency

The DFF shown in Figure 4-17 behaves sequentially transparent when the tool pulses its clock input, clock2. The sequential transparent procedure shows the events that enable transparent behavior.


Note

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

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

Rules checking determines if non-scan cells qualify for sequential transparency via these procedures. Specifically, the cells must satisfy rules P5, P6, P41, P44, P45, P46, D3, and D9. For more information on these rules, refer to "Design Rules Checking" in the *Design-for-Test Common*

Resources Manual. Clock rules checking treats sequential transparent elements the same as scan cells.

Limitations of sequential transparent cell handling include the following:

- Impaired ability to detect AC defects (transition fault type causes sequential transparent elements to appear as tie-X gates).
- Cannot make non-scan cells clocked by scan cells sequentially transparent without **condition** statements.
- Limited usability of the sequential transparent procedure if applying it disturbs the scan cells (contents of scan cells change during the **seq_transparent** procedure).
- Feedback paths to non-scan cells, unless broken by scan cells, prevent treating the non-scan cells as sequentially transparent.

- **Clocked sequential -** If a non-scan cell obeys the standard scan clock rules—that is, if the cell holds its value with all clocks off—FastScan treats it as a clocked sequential cell. In this case, after the tool loads the scan chain, it forces the primary inputs and pulses the clock/write/read lines multiple times (based on the sequential depth of the non-scan cells) to set up the conditions for a test. A normal observe cycle then follows. Figure 4-18 shows a clock sequential scan pattern.

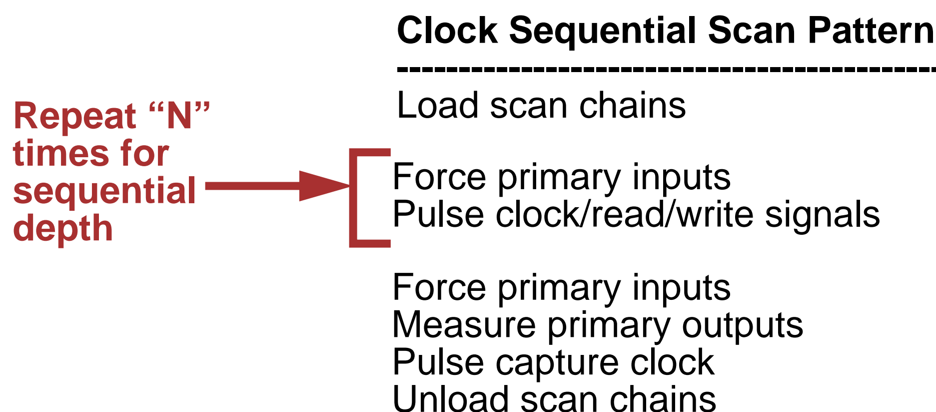


Figure 4-18. Clocked Sequential Scan Pattern Events

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

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

You instruct FastScan to use clocked sequential handling by selecting the -Depth option to the Set Simulation Mode command. During test generation, FastScan generates test patterns for target faults by first attempting combinational, and then RAM sequential techniques. If unsuccessful with these techniques, FastScan performs clocked sequential test generation (if you specify a non-zero sequential depth). To report on clocked sequential cells, you use the Report Nonscan Cells command. For more information on setting up and reporting on clocked sequential test generation, refer to the Set Simulation Mode and Report Nonscan Cells reference pages in the *FastScan and FlexTest Reference Manual.*

Limitations of clocked sequential non-scan cell handling include:

- You cannot use ATPG compression via the Set Atpg Compression command (although Compress Patterns allows static compression of the test pattern set).
- The maximum allowable sequential depth is 255 (a typical depth would range from 2 to 5).
- Copy and shadow cells cannot behave sequentially.
- The tool cannot detect faults on clock/set/reset lines.
- You cannot use the read-only mode of RAM testing with clock sequential pattern generation.
- There is no capability to hold the state of tristate devices.
- You must set sequential depth before rules checking.

- FastScan simulates cells that capture data on a trailing clock edge (when data changes on the leading edge) using the original values on the data inputs.
- This type of testing has high memory and performance costs.

FlexTest Handling of Non-Scan Cells

During circuit learning, FlexTest places non-scan cells in one of the following categories:

- **HOLD** - The learning process separates non-scan elements into two classes: those that change state during scan loading and those that hold state during scan loading. The HOLD category is for those non-scan elements that hold their values: that is, FlexTest assumes the element retains the same value after scan loading as prior to scan loading.
- **INITX -** When the learning process cannot determine any useful information about the non-scan element, FlexTest places it in this category and initializes it to an unknown value for the first test cycle.
- **INIT0 -** When the learning process determines that the **load_unload** procedure forces the non-scan element to a 0, FlexTest initializes it to a 0 value for the first test cycle.
- **INIT1 -** When the learning process determines that the **load_unload** procedure forces the non-scan element to a 1, FlexTest initializes it to a 1 value for the first test cycle.
- **TIE0 -** When the learning process determines that the non-scan element is always a 0, FlexTest assigns it a 0 value for all test cycles.
- **TIE1 -** When the learning process determines that the non-scan element is always a 1, FlexTest assigns it a 1 value for all test cycles.
- **DATA_CAPTURE** - When the learning process determines that the value of a non-scan element depends directly on primary input values, FlexTest places it in this category. Because primary inputs (other than scan inputs or

bi-directionals) do not change during scan loading, FlexTest considers their values constant during this time.

The learning process places the non-scan cells into one of the preceding categories. You can report on the non-scan cell handling with the Report Nonscan Handling command. You can override the default categorization with the Add Nonscan Handling command.

Clock Dividers

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

Figure 4-19 shows an example of a sequential element (B) driven by a clock divider signal and with the appropriate circuitry added to control the divided clock signal.

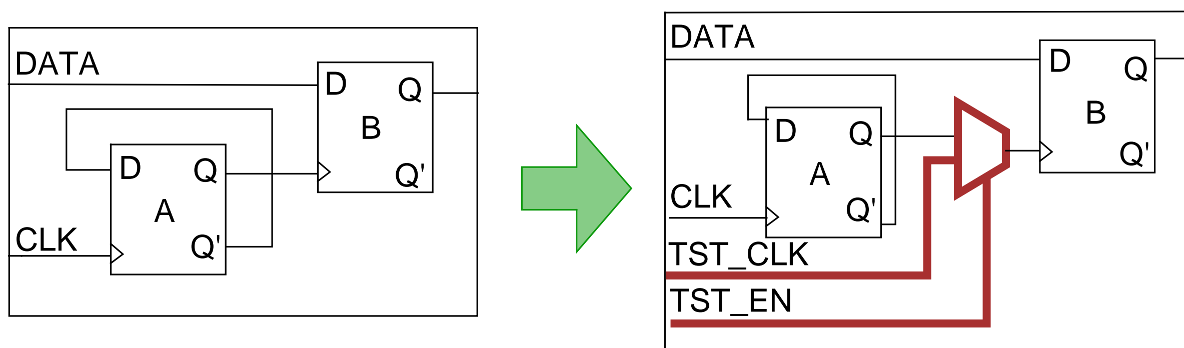


Figure 4-19. Clock Divider

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

For more information on test logic, refer to "Enabling Test Logic Insertion" on page 5-12.

Pulse Generators

Pulse generators are circuitry that create pulses when active. Figure 4-20 gives an example of pulse generator circuitry.

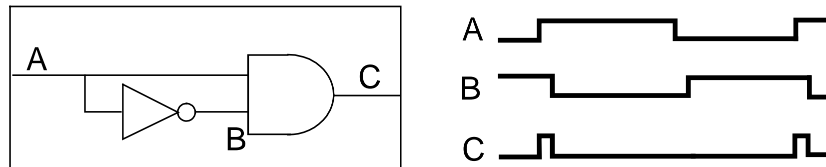


Figure 4-20. Example Pulse Generator Circuitry

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

FastScan and FlexTest identify the reconvergent pulse generator sink gates, or simply "pulse generators", during the learning process. For the tools to support "pulse generators", it must satisfy the following requirements:

- The "pulse generator" gate must have a connection to a clock input of a memory element or a write line of a RAM.
- The "pulse generator" gate must be an AND, NAND, OR, or NOR gate.
- Two inputs of the "pulse generator" gate must come from the reconvergent source gate.
- The two reconvergent paths may only contain inverters and buffers.
- There must be an inversion difference in the two reconvergent paths.
- The two paths must have different lengths.

- The input gate of the “pulse generator” gate in the long path must only go to gates of the same gate type. The tools model this input gate as tied to the non-controlling value of the “pulse generator” gate.

FastScan and FlexTest provide two commands that deal with pulse generators: Set Pulse Generators, which controls the identification of the “pulse generator” gates, and Report Pulse Generators, which displays the list of “pulse generator” gates. Refer to the *FastScan and FlexTest Reference Manual* for information on the Set Pulse Generators and Report Pulse Generators commands.

Additionally, rules checking includes some checking for “pulse generator” gates. Specifically, Trace rules #16 and #17 check to ensure proper usage of “pulse generator” gates. Refer to “T16 (Trace Rule #16)” and “T17 (Trace Rule #17)” in the *Design-for-Test Common Resources Manual* for more details on these rules.

JTAG-Based Circuits

Boundary scan circuitry, as defined by IEEE standard 1149.1, can result in a complex environment for the internal scan structure and the ATPG process. The two main issues with boundary scan circuitry are 1) connecting the boundary scan circuitry with the internal scan circuitry, and 2) ensuring that the boundary scan circuitry is set up properly during ATPG. For information on connecting boundary scan circuitry to internal scan circuitry, refer to “Connecting Internal Scan Circuitry” in the *Boundary Scan Process Guide*. For an example test procedure file that sets up a JTAG-based circuit, refer to page 6-103.

Built-In Self-Test (FastScan Only)

Built-In Self-Test, or BIST, which is becoming increasingly popular with designers, gives a circuit the ability to test itself. Although predominantly used for regular structures, such as embedded RAM and ROM, designers are using BIST technology more and more for random logic testing.

BIST circuitry can perform burn-in testing and at-speed testing, and allows for self-checking on critical portions of a design. BIST can minimize the need for ATPG, shorten the amount of ATE time, and require less complex external test equipment.

Sections “Setting Up for BIST (FastScan Only)” on page 6-42 and “Running Random/BIST Pattern Simulation (FastScan)” on page 6-53 give task-oriented information on testing with BIST.

Example BIST Configuration

BIST structures do not require externally generated ATPG patterns to test the circuitry. Using the BIST technique, the device itself generates test patterns and applies them to the circuitry. There are many different BIST architectures and strategies. However, this discussion covers an architecture that includes the capability to generate random patterns to test the device. The component that performs this task is a *linear feedback shift register*, or *LFSR*. An LFSR is an N bit register with feedback from the last bit back to the first bit. Instead of just shifting bits around the register, a special technique applies an XORing of the value of two or more register cells and places this value in the new data position. The XORed register bits, known as *tap points, are* either external to the register or an internal part of the register. The shift register, along with the tap points, create a *pseudo-random pattern generator*, or *PRPG*, which generates patterns for BIST testing.

Figure 4-21 shows an N bit LFSR with three external tap points used as a PRPG in BIST circuitry.

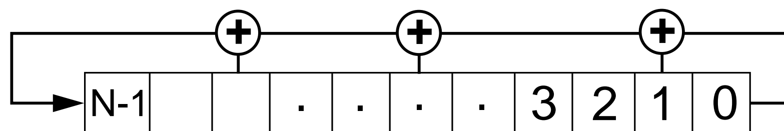


Figure 4-21. LFSR Configuration

BIST circuitry also uses another type of LFSR, the *multiple input signature register*, or *MISR*. The PRPG generates patterns for the logic and the MISR compresses the logic response of the circuit into a signature. The circuitry compares the signature of the actual circuit to a known good circuit response and then generates either a “go” or “no-go” signal. A “no-go” signal indicates a problem with the circuitry.

Figure 4-22 shows an example of a design containing BIST circuitry.

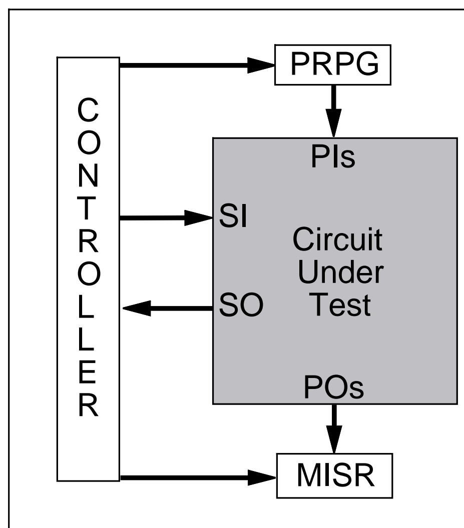


Figure 4-22. Simple BIST Configuration

Scan chain inputs and outputs typically connect to the LFSRs. More specifically, the BIST circuitry tests the circuit under test (CUT) by performing the following tasks:

1. Initialize the LFSR.
2. Load a pseudo-random pattern into the CUT via the scan path.
3. Generate and apply a new pseudo-random test pattern to the primary inputs.
4. Capture the response into the internal scan cells.
5. Load the response from the internal scan cells to the MISR.

FastScan's BIST Support

The BIST support features of FastScan include:

- Random pattern fault simulation, which predicts the expected BIST test coverage for a given number of random patterns.

- Simulation of user-defined LFSRs to calculate the actual BIST test coverage and expected signatures that result from the application of BIST patterns.
- Controllability analysis to identify points of low controllability and suggest where to add controllability to improve BIST test coverage.
- Observability analysis to identify points of low observability and suggest where to add observability to improve BIST test coverage.
- Insertion of control and observe points to evaluate their effect on test coverage.
- Automatic BIST testability analysis and circuit modifications to maximize test coverage given a selected number of inserted control and observe points.
- User-control of the primary input weighting factors to increase fault coverage during random pattern generation.

The limitations of FastScan's BIST support include:

- Use of only the user-defined LFSRs, not the physical BIST structure, when simulating BIST patterns.
- No checking of the operations of BIST circuitry or consistency with the user-defined LFSRs.
- No support of LFSRs connected to bidirectional pins.
- Ignored effect of MISR masking.
- BIST pattern support of only a single scan chain group.
- No support of a single LFSR used as both a PRPG and MISR.
- No support of circular BIST (configuration where internal scan cells function as the PRPG and MISR).

- RAM support includes only two methods: 1) initialize RAM and hold states during BIST test, and 2) disable the RAM outputs from propagating to observe points.

- Limited fault detection due to pin constraints or requirements of different capture clock or observe points.

For more information on FastScan's support of BIST structures, refer to "Setting Up for BIST (FastScan Only)" on page 6-42.

Table 4-1 shows the FastScan BIST support commands.

Table 4-1. FastScan BIST Commands

Command Name	Description
Add Control Points	Adds control points to output pins.
Add LFSRs	Adds LFSRs for use as PRPGs or MISRs.
Add LFSR Connections	Connects an external pin to an LFSR.
Add LFSR Taps	Adds the tap configuration to an LFSR.
Add Notest Points	Adds circuit points that cannot be used for testability insertion.
Add Observe Points	Adds observe points to output pins.
Add Random Weights	Specifies the random pattern weighting factors for primary inputs.
Analyze Control	Calculates zero and one-state controllability.
Analyze Observe	Calculates observability coverage.
Delete LFSRs	Deletes previously defined LFSRs.
Delete LFSR Connections	Deletes connections between LFSRs and primary pins.
Delete LFSR Taps	Deletes tap positions from an LFSR.
Delete Notest Points	Deletes added circuit points that cannot be used for testability insertion.
Delete Observe Points	Deletes added observe points.
Insert Testability	Performs testability analysis to achieve maximum test coverage.

Table 4-1. FastScan BIST Commands [continued]

Command Name	Description
Report Control Data	Displays information from the specified Analyze Control command.
Report Control Points	Displays a list of control points.
Report LFSRs	Displays a list of all defined LFSRs.
Report LFSR Connections	Displays a list of all connections between LFSRs and primary pins.
Report Notest Points	Displays a list of all added circuit points.
Report Observe Data	Displays information from the preceding Analyze Observe command.
Report Observe Points	Displays a list of all observe points.
Report Random Weights	Displays current weighting factors for primary inputs.
Report Testability Data	Analyzes collapsed faults for a selected fault class.
Set Bist Initialization	Specifies the states of the scan cells before applying BIST patterns.
Set Capture Clock	Specifies the capture clock name for random pattern simulation.
Set Control Threshold	Specifies the controllability value.
Set Observation Point	Specifies the observation point.
Set Observe Threshold	Specifies the minimum number of observations.
Set Pattern Source	Specifies the pattern source for a future ATPG or fault simulation run.
Set Random ATPG	Specifies random pattern usage.
Set Random Patterns	Specifies the number of random patterns to be simulated.
Setup LFSRs	Sets the default setting for the shift-type and tap-type switches.

For more information on any of these commands, refer to the Command Dictionary chapter in the *FastScan and FlexTest Reference Manual.*

Checking BIST Rules

If your circuit contains BIST (with at least one defined LFSR), the rules checker performs BIST rules checking in addition to the normal rules checks. You must correct all BIST rules violations, or remove the defined LFSRs, to pass rules checking. A BIST design must satisfy the following conditions:

- All LFSRs must have at least one tap point.
- Each scan chain input pin must connect to an LFSR that has a shift type of either serial or both. The tool supports deterministic scan chains controlled by a global template pattern, but you must change the handling of rule B2 to allow for this.
- Each scan chain output pin must connect to a MISR that has a shift type of either serial or both.
- LFSRs not connected to either a scan chain input or scan chain output must have a shift type of either parallel or both.
- LFSRs should not repeat during the first 32 patterns.

For more information on BIST rules checking, refer to “BIST Rules” in the *Design-for-Test Common Resources Manual*.

Testing with RAM and ROM

The three basic problems of testing designs with RAM and ROM are 1) modeling the behavior, 2) passing rules checking to allow testing, and 3) detecting faults during ATPG. The “RAM and ROM” section in the *Design-for-Test Common Resources Manual* discusses modeling RAM and ROM behavior. The “RAM Rules” section in the *Design-for-Test Common Resources Manual* discusses RAM rules checking. This section primarily discusses the techniques for detecting faults in circuits with RAM and ROM during ATPG.

The ATPG tools, FastScan and FlexTest, do *not* test the internals of the RAM/ROM, because stuck-at fault models are not effective in testing for the internal defects of RAM/ROM. Either direct access with chip pins (in test mode),

self-test structures within the chip itself, or scan circuitry are the best methods of testing the internal RAM or ROM circuitry.

However, FastScan and FlexTest need to model the behavior of the RAM/ROM so that faults can propagate through the RAM/ROM for detection at an observation point. This allows FastScan and FlexTest to generate tests for the circuitry around the RAM/ROM, as well as the read and write controls, data lines, and address lines of the RAM/ROM unit itself.

Figure 4-23 shows a typical configuration for a circuit containing embedded RAM.

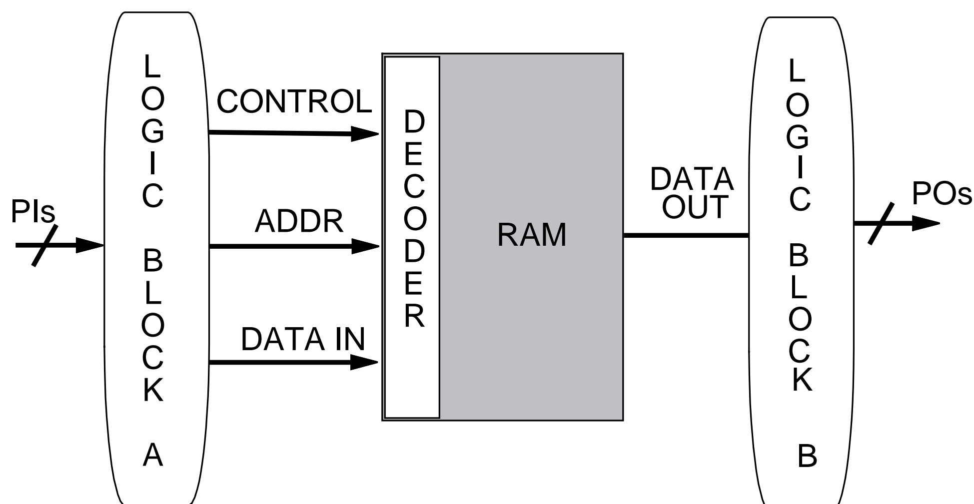


Figure 4-23. Design with Embedded RAM

If a fault occurs in Logic Block A, the tools cannot detect it unless they somehow propagate it through the RAM and Logic Block B and measure it at the primary outputs. FastScan and FlexTest each have unique strategies for handling this situation.

FastScan RAM/ROM Support

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

FastScan supports the following strategies for propagating fault effects through the RAM:

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

input and data output lines. It is not adequate for testing faults on read and write address lines.

  - **Dynamic pass-through -** This testing technique is similar to static pass-through testing except one pulse of the write clock performs both the write and read operation (if the write and read control lines are complementary). While static pass-through testing is comparable to transparent latch handling, dynamic pass-through testing compares to sequential transparent testing.

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

  RAM sequential testing requires its own specialized pattern type. RAM sequential patterns consist of one scan pattern with multiple scan chain loads. A typical RAM sequential pattern contains the events shown in Figure 4-24.

RAM Sequential Pattern

write into one address →	load scan chains force primary inputs pulse write control lines
write into second address →	load scan chains force primary inputs pulse write control lines
get data on outputs →	load scan chains force primary inputs pulse read control lines
basic pattern events →	load scan chain force primary inputs measure primary outputs pulse capture clock unload scan chains

Figure 4-24. RAM Sequential Example

In this example of an address line test, the first write would write data into a specific address, such as 1000. The second write operation would write different data into another address, such as 0000. The read operation then reads from the first address, 1000. If the highest order address bit is stuck-at-O, the faulty circuitry data would instead read data from 0000.

Another technique that may be useful for detecting faults in circuits with embedded RAM is clock sequential test generation. It is a more flexible technique, which effectively detects faults associated with RAM. "Clock Sequential Patterns" on page 6-11 discusses clock sequential test generation in more detail.

Common Read and Clock Lines

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

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

If the clock which captures the data from the RAM is not the same clock which is used for reading, then you will likely need to turn on multiple clocks to detect faults. If you issue the Set Clock Restriction Off command, FastScan will not allow these patterns, resulting in a loss in test coverage. If you issue the Set Clock Restriction On command, FastScan will allow these patterns, but there is a risk of inaccurate simulation results since the simulator will not propagate captured data effects.

Common Write and Clock Lines

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

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

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

FlexTest RAM/ROM Support

Like FastScan, FlexTest treats ROMs as strictly combinational gates. Once you initialize a ROM, it is a simple task to generate tests because the contents of the ROM do not change. However, testing RAM is more of a challenge because of the sequential behavior that occurs when writing data to and reading data from the RAM. Testing designs with RAM is a challenge for FastScan because of the combinational nature. FlexTest, however, due to its sequential nature, is able to handle designs with RAM without complication. RAMs are just treated as non-scan sequential blocks. However, in order to generate the appropriate RAM tests, you do need to specify the appropriate control lines.

FastScan and FlexTest RAM/ROM Support Commands

FastScan and FlexTest require certain knowledge about the design prior to test generation. For circuits with RAM, you must define write controls, and if the RAM had data hold capabilities, you must also define read controls. Just as you must define clocks so the tool can effectively write scan patterns, you must also define these control lines so it can effectively write patterns for testing RAM. And similar to clocks, you must define these signals in Setup mode, prior to rules checking. The FastScan (FS) and FlexTest(FT) commands in Table 4-2 support the testing of designs with RAM and/or ROM.

Table 4-2. FastScan and FlexTest RAM/ROM Commands

Command Name	FS	FT	Description
Add Read Controls	●	●	Adds an off-state value to read control lines.
Add Write Controls	●	●	Adds an off-state value to specified write control lines.
Create Initialization Patterns	●		Creates RAM initialization patterns for places them in the internal pattern set.
Delete Read Controls	●	●	Removes the read control line definitions from the specified primary input pins.
Delete Write Controls	●	●	Removes the write control line definitions from the specified primary input pins.
Read Modelfile	●	●	Initializes the specified RAM or ROM gate using the memory states contained in the specified modelfile.
Report Read Controls	●	●	Displays all of the currently defined read control lines.
Report Write Controls	●	●	Displays all of the currently defined write control lines.
Set Ram Initialization	●		Specifies whether to initialize RAM and ROM gates that do not have initialization files.
Set Ram Test	●		Sets the RAM testing mode to either read_only, pass_thru, or static_pass_thru.
Set Simulation Mode	●		Specifies whether the ATPG simulation run uses combinational or sequential RAM test patterns.
Write Modelfile	●	●	Writes all internal states for a RAM or ROM gate into the file that you specify.

For more information on any of these commands, refer to the Command Dictionary chapter in the *FastScan and FlexTest Reference Manual.*

Basic ROM/RAM Rules Checking

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

- The checker reads the RAM/ROM initialization files and checks them for errors. If you selected random value initialization, the tool gives random values to all RAM and ROM gates without an initialized file. If there are no initialized RAMs, you cannot use the read-only test mode. If any ROM is not initialized, an error condition occurs. A ROM must have an initialization file but it may contain all Xs. Refer to the Read Modelfile command in the *FastScan and FlexTest Reference Manual* for details on initialization of RAM/ROM.

- The RAM/ROM instance name given must contain a single RAM or ROM gate. If no RAM or ROM gate exists in the specified instance, an error condition occurs.

- If you define write control lines and there are no RAM gates in the circuit, an error condition occurs. To correct this error, delete the write control lines.

- When the write control lines are off, the RAM set and reset inputs must be off and the write enable inputs of all write ports must be off. You cannot use RAMs that fail this rule in read-only test mode. If any RAM fails this check, you cannot use dynamic pass-through. If you defined an initialization file for a RAM that failed this check, an error condition occurs. To correct this error, properly define all write control lines or use lineholds (pin constraints). You can ignore this error by using the -Force switch. If you use the -Force switch, you can only use the RAM for detection in static pass-through mode.

- A RAM gate must not propagate to another RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.

- A defined scan clock must not propagate directly (unbroken by scan or non-scan cells) to a RAM gate. If any RAM fails this check, you cannot use dynamic pass-through.
- The tool checks the write and read control lines for connectivity to the address and data inputs of all RAM gates. It gives a warning message for all occurrences and if connectivity fails, there is a risk of race conditions for all pass-through patterns.
- A RAM that uses the edge-triggered attribute must also have the **read_off** attribute set to hold. Failure to satisfy this condition results in an error condition when the design flattening process is complete.
- If the RAM rules checking identifies at least one RAM that the tool can test in read-only mode, it sets the RAM test mode to read-only. Otherwise, if the RAM rules checking passes all checks, it sets the RAM test mode to dynamic pass-through. If it cannot set the RAM test mode to read-only or dynamic pass-through, it sets the test mode to static pass-through.
- A RAM with the **read_off** attribute set to hold must pass Design Rule A7 (when read control lines are off, place read inputs at 0). The tool treats RAMs that fail this rule as:
    - a TIE-X gate, if the read lines are edge-triggered.
    - a **read_off** value of X, if the read lines are not edge-triggered.
- The read inputs of RAMs that have the **read_off** attribute set to hold must be at 0 during all times of all test procedures, except the **test_setup** procedure.
- The read control lines must be off at time 0 of the **load_unload** procedure.
- A clock cone stops at read ports of RAMs that have the **read_off** attribute set to hold, and the effect cone propagates from its outputs.

For more information on the RAM rules checking process, refer to “RAM Rules” in the *Design-for-Test Common Resources Manual*.

Chapter 5
Inserting Internal Scan and Test Circuitry

DFTAdvisor is the Mentor Graphics tool that provides comprehensive testability analysis and inserts internal test structures into your design. Figure 5-1 shows the layout of this chapter, as it applies to the process of inserting scan and other test circuitry.

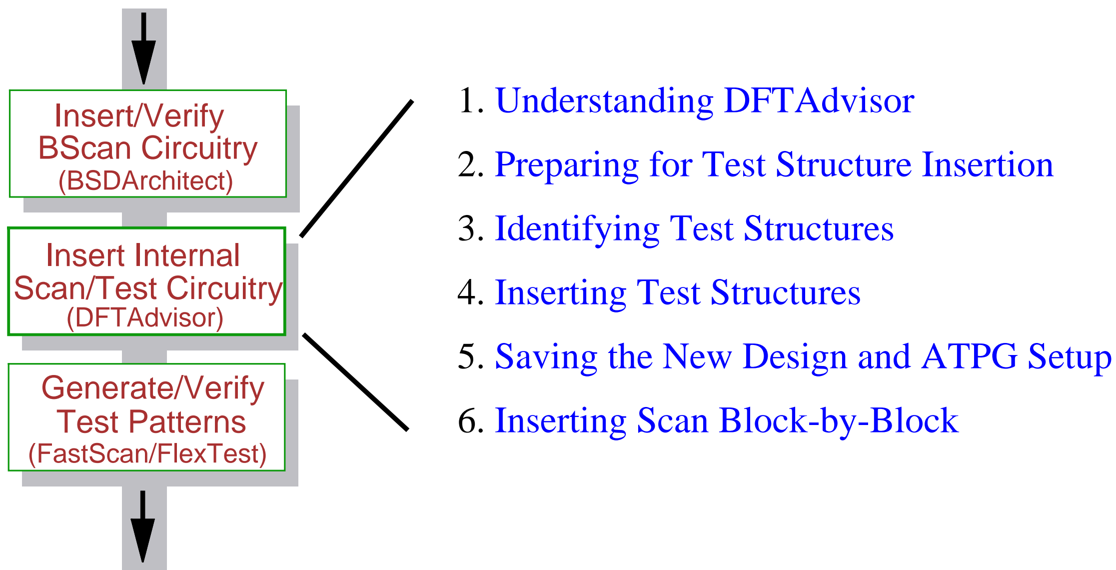


Figure 5-1. Internal Scan Insertion Procedure

This section discusses each of the tasks outlined in Figure 5-1, providing details on using DFTAdvisor in different environments and with different test strategies. For more information on all available DFTAdvisor functionality, refer to the *DFTAdvisor Reference Manual*.

Understanding DFTAdvisor

DFTAdvisor functionality is available in two modes: graphical user interface (GUI) or command-line. For information on using basic GUI functionality, refer to “User Interface Overview” on page 1-9 and “DFTAdvisor User Interface” on page 1-21.

Before you use either mode of DFTAdvisor, you should get familiar with the basic process flow, the inputs and outputs, the supported test structures, and the DFTAdvisor invocation as described in the following subsections.

You should also have a good understanding of the material in both Chapter 2, “Understanding Scan and ATPG Basics“, and Chapter 3, “Understanding Common Tool Terminology and Concepts“.

The DFTAdvisor Process Flow

Figure 5-2 shows the basic flow for synthesizing scan circuitry with DFTAdvisor.

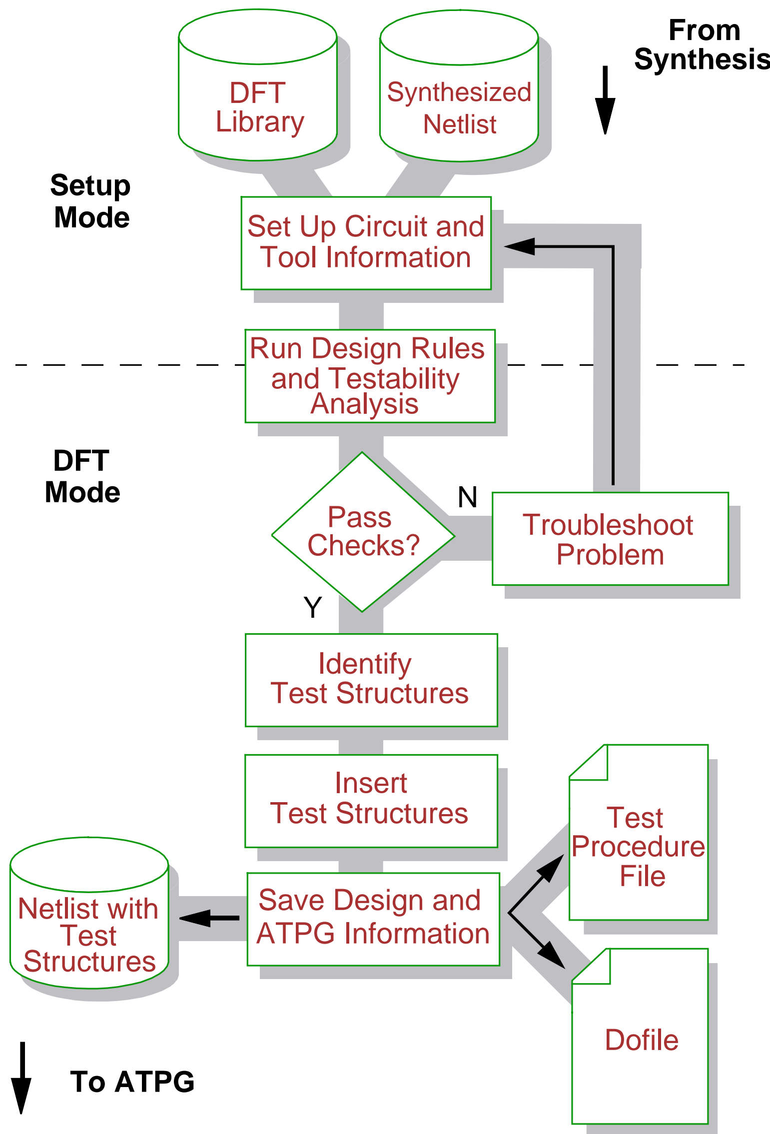
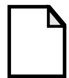


Figure 5-2. Basic Scan Insertion Flow with DFTAdvisor

You start with a DFT library and a synthesized design netlist. The library is the same one that FastScan and FlexTest use. "DFTAdvisor Inputs and Outputs" on page 5-5 describes the netlist formats you can use with DFTAdvisor. The design netlist you use as input may be an individual block of the design, or the entire design.

After invoking the tool, your first task is to set up information about the design—this includes both circuit information and information about the test structures you want to insert. "Preparing for Test Structure Insertion" on page 5-11 describes the procedure for this task. The next task after setup is to run rules checking and testability analysis, and debug any violations that you encounter. "Changing the System Mode (Running Rules Checking)" on page 5-18 documents the procedure for this task.


Note

To catch design violations early in the design process, you should run and debug design rules on each block as it is synthesized.

After successfully completing rules checking, you will be in the Dft system mode. At this point, if you have any existing scan you want to remove, you can do so. "Deleting Existing Scan Circuitry" on page 5-18 describes the procedure for doing this. You can then set up specific information about the scan or other testability circuitry you want added and identify which sequential elements you want converted to scan. "Identifying Test Structures" on page 5-20 describes the procedure for accomplishing this. Finally, with these tasks completed, you can insert the desired test structures into your design. "Inserting Test Structures" on page 5-37 describes the procedure for this insertion.

DFTAdvisor Inputs and Outputs

Figure 5-3 shows the inputs used and the outputs produced by DFTAdvisor.

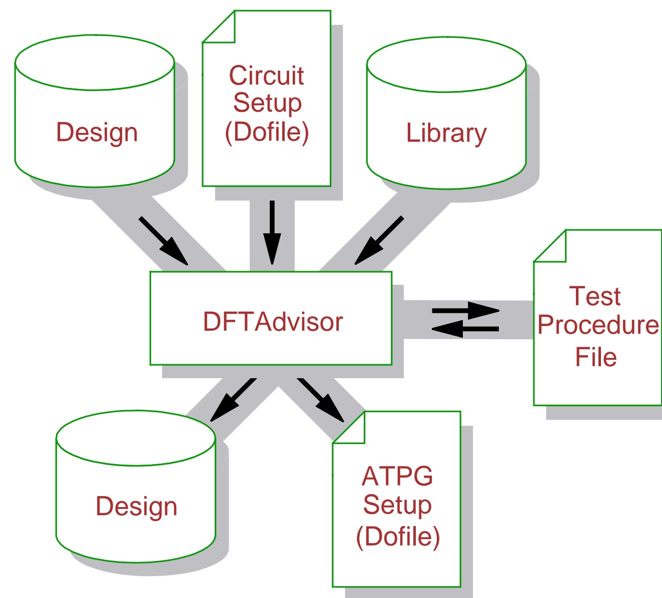


Figure 5-3. The Inputs and Outputs of DFTAdvisor

DFTAdvisor utilizes the following inputs:

- **Design (netlist)**
  The supported design data formats are Electronic Design Interchange Format (EDIF 2.0.0), GENIE, Tegas Design Language (TDL), VHDL, Verilog, and Spice.

- **Circuit Setup (or Dofile)**
  This is the set of commands that gives DFTAdvisor information about the circuit and how to insert test structures. You can issue these commands interactively in the DFTAdvisor session or place them in a dofile.

- **Library**
  The design library contains descriptions of all the cells the design uses. The library also includes information that DFTAdvisor uses to map non-scan cells to scan cells and to select components for added test logic circuitry.

The tool uses the library to translate the design data into a flat, gate-level simulation model on which it runs its internal processes.

- **Test Procedure File**
  This file defines the stimulus for shifting scan data through the defined scan chains. This input is only necessary on designs containing pre-existing scan circuitry or requiring test setup patterns.

DFTAdvisor produces the following outputs:

- **Design (Netlist)**
  This netlist contains the original design modified with the inserted test structures. The output netlist formats are the same type as the input netlist formats, with the exception of the NDL format. The NDL, or Network Description Language, format is a gate-level logic description language used in LSI Logic's C-MDE environment. This format is structurally similar to the TDL format.

- **ATPG Setup (Dofile)**
  DFTAdvisor can automatically create a dofile that you can supply to the ATPG tool. This file contains the circuit setup information that you specified to DFTAdvisor, as well as information on the test structures that DFTAdvisor inserted into the design. DFTAdvisor creates this file for you when you issue the command Write Atpg Setup.

- **Test Procedure File**
  When you issue the Write Atpg Setup command, DFTAdvisor writes a simple test procedure file for the scan circuitry it inserted into the design. You use this file with the downstream ATPG tools, FastScan and FlexTest.

Test Structures Supported by DFTAdvisor

DFTAdvisor can identify and insert a variety of test structures, including several different scan architectures and test points. Figure 5-4 depicts the types of scan and testability circuitry DFTAdvisor can add.

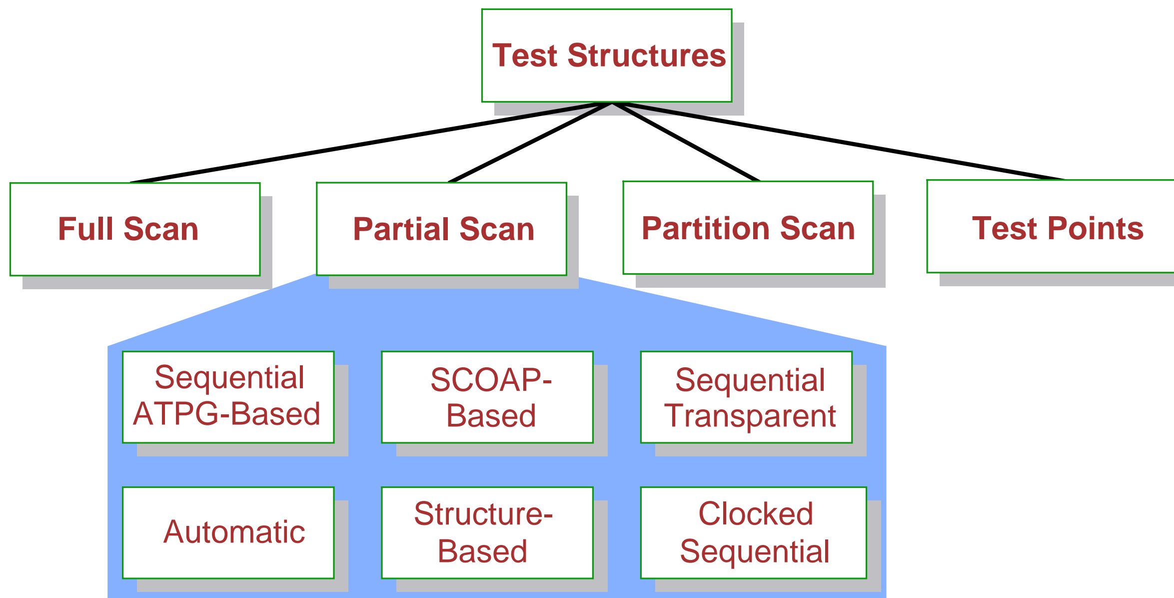


Figure 5-4. DFTAdvisor Supported Test Structures

The following list briefly describes the test structures DFTAdvisor supports:

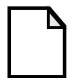
- **Full scan** — a style that identifies and converts all sequential elements (that pass scannability checking) to scan. "Understanding Full Scan" on page 2-4 discusses the full scan style.
- **Partial scan** — a style that identifies and converts a subset of sequential elements to scan. "Understanding Partial Scan" on page 2-5 discusses the partial scan style. DFTAdvisor provides five alternate methods of partial scan selection:
    - **Sequential ATPG-based** — chooses scan circuitry based on FlexTest's sequential ATPG algorithm. Because of its ATPG-based nature, this method provides predicable test coverage for the selected scan cells. This method selects scan cells using the sequential ATPG algorithm of FlexTest.

- **Automatic** — chooses as much scan circuitry as needed to achieve a high fault coverage. It combines several scan selection techniques. It typically achieves higher test coverage for the same allocation of scan. If it is limited, it attempts to select the best scan cells within the limit.

- **SCOAP-based** — chooses scan circuitry based on controllability and observability improvements determined by the SCOAP (Sandia Controllability Observability Analysis Program) approach. DFTAdvisor computes the SCOAP numbers for each memory element and chooses for scan those with the highest numbers. This method provides a fast way to select the best scan cells for optimum test coverage.

- **Structure-Based** — chooses scan circuitry using structure-based scan selection techniques. These techniques include loop breaking, self-loop breaking, and limiting the design's sequential depth.

- **Sequential Transparent** — chooses scan circuitry based on FastScan's scan sequential requirements. Scan cells are selected such that all sequential loops, including self loops, are cut. For more information on sequential transparent scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-20.


Note

This technique is useful for data path circuits.

- **Clocked Sequential** —chooses scannable cells by cutting sequential loops and limiting sequential depth. Typically, this method is used to create structured partial scan designs that can use FastScan's clock sequential ATPG algorithm. For more information on clock sequential scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-20.

- **Partition scan** — a style that identifies and converts certain sequential elements within design partitions to scan chains at the boundaries of the partitions. "Understanding Partition Scan" on page 2-8 discusses the partition scan style.

- **Test points** — a method that identifies and inserts control and observe points into the design to increase the overall testability of the design. "Understanding Test Points" on page 2-10 discusses the test points method.

DFTAdvisor first identifies and then inserts test structures. You use the Setup Scan Identification command to select scan during the identification process. You use Setup Test_point Identification for identifying test points during the identification process. If both scan and test points are enabled during an identification run, DFTAdvisor performs scan identification followed by test point identification. Table 5-1 shows which of the supported types may be identified together. The characters are defined as follows:

> * = Not recommended. Scan selection should be performed prior to test point selection.
> A = Allowed.
> N = Nothing more to identify.
> E = Error. Can not mix given scan identification types.

Table 5-1. Test Type Interactions

		Second Pass						
		Full Scan	**Clock Seq.**	**Seq. Trans-parent**	**Parti-tion Scan**	**Seq.**	**None**	**Test Point**
First Pass	**Full Scan**	N	N	N	A	N	A	A
	Clock Sequential	A	A	E	A	N	A	A
	Sequential Transparent	A	E	A	A	E	A	A
	Partition Scan	A	A	A	A	A	A	A
	Sequential	A	E	E	A	A	A	A
	None	A	A	A	A	A	A	A
	Test Point	*	*	*	*	*	A	A

"Selecting the Type of Test Structure" on page 5-20 discusses how to use the Setup Scan Identification command.

Invoking DFTAdvisor

You can invoke DFTAdvisor in two ways. Using the first option, you enter just the application name on the shell command line which opens DFTAdvisor in graphical mode.

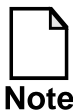
$MGC_HOME/bin/dftadvisor

Once the tool is invoked, a dialog box prompts you for the required arguments (design_name, design type, and library). Browser buttons are provided for navigating to the design and library. Once the design and library are loaded, the tool is in Setup mode, ready for you to begin working on your design. You can use the Setup mode to define the circuit and scan data, which is the next step in the process.

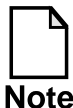
Using the second option requires you to enter all required arguments at the shell command line.

$MGC_HOME/bin/dftadvisor {***design_name*** {**-Edif** | **-TDl** | **-VHdl** |
-VERIlog | **-Genie** | **-SPice**} {**-LIBrary** ***filename***} [-SEnsitive]
[-LOg *filename*] [-Replace] [-TOp *module_name*] [-Dofile *dofile_name*]
[-LICense *retry_limit*] [-NOGui]} | -Help | -VERSion

When the tool is finished invoking, the design and library are also loaded. The tool is now in Setup mode, ready for you to begin working on your design. If you want to use the command-line interface, you must specify the -Nogui switch using the second invocation option.


Note

Your design must be in either EDIF, TDL, VHDL, Verilog, Genie, or Spice format.


Note

The invocation syntax for DFTAdvisor includes a number of other switches and options. For a list of available options and explanations of each, you can refer to "Shell Commands" in the *DFTAdvisor Reference Manual* or enter:

```
$  $MGC_HOME/bin/<application> -help
```

Preparing for Test Structure Insertion

The following subsections discuss the steps you would typically take to prepare for the insertion of test structures into your design. When the tool invokes, you are in Setup mode. All of the setup steps shown in the following subsections occur in Setup mode.

Selecting the Scan Methodology

If you want to insert scan circuitry into your design, you must select the type of architecture for the scan circuitry. Your choices are Mux_scan, Clocked_scan, or Lssd. For more information, refer to "Scan Architectures" on page 3-8.

You use the Set Scan Type command to specify the type of scan architecture you want to insert. The usage for this command is as follows:

SET SCan Type {**Mux_scan** | **Lssd** | **Clocked_scan**}

Defining Scan Cell and Scan Output Mapping

DFTAdvisor uses the default mapping defined within the ATPG library. Each scan model in the library describes how the non-scan models map to scan model in the scan_definition section of the model. For more information on the default mapping of the library model, refer to "Defining a Scan Cell Model" in the *Design-for-Test Common Resources Manual*.

You have the option to customize the scan cell and the cell's scan output mapping behavior. You can change the mapping for an individual instance, all instances under a hierarchical instance, all instances in all occurrences of a module in the design, or all occurrences of the model in the entire design using the Add Mapping Definition command. You can also delete scan cell mapping and report on its current status using the Delete Mapping Definition and Report Mapping Definition commands.

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

```
add mapping definition  fd1 -scan_model fd1s
```

In the following example, you can map the fd1 nonscan model to the fd1s scan model for all matching instances in the "counter" module and for all occurrences of that module in the design:

```
add mapping definition  counter -module -nonscan_model fd1
        -scan_model fd1s
```

Additionally, you can change the scan output pin of the scan model in the same manner as the scan cell. Within the scan_definition section of the model, the scan_out attribute defines which pin is used as the scan output pin. During the scan stitching process, the selection of the output pin is made by DFTAdvisor based on the lowest fanout count of each of the possible pins. If you have a preference as to which pin is used for a particular model or instance, you can also use the Add Mapping Definition command to define that pin.

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

```
add mapping definition  fd1s -output qn
```

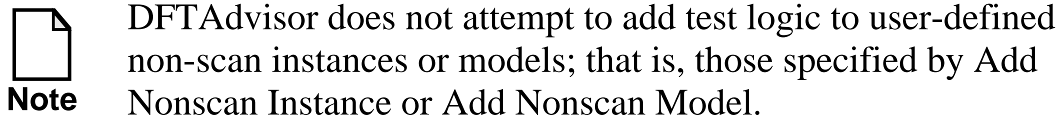
For additional information and examples on using these commands, refer to Add Mapping Definition, Delete Mapping Definition, or Report Mapping Definition in the *DFTAdvisor Reference Manual.*

Enabling Test Logic Insertion

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

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

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



Note DFTAdvisor does not attempt to add test logic to user-defined non-scan instances or models; that is, those specified by Add Nonscan Instance or Add Nonscan Model.

DFTAdvisor typically gates the uncontrollable circuitry with a chip-level test pin. Figure 5-5 shows an example of test logic circuitry.

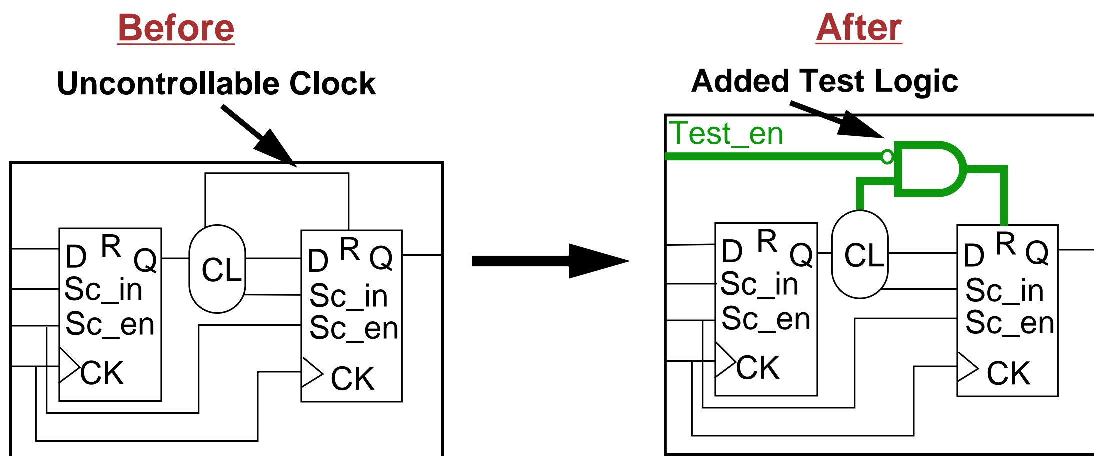


Figure 5-5. Test Logic Insertion

You can specify the types of signals for which you want test logic circuitry added, using the Set Test Logic command. This command's usage is as follows:

SET TEst Logic {**-Set** {**ON** | **OFf**} | **-REset** {**ON** | **OFf**} | **-Clock** {**ON** | **OFf**} |
-Tristate {**ON** | **OFf**} | **-RAm** {**ON** | **OFf**}}...

This command specifies whether or not you want to add test logic to all uncontrollable (set, reset, clock, or RAM write control) signals during the scan insertion process. Additionally, you can specify to turn on (or off) the ability to prevent bus contention for tri-state devices. By default, DFTAdvisor does not add test logic. You must explicitly enable the use of test logic by issuing this command.

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

Note

You must turn the appropriate test logic on if you want DFTAdvisor to consider latches as scan candidates. Refer to "D6 (Data Rule #6)" in the *Design-for-Test Common Resources Manual* for more information on scan insertion with latches.

Specifying the Models to use for Test Logic

When adding test logic circuitry, DFTAdvisor uses a number of gates from the library. The **cell_type** attribute in the library model descriptions tells DFTAdvisor which components are available for use as test logic. If the library does not contain this information, you can instead specify which library models to use with the Add Cell Models command. This command's usage is as follows:

ADD CEll Models ***dftlib_model*** {**-Type** {**INV** | **And** | {**Buf** -Max_fanout *integer*} | **OR** | **NAnd** | **NOr** | **Xor** | **INBuf** | **OUtbuf** | {**Mux** ***selector data0 data1***} | {**ScanCELL** ***clk data***} | {**DFf** ***clk data***} | {**DLat** ***enable dat*** [-Active {High | Low}]} }} [{-Noinvert | -Invert} *output_pin*]

The model_name argument specifies the exact name of the model within the library. The -Type option specifies the type of the gate. The possible cell_model_types are INV, AND, OR, NAND, NOR, XOR, BUF, INBUF, OUTBUF, DLAT, MUX, ScanCELL, and DFF.

Refer to the *DFTAdvisor Reference Manual* for more details on the Add Cell Models command.

Issues Concerning Test Logic Insertion and Test Clocks

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

In some cases, inserting test logic requires the addition of multiple test clocks. Analysis run during DRC determines how many test clocks DFTAdvisor needs to insert. The Report Scan Chains command reports the test clock pins used in the scan chains.

Related Test Logic Commands

Delete Cell Models - deletes the information specified by the Add Cell Models command.
Report Cell Models - displays a list of library cell models to be used for adding test logic circuitry.
Report Test Logic - displays a list of test logic added during scan insertion.

Specifying Clock Signals

DFTAdvisor must be aware of the circuit clocks to determine which sequential elements are eligible for scan. DFTAdvisor considers clocks to be any signals that have the ability to alter the state of a sequential device (such as system clocks, sets, and resets). Therefore, you need to tell DFTAdvisor about these “clock signals” by adding them to the clock list with the Add Clocks command. This command’s usage is as follows:

ADD CLocks ***off_state primary_input_pin****...*

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

For example, you might have two system clocks, called “clk1” and “clk2”, whose off-states are 0 and a global reset line called “rst_l” whose off-state is 1 in your circuit. You can specify these as clock lines as follows:

```
SETUP> add clocks 0 clk1 clk2
SETUP> add clocks 1 rst_1
```

You can specify multiple clock pins with the same command if they have the same off-state. You must define clock pins prior to entering Dft mode. Otherwise, none of the non-scan sequential elements will successfully pass through scannability checks. Although you can still enter Dft mode without specifying the

clocks, DFTAdvisor will not be able to convert elements which the unspecified clocks control.

Note

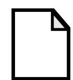
If you are unsure of the clocks within a design, you can use the Analyze Control Signals command to identify and then define all the clocks. It also defines the other control signals in the design.

Related Commands:

Delete Clocks - deletes primary input pins from the clock list.
Report Clocks - displays a list of all clocks.
Report Primary Inputs - displays a list of primary inputs.
Write Primary Inputs - writes a list of primary inputs to a file.

Specifying Existing Scan Information

You may have a design that already contains some existing internal scan circuitry. For example, one block of your design may be reused from another design, and thus, may already contain its own scan chain. If this is your situation, there are several ways in which you may want to handle the existing scan data, including leaving the existing scan alone, deleting the existing scan, and adding additional scan circuitry.


Note

If you are performing block-by-block scan synthesis, you should refer to "Inserting Scan Block-by-Block" on page 5-47.

If your design contains existing scan that you want to use, you must specify this information to DFTAdvisor while you are in Setup mode; that is, before design rules checking. If you do not specify existing scan circuitry, DFTAdvisor treats all the scan cells as non-scan cells and performs non-scan cell checks on them to determine if they are scan candidates.

If you so direct, DFTAdvisor can convert more registers from the existing design block to scan registers and connect them into another scan chain that it creates within the design. Additionally, you can remove the existing scan circuitry from the design and then treat the design as you would any other new design to which you want to add scan circuitry. This section discusses these tasks.

Specifying Existing Scan Groups

A scan chain group consists of a set of scan chains that are controlled through the same procedures; that is, the same test procedure file controls the operation of all chains in the group. If your design contains existing scan, you must specify the scan group to which they belong, as well as which test procedure file that controls the group. To specify an existing scan group, you use the Add Scan Groups command. This command's usage is as follows:

ADD SCan Groups ***group_name test_procedure_filename***

For example, you can specify a group name of "group1" controlled by the test procedure file "group1.test_proc" using the Add Scan Groups command as follows:

```
SETUP> add scan groups group1 group1.test_proc
```

For information on creating test procedure files, refer to "Test Procedure Files" on page 3-11.

Specifying Existing Scan Chains

After specifying the existing scan group, you need to tell DFTAdvisor which scan chains are part of this group. To specify existing scan chains, you use the Add Scan Chains command. This command's usage is as follows:

ADD SCan Chains ***chain_name group_name primary_input_pin primary_output_pin***

You need to specify the scan chain name, the scan group to which it belongs, and the primary input and output pins of the scan chain. For example, assume your design has two existing scan chains, "chain1" and "chain2", that are part of "group1". The scan input and output pins of chain1 are "sc_in1" and "sc_out1", and the scan input and output pins of chain2 are "sc_in2" and "sc_out2", respectively. You can specify this information as follows:

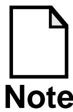
```
SETUP> add scan chain chain1 group1 sc_in1 sc_out1
SETUP> add scan chain chain2 group1 sc_in2 sc_out2
```

Deleting Existing Scan Circuitry

If your design contains existing scan that you want to delete, you must specify this information to DFTAdvisor while you are in Setup mode; that is, before design rules checking. The preceding subsection described this procedure. Then, to remove existing scan circuitry from the design, you switch to Dft mode and use the Ripup Scan Chains command as follows:

```
SETUP> set system mode dft
DFT>  ripup scan chains {chain_name ... | -all} [-Output]
```

You can specify one or more scan chain names, or use the -All option to remove all existing scan circuitry. You can also remove the scan-outs with the -Output option. Once DFTAdvisor removes the scan circuitry, it treats the design as if it never had any scan circuitry.


Note

This process involves backward mapping of scan to non-scan cells. Thus, the library you are using must have valid scan to non-scan mapping.

Handling Existing Boundary Scan Circuitry

If your design contains boundary scan circuitry and existing internal scan circuitry, you must integrate the boundary scan circuitry with the internal test circuitry. If you inserted boundary scan with BSDArchitect, then the two test structures should already be connected. "Connecting Internal Scan Circuitry" in the *Boundary Scan Process Guide* outlines the procedure. If you used some other method for generating the boundary scan architecture, you need to ensure that the scan chains' scan_in and scan_out ports connect properly to the TAP controller, in whatever manner you desire.

Changing the System Mode (Running Rules Checking)

DFTAdvisor performs model flattening, learning analysis, rules checking, and scannability checking when you try to exit the Setup system mode. "Understanding Common Tool Terminology and Concepts" on page 3-1 explains these processes in detail. If you are finished with all the setup you need to

perform, you can change the system mode by entering the Set System Mode command as follows:

SETUP> **set system mode dft**

If an error occurs during the rules checking process, the application remains in Setup mode, where you must correct the error. You can clearly identify and easily resolve the cause of many errors. Other errors, such as those associated with proper clock definitions and test procedure files, can be complex. "Troubleshooting Rules Violations" in the *Design-for-Test Common Resources Manual* discusses the procedure for debugging rules violations. You can also use DFTInsight to visually investigate the causes of DRC violations. "Using DFTInsight" in the *Design-for-Test Common Resources Manual* discusses how you can do this.

Identifying Test Structures

Prior to inserting test structures into your design, you must identify the type of test structure you want to insert. "Test Structures Supported by DFTAdvisor" on page 5-7 discusses the types of test structures DFTAdvisor supports. You identify the desired test structures in Dft mode. The following logically-ordered subsections discuss how to perform these tasks.

Selecting the Type of Test Structure

In Dft mode, you select the type of test structure you want using the Setup Scan Identification command. This command's usage for the type of test structure is as follows:

SETup SCan Identification
    Full_scan |
    {**Clock_sequential** *options*} |
    {**SEQ_transparent** *options*} |
    {**Partition_scan** *options*} |
    {**SEQUential**
        {Atpg *options*} |
        {AUtomatic *options*} |
        {SCoap *options*} |
        {STructure *options*}} |
    None

Most of these test structures include additional setup options (which are omitted from the preceding usage). Depending on your scan selection type, you should refer to one of the following subsections for additional details on the test structure type and its setup options:

- Full scan: "Setting Up for Full Scan Identification" on page 5-21
- Partial scan, clocked sequential based: "Setting Up for Clocked Sequential Identification" on page 5-21
- Partial scan, sequential transparent based: "Setting Up for Sequential Transparent Identification" on page 5-22

- Partition scan: "Setting Up for Partition Scan Identification" on page 5-22
- Sequential partial scan, including ATPG-based, Automatic, SCOAP-based, and Structure-based: "Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification" on page 5-25
- Test points (None): "Setting Up for Test Point Identification" on page 5-28
- Manual intervention for all types of identification: "Manually Including and Excluding Cells for Scan" on page 5-31

Setting Up for Full Scan Identification

If you select Full_scan as the identification type with the Setup Scan Identification command, you do not need to perform any additional setup:

SETup SCan Identification **Full_scan**

Full scan is the fastest identification method, converting all scannable sequential elements to scan. You can use FastScan for ATPG on full scan designs. This is the default upon invocation of the tool. For more information on full scan, refer to "Understanding Full Scan" on page 2-4.

Setting Up for Clocked Sequential Identification

If you select Clock_sequential as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **Clock_sequential** [-Depth integer]

Clock sequential identification selects scannable cells by cutting sequential loops and limiting sequential depth based on the -Depth switch. Typically, this method is used to create structured partial scan designs that can use FastScan's clock sequential ATPG algorithm. For more information on clock sequential scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-20.

Setting Up for Sequential Transparent Identification

If you select Seq_transparent as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQ_transparent** [-Reconvergence {ON | OFf}]

Note

This technique is useful for data path circuits. Scan cells are selected such that all sequential loops, including self loops, are cut. The -Reconvergence option specifies to remove sequential reconvergent paths by selecting a scannable instance on the sequential path for scan. For more information on sequential transparent scan, refer to "FastScan Handling of Non-Scan Cells" on page 4-20.

With the sequential transparent identification type, you do not necessarily need to perform any other tasks prior to the identification run. However, if a clock enable signal gates the clock input of a sequential element, the sequential element will not behave sequentially transparent without proper constraints on the clock enable signal.

You specify these constraints, which constrain the clock enable signals during the sequential transparent procedures, with the Add Seq_transparent Constraints command. This command's usage is as follows:

ADD SEq_transparent Constraints {**C0** | **C1**} ***model_name pin_name***...

You specify either a C0 or C1 value constraint, a library model name, and one or more of the model's pins that you wish to constrain.

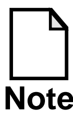
Setting Up for Partition Scan Identification

If you choose Partition_scan as the identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **Partition_scan** [-Input_threshold {*integer* | Nolimit}]
[-Output_threshold {*integer* | Nolimit}]

Partition scan identification provides controllability and observability of embedded blocks. You can also set threshold limits to control the overhead sometimes associated with partition scan identification. For example, overhead extremes may occur when DFTAdvisor identifies a large number of partition cells for a given uncontrollable primary input or unobservable primary output. By setting the partition threshold limit for primary inputs (-Input_threshold switch) and primary outputs (-Output_threshold switch), you maintain control over the trade-off of whether to scan these partitioned cells or, instead, insert a controllability/observability scan cell.

When DFTAdvisor reaches the specified threshold for a given primary input or primary output, it terminates the partition scan identification process on that primary input or primary output and unmarks any partition cell identified for that pin. For more information on partition scan, refer to "Understanding Partition Scan" on page 2-8.


Note

With the partition scan identification type, you must perform several tasks before exiting Setup mode. These tasks include specifying partition pins and setting the partition threshold. Partition pins may be input pins or output pins. You must constrain input pins to an X value and mask output pins from observation.

Constraining Input Partition Pins

Input partition pins are block input pins that you cannot directly control from chip-level primary inputs. Referring to Figure 2-7 on page 2-10, the input partition pins are those inputs that come into Block A from Block B. Because these are uncontrollable inputs, you must constrain them to an X value using the Add Pin Constraints command. This command's usage is as follows:

ADD PIn Constraints ***primary_input_pin constant_value***

Masking Output Partition Pins

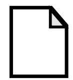
Output partition pins are block output pins that you cannot directly observe from chip-level primary outputs. Referring to Figure 2-7 on page 2-10, the output partition pins are those outputs that go to Block B and Block C. Because these are unobservable outputs, you must mask them with the Add Output Masks command.

This command's usage is as follows:

ADD OUtput Masks ***primary_output***... [-Hold {0 | 1}]

To ensure that masked primary outputs drive inactive values during the testing of other partitions, you can specify that the primary outputs hold a 0 or 1 value during test mode. Special cells called output hold-0 or output hold-1 partition scan cells serve this purpose. By default, the tool uses regular output partition scan cells.

Analyzing Controllability of Input Partition Pins


Note

This task must be performed in Dft mode.

After constraining the input partition pins to X values, you can analyze the controllability for each of these inputs. This analysis is useful because sometimes there is combinational logic between the constrained pin and the sequential element that gets converted to an input partition scan cell. Constraining a partition pin can impact the fault detection of this combinational logic. DFTAdvisor determines the controllability factor of a partition pin by removing the X constraint and calculating the controllability improvement on the affected combinational gates. You can analyze controllability of input partition pins as follows:

ANAlyze INput Control

The analysis reports the data by primary input, displaying those with the highest controllability impact first. Based on this information, you may choose to make one or more of the inputs directly controllable at the chip level by multiplexing the inputs with primary inputs.

Analyzing Observability of Output Partition Pins

Note This task must be performed in Dft mode.

Similar to the issue with input partition pins, there may be combinational logic between the sequential element (which gets converted to an output partition cell) and a masked primary output. Thus, it is useful to also analyze the observability of each of these outputs because masking an output partition pin can impact the fault detection of this combinational logic. DFTAdvisor determines the observability factor of a partition pin by removing the mask and calculating the observability improvement on the affected combinational gates. You can analyze observability of output partition pins as follows:

ANAlyze OUtput Observe

The analysis reports the data by primary output, displaying those with the highest observability impact first. Based on this information, you may choose to make one or more of the outputs directly observable by extending the output to the chip level.

Setting Up for Sequential (ATPG, Automatic, SCOAP, and Structure) Identification

If you choose to have DFTAdvisor identify instances for partial scan (Sequential), you can choose to use either the sequential ATPG algorithm of FlexTest, the SCOAP-based algorithm, or the structure-based algorithm. The following subsections discuss the ways in which you can control the process of sequential scan selection. "Running the Identification Process" on page 5-36 tells you how to identify scan cells, after setting up for partial scan identification.

Sequential ATPG-Based Identification

If you choose ATPG as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential Atpg** [{-Percent *integer*} | {-Number *integer*}] [-Internal | -External *filename*] [-COntrollability *integer*] [-Observability *integer*] [-Backtrack *integer*] [-CYcle *integer*] [-Time *integer*] [-Min_detection *floating_point*]

The benefit of ATPG-based scan selection is that ATPG runs as part of the process, giving test coverage results along the way.

Sequential Automatic Identification

If you choose Automatic as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential Automatic** [-Percent *integer* | -Number *integer*]

It is recommended that during the first scan selection and ATPG iteration, you use the default (not specifying -Percent and -Number) to allow the tool to determine the amount of scan needed. Then based on the ATPG results and how they compare to the required test coverage criteria, you can specify the exact amount of scan to select. The amount of scan selected in the first (default) iteration can be used as a reference point for determining how much more or less scan to select in subsequent iterations (i.e. what limit to specify).

Sequential SCOAP-Based Identification

If you choose SCOAP as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential SCoap** [-Percent *integer* | -Number *integer*]

SCOAP-based selection is typically faster than ATPG-based selection, and produces an optimal set of scan candidates.

Sequential Structure-Based Identification

If you choose Structure as the sequential identification type with the Setup Scan Identification command, you have the following options:

SETup SCan Identification **SEQUential STructure** [-Percent *integer* | -Number *integer*] [-Loop {ON | OFf}] [-Self_loop {*integer* | Nolimit}] [-Depth {*integer* | Nolimit}]

The Structure technique includes loop breaking, self-loop breaking, and limiting the design's sequential depth. These techniques are proven to reduce the sequential ATPG problem and quickly provide a useful set of scan candidates.

Setting Contention Checking During Partial Scan Identification

DFTAdvisor can use contention checking on tri-state bus drivers and multiple port flip-flops and latches when identifying the best elements for partial scan. You can set contention checking parameters with the Set Contention Check command, whose usage is as follows:

SET COntention Check **OFf** | {**ON** [-Warning | -Error] [-ATpg] [-Start *frame#*]} [-Bus | -Port | -ALl]

By default, contention checking is on for buses, with violations considered warnings. This means that during the scan identification process, DFTAdvisor considers the effects of bus contention and issues warning messages when two or more devices concurrently drive a bus. If you want to consider contention of clock ports of flip-flops or latches, or change the severity of this type of problem to error instead of warning, you can do so with this command. For further information on this command, refer to the Set Contention Check command page in the *DFTAdvisor Reference Manual*.

Setting Up for Test Point Identification

If you want DFTAdvisor to identify test points, you can also set a number of parameters to control the process. DFTAdvisor considers the test points it selects as system-class test points, while those you manually specify are user-class test points.

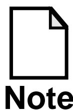
Automatically Choosing Control and Observe Points

To only identify and insert system-class test points, you must specify Setup Scan Identification command with the None option (you do not need to do this for user-added test points):

SETup SCan Identification **None**

You set the number of control and observe points with the Setup Test_point Identification command. This command's usage is as follows:

SETup TEst_point IDentification [-COntrol *integer*] [-OBserve *integer*]
[-Verbose | -NOVerbose] [-BAse {SCoap [-Internal | {-External *filename*}]}]


Note

The -Base Simulation and -Base Multiphase options (not shown here) are only available if you have a LBISTArchitect license.

DFTAdvisor bases identification on the information found in the testability analysis process. DFTAdvisor selects the pins with the highest control and observe numbers, up to the limit of test points you specify with this command. After analyzing testability and setting up for test point identification, you must then perform test point identification, which you do with the Run command. Identifying test points simply identifies, or tags, the individual test points for later insertion. Refer to "Changing the System Mode (Running Rules Checking)" on page 5-18 and "Running the Identification Process" on page 5-36 for more details on the next steps in the process.

The following locations in the design will not have test points automatically added by DFTAdvisor:

- Any site in the fanout cone of a declared clock (defined with the Add Clock command).
- The outputs of scanned latches or flip flops.
- The internal gates of library cells. Only gates driving the top library boundary can have test points.
- Notest points which are set using the Add Notest Points command.
- The outputs of primitives that can be tri-state.
- The primary inputs for control or observation points.
- The primary outputs for observation points. A primary output driver which also fans out to internal logic could have a control point added, if needed.
- No control points at unobservable sites.
- No observation points at uncontrollable sites.

Related Test Point Commands:

Delete Test Points - deletes the information specified by the Add Test Points command.

Report Test Points - displays identified/specified test points.

Manually Specifying Control and Observe Points

If you already know the places in your design that are difficult to control or observe, you can manually specify which control and observe points to add using the Add Test Points command. This command's usage is as follows:

ADD TEst Points ***tp_pin_pathname*** {{**Control** ***model_name*** ***input_pin_pathname*** [*mux_sel_input_pin*] [*scan_cell*]} | {**Observe** ***output_pin_pathname*** [*scan_cell*]} | {**Lockup** ***lockup_latch_model*** ***clock_pin*** [-INVert | -NOInvert]}}

The tp_pin_pathname argument specifies the pin pathname of the location where you want to add a control or observe point. If the location is to be a control point, you specify the Control argument with the name of the model to insert (which you define with Add Cell Models or the cell_type attribute in the library description) and pin(s) to which you want to connect the added gate. If the location is to be an observe point, you must specify the primary output in which to connect the observe point. You can also specify whether to add a scan cell at the control or observe point. Because this command encapsulates much functionality, you should refer to the Add Test Points command description in the *DFTAdvisor Reference Manual* for more details.

Analyzing the Design for Controllability and Observability of Gates

Typically, you do not know your design's best control and observe points. DFTAdvisor can analyze your design based on the SCOAP (Sandia Controllability Observability Analysis Program) approach and determine the locations of the difficult-to-control and difficult-to-observe points. To analyze the design for controllability and observability, you use the Analyze Testability command with the -Scoap_only switch:

ANAlyze TEstability -Scoap_only

To report information from the controllability and observability analysis, you use the Report Testability Analysis command, whose usage is as follows:

REPort TEstability Analysis [*pathname*] [-Controllability | -OBservability]
[{-Number *integer*} | {-Percent *integer*} | {-OVer *integer*}]

By default, the tool reports analysis information for all gates in the design. To restrict the information to all gates beneath a certain instance, you can specify an instance pathname. By default, it also lists both controllability and observability information. To list only controllability or only observability information, you can specify the -Controllability or -Observability options, respectively. The larger the controllability/observability number of a gate, the harder it is to control/observe. You can control the amount of information shown by limiting the gates reported

to an absolute number (-Number), a percentage of gates in the design (-Percent), or only those whose controllability/observability is over a certain number (-Over).

Note

The Analyze Testability and Report Testability Analysis are general purpose commands. You can use these commands at any time—not just in the context of automatic test point identification—to get a better understanding of your design's testability. They are presented in this section because they are especially useful with regards to test points.

Manually Including and Excluding Cells for Scan

Regardless of what type of scan you want to insert, you can manually specify instances or models to either convert or not convert to scan. DFTAdvisor uses lists of scan cell candidates and non-scan cells when it selects which sequential elements to convert to scan. You can add specific instances or models to either of these lists. When you manually specify instances or models to be in these lists, these instances are called *user-class* instances. *System-class* instances are those DFTAdvisor selects. The following subsections describe how you accomplish this.

Handling Cells Without Scan Replacements

When DFTAdvisor switches from Setup to Dft mode, it issues warnings when it encounters sequential elements that have no corresponding scan equivalents. DFTAdvisor treats elements without scan replacements as non-scan models and automatically adds them as system-class elements to the non-scan model list. You can display the non-scan model list using the Report Nonscan Model or Report Dft Check command.

In many cases, a sequential element may not have a scan equivalent of the currently selected scan type. For example, a cell may have an equivalent mux-DFF scan cell but not an equivalent LSSD scan cell. If you set the scan type to LSSD, DFTAdvisor places these models in the non-scan model list. However, if you change the scan type to mux-DFF, DFTAdvisor updates the non-scan model list, in this case removing the models from the non-scan model list.

Specifying Non-Scan Components

DFTAdvisor keeps a list of which components it must exclude from scan identification and replacement. To exclude particular instances from the scan identification process, you use the Add Nonscan Instance command. This command's usage is as follows:

ADD NONscan Instances ***pathname***... [-INStance | -Control_signal | -Module]

For example, you can specify that I$155/I$117 and /I$155/I$37 are sequential instances you *do not* want converted to scan cells by specifying:

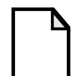
SETUP> **add nonscan instance /I$155/I$117 /I$155/I$37**

Another method of eliminating some components from consideration for scan cell conversion is to specify that certain models should not be converted to scan. To exclude all instances of a particular model type, you can use the Add Nonscan Models command. This command's usage is as follows:

ADD NONscan Models ***model_name***...

For example, the following command would exclude all instances of the dff_3 and dff_4 components from scan cell conversion.

SETUP> **add nonscan models dff_3 dff_4**


Note

DFTAdvisor automatically treats sequential models without scan equivalents as non-scan models, adding them to the nonscan model list.

Using the Dont_Touch Property

If you are using a Genie format, you have a third option in which to specify non-scan components. DFTAdvisor recognizes the "dont_touch" property associated with memory elements in the Genie netlist. Instances tagged with the "dont_touch" property are added to the non-scan instance list and treated the same as instances you specify with the Add Nonscan Instance command. However, if DFTAdvisor tags the instance as non-scan in this manner, it lists the instance as a system-class non-scan instance, rather than a user-class non-scan instance, when it reports information.

Specifying Scan Components

After you decide which specific instances or models you do *not* want included in the scan conversion process, you are ready to identify those sequential elements you *do* want converted to scan. The instances you add to the scan instance list are called user-class instances.

To include particular instances in the scan identification process, use the Add Scan Instances command. This command's usage is as follows:

ADD SCan Instances ***pathname***... [-INStance | -Control_signal | -Module]
[-INPut | -Output | {-Hold {0 | 1}}]

This command lets you specify individual instances, hierarchical instances (for which all lower-level instances are converted to scan), or control signals (for which all instances controlled by the signals are converted to scan).

For example, the following command ensures the conversion of instances */I$145/I$116* and */I$145/I$138* to scan cells when DFTAdvisor inserts scan circuitry.

SETUP> **add scan instances /I$145/I$116 /I$145/I$138**

To include all instances of a particular model type for conversion to scan, use the Add Scan Models command. This command's usage is as follows

ADD SCan Models ***model_name***...

For example, the following command ensures the conversion of all instances of the component models dff_1 and dff_2 to scan cells when DFTAdvisor inserts scan circuitry.

SETUP> **add scan models dff_1 dff_2**

For more information on these commands, refer to the Add Scan Instances and Add Scan Models reference pages in the *DFTAdvisor Reference Manual*.

Related Scan and Nonscan Commands

Delete Nonscan Instances - deletes instances from the non-scan instance list.
Delete Nonscan Models - deletes models from the non-scan model list.
Delete Scan Instances - deletes instances from the scan instance list.

Delete Scan Models - deletes models from the scan model list.
Report Nonscan Instances - displays the instances in the non-scan instance list.
Report Nonscan Models - displays the models in the non-scan instance list.
Report Scan Instances - displays instances in the scan instance list.
Report Scan Models - displays models in the scan model list.

Reporting Scannability Information

Scannability checking is a modified version of clock rules checking that determines which non-scan sequential instances to consider for scan. You may want to examine information regarding the scannability status of all the non-scan sequential instances in your design. To display this information, you use the Report Dft Check command, whose usage is as follows:

REPort DFt Check [-All | *instance_pathname*...] {[-FIlename *filename*]
[-REplace]} [-FUll | -Scannable | -Nonscannable | {-Defined {Scan | Nonscan}
| -Identified | -Unidentified | {-RUle {S1 | S2 | S3}} | -Tristate | -RAm]

This command displays the results of scannability checking for the specified non-scan instances, for either the entire design or the specified (potentially hierarchical instance).

When you perform a Report Dft Check command there is typically a large number of nonscan instances displayed, as shown in the sample report in Figure 5-6.

```
SCANNABLE       IDENTIFIED      CLK0_7      /I_3    dff (156)
SCANNABLE       IDENTIFIED      CLK0_7      /I_2    dff (157)
SCANNABLE       IDENTIFIED      CLK0_7      /I_235  dff (158)
SCANNABLE       IDENTIFIED      CLK0_7      /I_237  dff (159)
SCANNABLE       IDENTIFIED      CLK0_7      /I_236  dff (160)
SCANNABLE       IDENTIFIED      Test-logic  /I_265  dff (161)
  Clock #1: F   /I_265/clk
SCANNABLE       IDENTIFIED      Test-logic  /I_295  dff (162)
  Clock #1: F   /I_295/clk
SCANNABLE       IDENTIFIED      Test-logic  /I_298  dff (163)
  Clock #1: F   /I_298/clk
SCANNABLE       IDENTIFIED      Test-logic  /I_296  dff (164)
  Clock #1: F   /I_296/clk
SCANNABLE       IDENTIFIED      Test-logic  /I_268  dff (165)
  Clock #1: F   /I_268/clk
SCANNABLE       IDENTIFIED      CLK0_7      /I_4    dff (166)
SCANNABLE       IDENTIFIED      CLK0_7      /I_1    dff (167)
SCANNABLE       DEFINED-NONSCAN Test-logic  /I_266  dfscc (168) Stable-high
  Clock #1: F   /I_266/clk
SCANNABLE       DEFINED-NONSCAN CLK0_7      /I_238  dfscc (169)
SCANNABLE       DEFINED-NONSCAN Test-logic  /I_297  dfscc (170) Stable-high
  Clock #1: F   /I_297/clk
SCANNABLE       DEFINED-NONSCAN Test-logic  /I_267  dfscc (171) Stable-high
  Clock #1: F   /I_267/clk
```

Figure 5-6. Example Report from Report Dft Check Command

The fields at the end of each line in the nonscan instance report provide additional information regarding the classification of a sequential instance. Using the instance /I_266 (highlighted in maroon), the “Clock” statement indicates a problem with the clock input of the sequential instance. In this case when a trace back of the clock is performed. The signal doesn’t trace back to a defined clock. The message indicates that the signal traced is connected to the clock input of this non-scan instance and doesn’t trace back to a primary input defined as a clock. If several nodes are listed (similarly for “Reset” and “Set), it means that the line is connected to several endpoints (sequential instances or primary inputs).

This “Clock # 1 F /I_266/clk” issue can be resolved by either defining the specified input as a clock or allowing DFTAdvisor to add a test clock for this instance.

Related Commands:

Report Control Signals - displays control signal information.
Report Statistics - displays a statistics report.

Report Scan Identification - displays identified and/or defined scan instances.

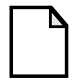
Running the Identification Process

Once you complete the proper setup, you can simply run the identification process for any of the test structures as follows:

DFT> **run**

While running the identification process, this command issues a number of messages about the identified structures.

You may perform multiple identification runs within a session, changing the identification parameters each time. However, be aware that each successive scan identification run adds to the results of the previous runs. For more information on which scan types you can mix in successive runs, refer to Table 5-1 on page 5-9.


Note

If you want to start the selection process anew each time, you must use the Reset State command to clear the existing scan candidate list.

Reporting Identification Information

If you want a statistical report on all aspects of scan cell identification, you can enter the DFTAdvisor command:

DFT> **report statistics**

This command lists the total number of sequential instances, user-defined non-scan instances, user-defined scan instances, system-identified scan instances, scannable instances with test logic, and the scan instances in pre-existing chains identified by the rules checker.

Related Commands:

Report Scan Identification - displays identified/specified scan instances.
Write Scan Identification - writes identified/specified scan instances to a file.

Inserting Test Structures

Typically, after identifying the test structures you want, you perform some test synthesis setup and then insert the structures into the design. The additional setup varies somewhat depending on the type of test structure you select for insertion. The following logically-ordered subsections discuss how to perform these tasks.

Setting Up for Internal Scan Insertion

As part of the internal scan insertion setup, you may want to set some scan chain parameters, such as the scan input and output port names and the enable and clock ports. If you specify a port name that matches an existing port of the design, the existing port is used as the scan port. If the specified port name does not exist, DFTAdvisor creates a new port with the specified name. If you use an existing connected output port, DFTAdvisor also inserts a mux at the output to select data from either the scan chain or the design, depending on the value of the scan enable signals.

Naming Scan Input and Output Ports

Before DFTAdvisor stitches the identified scan instances into a scan chain, it needs to know the names of various pins, such as the scan input and scan output. If the pin names you specify are existing pins, DFTAdvisor will connect the scan circuitry to those pins. If the pin names you specify do not exist, DFTAdvisor adds these pins to the design. By default, DFTAdvisor adds pins for chainX scan ports and names them scan_inX and scan_outX (where X represents the number of the chain).

To give scan ports specific names (other than those created by default), you can use the Add Scan Pins command. This command's usage is as follows:

ADD SCan Pins ***chain_name scan_input_pin scan_output_pin*** [-Clock *pin_name*] [-Cut] [-Registered]

You must specify the scan chain name, the scan input pin, and the scan output pin. Additionally, you may specify the name of the scan chain clock. For existing pins, you can specify top module pins or dangling pins of lower level modules.

Related Commands:

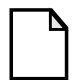
Delete Scan Pins - deletes scan chain inputs, outputs, and clock names.
Report Scan Pins - displays scan chain inputs, outputs, and clock names.
Setup Scan Pins - specifies the index or bus naming conventions for scan input and output pins.

Naming the Enable and Clock Ports

The enable and clock parameters include the pin names of the scan enable, test enable, test clock, new scan clock, scan master clock, and scan slave clock. Additionally, you can specify the names of the set and reset ports and the RAM write and read ports in which you want to add test logic, along with the type of test logic to use. You do this using the Setup Scan Insertion command. This command's usage is as follows:

SETup SCan INsertion [{-SEN name | -TEn name} [-Active {Low | High}]}]
[-TClk name] [-SClk name] [-SMclk name] [-SSclk name] {{[-SET name] |
[-RESet name] | [-Write name] | [-REAd name]}... [-Muxed | -Disabled |
-Gated]}

If you do not specify this command, the default pins names are scan_en, test_en, test_clk, scan_clk, scan_mclk, scan_sclk, scan_set, scan_reset, write_clk, and read_clk, respectively. If you want to specify the names of existing pins, you can specify top module pins or dangling pins of lower level modules.


Note

if DFTAdvisor adds more than one test clock, it names the first test clock the specified or default <name> and names subsequent test clocks based on this name plus a unique number.

The -Muxed and -Disabled switches specify whether DFTA uses an AND gate or MUX gate when performing the gating. If you specify the -Disabled option, then for gating purposes DFTAdvisor ANDs the test enable signal with the set and reset to disable these inputs of flip-flops. If you specify the -Muxed option, then for muxing purposes DFTA uses any set and reset pins defined as clocks to multiplex with the original signal. You can specify the -Muxed and -Disabled switches for individual pins by successively issuing the Setup Scan Insertion command.

If DFTAdvisor writes out a test procedure file, it places the scan enable at 1 (0) if you specify -Active high (low).

Note

If the test enable and scan enable have different active values, you must specify them separately in different Setup Scan Insertion commands. For more information on the Setup Scan Insertion command, refer to the *DFTAdvisor Reference Manual.*

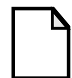
After setting up for internal scan insertion, refer to "Running the Insertion Process" on page 5-41 to complete insertion of the internal scan circuitry.

Attaching Head and Tail Registers to the Scan Chain

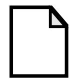
You can have DFTAdvisor attach the head and tail registers to the scan chain for MUX scan type. A *head register* is a non-scan DFF connected at the beginning of a scan chain. This DFF is clocked using the shift clock of the scan chain. If the scan chain has multiple shift clocks, any one of those clocks can be used for the head register. A *tail register* is a scan, DFF connected at the end of the scan chain. Clocking of the tail register is similar to that of the head register.

DFTAdvisor uses the head register (specified by the scan_input_pin) and the tail register (specified by the scan_output_pin) to determine the beginning and ending points of the scan chain. Scan cells are inserted between these registers.

During test logic insertion, DFTAdvisor attaches the non-scan head register's output to the beginning of the scan chain, performs scan replacement on the tail register, and then attaches the scan tail register's input to the end of the scan chain. If there is no scan replacement in the ATPG library for the tail register, a MUX is added to include the tail DFF into the scan chain.


Note

No design rule checks are performed from the scan_in pin to the output of the head register and from the output of the tail register to the scan_out pin. You are responsible for making those paths transparent for scan shifting.


Note

DFTAdvisor does not determine the associated top-level pins that are required to be identified for the Add Scan Chains command. You are responsible for adding this information to the dofile that DFTAdvisor creates using the Write ATPG Setup command. You must also provide the pin constraints that cause the correct behavior of the head and tail registers.

To attach registers to the head and tail of the scan chain, you can use the Add Scan Pins command, specifying the scan input (head register output pin) and scan output (tail register input pin) of the registers along with the -Registered switch. This command's usage is as follows:

ADD SCan Pins ***chain_name scan_input_pin scan_output_pin*** [-Clock *pin_name*] [-Cut] [-Registered]

For more information on the Add Scan Pins command, refer to the *DFTAdvisor Reference Manual.*

Setting Up for Test Point Insertion

When adding test points, you can specify whether control inputs come from primary inputs or scan cells. Likewise, you can specify whether observe outputs go to primary outputs or scan cells. You perform these tasks using the Setup Test_point Insertion command. This command's usage is as follows:

SETup TEst_point INsertion [-Control *input_pin_name*] [-Observe *output_pin_name*] [-None | -Model *modelname*] [-REconvergence {OFf | ON}]

If you want the control input to be a DFF/SDFF scan cell or the observe output to be a SDFF scan cell, you specify the -Model switch with the name of the appropriate library cell. The -Control switch either specifies the pin_pathname to the clock input of the DFF/SDFF scan cell (if the -Model switch was used) or the pin_pathname of the control input. The -Observe switch either specifies the pin_pathname of the clock input of the SDFF scan cell (if the -Model switch was used) or the pin_pathname of the observe output.

After setting up for test point insertion, refer to "Running the Insertion Process" on page 5-41 to complete insertion of the test point circuitry.

Buffering Test Pins

When the tool inserts scan into a design, the test pins (such as scan enable, test enable, test clock, scan clock, scan master clock, and scan slave clock) may end up driving a lot of fanouts. If you want DFTAdvisor to limit the number of fanouts and insert buffer trees instead, you can use the Add Buffer Insertion command.

This command's usage is as follows:

ADD BUffer Insertion ***max_fanout test_pin*** [-Model *modelname*]

The max_fanout option must be a positive integer greater than one. The test_pin option must have one of the following values: SEN, TEN, SCLK, SMCLK, SSCLK, TCLK, SET, or RESET. The -Model option specifies the name of the library buffer model to use to buffer the test pins.

Related Commands:
Delete Buffer Insertion - deletes added buffer insertion information.
Report Buffer Insertion - displays inserted buffer information.

Running the Insertion Process

The Insert Test Logic command inserts all of the previously identified test structures into the design. This includes internal scan (full, sequential, and scan-sequential types), partition scan, test logic, and test points.

When you issue this command for scan insertion (assuming appropriate prior setup), DFTAdvisor converts all identified scannable memory elements to scan elements and then stitches them into one or more scan chains. If you select partition scan for insertion, DFTAdvisor converts the non-scan cells identified for partition scan to partition scan cells and stitches them into scan chains separate from internal scan chains.

The scan circuitry insertion process may differ depending on whether you insert scan cells and connect them up front or insert and connect them after layout data is available. DFTAdvisor allows you to insert scan using both methods.

To insert scan chains and other test structures into your design, you use the Insert Test Logic command. This command's usage is as follows:

INSert TEst Logic [*filename* [-Fixed]] [-Scan {ON | OFf}] [-Test_point {ON | OFf}] [-Ram {ON | OFf}] {[-NOlimit] | [-Max_length *integer*] | [-NUmber [*integer*]]} [-Clock {Nomerge | Merge}] [-Edge {Nomerge | Merge}] [-COnnect {ON | OFf | Tied}] [-Output {Share | New}] [-MOdule {Norename | Rename}]

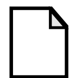
The Insert Test Logic command has a number of different options, most of which apply primarily to internal scan insertion.

- If you are using specific cell ordering, you can specify a filename of user-identified instances (in either a fixed or random order) for the stitching order.
- The -Max_length option lets you specify a maximum length to the chains.
- The -NOlimit switch allows an unlimited chain length.
- The -NUmber option lets you specify the number of scan chains for the design.
- The -Clock switch lets you choose whether to merge two or more clocks on a single chain.
- The -Edge switch lets you choose whether to merge stable high clocks with stable low clocks on chains.

  The subsection that follows, "Merging Chains with Different Shift Clocks", discusses some of the issues surrounding merging chains with different clocks.
- The -COnnect option lets you specify whether to connect the scan cells and scan-specific pins (scan_in, scan_enable, scan_clock, etc.) to the scan chain (which is the default mode), or just replace the scan candidates with scan equivalent cells. If you want to use layout data, you should replace scan cells (using the -connect off switch), perform layout, obtain a placement order file, and then connect the chain in the appropriate order (using the -filename <filename> -fixed options).

- The -Scan, -Test_point, and -Ram switches let you turn scan insertion, test point insertion and RAM gating on or off.

If you do not specify any options, DFTAdvisor stitches the identified instances into default scan chain configurations. Since this command contains many options, refer to the Insert Test Logic command reference page for additional information.


Note

Because the design is significantly changed by the action of this command, DFTAdvisor frees up (or deletes) the original flattened, gate-level simulation model it created when you entered the DFT system mode.

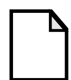
Merging Chains with Different Shift Clocks

DFTAdvisor lets you merge scan cells with different shift clocks into the same scan chain. However, to avoid synchronization problems, DFTAdvisor can do two things: 1) place cells using the same clock adjacent to each other in the chain, and 2) place synchronization latches between the differently-clocked groups.

When you have cells that do not share the same shift clock, you can have them use the same scan chain by adding them to a *clock group*. This informs DFTAdvisor which scan cells to place together in the chain. You specify clock groups using the Add Clock Groups command, whose usage is as follows:

ADD CLock Groups ***group_name clk_pin*** [-Tclk]

You must give a name to the group containing scan cells controlled by the specified clock(s). The clock pins you specify include those you added with the Add Clocks command as well as the test clock pin (added during scan insertion).


Note

To have the clocks merged into one, you must specify the “-Clock merge” option when specifying the Insert Scan Chains command.

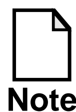
Once DFTAdvisor has the clock group information, it determines where to place the synchronization latches, or *lockup latches*. These latches synchronize the

clock domains within the chain. Lockup latches are only inserted between clock domains of a clock group, not between clock groups.

If you want to insert lockup latches, you must first specify the two-input D latch you want to use with the Add Cell Models command. You specify for DFTAdvisor to insert lockup latches with the Set Lockup Latch command. This command's usage is as follows:

SET LOckup Latch {**ON** | **OFf**} [-NOLast | -Last] [-First_clock | -SEcond_clock] [-STABLE_High latch_model1] [-STABLE_Low latch_model2] [-Internal | -NOInternal]

By default, DFTAdvisor does not insert lockup latches between clock domains. You must turn this functionality on if you want lockup latches inserted. If you turn the functionality on, DFTAdvisor inserts lockup latches between the last scan cell of one clock group and the first scan cell of the next clock group. In order to insert lockup latches you must have defined a clock group.



If you want to insert one scan chain with lockup latches, you must add *all* clocks to a clock group.

Figure 5-7 illustrates lockup latch insertion.

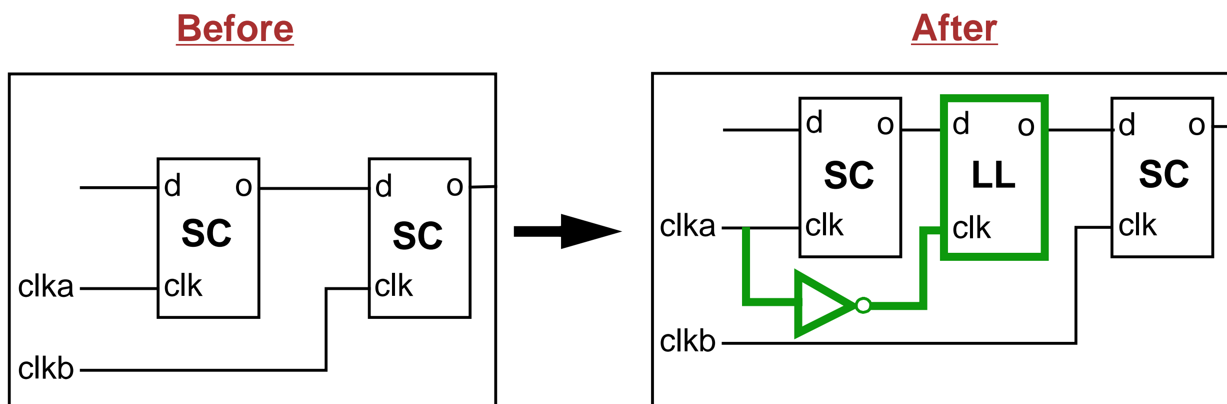


Figure 5-7. Lockup Latch Insertion

DFTAdvisor can also insert a lockup latch between the last scan cell in the chain and the scan out pin, if you specify the -Last option. The -Nolast option is the

default, which means DFTAdvisor does not insert a lockup latch as the last element in the chain.

Related Commands:

Delete Clock Groups - deletes the specified clock groups.
Report Clock Groups - reports the added clock groups.
Report Dft Check - displays and writes the scannability check status for all non-scan instances.
Report Scan Cells - displays a list of all scan cells.
Report Scan Chains - displays scan chain information.
Report Scan Groups - displays scan chain group information.

Saving the New Design and ATPG Setup

After test structure insertion, DFTAdvisor releases the current flattened model and has a new hierarchical netlist in memory. Thus, you should save this new version of your design. Additionally, you should save any design information that the ATPG process might need.

Writing the Netlist

You can save the netlist for your new design by issuing the Write Netlist command. This command's usage is as follows:

WRIte NEtlist ***filename*** [-Edif | -Tdl | -Verilog | -VHdl | -Genie | -Ndl] [-Replace]

Issues with the New Version of the Netlist

The following lists some important issues concerning netlist writing:

- DFTAdvisor is not intended for use as a robust netlist translation tool. Thus, you should always write out the netlist in the same format in which you read the original design.
- If a design contains only one instantiation of a module, and DFTAdvisor modifies the instance by adding test structures, the instantiation retains the original module name.

- When DFTAdvisor identically modifies two or more instances of the same module, all modified instances retain the original module name. This generally occurs for full scan designs.
- If a design contains multiple instantiations of a module, and DFTAdvisor modifies them differently, DFTAdvisor derives new names for each instance based on the original module name.
- DFTAdvisor assigns "net" as the prefix for new net names and "uu" as the prefix for new instance names. It then compares new names with existing names (in a case-insensitive manner) to check for naming conflicts. If it encounters naming conflicts, it changes the new name by appending an index number.
- When writing directory-based Genie netlists, DFTAdvisor writes out modules based on directory names in uppercase. Instance names within the netlist remain in their original case.

Writing the Test Procedure File and Dofile for ATPG

If you plan to use FastScan or FlexTest for ATPG, you can use DFTAdvisor to create a dofile (for setting up the scan information) and a test procedure file (for operating the inserted scan circuitry). For details on test procedure files, refer to "Test Procedure Files" on page 3-11.

You can tell DFTAdvisor to create these files for you by issuing the Write Atpg Setup command. This command's usage is as follows:

WRIte ATpg Setup ***basename*** [-Replace]

The tool uses the <basename> argument to name the dofile (*<basename>.dofile*) and test procedure file (*<basename>.testproc*). You can overwrite existing files using the -Replace switch.

Running Rules Checking on the New Design

You can verify the correctness of the added test circuitry by running the full set of rules checks on the new design. To do this, return to Setup mode after scan insertion, delete the circuit setup, run the dofile produced for ATPG, and then

return to Dft mode. This enables rules checking on the added scan circuitry to ensure it operates properly before you go to the ATPG process.

For example, if DFTAdvisor added a single scan chain and wrote out an ATPG setup file named *scan_design.dofile*, you could enter:

```
DFT> set system mode setup
SETUP> delete clocks -all
SETUP> dofile scan_design.dofile
SETUP> set system mode dft
```

Exiting DFTAdvisor

When you are finished with the DFTAdvisor session, you exit the application by executing the **File > Exit** menu item, by clicking on the Exit button in the Control Panel window, or by typing:

```
DFT> exit
```

Inserting Scan Block-by-Block

Scan insertion is “block-by-block” when DFTAdvisor first inserts scan into lower-level hierarchical blocks and then connects them together at a higher level of hierarchy. For example, Figure 5-8 shows a module (Top) with three submodules (A, B, and C).

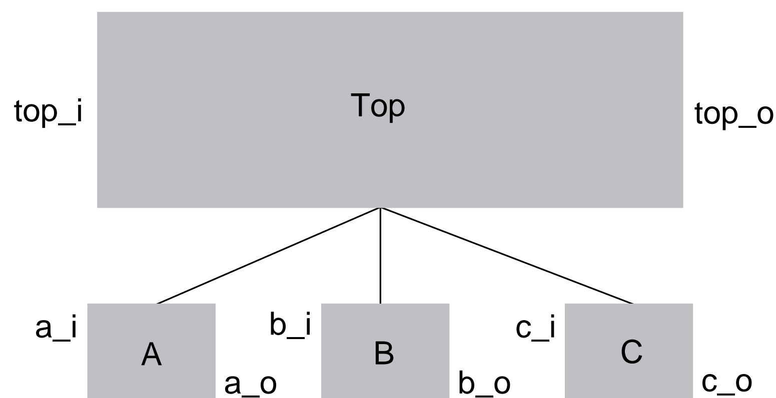


Figure 5-8. Hierarchical Design Prior to Scan

Using block-by-block scan insertion, the tool inserts scan (referred to as "sub-chains) into blocks A, B, and C, prior to insertion in the Top module. When A, B, and C already contain scan, inserting scan into the Top module is equivalent to inserting any scan necessary at the top level and then connecting the existing scan circuitry in A, B, and C at the top level.

Verilog and EDIF Flow Example

The following shows the basic procedure for adding scan circuitry block-by-block, as well as the input and results of each step. Assume the design is a Verilog netlist (although EDIF netlists follow the same flow).

1. **Insert scan into block A.**

   a. Invoke DFTAdvisor on *a.hdl*.
   Assume that the module interface is:

   ```
   A(a_i, a_o)
   ```

   b. Insert scan.
   Set up the circuit, run rules checking, insert the desired scan circuitry.

   c. Write out scan-inserted netlist.
   Write the scan-inserted netlist to a new filename, such as *a_scan.hdl*. The new module interface may differ, for example:

   ```
   A(a_i, a_o, sc_i, sc_o, sc_en)
   ```

   d. Write out the subchain dofile.
   Use the Write Subchain Setup command to write a dofile called *a.do* for the scan-inserted version of A. The Write Subchain Setup command uses the Add Sub Chain command to specify the scan circuitry in the individual module of the design. Assuming that you use the mux-DFF scan style and the design block contains 7 sequential elements converted to scan, the subchain setup dofile could appear as follows:

   DFT> **add sub chains /user/jdoe/designs/design1/A chain1 sc_i sc_o 7 mux_scan sc_en**

   e. Exit DFTAdvisor.

2. **Insert scan into block B.**
   Follow the same procedure as in block A.

3. **Insert scan into block C.**
   Follow the same procedure as in blocks A and B.

4. **Concatenate the individual scan-inserted netlists into one file.**

```
$ cat top.hdl a_scan.hdl b_scan.hdl c_scan.hdl > all.hdl
```

5. **Stitch together the chains in blocks A, B, and C.**

   a. Invoke DFTAdvisor on *all.hdl*.
      Assume at this point that the module interface is:

```
TOP(top_i, top_o)
A(a_i, a_o, sc_i, sc_o, sc_en)
B(b_i, b_o, sc_i, sc_o, sc_en)
C(c_i, c_o, sc_i, sc_o, sc_en)
```

   b. Run each of the scan subchain dofiles (*a.do, b.do, c.do*).

   c. Insert the desired scan circuitry into the *all.hdl* design.

6. **Write out the netlist and exit.**

   At this point the module interface is:

```
TOP(top_i, top_o, sc_i, sc_o, sc_en)
A(a_i, a_o, sc_i, sc_o, sc_en)
B(b_i, b_o, sc_i, sc_o, sc_en)
C(c_i, c_o, sc_i, sc_o, sc_en)
```

   Figure 5-9 shows a schematic view of the design with scan connected in the Top module.

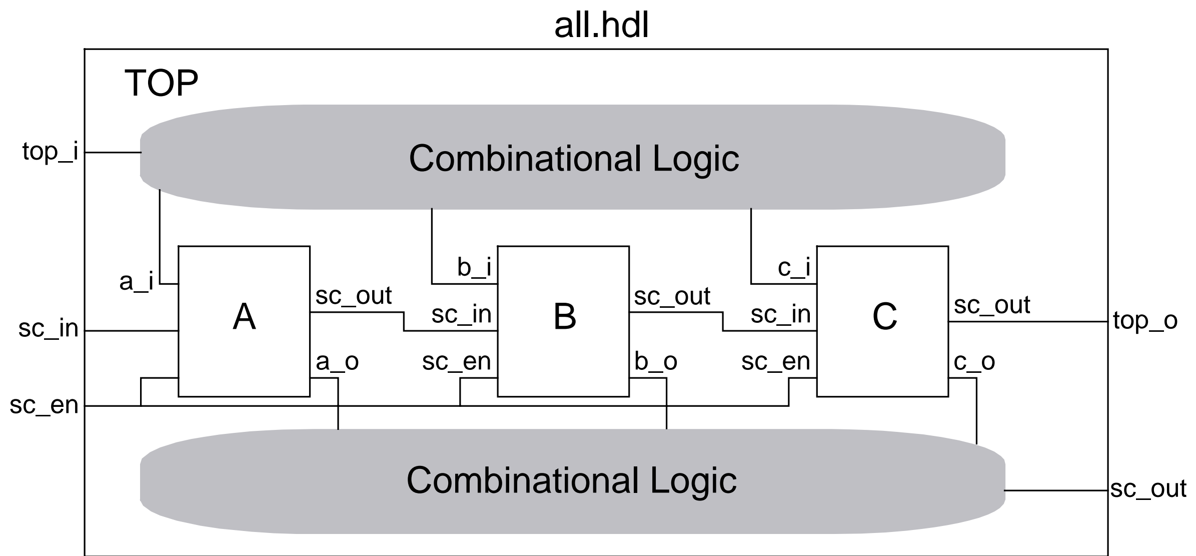
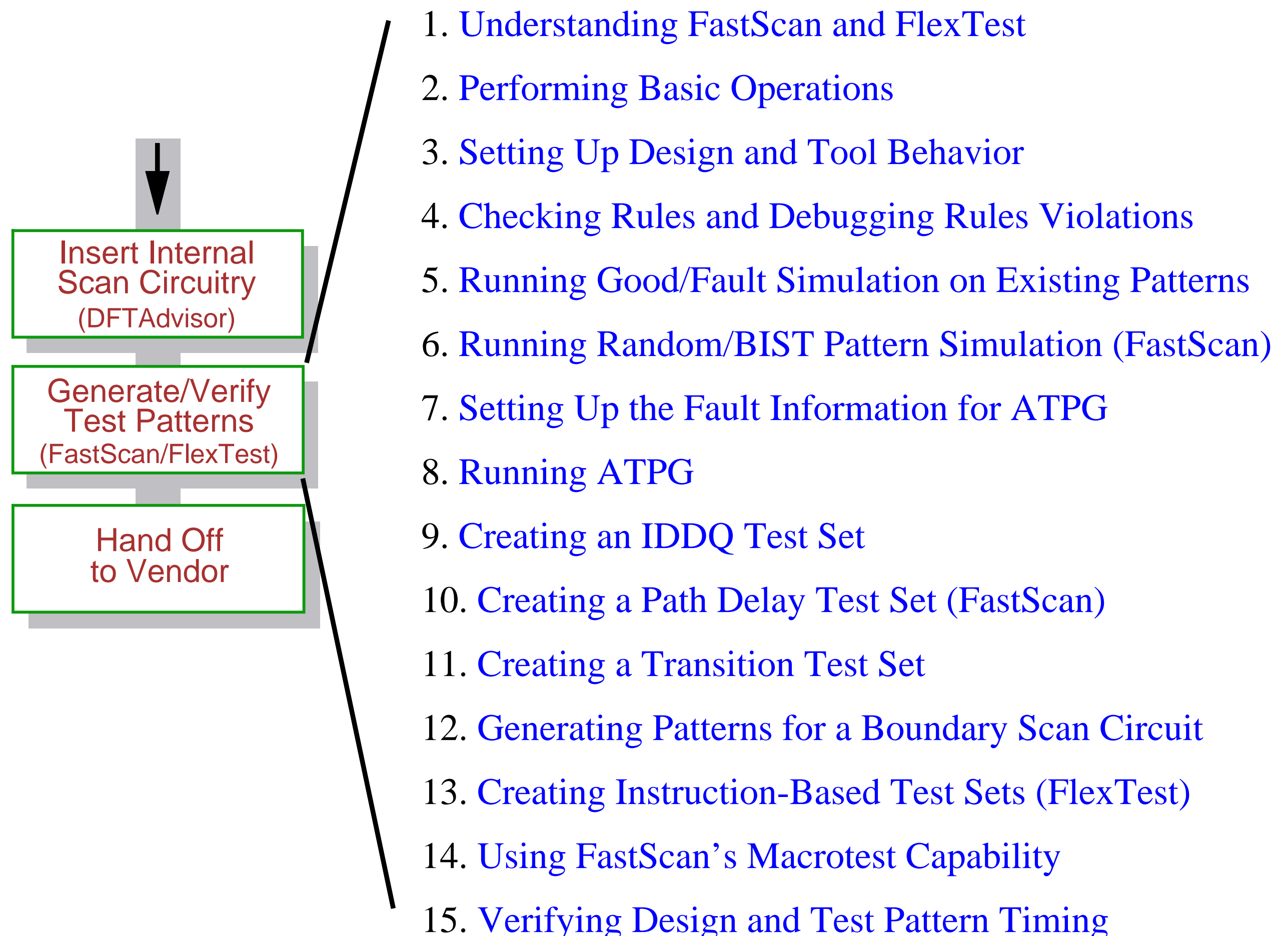


Figure 5-9. Final Scan-Inserted Design

Chapter 6
Generating Test Patterns

FastScan and FlexTest are the Mentor Graphics ATPG tools for generating test patterns. Figure 6-1 shows the layout of this chapter and the process for generating test patterns for your design.



1. Understanding FastScan and FlexTest
2. Performing Basic Operations
3. Setting Up Design and Tool Behavior
4. Checking Rules and Debugging Rules Violations
5. Running Good/Fault Simulation on Existing Patterns
6. Running Random/BIST Pattern Simulation (FastScan)
7. Setting Up the Fault Information for ATPG
8. Running ATPG
9. Creating an IDDQ Test Set
10. Creating a Path Delay Test Set (FastScan)
11. Creating a Transition Test Set
12. Generating Patterns for a Boundary Scan Circuit
13. Creating Instruction-Based Test Sets (FlexTest)
14. Using FastScan's Macrotest Capability
15. Verifying Design and Test Pattern Timing

Figure 6-1. Test Generation Procedure

This section discusses each of the tasks outlined in Figure 6-1. You will use FastScan and/or FlexTest (and possibly QuickSim II and QuickFault, depending on your test strategy) to perform these tasks.

Understanding FastScan and FlexTest

FastScan and FlexTest functionality is available in two modes: graphical user interface (GUI) or command-line. For more information on using basic GUI functionality, refer to the following sections in Chapter 1: "User Interface Overview" on page 1-9, "FastScan User Interface" on page 1-23 and "FlexTest User Interface" on page 1-25.

Before you use FastScan and/or FlexTest, you should learn the basic process flow, the tool's inputs and outputs, and its basic operating methods. The following subsections describe this information.

You should also have a good understanding of the material in both Chapter 2, "Understanding Scan and ATPG Basics", and Chapter 3, "Understanding Common Tool Terminology and Concepts".

FastScan and FlexTest Basic Tool Flow

Figure 6-2 shows the basic tool flow for FastScan and/or FlexTest.

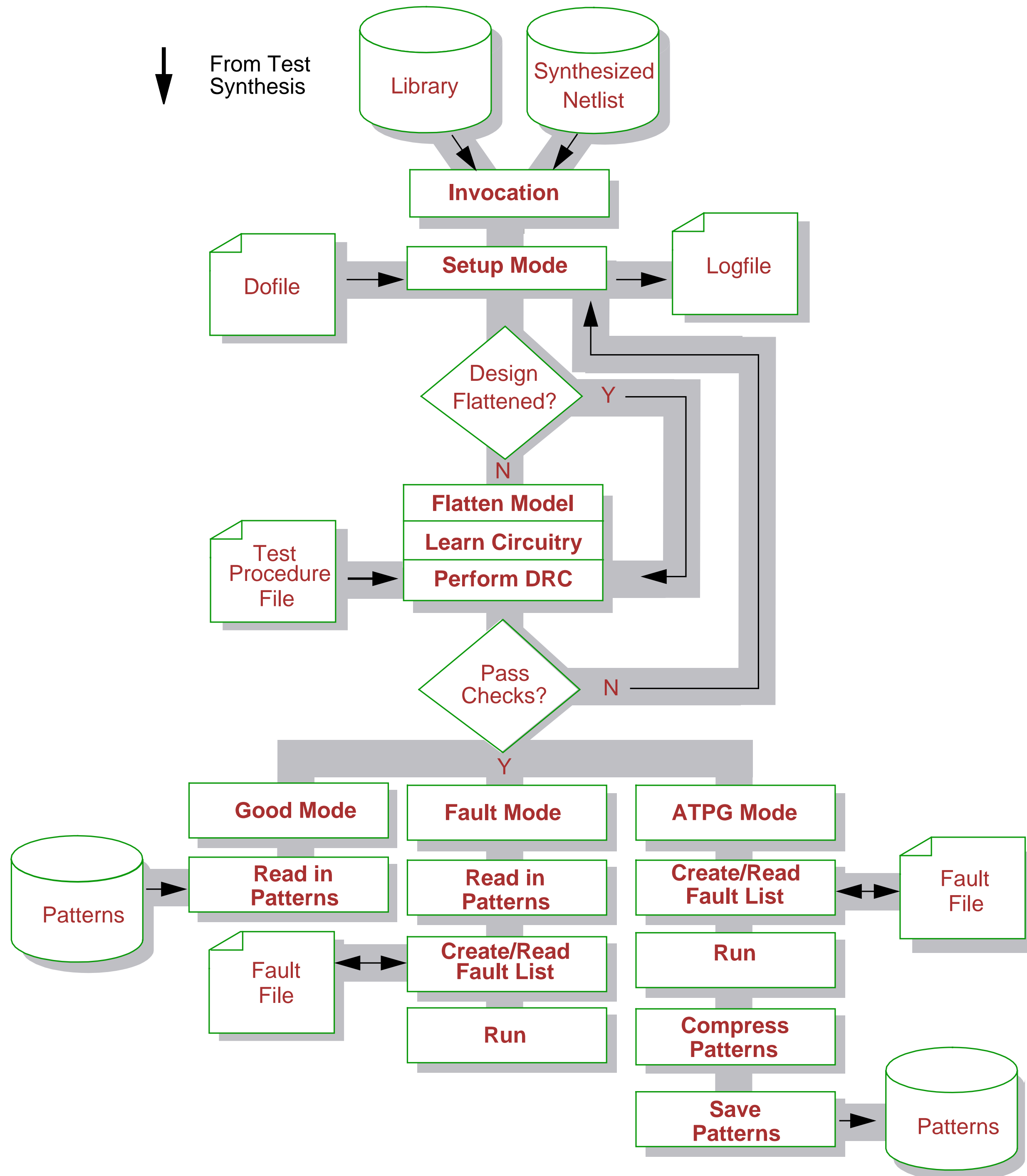


Figure 6-2. Overview of FastScan/FlexTest Usage

The following list describes the basic process for using FastScan and/or FlexTest:

1. FastScan and FlexTest require a structural (gate-level) design netlist and a DFT library. "FastScan and FlexTest Inputs and Outputs" on page 6-6 describes which netlist formats you can use with FastScan and FlexTest. Every element in the netlist must have an equivalent description in the specified DFT library. The "Design Library" section in the *Design-for-Test Common Resources Manual* gives information on the DFT library. At invocation, the tool first reads in the library and then the netlist, parsing and checking each. If the tool encounters an error during this process, it issues a message and terminates invocation.

2. After a successful invocation, the tool goes into Setup mode. Within Setup mode, you perform several tasks, using commands either interactively or through the use of a dofile. You can set up information about the design and the design's scan circuitry. "Setting Up Design and Tool Behavior" on page 6-24 documents this setup procedure. Within Setup mode, you can also specify information that influences simulation model creation during the design flattening phase.

3. After performing all the desired setup, you can exit the Setup mode. Exiting Setup mode triggers a number of operations. If this is the first attempt to exit Setup mode, the tool creates a flattened design model. This model may already exist if a previous attempt to exit Setup mode failed or you used the Flatten Model command. "Model Flattening" on page 3-29 provides more detail on design flattening.

4. Next, the tool performs extensive learning analysis on this model. "Learning Analysis" on page 3-36 explains learning analysis in more detail.

5. Once the tool creates a flattened model and learns its behavior, it begins design rules checking. The "Design Rules Checking" section in the *Design-for-Test Common Resources Manual* gives a full discussion of the design rules.

6. Once the design passes rules checking, the tool enters either Good, Fault, or Atpg mode. While typically you would enter the Atpg mode, you may want to perform good machine simulation on a pattern set for the design. "Good Machine Simulation" on page 6-50 describes this procedure.

7. You may also just want to fault simulate a set of external patterns. "Fault Simulation" on page 6-45 documents this procedure.

8. At this point, you might typically want to create patterns. However, you must perform some additional setup, such as creating the fault list. "Setting Up the Fault Information for ATPG" on page 6-62 details this procedure. You can then run ATPG on the fault list. During the ATPG run, the tool also performs fault simulation to verify that the generated patterns detect the targeted faults.

   If you started ATPG by using FastScan, and your test coverage is still not high enough because of sequential circuitry, you can repeat the ATPG process using FlexTest. Because the FlexTest algorithms differ from those of FastScan, using both applications on a design may lead to a higher test coverage. In either case (full or partial scan), you can run ATPG under different constraints, or augment the test vector set with additional test patterns, to achieve higher test coverage. "Running ATPG" on page 6-69 covers this subject.

   After generating a test set with FastScan or FlexTest, you should apply timing information to the patterns and verify the design and patterns before handing them off to the vendor. "Verifying Design and Test Pattern Timing" on page 6-125 documents this operation.

FastScan and FlexTest Inputs and Outputs

Figure 6-3 shows the inputs and outputs of the FastScan and FlexTest applications.

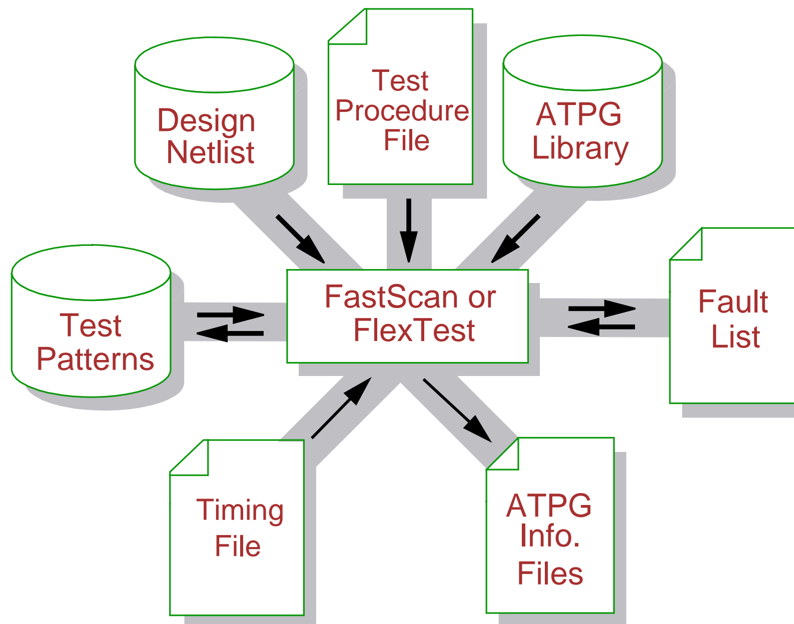


Figure 6-3. FastScan/FlexTest Inputs and Outputs

FastScan and FlexTest utilize the following inputs:

- **Design**
  The supported design data formats are EDDM, Electronic Design Interchange Format (EDIF 2.0.0), GENIE, Tegas Design Language (TDL), Verilog, VHDL, and SPICE. Other inputs also include 1) a cell model from the design library and 2) a previously saved flattened model (FastScan Only).

- **Test Procedure File**
  This file defines the operation of the scan circuitry in your design. You can generate this file by hand, or DFTAdvisor can create this file automatically when you issue the command Write Atpg Setup.

- **Library**
  The design library contains descriptions of all the cells used in the design.

FastScan/FlexTest use the library to translate the design data into a flat, gate-level simulation model for use by the fault simulator and test generator.

- **Fault List**
  FastScan and FlexTest can both read in an external fault list. They can use this list of faults and their current status as a starting point for test generation.

- **Timing File**
  If you want FastScan and FlexTest to write non-default timing into the test patterns, you must specify the timing information in this file.

- **Test Patterns**
  FastScan and FlexTest can both read in externally generated test patterns and use those patterns as the source of patterns to be simulated.

FastScan and FlexTest produce the following outputs:

- **Test Patterns**
  FastScan and FlexTest generate files containing test patterns. They can generate these patterns in a number of different simulator and ASIC vendor formats. "Test Pattern Formatting and Timing" on page 7-1 discusses the test pattern formats in more detail.

- **ATPG Information Files**
  These consist of a set of files containing information from the ATPG session. For example, you can specify creation of a log file for the session.

- **Fault List**
  This is an ASCII readable file containing internal fault information in the standard Mentor Graphics fault format.

Understanding FastScan's ATPG Method

To understand how FastScan operates, you should understand the basic ATPG process, timing model, and basic pattern types that FastScan produces. The following subsections discuss these topics.

FastScan's Basic ATPG Process

FastScan has default values set so that when you invoke ATPG for the first time (by issuing the Run command), it performs an efficient combination of random pattern fault simulation and deterministic test generation on the target fault list. "The ATPG Process" on page 2-14 discusses the basics of random and deterministic pattern generation.

Random Pattern Generation with FastScan

FastScan first performs random pattern fault simulation for each capture clock, stopping when a simulation pattern fails to detect at least 0.5% of the remaining faults. FastScan then performs random pattern fault simulation for patterns without a capture clock, as well as those that measure the primary outputs connected to clock lines.

Note

ATPG constraints and circuitry that can have bus contention are not optimal conditions for random pattern generation. If you specify ATPG constraints, FastScan will not perform random pattern generation.

Deterministic Test Generation with FastScan

Some faults have a very low chance of detection using a random pattern approach. Thus, after it completes the random pattern simulation, FastScan performs deterministic test generation on selected faults from the current fault list. This process consists of creating test patterns for a set of somewhat randomly chosen faults from the fault list.

During this process, FastScan identifies and removes redundant faults from the fault list. After it creates enough patterns for a fault simulation pass, it displays a message indicating the number of redundant faults, the number of ATPG

untestable faults, and the number of aborted faults that the test generator identifies. FastScan then once again invokes the fault simulator, removing all detected faults from the fault list and placing the effective patterns in the test set. FastScan then selects another set of patterns and iterates through this process until no faults remain in the current fault list, except those aborted during test generation (that is, those in the UC or UO categories).

FastScan Timing Model

FastScan uses a cycle-based timing model, grouping the test pattern events into test cycles. The FastScan simulator uses the non-scan events **force_pi**, **measure_po**, **capture_clock_on**, **capture_clock_off**, **ram_clock_on**, and **ram_clock_off**. FastScan uses a fixed test cycle type for ATPG; that is, you cannot modify it.

The most commonly used test cycle contains the events **force_pi**, **measure_po**, **capture_clock_on**, and **capture_clock_off**. The test vectors used to read or write into RAMs contain the events **force_pi**, **ram_clock_on**, and **ram_clock_off**. You can associate real times with each event via the timing file. Refer to "FastScan Non-Scan Event Timing" on page 7-13 for more details.

FastScan Pattern Types

FastScan has several different types of testing modes. That is, it can generate several different types of patterns depending on the style and circuitry of the design and the information you specify. By default, FastScan generates basic scan patterns, which assumes a full-scan design methodology. The following subsections describe basic scan patterns, as well as the other types of patterns that FastScan can generate.

Basic Scan Patterns

As mentioned, FastScan generates basic scan patterns by default. A scan pattern contains the events that force a single set of values to all scan cells and primary inputs (force_pi), followed by observation of the resulting responses at all primary outputs and scan cells (measure_po). FastScan uses any defined scan clock to capture the data into the observable scan cells (capture_clock_on, capture_clock_off). Scan patterns reference the appropriate test procedures to define how to control and observe the scan cells. FastScan requires that each scan

pattern be independent of all other scan patterns. The basic scan pattern contains the following events:

1. Load values into scan chains
2. Force values on all non-clock primary inputs (with clocks off and constrained pins at their constrained values).
3. Measure all primary outputs (except those connected to scan clocks).
4. Pulse a capture clock or apply selected clock procedure.
5. Unload values from scan chains.

While the list shows the loading and unloading of the scan chain as separate events, more typically, the pattern would perform load and unload simultaneously. Thus, when applying the patterns at the tester, you have a single operation that loads in a new pattern while unloading a previous pattern.

Because FastScan is an ATPG tool optimized for use with scan designs, the basic scan pattern contains the events from which it derives all other pattern types.

Clock PO Patterns

Figure 6-4 shows that in some designs, a clock signal may go to a primary output through some combinational logic.

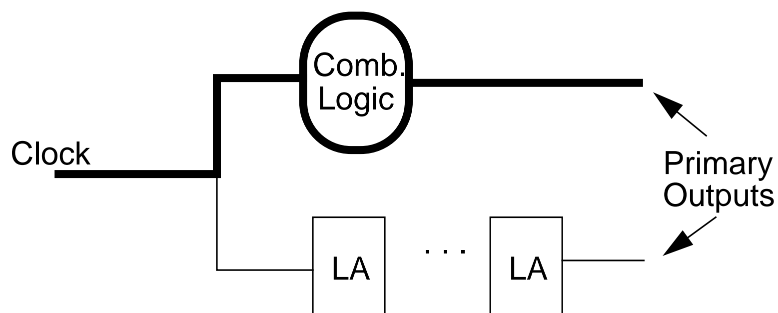


Figure 6-4. Clock-PO Circuitry

FastScan considers any pattern that measures a PO with connectivity to a clock, regardless of whether or not the clock is active, a clock PO pattern. A normal scan

pattern has all clocks off during the force of the primary inputs and the measure of the primary outputs. However, in the clocked primary output situation, if the clock is off, a condition necessary to test a fault within this circuitry might not be met and the fault may go undetected. In this case, in order to detect the fault, the pattern must turn the clock on during the force and measure. This does not happen in the basic scan pattern. FastScan allows this within a clock PO pattern, to observe primary outputs connected to clocks.

Clock PO patterns contain the following events:

1. Load values into the scan chains.
2. Force values on all primary inputs, (potentially) including clocks (with constrained pins at their constrained values).
3. Measure all primary outputs that are connected to scan clocks.

FastScan generates clock PO patterns whenever it learns that a clock connects to a primary output and if it determines that it can only detect faults associated with the circuitry by using a clock PO pattern. If you do not want FastScan to generate clock PO patterns, you can turn off the capability as follows:

SETUP> **Set Clockpo Patterns off**

Clock Sequential Patterns

The FastScan clock sequential pattern type handles limited sequential circuitry, and can also help in testing designs with RAM. This kind of pattern contains the following events:

1. Load values into the scan chains.
2. Force values on all primary inputs, except clocks (with constrained pins at their constrained values).
3. Pulse the write lines, read lines, capture clock, and/or apply selected clock procedure.
4. Repeat steps 2 and 3 up to “N” times, where N is the design circuitry’s sequential depth.

5. Measure all primary outputs (except those connected to clocks).

6. Optionally apply the selected clock procedure.

7. Unload values from scan cells.

To instruct FastScan to generate clock sequential patterns, you must set the sequential depth to some number greater than one, using the Set Simulation Mode command as follows:

SETUP> **Set Simulation Mode Combinational -depth 2**

A depth of zero indicates combinational circuitry. A depth greater than one indicates limited sequential circuitry. You should, however, be careful of the depth you specify. You should start off using the lowest sequential depth and analyzing the run results. You can perform several runs if necessary, increasing the sequential depth each time. Although the maximum allowable depth limit is 255, for performance reasons you should typically limit the value to specify to five or less.

RAM Sequential Patterns

To propagate fault effects through RAM, and to thoroughly test the circuitry associated with a RAM, FastScan generates a special type of pattern called RAM sequential. RAM sequential patterns are single patterns with multiple loads, which model some sequential events necessary to test RAM operations. The multiple load events include two address writes and possibly a read (if the RAM has data hold). This type of pattern contains the following events:

1. Load scan cells.

2. Force primary inputs.

3. Pulse write line(s).

4. Repeat steps 1 through 3 for a different address.

5. Load scan cells.

6. Force primary inputs.

7. Pulse read lines (optional, depending on the RAM's data hold attribute).
8. Load scan cells.
9. Force primary inputs
10. Measure primary outputs.
11. Pulse capture clock.
12. Unload values from scan cells.

The following example explains the operations depicted in this type of pattern. Assume you want to test a stuck-at-0 fault on the highest order bit of the address lines. You could do this by writing some data, D, to location 1000. You could then write different data, D', to location 0000. If a stuck-at-1 fault were present on the highest address bit, the faulty machine would overwrite location 1000 with the value D'. Next, you would attempt to read from address location 1000. With the stuck-at-1 fault on the address line, you would read D'.

Similarly, if the stuck-at-0 fault were present on the highest address bit, you write D' into 0000 would read D' from location 0000 (instead of 1000). In the good machine, you would expect to read the value D. In the faulty machine (whether stuck-at-0 or stuck-at-1 faults), you would read the value D'.

You can instruct FastScan to generate RAM sequential patterns by issuing the Set Simulation Mode command as follows:

SETUP> **Set Simulation Mode Ram_sequential**

Sequential Transparent Patterns

Designs containing some non-scan latches can use basic scan patterns if the latches behave transparently between the time of the primary input force and the primary output measure. A latch behaves transparently if it passes rule D6.

For latches that do not behave transparently, a user-defined procedure can force some of them to behave transparently between the primary input force and primary output measure. A test procedure, which is called **seq_transparent**,

defines the appropriate conditions necessary to force transparent behavior of some latches. The events in sequential transparent patterns include:

1. Load scan chains.
2. Force primary inputs.
3. Apply **seq_transparent** procedure(s).
4. Measure primary outputs.
5. Unload scan chains.

For more information on sequential transparent procedures, refer to "The Procedures" on page 3-15.

Understanding FlexTest's ATPG Method

Some sequential ATPG algorithms must go forward and backward in time to generate a test. These algorithms are not practical for large and deep sequential circuits, due to high memory requirements. FlexTest uses a general sequential ATPG algorithm, called the BACK algorithm, that avoids this problem. The BACK algorithm uses the behavior of a target fault to predict which primary output (PO) to use as the fault effect observe point. Working from the selected PO, it sensitizes the path backward to the fault site. After creating a test sequence for the target fault, FlexTest uses a parallel differential fault simulator for synchronous sequential circuits to calculate all the faults detected by the test sequence. To facilitate the ATPG process, FlexTest first performs redundancy identification when exiting the Setup mode.

This is typically how FlexTest performs ATPG. However, FlexTest can also generate functional vectors based on the instruction set of a design. The ATPG method it uses in this situation is significantly different from the sequential-based ATPG method it normally uses. For information on using FlexTest in this capacity, refer to "Creating Instruction-Based Test Sets (FlexTest)" on page 6-111.

Cycle-Based Timing Circuits

Circuits have cycle-based behavior if their output values are always stable at the end of each cycle period. Most designers of synchronous and asynchronous circuits use this concept. Figure 6-5 gives an example of a cycle-based circuit.

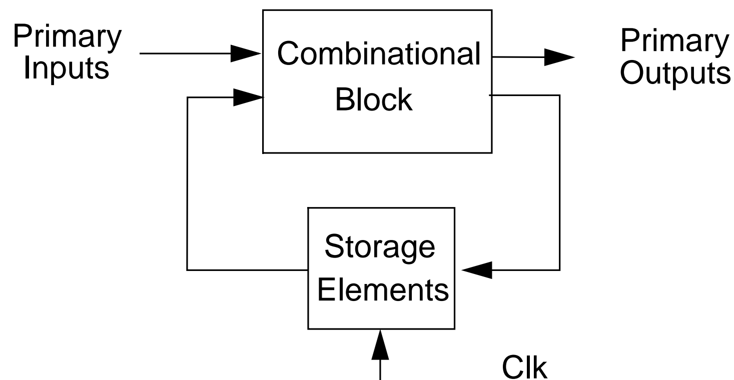


Figure 6-5. Cycle-Based Circuit with Single Phase Clock

In Figure 6-5, all the storage elements are edge-triggered flip-flops controlled by the rising edge of a single clock. The primary outputs and the final values of the storage elements are always stable at the end of each clock cycle, as long as the data and clock inputs of all flip-flops do not change their values at the same time. The clock period must be longer than the longest signal path in the combinational block. Also, stable values depend only on the primary input values and the initial values on the storage elements.

For the multiple-phase design, relative timing among all the clock inputs determines whether the circuit maintains its cycle-based behavior.

In Figure 6-6, the clocks PH1 and PH2 control two groups of level-sensitive latches which make up this circuit's storage elements.

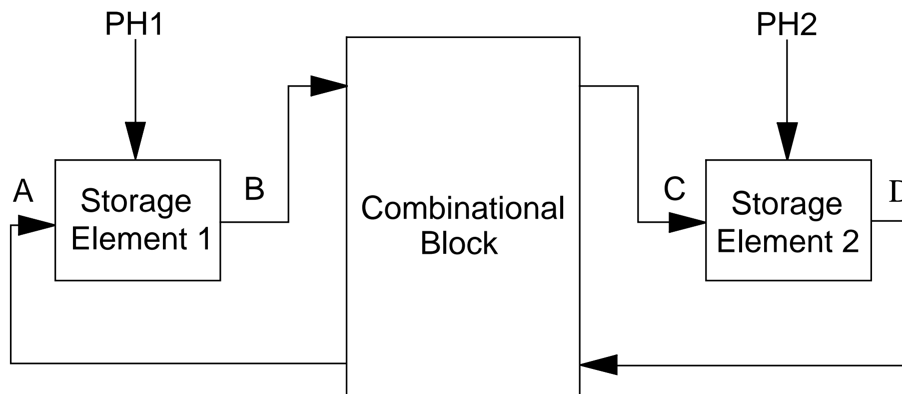


Figure 6-6. Cycle-Based Circuit with Two Phase Clock

When PH1 is on and PH2 is off, the signal propagates from point D to point C. On the other hand, the signal propagates from point B to point A when PH1 is off and PH2 is on. Designers commonly use this cycle-based methodology in two-phase circuits because it generates systematic and predictable circuit behavior. As long as PH1 and PH2 are not on at the same time, the circuit exhibits cycle-based behavior. If these two clocks are on at the same time, the circuit can operate in an unpredictable manner and can even become unstable.

Cycle-Based Timing Model

All automatic test equipment (ATE) are cycle-based, unlike event-based digital simulators. A *test cycle* for ATE is the waveform (stored pattern) applied to all primary inputs and observed at all primary outputs of the device under test (DUT). Each test cycle has a corresponding timing definition for each pin.

In FlexTest, as opposed to FastScan, you must specify the timing information for the test cycles. FlexTest provides a sophisticated timing model that you can use to properly manage timing relationships among primary inputs--especially for critical signals, such as clock inputs.

FlexTest uses a test cycle, which is conceptually the same as an ATE test cycle, to represent the *period* of each primary input. If the input cycle of a primary input is longer (for example, a signal with a slower frequency) than the length you set for the test cycle, then you must represent its period as a multiple of test cycles.

A test cycle further divides into timeframes. A *timeframe* is the smallest time unit that FlexTest can simulate. The tool simulates whatever events occur in the timeframe until signal values stabilize. For example, if data inputs change during a timeframe, the tool simulates them until the values stabilize. The number of timeframes equals the number of simulation processes FlexTest performs during a test cycle. At least one input must change during a defined timeframe. You use timeframes to define the test cycle terms *offset* and the *pulse width*. The offset is the number of timeframes that occur in the test cycle before the primary input goes active. The pulse width is the number of timeframes the primary input stays active.

Figure 6-7 shows a primary input with a positive pulse in a six timeframe test cycle. In this example, the period of the primary input is one test cycle. The length of the test cycle is six timeframes, the offset is two timeframes, and the width of its pulse is three timeframes.

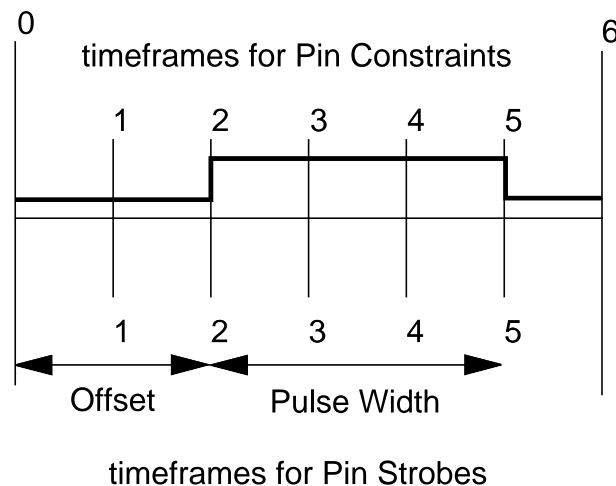
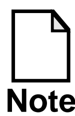


Figure 6-7. Example Test Cycle

In this example, if other primary inputs have periods longer than the test cycle, you must define them in multiples of six timeframes (the defined test cycle period). Time 0 is the same as time 6, except time 0 is treated as the beginning of the test cycle, while time 6 is treated as the end of the test cycle.



To increase the performance of FlexTest fault simulation and ATPG, you should try to define the test cycle to use as few timeframes as possible.

For most automatic test equipment, the tester strobes each primary output only once in each test cycle and can strobe different primary outputs at different timeframes. In the non-scan environment, FlexTest strobes primary outputs at the end of each test cycle by default.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

In the scan environment, if any scan memory element capture clock is on, the scan-in values in the scan memory elements change. Therefore, in the scan test, right after the scan load/unload operation, no clocks can be on. Also, the primary output strobe should occur before any clocks turn on. Thus, in the scan environment, FlexTest strobes primary outputs after the first timeframe of each test cycle by default.

If you strobe a primary output while the primary inputs are changing, FlexTest first strobes the primary output and then changes the values at the primary inputs. To be consistent with the boundary of the test cycle (using Figure 6-7 as an example), you must describe the primary input's value change at time 6 as the change in value at time 0 of the next test cycle. Similarly, the strobe time at time 0 is the same as the strobe time at time 6 of the previous test cycle.

Cycle-Based Test Patterns

Each primary input has its own signal frequency and cycle. Test patterns are cycle-based if each individual input either holds its value or changes its value at a specific time in each of its own input cycle periods. Also, the width of the period of every primary input has to be equal to or a multiple of test cycles used by the automatic test equipment.

Cycle-based test patterns are easy to use and tend to be portable among the various automatic test equipment. For most ATE, the tester allows each primary input to change its value up to two times within its own input cycle period. A constant value means that the value of the primary input does not change. If the value of the primary input changes only once (generally for data inputs) in its own cycle, then the tester holds the new value for one cycle period. A pulse input

means that the value of the primary input changes twice in its own cycle. For example, clock inputs behave in this manner.

Performing Basic Operations

This section describes the most basic operations you may need to perform with FastScan and FlexTest.

Also refer to "User Interface Overview" on page 1-9 for more general information.

Invoking the Applications

You can invoke FastScan and FlexTest in two ways. Using the first option, you enter just the application name on the shell command line which opens the application in graphical mode.

For FastScan:

```
$MGC_HOME/bin/fastscan [-Falcon]
```

For FlexTest:

```
$MGC_HOME/bin/flextest [-Falcon]
```

Once the tool is invoked, a dialog box prompts you for the required arguments (design name, design format, and library). Browser buttons are provided for navigating to the appropriate files. Once the design and library are loaded, the tool is in Setup mode and ready for you to begin working on your design.

Using the second option requires you to enter all required arguments at the shell command line.

For FastScan:

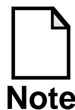
$MGC_HOME/bin/fastscan {{{***design_name*** {{**-EDDM** [-I | {-S *root_name*}]} | **-EDIF** | **-TDL** | **-VERILOG** | **-VHDL** | **-GENIE** | **-SPICE** | -**FLAT**}} **|** {**-MODEL** ***cell_name***}} [-LIBrary *library_name*] [-SENsitive] [-LOG *filename*] [-REPlace] [-NOGui] [-FAlcon][-TOP *model_name*] [-DOFile *dofile_name*] [-LICense *retry_limit*] [-SETup *setup_name*] [-DIAG]} | {[-HELP] | [-USAGE] | [-VERSION]}

For FlexTest:

$MGC_HOME/bin/flextest {{{***design_name*** {{**-EDDM** [-I | {-S *root_name*}]} | **-EDIF** | **-TDL** | **-VERILOG** | **-VHDL** | **-GENIE** | -**SPICE**}} | {**-MODEL** ***cell_name***}} [**-Library *filename***] [-SENsitive] [-LOG *filename*] [-REPlace] [-NOGui] [-FAlcon] [-FaultSIM] [-TOP *model_name*] [-DOFile *dofile_name*] [-LICense *retry_limit*] [-Hostfile *host_filename*]} | {[-HELP] | [-USAGE] | [-VERSION]}

When the tool is finished invoking, the design and library are also loaded. The tool is now in Setup mode and ready for you to begin working on your design. By default, the tool invokes in graphical mode so if you want to use the command-line interface, you must specify the -Nogui switch using the second invocation option.

The application argument is either "fastscan" or "flextest". The design_name is an netlist in one of the appropriate formats. If you invoke using the -Falcon option, (with or without using the GUI), the EDDM format is the default netlist format. For the point tool version, EDIF is the default format. The library contains descriptions of all the library cells used in the design.


Note

The invocation syntax for both FastScan and FlexTest includes a number of other switches and options. For a list of available options and explanations of each, you can refer to "Shell Commands" in the *FastScan and FlexTest Reference Manual* or enter:

```
$  $MGC_HOME/bin/<application> -help
```

Invoking the Point Tool and Falcon Versions

FastScan and FlexTest are both available as point tools; that is, they are available without the overhead of the Falcon Framework. As a result of this decoupling from the framework, the point tool versions of the tools do not have access to the EDDM format netlist read and write capabilities, or the MGC WDB output pattern format capabilities.

Despite the different package names, you still invoke the application in the same manner as shown previously. The only difference occurs with the invocation switches. If you can access both the Falcon and point tool version of the tools, you must use the Falcon switch to invoke the Falcon version of the tool.

Invoking the FastScan Diagnostics-Only Version

FastScan is also available in a diagnostics-only package. This version of the tool has only three system modes: Setup, Good, and Fault. An error condition occurs if you attempt to enter the Atpg system mode.

You invoke this version of FastScan using the -Diag switch. Using the -Diag switch checks for the diagnostics-only license, and if found, invokes the FastScan diagnostics-only capabilities.

Invoking Distributed FlexTest

FlexTest has the ability to divide ATPG processes into smaller sets and run these sets simultaneously on multiple workstations. This capability is called *Distributed FlexTest*. For more information on this capability, refer to "Distributed FlexTest" in the *FastScan and FlexTest Reference Manual*.

Invoking the FlexTest Fault Simulation Version

Similarly, FlexTest is available in a fault simulation only package called FlexTest FaultSim. This version of the tool has only the Setup, Drc, Good, and Fault system modes. An error condition occurs if you attempt to enter the Atpg system mode.

You invoke this version of FlexTest using the -Fsim switch. Using the -Fsim switch checks for the fault simulation license, and if found, invokes the fault simulation package.

FlexTest Interrupt Capabilities

Instead of aborting the current process, FlexTest optionally allows you to interrupt a process. An interrupted process remains in a suspended state. While in a suspended state, you may execute any of the following commands:

- Help
- all Report commands
- all Write commands
- Set Abort Limit
- Set Atpg Limits
- Set Checkpoint
- Set Fault Mode
- Set Gate Level
- Set Gate Report
- Set Logfile Handling
- Save Patterns

You may find these commands useful in determining whether or not to resume the process. By default, interrupt handling is off, thus aborting interrupted processes. If instead of aborting, you want an interrupted process to remain in a suspended state, you can issue the Set Interrupt Handling command as follows:

SETUP> **set interrupt handling on**

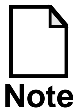
After you turn interrupt handling on and interrupt a process, you can either abort the suspended process using the Abort Interrupted Process command or continue the process using the Resume Interrupted Process command.

For more information on interrupt capabilities see "Interrupting the Session" on page 1-20.

Setting the System Mode

When FastScan and FlexTest invoke, they assume the first thing you want to do is set up circuit behavior, so they automatically put you in Setup mode. The entire set of system modes includes:

- **SETUP** - use to set up circuit behavior.
- **DRC** - use (FlexTest only) to retain the flattened design model for design rules checking.
- **ATPG** - use to run test pattern generation.
- **FAULT** - use to run fault simulation.
- **GOOD** - use to run good simulation.


Note

Drc mode applies to FlexTest only. While FastScan uses the same model for design rules checking and other processes, FlexTest creates a slightly different version of the design after successfully passing rules checking. Thus, Drc mode allows FlexTest to retain this intermediate design model.

To change the system mode, you use the Set System Mode command, whose usage is as follows:

SET SYstem Mode {**Setup** | {{**Atpg** | **Fault** | **Good | Drc**} [-Force]}

If you are using the graphical user interface, you can click on the palette menu items "SETUP", "ATPG", "FAULT", or "GOOD". Notice how the palette changes for each system mode selection you make.

Setting Up Design and Tool Behavior

The first real task you must perform in the basic ATPG flow is to set up information about design behavior and existing scan circuitry. The following subsections describe how to accomplish this setup.

Setting Up the Circuit Behavior

FastScan and FlexTest provide a number of commands that let you set up circuit behavior. You must execute these commands while in Setup mode. A convenient way to execute the circuit setup commands is to place these commands in a dofile, as explained previously in "Running Batch Mode Using Dofiles" on page 1-18. The following subsections describe typical circuit behavior set up tasks.

Defining Equivalent or Inverted Primary Inputs

Within the circuit application environment, often multiple primary inputs of the circuit being tested must always have the same (equivalent) or opposite values. Specifying pin equivalences constrains selected primary input pins to equivalent or inverted values relative to the last entered primary input pin. To add pin equivalences, you use the Add Pin Equivalences command. This command's usage is as follows:

ADD PIn Equivalences ***primary_input_pin...*** [-Invert *primary_input_pin*]

Or, if you are using the graphical user interface, you can select the **Add > Pin Equivalences...** pulldown menu item and specify the pin information in the dialog box that appears.

Related Commands:

Delete Pin Equivalences - deletes the specified pin equivalences.
Report Pin Equivalences - displays the specified pin equivalences.

Adding Primary Inputs and Outputs

In some cases, you may need to change the test pattern application points (primary inputs) or the output value measurement points (primary outputs). When you add

previously undefined primary inputs, they are called user class primary inputs, while the original primary inputs are called system class primary inputs.

To add primary inputs to a circuit, at the Setup mode prompt, you use the Add Primary Inputs command. This command's usage is as follows:

ADD PRimary Inputs ***net_pathname...*** [-Cut] [-Module]

Or, if you are using the graphical user interface, you can select the ADD PRIM INPUTS palette menu item or the **Add > Primary Inputs...** pulldown menu item and specify the information in the dialog box that appears.

When you add previously undefined primary outputs, they are called user class primary outputs, while the original primary outputs are called system class primary outputs.

To add primary outputs to a circuit, at the Setup mode prompt, you use the Add Primary Outputs command. This command's usage is as follows:

ADD PRimary Outputs ***net_pathname...***

Or, if you are using the graphical user interface, you can select the ADD PRIM OUTPUTS palette menu item or the **Add > Primary Outputs...** pulldown menu item.

Related Commands:

Delete Primary Inputs - deletes the specified types of primary inputs.
Report Primary Inputs - reports the specified types of primary inputs.
Write Primary Inputs - writes the current list of primary inputs to a file.
Delete Primary Outputs - deletes the specified types of primary outputs.
Report Primary Outputs - reports the specified types of primary outputs.
Write Primary Outputs - writes the current list of primary outputs to a file.

Tying Undriven Signals

Within your design, there could be several *undriven* nets, which are input signals not tied to fixed values. When you invoke FastScan or FlexTest, the application issues a warning message for each undriven net or floating pin in the module. The

ATPG tool must "virtually" tie these pins to a fixed logic value during ATPG. If you do not specify a value, the application uses the default value X, which you can change with the Setup Tied Signals command.

To add tied signals, at the Setup mode prompt, you use the Add Tied Signals command. This command's usage is as follows:

ADD TIed Signals {**0** | **1** | **X** | **Z**} ***floating_object_name...*** [-Pin]

Or, if you are using the graphical user interface, you can select the ADD TIED SIGNAL palette menu item or the **Add > Tied Signals...** pulldown menu item.

This command assigns a fixed value to every named floating net or pin in every module of the circuit under test.

Related Commands:

Setup Tied Signals - sets default for tying unspecified undriven signals.
Delete Tied Signals - deletes the current list of specified tied signals.
Report Tied Signals - displays current list of specified tied nets and pins.

Constraining Primary Inputs

FastScan and FlexTest can constrain primary inputs during the ATPG process. To add pin constraints to a specific pin, you use the Add Pin Constraints command. This command's usage is as follows:

ADD PIn Constraints ***primary_input_pin... constraint_format***

Or, if you are using the graphical user interface, you can select the ADD PIN CONSTRAINT palette menu item or the **Add > Pin Constraints...** pulldown menu item.

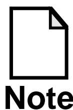
You can specify one or more primary input pin pathnames to be constrained to one of the following formats: constant 0 (C0), constant 1 (C1), high impedance (CZ), or unknown (CX). For FlexTest, the Add Pin Constraints command supports a number of additional constraint formats for specifying the cycle-based timing of primary input pins. Refer to "Defining the Cycle Behavior of Primary Inputs" on page 6-35 for the FlexTest-specific timing usage of this command.

For detailed information on the tool-specific usages of this command, refer to Add Pin Constraints in the *FastScan and FlexTest Reference Manual*.

Masking Primary Outputs

Your design may contain certain primary output pins that have no strobe capability. Or in a similar situation, you may want to mask certain outputs from observation for design trade-off experimentation. In these cases, you could mask these primary outputs using the Add Output Masks command. This command's usage is as follows:

ADD OUtput Masks ***primary_output...***



Note

FastScan and FlexTest place faults they can only detect through masked outputs in the AU category--not the UO category.

Adding Slow Pads (FastScan Only)

While running tests at high speed, as might be used for path delay test patterns, it is not always safe to assume that the loopback path from internal registers, via the I/O pad back to internal registers, can stabilize within a single clock cycle. Assuming that the loopback path stabilizes within a single clock cycle may cause problems verifying ATPG patterns or may lead to yield loss during testing.

To prevent a problem caused by this loopback, use the Add Slow Pad command to modify the simulated behavior of the bidirectional I/O pin, on a pin by pin basis. This command's usage is as follows:

ADD SLow Pad {***pin_name*** [-Cell *cell_name*]} | -**All**

For a slow pad, the simulation of the I/O pad is changed such that the value propagated into the internal logic is X whenever the primary input is not driven. This causes an X to be captured for all observation points dependent on the loopback value.

Related Commands:

Delete Slow Pad - resets the specified I/O pin back to the default simulation mode.
Report Slow Pads - displays all I/O pins marked as slow.

Setting Up Tool Behavior

In addition to specifying information about the design to the ATPG tool, you can also set up how you want the ATPG tool to handle certain situations and how much effort to put into various processes. The following subsections discuss the typical tool setup.

Related Commands:

Set Learn Report - enables access to certain data learned during analysis.
Set Loop Handling - specifies the method in which to break loops.
Set Possible Credit - sets credit for possibly-detected faults.
Set Pulse Generators - specifies whether to identify pulse generator sink gates during learning analysis.
Set Race Data - specifies how to handle flip-flop race conditions.
Set Rail Strength - sets the strongest strength of a fault site to a bus driver.
Set Redundancy Identification - specifies whether to perform redundancy identification during learning analysis.

Checking Bus Contention

If you use contention checking on tri-state™ driver busses and multiple-port flip-flops and latches, FastScan and FlexTest will reject (from the internal test pattern set) patterns generated by the ATPG process that can cause bus contention. To set contention checking, you use the Set Contention Check command. This command's usage is as follows:

For FastScan:

SET COntention Check **OFf** | {{**ON** | **Capture_clock**} [-Warning | -Error] [-Bus | -Port | -ALl] [-BIdi_retain | -BIDI_Mask] [-ATpg] [-NOVerbose | -Verbose | -VVerbose]}

For FlexTest:

SET COntention Check **OFf** | {**ON** [-Warning | -Error] [-Bus | -Port | -ALl] [-ATpg] [-Start *frame#*]}

By default, contention checking is on, as are the switches -Warning and -Bus, causing the tool to check tri-state driver buses and issue a warning if bus contention occurs during simulation. FastScan and FlexTest vary somewhat in their contention checking options. For more information on the different contention checking options, refer to the Set Contention Check command page in the *FastScan and FlexTest Reference Manual*.

To display the current status of contention checking, use the Report Environment command.

Related Commands:

Analyze Bus - analyzes the selected buses for mutual exclusion.
Set Bus Handling - specifies how to handle contention on buses**.**
Set Driver Restriction - specifies whether only a single driver or multiple drivers can be on for buses or ports.
Report Bus Data - reports data for either a single bus or a category of buses.
Report Gates - reports netlist information for the specified gates.

Setting Multi-Driven Net Behavior

When you specify the fault effect of bus contention on tri-state nets with the Set Net Dominance command, you are giving the tool the ability to detect some faults on the enable lines of tri-state drivers that connect to a tri-state bus. At the Setup mode prompt, you use the Set Net Dominance command. This command's usage is as follows:

SET NEt Dominance **Wire** | **And** | **Or**

The three choices for bus contention fault effect are And, Or, and Wire (unknown behavior), Wire being the default. The Wire option means that any different binary value results in an X state. The truth tables for each type of bus contention fault effect are shown on the references pages for the Set Net Dominance command in the *FastScan and FlexTest Reference Manual*.

On the other hand, if you have a net with multiple non-tri-state drivers, you may want to specify this type of net's output value when its drivers have different values. Using the Set Net Resolution command, you can set the net's behavior to And, Or, or Wire (unknown behavior). The default Wire option requires all inputs to be at the same state to create a known output value. Some loss of test coverage can result unless the behavior is set to And (wired-and) or Or (wired-or). To set the multi-driver net behavior, at the Setup mode prompt, you use the Set Net Resolution command. This command's usage is as follows:

SET NEt Resolution **Wire** | **And** | **Or**

Setting Z-State Handling

If your tester has the ability to distinguish the high impedance (Z) state, you should use the Z state for fault detection to improve your test coverage. If the tester can distinguish a high impedance value from a binary value, certain faults may become detectable which otherwise would at best be possibly detected (pos_det). This capability is particularly important for fault detection in the enable line circuitry of tri-state drivers.

The default for FastScan and FlexTest is to treat a Z state as an X state. If you want to account for Z state values during simulation, you can issue the Set Z Handling command.

Internal Z handling specifies how to treat the high impedance state when the tri-state network feeds internal logic gates. External handling specifies how to treat the high impedance state at the circuit primary outputs. The ability of the tester normally determines this behavior.

To set the internal or external Z handling, use the Set Z Handling command at the Setup mode prompt. This command's usage is as follows:

SET Z Handling {**Internal** ***state***} | {**External** ***state***}

For internal tri-state driver nets, you can specify the treatment of high impedance as a 0 state, a 1 state, an unknown state, or (for FlexTest only) a hold of its previous state.

Note

This command is not necessary if the circuit model already reflects the existence of a pull gate on the tri-state net.

For example, to specify that the tester does not measure high impedance, enter the following:

SETUP> **set z handling external X**

For external tri-state nets, you can also specify that the tool measure high impedance as a 0 state and distinguished from a 1 state (0), measure high impedance as a 1 state and distinguished from a 0 state (1), measure high impedance as unique and distinguishable from both a 1 and 0 state (Z), or (for FlexTest only) measure high impedance from its previous state (Hold).

Controlling the Learning Process

FastScan and FlexTest perform extensive learning on the circuit during the transition from Setup to some other system mode. This learning reduces the amount of effort necessary during ATPG. FastScan and FlexTest allow you to control this learning process.

For example, FastScan and FlexTest lets you turn the learning process off or change the amount of effort put into the analysis. You can accomplish this for combinational logic using the Set Static Learning command, whose usage is as follows:

SET STatic Learning {**ON** [-Limit *integer*]} | **OFf**

By default, static learning is on and the simulation activity limit is 1000. This number ensures a good trade-off between analysis effort and process time. If you want FastScan to perform maximum circuit learning, you should set the activity limit to the number of gates in the design.

You can also use the Set Sequential Learning command to turn the learning process off for sequential elements. This command's usage is as follows:

SET SEquential Learning OFf | ON

FlexTest also performs state transition graph extraction as part of its learning analysis activities in an attempt to reduce the state justification effort during ATPG. FlexTest gives you the ability to turn on or off the state transition process. You accomplish this using the Set Stg Extraction command, whose usage is as follows:

SET STg Extraction **ON** | **OFf**

By default, state transition graph extraction is on. For more information on the learning process, refer to "Learning Analysis" on page 3-36.

Setting the Capture Handling (FastScan Only)

FastScan evaluates gates only once during simulation, simulating all combinational gates before sequential gates. This default simulation behavior correlates well with the normal behavior of a synchronous design, if the design model passes design rules checks--particularly rules C3 and C4. However, if your design fails these checks, you should examine the situation to see if your design would benefit from a different type of data capture simulation.

For example, examine the design of Figure 6-8. It shows a design fragment which fails the C3 rules check.

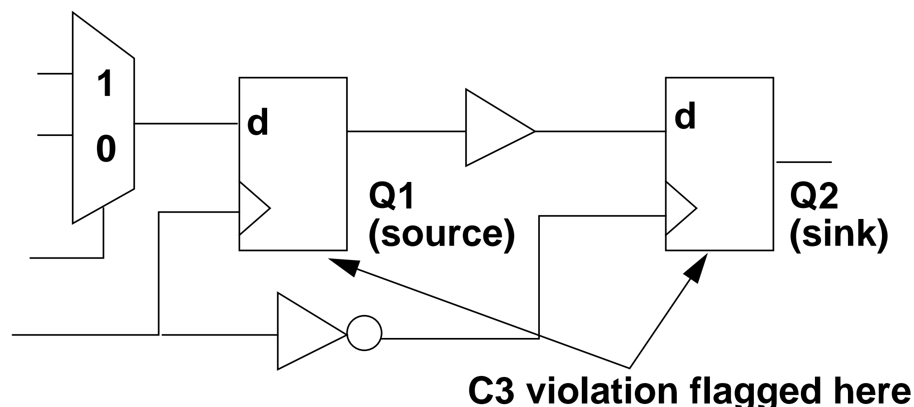


Figure 6-8. Data Capture Handling Example

The rules checker flags the C3 rule because Q2 captures data on the trailing edge of the same clock that Q1 uses. FastScan considers sequential gate Q1 as the data *source* and Q2 as the data *sink*. By default, FastScan simulates Q2 capturing old data from Q1. However, this behavior most likely does not correspond to the way the circuit really operates. In this case, the C3 violation should alert you that simulation could differ from real circuit operation.

To allow greater flexibility of capture handling for these types of situations, FastScan provides some commands that alter the default simulation behavior. The Set Capture Handling command changes the default data capture handling for gates failing the C3 or C4 design rules. The usage for this command is as follows:

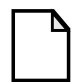
SET CApture Handling {**-Ls** {**Old** | **New** | **X**} | **-Te** {**Old** | **New** | **X**}} [-Atpg | -NOAtpg]

You can select modified capture handling for level sensitive or trailing edge gates. For these types of gates, you select whether you want simulation to use old data, new data, or X values. If you specify the -Atpg option, FastScan not only uses the specified capture handling for rules checking but for the ATPG process as well.

The Set Capture Handling command changes the data capture handling globally for all the specified types of gates that fail C3 and C4. If you want to selectively change capture handling, you can use the Add Capture Handling command. The usage for this command is as follows:

ADD CApture Handling {**Old** | **New** | **X**} ***object...*** [-SInk | -SOurce]

You can specify the type of data to capture, whether the specified gate(s) is a source or sink point, and the gates or objects (identified by ID number, pin names, instance names, or cell model names) for which to apply the special capture handling.


Note

When you change capture handling to simulate new data, FastScan just performs new data simulation for one additional level of circuitry. That is, sink gates capture new values from their sources. However, if the sources are also sinks that are set to capture new data, FastScan does not simulate this effect.

Fore more information on Set Capture Handling or Add Capture Handling, refer to the *FastScan and FlexTest Reference Manual*. For more information on C3 and C4 rules violations, refer to "Clock Rules" in the *Design-for-Test Common Resources Manual*.

Related Commands:

Delete Capture Handling - removes special data capture handling for the specified objects.
Set Drc Handling - specifies violation handling for a design rules check.
Set Sensitization Checking - specifies if DRC must determine path sensitization during the C3 rules check.

Checking the Environment Setup

You can check the environment you have set up by using the Report Environment command as follows:

REPort ENvironment

If you are using the graphical user interface, select the **Report > Environment** pulldown menu item.

This command reports on the tool's current user-controllable settings. If you issue this command before specifying any setup commands, the application lists the system defaults for all the setup commands. To write this information to a file, use the Write Environment command

Setting the Circuit Timing (FlexTest Only)

As "Understanding FlexTest's ATPG Method" on page 6-14 explains, to create reliable test patterns with FlexTest, you need to provide proper timing information for certain primary inputs. The following subsections describe how to set circuit timing. If you need to better understand FlexTest timing, you should refer to "Test Pattern Formatting and Timing" on page 7-1.

Setting the Test Cycle Width

When you set the test cycle width, you specify the number of timeframes needed per test cycle. The larger the number you enter for timeframes, the better the resolution you have when adding pin constraints. The smaller the number of timeframes you specify per cycle, the better the performance FlexTest has during ATPG.

By default, FlexTest assumes a test cycle of one timeframe. However, typically you will need to set the test cycle to two timeframes. And if you define a clock using the Add Clocks command, you must specify at least two timeframes. In a typical test cycle, the first timeframe is when the data inputs change (forced and measured) and the second timeframe is when the clock changes. If you have multi-phased clocks, or want certain data pins to change when the clock is active, you should set three or more timeframes per test cycle.

At least one input or set of inputs should change in a given timeframe. If not, the timeframe is unnecessary. Unnecessary timeframes adversely affect FlexTest performance. When you attempt to exit Setup mode, FlexTest checks for unnecessary timeframes, just prior to design flattening. If the check fails, FlexTest issues an error message and remains in Setup mode.

To set the number of timeframes in a test cycle, you use the Set Test Cycle command. This command's usage is as follows:

SET TEst Cycle ***integer***

Or, if you are using the graphical user interface, you can select the SET TEST CYCLE palette menu item or the **Setup > Test Cycle...** pulldown menu item.

Defining the Cycle Behavior of Primary Inputs

As discussed previously, testers are naturally cyclic and the test patterns FlexTest generates are also cyclic. Events occur repeatedly, or in cycles. Cycles further divide into timeframes. Clocks exhibit cyclic behavior and you must define this behavior in terms of the test cycle. Thus, after setting the test cycle width, you need to define the cyclic behavior of the circuit's primary inputs.

There are three components to describing the cyclic behavior of signals. A pulse signal contains a period (that is equal to or a multiple of test cycles), an offset time, and a pulse width. Constraining a pin lets you define when its signal can change in relation to the defined test cycle. To add pin constraints to a specific pin, you use the Add Pin Constraints command. This command's usage is as follows:

ADD PIn Constraints ***primary_input_pin... constraint_format***

You define a signal with a constant value using the constant constraint formats only. The definition for a signal with a hold value includes a period and an offset time. There are eleven constraint formats from which to chose. The constraint values (or waveform types) further divide into the three waveform groups used in all automatic test equipment:

- **Group 1: Non-return waveform (Signal value changes only once)**
  These include hold (NR <period> <offset>), constant zero (C0), constant one (C1), constant unknown (CX), and constant Z (CZ).

- **Group 2: Return-zero waveform (Signal may go to a 1 and then return to 0)**
  These include one positive pulse per period (R0 <period> <offset><width>), one suppressible positive pulse (SR0 <period><offset> <width>), and no positive pulse during non-scan (CR0 <period> <offset> <width>).

- **Group 3: Return-one waveform (Signal may go to a 0 and then return to 1**
  These include one negative pulse per cycle (R1 <period> <offset><width>), one suppressible negative pulse (SR1 <period><offset> <width>), and no negative pulse during non-scan (CR1 <period> <offset> <width>).

Pins not specifically constrained with Add Pin Constraints adopt the default constraint format of NR 1 0. You can change the default constraint format using the Setup Pin Constraints command, whose usage is as follows:

SETUp PIn Constraints ***constraint_format***

Related Commands:

Delete Pin Constraints - deletes the specified pin constraints.
Report Pin Constraints - displays cycle behavior of the specified inputs.

Defining the Strobe Time of Primary Outputs

After setting the cyclic behavior of all primary inputs, you need to define the strobe time of primary outputs. As "Understanding FlexTest's ATPG Method" on page 6-14 explains, each primary output has a strobe time--the time at which the tool measures its value--in each test cycle. Typically, all outputs are strobed at once, however different primary outputs can have different strobe times.

To specify a unique strobe time for certain primary outputs, you use the Add Pin Strobes command. You can also optionally specify the period for each pin strobe. This command's usage is as follows:

ADD PIn Strobes ***strobe_time primary_output_pin...*** [-Period *integer*]

Or, if you are using the graphical user interface, you can select the **Add > Pin Strobes...** pulldown menu item.

Any primary output without a specified strobe time uses the default strobe time. To set the default strobe time for all unspecified primary output pins, you use the Setup Pin Strobes command. This command's usage is as follows:

SETup PIn Strobes ***integer*** | **-Default**

The -Default switch resets the strobe time to the FlexTest defaults, such that the strobe takes place in the last timeframe of each test cycle, unless there is a scan operation during the test period. If there is a scan operation, FlexTest sets time 1 as the strobe time for each test cycle.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

Related Commands:

Delete Pin Strobes - deletes the specified pin strobes.
Report Pin Strobes - displays the strobe time of the specified outputs.

Defining the Scan Data

You must define the scan clocks and scan chains before the application performs rules checking (which occurs upon exiting the Setup mode). The following subsections describe how to define the various types of scan data.

Defining Scan Clocks

FastScan and FlexTest consider any signals that capture data into sequential elements (such as system clocks, sets, and resets) to be scan clocks. Therefore, to take advantage of the scan circuitry, you need to define these "clock signals" by adding them to the clock list.

You must specify the *off-state* for pins you add to the clock list. The off-state is the state in which clock inputs of latches are inactive. For edge-triggered devices, the off-state is the clock value prior to the clock's capturing transition. You add clock pins to the list by using the Add Clocks command. This command's usage is as follows:

ADD CLocks ***off_state primary_input_pin...***

Or, if you are using the graphical user interface, you can select the ADD CLOCK palette menu item or the **Add > Clocks...** pulldown menu item.

You can constrain a clock pin to its off-state to suppress its usage as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state, otherwise an error occurs. If you add an equivalence pin to the clock list, all of its defined equivalent pins are also automatically added to the clock list.

Related Commands:

Delete Clocks - deletes the specified pins from the clock list.
Report Clocks - reports all defined clock pins.

Defining Scan Groups

A scan group contains a set of scan chains controlled by a single test procedure file. You must create this test procedure file prior to defining the scan chain group that references it. To define scan groups, you use the Add Scan Group command, whose usage is as follows:

ADD SCan Groups ***group_name test_procedure_filename***

Or, if you are using the graphical user interface, you can select the ADD SCAN GROUP palette menu item or the **Add > Scan Groups...** pulldown menu item.

Related Commands:

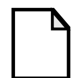
> **Delete Scan Groups** - deletes specified scan groups and associated chains.
> **Report Scan Groups** - displays current list of scan chain groups.

Defining Scan Chains

After defining scan groups, you can define the scan chains associated with the groups. For each scan chain, you must specify the name assigned to the chain, the name of the chain's group, the scan chain input pin, and the scan chain output pin. To define scan chains and their associated scan groups, you use the Add Scan Chains command, whose usage is as follows:

ADD SCan Chains ***chain_name group_name primary_input_pin primary_output_pin***

Or, if you are using the graphical user interface, you can select the ADD SCAN CHAIN palette menu item or the **Add > Scan Chains...** pulldown menu item.



Note Scan chains of a scan group can share a common scan input pin, but this condition requires that both scan chains contain the same data after loading.

Related Commands:

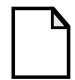
> **Delete Scan Chains** - deletes the specified scan chains.
> **Report Scan Chains** - displays current list of scan chains.

Setting the Clock Restriction

You can specify whether or not to allow the test generator to create patterns that have more than one non-equivalent capture clock active at the same time. To set the clock restriction, you use the Set Clock Restriction command. This command's usage is as follows:

SET CLock Restriction **ON** | **OFf** | **Clock_po**

The ON option only allows creation of patterns with a single active clock. The OFf option, which is the FlexTest default, allows creation of patterns with multiple active clocks. The Clock_po option (FastScan only), which is the FastScan default, allows only clock_po patterns to have multiple active clocks.


Note

If you choose to turn off the clock restriction, to avoid potential timing errors, you should verify the generated pattern set using a timing simulator.

Adding Constraints to Scan Cells

FastScan and FlexTest can constrain scan cells to a constant value (C0 or C1) during the ATPG process to enhance controllability or observability. Additionally, the tools can constrain scan cells to be either uncontrollable (CX), unobservable (OX), or both (XX).

You identify a scan cell by either a pin pathname or a scan chain name plus the cell's position in the scan chain.

To add constraints to scan cells, you use the Add Cell Constraints command. This command's usage is as follows:

ADD CEll Constraints {***pin_pathname*** | {***chain_name cell_position***}} **C0** | **C1** | **CX** | **Ox** | **Xx**

Or, if you are using the graphical user interface, you can select the **Add > Cell Constraints...** pulldown menu item.

If you specify the pin pathname, it must be the name of an output pin directly connected (through only buffers and inverters) to a scan memory element. In this case, the tool sets the scan memory element to a value such that the pin is at the

constrained value. An error condition occurs if the pin pathname does not resolve to a scan memory element.

If you identify the scan cell by chain and position, the scan chain must be a currently-defined scan chain and the position is a valid scan cell position number. The scan cell closest to the scan-out pin is in position 0. The tool constrains the scan cell's MASTER memory element to the selected value. If there are inverters between the MASTER element and the scan cell output, they may invert the output's value.

Related Commands:

> **Delete Cell Constraints** - deletes the constraints from the specified scan cells.
> **Report Cell Constraints** - reports all defined scan cell constraints.

Adding Nofault Settings

Within your design, you may have instances that should not have internal faults included in the fault list. You can label these parts with a *nofault* setting. To add a nofault setting, you use the Add Nofaults command. This command's usage is as follows:

ADD NOfaults ***pathname...*** [-Instance] [-Stuck_at {01 | 0 | 1}]

Or, if you are using the graphical user interface, you can select the **Add > Nofaults...** pulldown menu item.

You can specify that the listed pin pathnames, or all the pins on the boundary and inside the named instances, are not allowed to have faults included in the fault list.

Related Commands:

> **Delete Nofaults** - deletes the specified nofault settings.
> **Report Nofaults** - displays all specified nofault settings.

Setting Up for BIST (FastScan Only)

BIST support is available through FastScan only. For basic information on FastScan's BIST capabilities, refer to "Built-In Self-Test (FastScan Only)" on page 4-28. The following subsections discuss the extra setup FastScan typically needs for designs containing BIST circuitry.

Modifying Scan Chain Access

If your scan chain inputs and outputs do not connect to external pins, you must modify the circuit to make it appear so. This is a requirement for rules checking, but additionally, it provides the connect points for your LFSR.

To make scan chain I/O pins externally accessible, you use the Add Primary Inputs and Add Primary Outputs commands. The usage for these commands follows:

ADD PRimary Inputs ***net_pathname...*** [-Cut] [-Module]

ADD PRimary Outputs ***net_pathname...***

The net_pathname in the Add Primary Inputs command is the circuit connection to which the tool adds a primary input. This should be the scan_in pin. The -Cut option to Add Primary Inputs disconnects the original drivers of the specified pin so the primary input becomes the only driver. The net_pathname in the Add Primary Outputs command is the circuit connection to which the tool adds a primary output. This should be the scan_out pin.

Setting Random or BIST Patterns

To specify the number of random or BIST patterns to apply, you use the Set Random Patterns command. This command's usage is as follows:

SET RAndom Patterns ***integer***

The integer represents the number of patterns for random pattern simulation. By default, this number is 1024.

Selecting the Capture Clock

To specify either which capture clock random pattern simulation should use or which clock procedure to use, you use the Set Capture Clock command. This command's usage is as follows:

SET CApture Clock {***primary_input_pin*** | ***clock_procedure_name***} [-Atpg]

The clock_pin you specify must be a currently defined clock pin. The clock_procedure you specify must be the name of a clock procedure defined in the test procedure file. The -Atpg switch forces all patterns created during ATPG to apply either the selected capture clock or the specified clock procedure.

Selecting the Observation Point

To specify the observation point of the random patterns, you use the Set Observation Point command. This command's usage is as follows:

SET OBservation Point **Master** | **SLave** | **SHadow** | **Clockpo**

You can set observation to master latches and normal primary outputs (the default), slave latches and normal primary outputs, observable shadow latches and normal primary outputs, or only primary outputs directly connected to clocks.

Defining the LFSRs in the BIST Circuitry

If you want to perform BIST simulation (this is not necessary for random pattern simulation), you need to specify the pattern generation and response compression LFSRs, as well as their tap locations and external pin connections. The usage for the LFSR setup commands is as follows:

ADD LFsrs ***ref_name*** {**Prpg** | **Misr**} ***length seed*** [*shift_type*] [*tap_type*]

ADD LFsr Connections ***primary_pin lfsr_name position_list***

ADD LFsr Taps ***lfsr_name position_list***

The Add Lfsrs command specifies the LFSR name, its usage (whether it is a PRPG or MISR), the length of the register (in bits), the seed value for initializing the LFSR, the shift type (either -serial, -parallel, or -both), and the tap type (either -in or -out).

The Add Lfsr Connections command specifies which primary pin to connect to the LFSR, the name of the LFSR, and a list of the LFSR position bits to which the pin connects. The Add Lfsr Taps command specifies the LFSR name and indicates which LFSR bit positions to tap.

Related Commands:

Delete LFSRs - deletes the previously-defined LFSRs.

Delete LFSR Connections - deletes the connections between LFSRs and primary pins.
Delete LFSR Taps - deletes the specified tap positions from an LFSR.
Report LFSRs - displays a list of all defined LFSRs.
Report LFSR Connections - displays a list of all connections between LFSRs and primary pins.
Setup LFSRs - sets the default setting for the shift-type and tap-type switches.

Checking Rules and Debugging Rules Violations

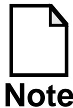
If an error occurs during the rules checking process, the application remains in Setup mode so you can correct the error. You can easily resolve the cause of many such errors; for instance, those that occur during parsing of the test procedure file. Other errors may be more complex and difficult to resolve, such as those associated with proper clock definitions or with shifting data through the scan chain.

FastScan and FlexTest perform model flattening, learning analysis, and rules checking when you try to exit the Setup mode. Each of these processes is explained in detail in “Understanding Common Tool Terminology and Concepts” on page 3-1. As mentioned previously, to change from Setup to one of the other system modes, you enter the Set System Mode command, whose usage is as follows:

SET SYstem Mode {**Setup** | {{**Atpg** | **Fault** | **Good** | **Drc**} [-Force]}

If you are using the graphical user interface, you can click on the palette menu item MODE and then select either "SETUP", "ATPG", "FAULT", or "GOOD".

If you are using FlexTest, you can also troubleshoot rules violations from within the Drc mode. This system mode retains the internal representation of the design used during the design rules checking process.


Note

FastScan does not require the Drc mode because it uses the same internal design model for all of its processes.

The "Troubleshooting Rules Violations" section in the *Design-for-Test Common Resources Manual* discusses the procedure for debugging rules violations. The schematic viewing tool, DFTInsight, is especially useful for analyzing and debugging certain rules violations. The "Using DFTInsight" section in the *Design-for-Test Common Resources Manual* discusses DFTInsight in detail.

Running Good/Fault Simulation on Existing Patterns

The purpose of fault simulation is to determine the fault coverage of the current pattern source for the faults in the active fault list. The purpose of "good" simulation is to verify the simulation model. Typically, you use the good and fault simulation capabilities of FastScan and FlexTest to grade existing hand- or ATPG-generated pattern sets.

Fault Simulation

The following subsections discuss the procedures for setting up and running fault simulation using FastScan and FlexTest.

Changing to the Fault System Mode

Fault simulation runs in Fault mode. Enter the Fault mode as follows:

SETUP> **set system mode fault**

This places the tool in Fault mode, from which you can enter the commands shown in the remaining fault simulation subsections.

If you are using the graphical user interface, you can click on the palette menu item **MODES > Fault**.

Setting the Fault Type

By default, the fault type is stuck-at. If you want to simulate patterns to detect stuck-at faults, you *do not* need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay (FastScan only), you can issue the Set Fault Type command. This command's usage is as follows:

SET FAult Type **Stuck** | **Iddq** | **TOggle** | **TRansition** | **Path_delay**

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Creating the Faults List

Before you can run fault simulation, you need an active fault list from which to run. You create the faults list using the Add Faults command, whose usage is follows:

ADD FAults ***object_pathname...*** | **-All** [-Stuck_at {01 | 0 | 1}]

Typically, you would create this list using all faults as follows:

```
FAULT> add faults -all
```

"Setting Up the Fault Information for ATPG" on page 6-62 provides more information on creating the fault list and specifying other fault information.

Setting the Pattern Source

You can have the tools perform simulation and test generation on a selected pattern source, which you can change at any time. To set the test pattern source, you use the Set Pattern Source command, which varies in its options between FastScan and FlexTest. This command's common usage is as follows:

SET PAttern Source **Internal** | {**External** ***filename***} [-NOPadding]}

For either application, the pattern source may be internal or external. The ATPG process creates internal patterns, which are the default source. In Atpg mode, the internal pattern source indicates that the test pattern generator will create the patterns. The External option uses patterns that reside in a named external file.

For FastScan only, the tool can perform simulation with a select number of random patterns, or a set of BIST patterns. FlexTest can additionally read in Table format, and also lets you specify what value to use for pattern padding. Refer to the *FastScan and FlexTest Reference Manual* for additional information on these application-specific Set Pattern Source command options.

Related Commands: The following related commands apply if you select the Random or Bist pattern source option:

> **Set Capture Clock** - specifies the capture clock for random pattern simulation.
> **Set Random Clocks** - specifies the selection of clock_sequential patterns for random pattern simulation.
> **Set Random Patterns** - specifies the number of random patterns to be simulated.

Executing Fault Simulation

You execute the fault simulation process by using the Run command in Fault mode. You can repeat the Run command as many times as you want for different pattern sources. To execute the fault simulation process, enter the Run command from the Fault system mode as follows:

FAULT> **run**

FlexTest has some options to the run command, which can aid in debugging fault simulation and ATPG. Refer to the *FastScan and FlexTest Reference Manual* for information on the Run command options.

Related Commands:

Report Faults - displays faults for selected fault classes.
Report AU Faults - displays information on undetected faults.
Report Statistics - displays a statistics report.
Report Core Memory - displays real memory required during ATPG and fault simulation.

Writing the Undetected Faults List

Typically, after performing fault simulation on an external pattern set, you will want to save the faults list. You can then use this list as a starting point for ATPG. To save the faults, you use the Write Faults command, whose usage is as follows:

WRIte FAults ***filename*** [-Replace] [-Class *class_type*] [-Stuck_at {01 | 0 | 1}] [-All | *object_pathname*...] [-Hierarchy *integer*] [-Min_count *integer*] [-Noeq]

Refer to "Writing Faults to an External File" on page 6-65 or the Write Faults command page in the *FastScan and FlexTest Reference Manual* for command option details.

To read the faults back in for ATPG, go to Atpg mode (using Set System Mode) and enter the Load Faults command. This command's usage is as follows:

LOAd FAults ***filename*** [-Restore | -Delete | -Delete_Equivalent]

Debugging the Fault Simulation

To debug your fault simulation, you can write a list of pin values that differ between the faulty and good machine. You do this using the Add Lists and Set List File commands. The usage for these commands follows:

ADD LIsts ***pin_pathname...***

SET LIst File {***filename*** [-Replace]}

The Add Lists command specifies which pins you want reported. The Set List File command specifies the name of the file in which to place simulation values for the selected pins. The default behavior is to write pin values to standard output.

Resetting Circuit and Fault Status

You can reset the circuit status and status of all testable faults in the fault list to undetected. Doing so lets you redo the fault simulation using the current fault list. In Fault mode this does not cause deletion of the current internal pattern set. To reset the testable faults in the current fault list enter the Reset State command at the Fault mode prompt as follows:

```
FAULT> reset state
```

Fault Simulation on MGC WDB Format Vectors (FlexTest Only)

In many cases, you begin test generation with a set of vectors previously derived from a simulator. You can read in these external patterns (in Mentor Graphics Waveform Database format), convert them to FlexTest Table format, and have FlexTest perform fault simulation on them. FlexTest uses these existing patterns to initialize the circuit and give some initial fault coverage. Then you can perform ATPG on the remaining faults. This method can result in more efficient test pattern sets and shorter test generation run times.

Converting MGC WDB to FlexTest Table Format

A shell utility called "*wdb2flex*"provides the means for reading MGC WDB into FlexTest. The invocation for **wdb2flex** is as follows:

```
shell> WDB2FLEX [-o <output_file>] <control_file>
<forces_wdb> [<results_wdb>]
```

The utility applies the name *table.flex* to the default output file. If you want to choose a different output file name, specify the -o switch with a different <output_file> name. The file named in <control_file> lets you set up the sampling of the waveforms in the file named in <forces_wdb>. You can optionally read in the <results_wdb> file. However, if the circuit contains bidirectionals, this argument is required to properly identify these signals.

For more information on the **wdb2flex** utility, including the available control file commands, refer to "FlexTest WDB Translation Support" in the *FastScan and FlexTest Reference Manual*.

Running Fault Simulation on the Functional Vectors

To run fault simulation on the vectors you converted to FlexTest table format, use the following commands:

```
SETUP> set system mode atpg
ATPG> set pattern source external table.flex-table
ATPG> add faults -all
ATPG> run
ATPG> set pattern source internal
ATPG> run
```

First, set the system mode to Atpg if you are not already in that system mode. Next, you must specify that the patterns you want to simulate are in an external file (by default, named *table.flex*). Then generate the fault list including all faults, and run the simulation. You could then set the pattern source to be internal and run the basic ATPG process on the remaining undetected faults.

Saving and Restoring Undetected Faults for Use with FastScan

The preceding procedure assumes you are running ATPG with FlexTest. You can also run ATPG with FastScan. In this case, you need to write all the faults to an external list using the Write Faults -All command in FlexTest. Then you use the Load Faults -Restore command in FastScan, which loads in all faults while preserving their categorization. You can then run ATPG using FastScan on this fault list.

Good Machine Simulation

Given a test vector, you use *good machine simulation* to predict the logic values in the good (fault-free) circuit at all the circuit outputs. The following subsections discuss the procedures for running good simulation on existing hand- or ATPG-generated pattern sets using FastScan and FlexTest.

Changing to the Good System Mode

You run good machine simulation in the Good system mode. Enter the Good system mode as follows:

ATPG> **set system mode good**

Specifying an External Pattern Source

By default, good machine simulation runs using an internal ATPG-generated pattern source. To run good machine simulation using an external hand-generated pattern set, enter the following command:

GOOD> **set pattern source external *filename***

Executing Good Machine Simulation

During good machine simulation, the tool compares good machine simulation results to an external pattern source, primarily for debugging purposes. To set up good circuit simulation comparison within FlexTest, you use the Set Output Comparison command from the Good system mode. This command's usage is as follows:

SET OUtput Comparison **OFf** | {**ON** [**-X_ignore** [**None** | **Reference** |
Simulated | **Both**]]} [-Io_ignore]

By default, the output comparison of good circuit simulation is off. FlexTest performs the comparison if you specify ON. The -X_ignore options will allow you to control whether X values in either simulated results or reference output should be ignored when output comparison capability is used.

To execute the simulation comparison, enter the Run command at the Good mode prompt as follows:

GOOD> **run**

Debugging the Good Machine Simulation

You can debug your good machine simulation in several ways. If you want to run the simulation and save the values of certain pins in batch mode, you can use the Add Lists and Set List File commands. The usage for these commands is as follows:

ADD LIsts ***pin_pathname...***

SET LIst File {***filename*** [-Replace]}

The Add Lists command specifies which pins to report. The Set List File command specifies the name of the file in which you want to place simulation values for the selected pins.

If you prefer to perform interactive debugging, you can use the Run and Report Gates commands to examine internal pin values. If using FlexTest, you can use the -Record switch with the Run command to store the internal states for the specified number of test cycles.

Resetting Circuit Status

You can reset the circuit status by using the Reset State command as follows:

GOOD> **reset state**

Running Random/BIST Pattern Simulation (FastScan)

In a circuit containing BIST, an LFSR generates a select number of pseudo-random patterns for testing the circuit. To determine the test coverage of these patterns, you can use one of two methods: *random pattern simulation*, or *BIST pattern simulation*.

Random pattern simulation simulates an equivalent number of random patterns to predict the test coverage of the BIST patterns generated by the LFSR. Although the patterns actually differ, there is a strong statistical correlation between the simulated and actual results because all the patterns are generated randomly. To get an even better correlation, you could take the average of the test coverages from several random pattern simulation runs.

BIST pattern simulation simulates the user-defined LFSRs to calculate the actual BIST test coverage and expected signatures that result from the application of the BIST patterns.

The following subsections outline the procedures for running both random pattern and BIST pattern simulations.

Random Pattern Simulation

The following subsections show the typical procedure for running random pattern simulation.

Changing to the Fault System Mode

You run random pattern simulation in the Fault system mode. If you are not already in the fault system mode, enter the Fault system mode as follows:

SETUP> **set system mode fault**

If you are using the graphical user interface, you can click on the palette menu item MODES > Fault.

Setting the Pattern Source to Random

To set the pattern source to random, use the Set Pattern Source command as follows:

FAULT> **set pattern source random**

Creating the Faults List

To generate the faults list and eliminate all untestable faults, use the Add Faults and Delete Faults commands together as follows:

FAULT> **add faults -all**

FAULT> **delete faults -untestable**

The Delete Faults command with the -untestable switch removes faults from the fault list that are untestable using BIST or random patterns.

Running the Simulation

To run the random pattern simulation, specify the Run command as follows:

FAULT> **run**

After the simulation run, you can display the undetected faults with the Report Faults command. Some of the undetected faults may be redundant. You can run ATPG on the undetected faults to identify those that are redundant.

BIST Pattern Simulation

The following subsections show the typical procedure for running BIST pattern simulation. Because of the appearance of the commands and their usage in previous sections, this section shows the relevant commands only in the context of their BIST usage.

Running BIST Pattern Simulation

The procedure for running BIST pattern simulation is identical to the previous procedure for running random pattern simulation. The only difference lies with the Set Pattern Source command. To run BIST pattern simulation, specify the BIST option, instead of the random option, to this command as follows:

```
FAULT> set pattern source bist
```

All other steps in the process are exactly the same.

Troubleshooting the Simulation

If, during BIST pattern simulation, an X state propagates to a MISR, an error condition occurs and simulation stops. The system reports the BIST pattern number and the scan cell or primary output with the X value. To display the final MISR signature values, you can use the Report Lfsrs command as follows:

```
FAULT> report lfsrs
```

To identify the source of an unknown state that propagates to a MISR, use the Set Gate Report and Report Gates commands as follows:

```
FAULT> set gate report error_pattern
FAULT> report gates <gate_id#>
```

The Set Gate Report command sets the gate reporting to display the simulated gate values and input conditions for the pattern at which the error occurred. The Report Gates command displays information on the gate that caused the X condition. Using the input values and the input connectivity of the previous Report Gate command, you can repeatedly use the Report Gate command until you identify the source of the X condition.

Storing BIST Patterns

After you run a successful BIST pattern simulation, you may want to store the generated BIST patterns. To store BIST patterns, use the following commands:

```
FAULT> set pattern source bist -store_patterns
FAULT> set system mode good
GOOD> run
GOOD> save patterns <pattern_filename>
```

The -store_patterns option to the Set Pattern Source command allows storage in the internal pattern set of patterns simulated in the Good system mode. Setting the system mode to Good and executing the Run command simulates the BIST patterns. And the Save Patterns commands saves the internal pattern set to the specified pattern filename in ASCII format.

Obtaining Optimum BIST Coverage

A BIST circuit's testability depends on the effectiveness of random pattern testing. Thus, the challenge is to intelligently add artificial controllability and observability to the design to increase its test coverage. FastScan can help you achieve this goal.

To improve controllability and observability in a BIST circuit, you should analyze, and possibly modify, the control and observe points in your circuit. The following subsections describe how you can accomplish this using FastScan.

Analyzing Controllability

The tool calculates controllability test coverage by examining the percentage of adequately-controlled pins. It considers a pin adequately controlled if it is both a 0 and a 1 for the minimum number of patterns (threshold) within the simulated random pattern set. Therefore, the higher the threshold number, the more pins that fail controllability checks and become candidates for test points.

For each output pin that is not adequately controlled, the system calculates a potential source of its control problem by tracing back from the pin, through its most difficult to control input, until it encounters a gate whose inputs all have a controllability value greater than the threshold. You can list the source gates, ordered by the number of inadequately controlled pins they contain, to identify the most productive points at which to add controllability.

To analyze the controllability of your circuit, you must first set up the number of random patterns, the capture clock, and the observation point. Then, from the Good simulation mode, use the following commands:

GOOD> **set control threshold <integer>**
GOOD> **analyze control**
GOOD> **report control data <filename>**

The Set Control Threshold command lets you specify the controllability threshold; that is, the minimum number of times a gate must be a zero or one state during random pattern simulation to be considered adequately controlled. The default value is four.

The Analyze Control command calculates the actual zero and one-state controllability. Then the Report Control Data command writes in the specified file a list of low-controllability gates, the gate values, the minimum threshold value, and the possible source of the controllability problem.

Analyzing Observability

FastScan calculates observability test coverage by performing fault simulation for a selected number of random patterns and examining the percentage of adequately-observed pins. It considers a pin adequately observed if it is both a 0 and a 1 for the minimum number of patterns (threshold) within the simulated random pattern set. Therefore, the higher the threshold number, the more pins that fail observability checks and become candidates for test points.

The tool calculates the potential problem of inadequately observed pins by tracing forward from the pin, through the most difficult to observe fanout gate, until encountering a gate that has no fanout with an observability value less than the threshold.

To analyze the observability of your circuit, you must first set up the number of random patterns, the capture clock, and the observation point. Then from the Fault simulation mode, use the following commands:

```
FAULT> set observe threshold <integer>
FAULT> analyze observe
FAULT> report observe data <filename>
```

The Set Observe Threshold command lets you specify the observability threshold; that is, the minimum number of observations during the selected patterns for adequate observation of a point. The Analyze Observe command calculates the actual observability coverage. Then the Report Observe Data command writes in the specified file a list of low-observability gates, the number of patterns in which the pin achieved the state, and the calculated source of the observability problem.

Inserting Control and Observe Points

Fault simulation, controllability analysis, and observability analysis can indicate problems with BIST test coverage that may require you to add additional control and observe points to the circuit. FastScan lets you select control and observe points for insertion into the circuit. You can then observe their effects on the circuit by performing additional controllability or observability analysis, or fault simulation.

You add control and observe points using the Add Control Points and Add Observe Points commands. The usage for these commands is as follows:

ADD COntrol Points ***pin_pathname...*** [-Type {Xor | And | Or}] [-Group]

ADD OBserve Points ***pin_pathname...***

The Add Control Points command adds control points to the output pins of cells, modeled in the selected way. You can model the control effect by either eXclusive-ORing, ANDing, or ORing the pin's value with random values. The Add Observe Points command adds observe points at the specified output pins.

Related Commands:

Add Notest Points - adds circuit points that cannot be used for testability insertion.
Report Control Points - displays a list of control points.
Report Observe Points - displays a list of all observe points.

Performing Automatic Testability Analysis

FastScan can perform a complete testability analysis of your design. Using "soft" circuit modifications, it produces a maximum test coverage with a maximum number of inserted control and observe points from a selected number of patterns. Prior to running an automatic testability analysis, you must set the number of random patterns, the capture clock, the observation point, the control threshold, and the observe threshold. Then to obtain an automatic testability analysis of your design, you use the Insert Testability command. This command's usage is as follows:

INSert TEstability [-Control_max *integer*] [-Observe_max *integer*]

The Insert Testability command performs the following actions:

- Deletes all learned circuit information (because of circuit modifications).
- Determines the testable nodes for control and observe analysis (considering the effects of pin constraints and connectivity to observe points).
- Performs control analysis and inserts control points until either all testable circuit nodes achieve minimum controllability or it inserts the maximum number of control points.
- Performs observe analysis and inserts observe points until either all testable circuit nodes have achieved minimum observability or it inserts the maximum number of observe points.
- Performs random pattern fault simulation to calculate the modified circuit's expected random pattern fault coverage.
- Performs test pattern generation on untested faults to identify redundant faults (which the tool excludes from the final test coverage calculation).

Following this command, you can report all control and observe points, using the Report Control Points and Report Observe Points commands, respectively.

Example ATPG Run on a BIST Circuit

This scan design is an 8-bit binary counter. The pins include D (system data port), CD (active-low, asynchronous clear port), TI (scan data port), TE (test mode selection port), CLK (system clock port), Q (output), and QN (complementary output).

This design additionally contains BIST circuitry. Figure 6-9 gives a simple block diagram of the added BIST circuitry.

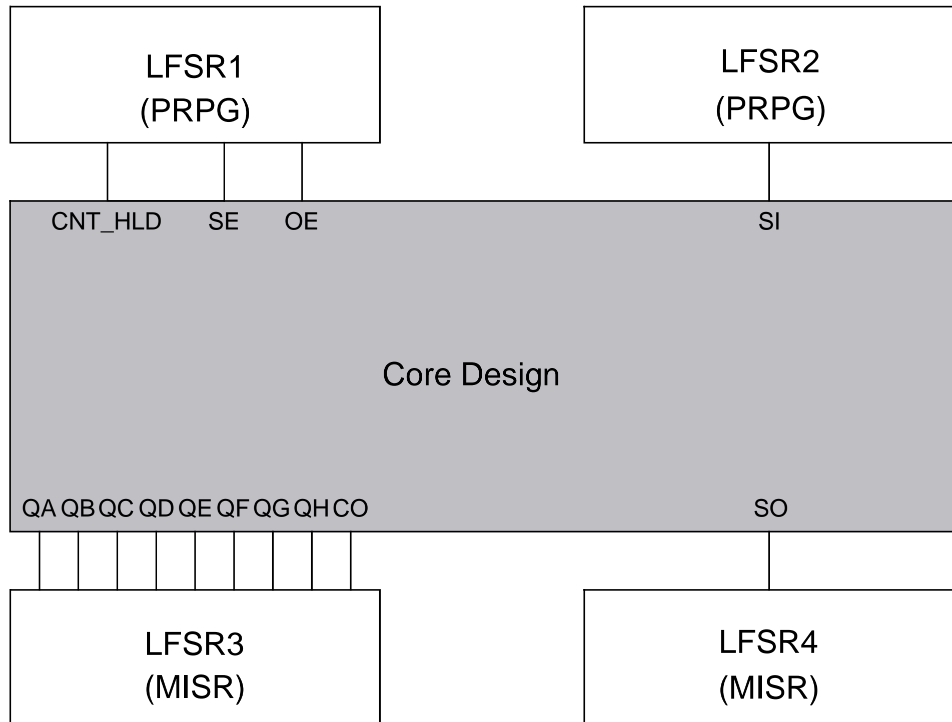


Figure 6-9. Block Diagram of BIST Example Circuit

The test procedure file (*counter.g1*) for this design follows:

```
proc shift =
  force_sci        0;
  measure_sco      0;
  force CLK      1 1;
  force CLK      0 2;
end;

proc load_unload =
  force SE       1 0;
  force CLEAR    1 0;
  force CLK      0 0;
  apply shift   10 1;
end;
```

The FastScan commands you could run (probably interactively--at least for the BIST-specific commands) to simulate BIST patterns are as follows:

```
//Setup scan and circuit info
add scan groups g1 counter.g1
add scan chains c1 g1 si so
add clocks 0 clk
add clocks 1 clear
set z handling external 0

//Define and specify LSFR info for LSFR1
add lfsrs lfsr1 prpg 12 2fb -parallel -out
add lfsr taps lfsr1 1 2 5 8 9 11
add lfsr connections cnt_hld lfsr1 2
add lfsr connections se lfsr1 6
add lfsr connections oe lfsr1 10

//Define and specify LSFR info for LSFR2
add lfsrs lfsr2 prpg 20 abc -serial -in
add lfsr taps lfsr2 5 8 10 11 13 15 19
add lfsr connections si lfsr2 16

//Define and specify LSFR info for LSFR3
add lfsrs lfsr3 misr 18 def -parallel -in
add lfsr taps lfsr3 1 7 9 12 14
add lfsr connections qa lfsr3 2
add lfsr connections qb lfsr3 4
add lfsr connections qc lfsr3 6
add lfsr connections qd lfsr3 7
add lfsr connections qe lfsr3 8
add lfsr connections qf lfsr3 9
add lfsr connections qg lfsr3 10
add lfsr connections qh lfsr3 13
add lfsr connections co lfsr3 17

//Define and specify LSFR info for LSFR4
add lfsrs lfsr4 misr 16 89ab -both -out
add lfsr taps lfsr4 2 5 8 9 12 15
add lfsr connections so lfsr4 211
```

```
//Fault simulate BIST patterns
set system mode fault
set pattern source bist
add faults -all
delete faults -untestable run

//Analyze controllability and observability
set control threshold 20
analyze control report control data control_info -replace
analyze observe report observe data observe_info -replace
insert testability exit
```

Setting Up the Fault Information for ATPG

Prior to performing test generation, you must set up a list of all faults the application has to evaluate. The tool can either read the list in from an external source, or generates the list itself. The type of faults in the fault list vary depending on the fault model and your targeted test type. For more information on fault modeling and the supported models, refer to "Fault Modeling" on page 2-22.

After the application identifies all the faults, it implements a process of structural equivalence fault collapsing from the original uncollapsed fault list. From this point on, the application works on the collapsed fault list. However, the results are reported for both the uncollapsed and collapsed fault lists. Executing any command that changes the fault list causes the tool to discard all patterns in the current internal test pattern set due to the probable introduction of inconsistencies. Also, whenever you re-enter the Setup mode, it deletes all faults from the current fault list. The following subsections describe how to create a fault list and define fault related information.

Changing to the ATPG System Mode

You can enter the fault list commands from the Good, Fault, or Atpg system modes. However, in the context of running ATPG, you must switch from Setup to the Atpg mode.

Assuming your circuit passes rules checking with no violations, you can exit the Setup system mode and enter the Atpg system mode as follows:

SETUP> **set system mode atpg**

If you are using the graphical user interface, you can click on the palette menu item MODES > ATPG.

Setting the Fault Type

By default, the fault type is stuck-at. If you want to generate patterns to detect stuck-at faults, you *do not* need to issue this command.

If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, or path delay (FastScan only), you can issue the Set Fault Type command. This command's usage is as follows:

SET FAult Type **Stuck** | **Iddq** | **TOggle** | **TRansition** | **Path_delay**

Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Creating the Faults List

The application creates the internal fault list the first time you add faults or load in external faults. Typically, you would create a fault list with all possible faults of the selected type, although you can place some restrictions on the types of faults in the list. To create a list with all faults of the given type, enter the Add Faults command using the -All switch as follows:

ATPG> **add faults -all**

If you are using the graphical user interface, you can click on the palette icon item ADD FAULTS and specify All in the dialog box that appears.

If you do not want all possible faults in the list, you can use other options of the Add Faults command to restrict the added faults. You can also specify no-faulted instances to limit placing faults in the list. You flag instances as "Nofault" while in Setup mode. For more information, refer to "Adding Nofault Settings" on page 6-41.

When the tool first generates the fault list, it classifies all faults as uncontrolled (UC).

Related Commands:

> **Delete Faults** - deletes the specified faults from the current fault list.
> **Report Faults** - displays the specified types of faults.

Adding Faults to an Existing List

To add new faults to the current fault list, enter the Add Faults command as follows:

ADD FAults ***object_pathname...*** | **-All** [-Stuck_at {01 | 0 | 1}]

If you are using the graphical user interface, you can click on the palette icon item ADD FAULTS and specify which faults you want to add in the dialog box that appears.

You must enter either a list of object names (pin pathnames or instance names) or use the -All switch to indicate the pins whose faults you want added to the fault list. You can use the -Stuck-at switch to indicate which stuck faults on the selected pins you want added to the list. If you do not use the Stuck-at switch, the tool adds both stuck-at-0 and stuck-at-1 faults. FastScan and FlexTest initially place faults added to a fault list in the undetected-uncontrolled (UC) fault class.

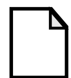
Loading Faults from an External List

You can place faults from a previous run (from an external file) into the internal fault list. To load faults from an external file into the current fault list, enter the Load Faults command. This command's usage is as follows:

LOAd FAults ***filename*** [-Restore | -Delete | -Delete_Equivalent]

The applications support external fault files in the 3, 4, or 5 column formats. The only data they use from the external file is the first column (stuck-at value) and the last column (pin pathname)--unless you use the -Restore option.

The -Restore option causes the application to retain the fault class (second column of information) from the external fault list. The -Delete option deletes all faults in the specified file from the internal faults list. The -DELETE_Equivalent option deletes from the internal fault list all faults in the file, as well as all their equivalent faults.


Note

The ***filename*** specified cannot have fault information lines with comments appended to the end of the lines or fault information lines greater than 5 columns. The tool will not recognize the line properly and will not add the fault on that line to the faultlist.

Writing Faults to an External File

You can write all or only selected faults from a current fault list into an external file. You can then edit or load this file to create a new fault list. To write faults to a file, enter the Write Faults command as follows:

WRIte FAults ***filename*** [-Replace] [-Class *class_type*] [-Stuck_at {01 | 0 | 1}] [-All | *object_pathname...*] [-Hierarchy *integer*] [-Min_count *integer*] [-Noeq]

You must specify the name of the file you want to write. For information on the remaining Write Faults command options, refer to the *FastScan and FlexTest Reference Manual*.

ATPG Library Verification (FlexTest Only)

FlexTest has the capability to generate ATPG library verification setup files. You can access this feature by using the Write Library_verification Setup command.

WRIte LIbrary_verification Setup ***basename*** [-Replace]

This command creates three dofiles for verification of test vectors and simulation results. For more information on this command and its features, refer to the Write Library_verification Setup command page in the *FastScan and FlexTest Reference Manual*.

Setting Self-Initialized Test Sequences (FlexTest Only)

FlexTest generates test sequences for target faults that are self-initialized. With the knowledge of self-initialized test sequences, static vector compaction by reordering is possible as well as splitting the test set without losing test coverage. Some pattern compaction routines also rely on self-initializing properties of sequences. Each self-initialized test sequence is defined as a *test pattern* (to be compatible with FastScan).

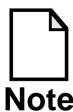
The Set Self Initialization command allows you to turn this feature on or off. By default, self-initializing behavior is on.

SET SElf Initialization **ON | OFf**

If the self-initializing property is enabled during ATPG:

- self-initializing boundaries in the test set will be determined
- during fault simulation, all state elements (except the ones with TIED properties) at self-initializing boundaries are set to X. Therefore, the reported fault coverage is actually the lower bound to the real fault coverage if state information were maintained between self-initializing sequences (the reported coverage will be close to or equal to the real fault coverage).

The self-initializing results can be saved by issuing the Save Patterns -Ascii command.


Note

Only the ASCII pattern format includes this test pattern information.

Setting the Fault Sampling Percentage (FlexTest Only)

By reducing the fault sampling percentage (which by default is 100%), you can decrease the process time to evaluate a large circuit by telling the application to process only a fraction of the total collapsed faults. To set the fault sampling

percentage, you use the Set Fault Sampling command. This command's usage is as follows:

SET FAult Sampling ***percentage***

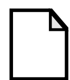
You must specify a percentage (between 1 and 100) of the total faults you want processed.

Setting the Fault Mode

You can specify use of either the collapsed or uncollapsed fault list for fault counts, test coverages, and fault reports. The default is to use uncollapsed faults.

To set the fault mode, you use the Set Fault Mode command. This command's usage is as follows:

SET FAult Mode **Uncollapsed** | **Collapsed**


Note

The Report Statistics command always reports both uncollapsed and collapsed statistics. Therefore, the Set Fault Mode command is useful only for the Report Faults and Write Faults commands.

Setting the Hypertrophic Limit (FlexTest Only)

To improve fault simulation performance, you can reduce or eliminate hypertrophic faults with little consequence to the accuracy of the fault coverage. In fault simulation, hypertrophic faults require additional memory and processor time. These type of faults do not occur often, but do significantly affect fault simulation performance. To set the hypertrophic limit, enter the Set Hypertrophic Limit command as follows:

SET HYpertrophic Limit **Off** | **Default** | **To** ***percentage***

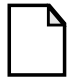
You can specify a percentage between 1 and 100, which means that when a fault begins to cause more than that percent of the state elements to deviate from the good machine status, the simulator will drop that fault from simulation. The default is a 30% difference (between good and faulty machine status) to classify a fault as hypertrophic. To improve performance, you can reduce the percentage number.

Setting the Possible-Detect Credit

Before reporting test coverage, fault coverage, and ATPG effectiveness, you should specify the credit you want given to possible detected faults. To set the credit given possible detected faults, you use the Set Possible Credit command. This command's usage is as follows:

SET POssible Credit ***percentage***

The selected credit may be any positive integer less than or equal to 100, the default being 50%.



If you are using FlexTest and you set the possible detection credit to 0, it does not place any faults in the possible-detected category. If faults already exist in these categories, the tool reclassifies PT faults as UO and PU faults as AU.

Running ATPG

Obtaining the optimal test set in the least amount of time is a desirable goal. Figure 6-10 outlines how to most effectively meet this goal.

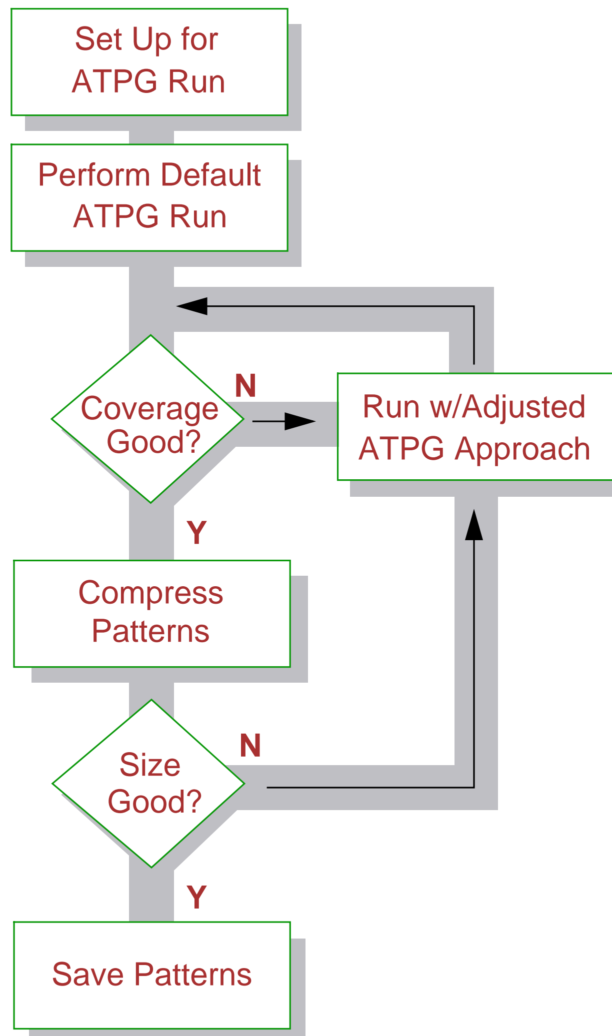


Figure 6-10. Efficient ATPG Flow

The first step in the process is to perform any special setup you may want for ATPG. This includes such things as setting limits on the ATPG process itself. The second step is to perform an ATPG run with default settings (see page 6-76). This is a very fast way to determine how close you are to your testability goals. In fact, you may even obtain the test coverage you desire from this very first run.

However, if your test coverage is not at the required level, you may have to troubleshoot the reasons for the inadequate coverage and perform the ATPG run again using other approaches (see page 6-78). Once you achieve the desired test coverage, you should statically compress the generated pattern set (see page 6-76). If the size of the test set is adequate, you can save the patterns and be finished with ATPG. However, if the test set is still too large, you can re-run ATPG with dynamic compression turned on during pattern generation.

The following subsections discuss each of these tasks in more detail.

Setting Up for ATPG

Prior to running ATPG, you may need to set certain criteria that aid the test generators in the test generation process. The following subsections discuss the typical tasks you may need to perform before running ATPG. If you just want to perform a quick ATPG run using default settings, refer to "Performing a Default ATPG Run" on page 6-76.

Defining ATPG Constraints

ATPG constraints are similar to pin constraints and scan cell constraints. Pin constraints and scan cell constraints let you restrict the values of pins and scan cells, respectively. ATPG constraints let you place restrictions on the acceptable kinds of values at any location in the circuit. For example, you can use ATPG constraints to prevent bus contention or other undesirable events within a design. Additionally, your design may have certain conditions that can never occur under normal systems operation. If you want to place these same constraints on the circuit during ATPG, you would use ATPG constraints to do so.

During deterministic pattern generation, the tool allows only the restricted values on the constrained circuitry. Unlike pin and scan cell constraints, which are only available in Setup mode, you can define ATPG constraints in any system mode--after design flattening. If you want to set ATPG constraints prior to performing design rules checking, you must first create a flattened model of the design using the Flatten Model command.

ATPG constraints are useful when you know something about the way the circuit behaves that you want the ATPG process to examine. For example, the design

may have a portion of circuitry that behaves like a bus system; that is, only one of various inputs may be on, or selected, at a time. Using ATPG constraints, combined with a defined *ATPG function*, you can specify this information to FastScan or FlexTest. ATPG functions let you place artificial boolean relationships on circuitry within your design. After defining the functionality of a portion of circuitry with an ATPG function, you can then constrain the value of the function as desired with an ATPG constraint. This can be far more useful than just constraining a point in a design to a specific value.

FlexTest allows you to specify temporal ATPG functions by using a Delay primitive to delay the signal for one timeframe. Temporal constraints can be achieved by combining ATPG constraints with the temporal function options.

You can define ATPG functions with the Add Atpg Functions command. This command's usage is as follows:

ADD ATpg Functions ***function_name type*** {***pin_pathname*** | ***gate_id#*** | ***function_name*** | {**-Cell** ***cell_name*** {***pin_name***...}}}...

To define a function, you specify a name, a function type, and the object to which the function applies. FlexTest has additional options for temporal functions and supports function application to specific net path names. For more information on these options, refer to the *FastScan and FlexTest Reference Manual*.

You can specify ATPG constraints with the Add Atpg Constraints command. This command's usage is as follows:

ADD ATpg Constraints {**0** | **1** | **Z**} ***object...*** [-Cell *cell_name pin_name*...] [-Dynamic | -Static]

To define ATPG constraints, you specify a value, an object, and whether the constraint is static or dynamic. FlexTest supports constraint additions to specific net path names as well. For more information, refer to the F*astScan and FlexTest Reference Manual*.

Test generation considers all current constraints. However, design rules checking considers only static constraints. You can only add or delete static constraints in Setup mode. Design rules checking does not consider dynamic constraints unless you explicitly use the -ATPGC switch with the Set Drc Handling command. You

can add or delete dynamic constraints at any time during the session. By default, ATPG constraints are dynamic.

Figure 6-11 and the following commands give an example of how you use ATPG constraints and functions together.

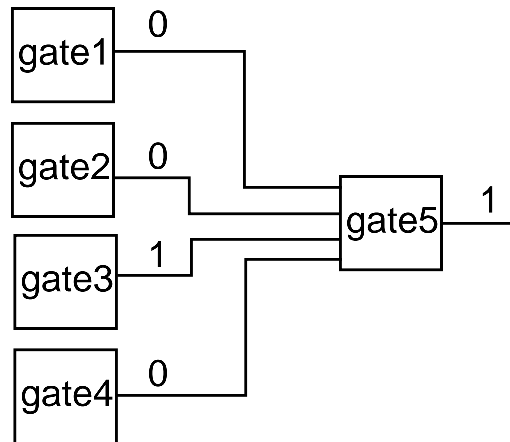


Figure 6-11. Circuitry with Natural "Select" Functionality

The circuitry of Figure 6-11 includes four gates whose outputs are the inputs of a fifth gate. Assume you know that only one of the four inputs to gate5 can be on at a time. You can specify this using the following commands:

```
ATPG> add atpg functions sel_func1 select1 1 2 3 4
ATPG> add atpg constraints 1 sel_func1
```

These commands specify that the "select1" function applies to gate1, gate2, gate3, and gate4 (gate IDs 1, 2, 3, and 4, respectively), and the output of the select1 function should always be a 1. Deterministic pattern generation must ensure these conditions are met. The conditions causing this constraint to be true are shown in Table 6-1. When this constraint is true, the output of gate5 will be on.

Table 6-1. ATPG Constraint Conditions

gate1	gate2	gate3	gate4	gate5
0	0	0	1	1
0	0	1	0	1
0	1	0	0	1

Table 6-1. ATPG Constraint Conditions

gate1	gate2	gate3	gate4	gate5
1	0	0	0	1

Given the defined function and ATPG constraint you placed on the circuitry, FastScan and FlexTest only generate patterns using the values shown in Table 6-1.

Typically, if you have defined ATPG constraints, the tools do not perform random pattern generation during the ATPG run. However, using FastScan you can force the pattern source to random (using Set Pattern Source Random). In this situation, FastScan rejects patterns during fault simulation that do not meet the currently-defined ATPG constraints.

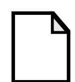
Related Commands:

> **Analyze Atpg Constraints** - analyzes a given constraint for either its ability to be satisfied or for mutual exclusivity.
> **Analyze Restrictions** - performs an analysis to automatically determine the source of the problems from a failed ATPG run.
> **Delete Atpg Constraints** - removes the specified constraint from the list.
> **Delete Atpg Functions** - removes the specified function definition from the list.
> **Report Atpg Constraints** - reports all ATPG constraints in the list.
> **Report Atpg Functions** - reports all defined ATPG functions.

Setting ATPG Limits

You can have FastScan and FlexTest terminate the ATPG process if CPU time, test coverage, or pattern (cycle) count limits are met. To set these limits, use the Set ATPG Limits command. This command's usage is as follows:

SET ATpg Limits [-Cpu_seconds {*integer* | OFf}] [-Test_coverage {*real* | OFf}] [-Pattern_count {*integer* | OFf} | -CYcle_count {*integer* | OFf}]


Note

The -Pattern_count option applies only to FastScan and the -Cycle_count option applies only to FlexTest.

FlexTest Only - The last test sequence generated by an atpg process is truncated to make sure the total test cycles do not exceed cycle limit.

Setting Event Simulation (FastScan Only)

By default, FastScan simulates a single event per test cycle, which corresponds to the point in the simulation cycle when clocks have pulsed and combinational logic has updated, but the state elements have not yet changed. This is adequate for most circuits. However, circuits that use both clock edges or have level sensitive logic may require the multiple event simulation mode.

FastScan uses its clock sequential fault simulator to simulate multiple events in a single cycle. Figure 6-12 illustrates the possible events.

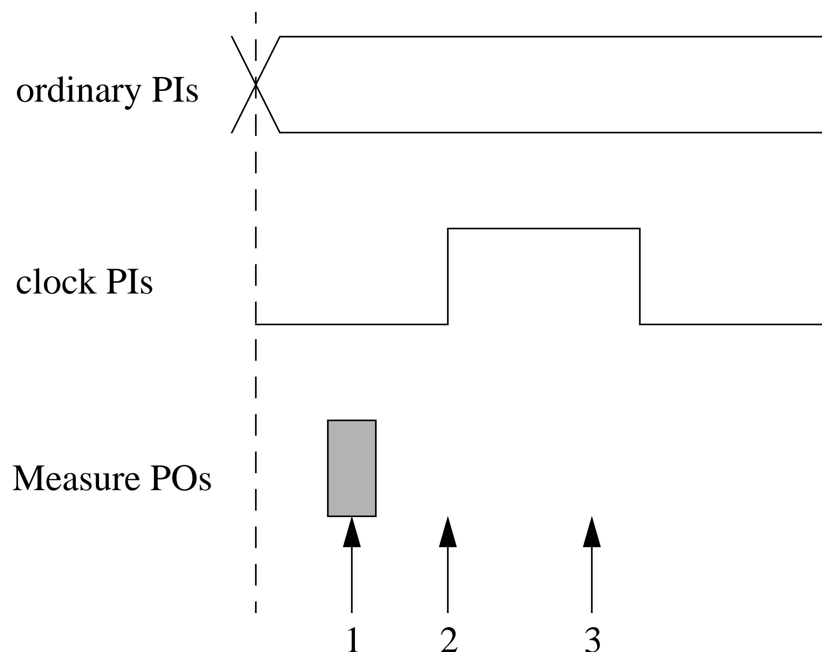


Figure 6-12. Single Cycle Multiple Events

Event 1 represents a simulation where all clock primary inputs are at their “off” value, other primary inputs have been forced to values, and state elements are at the values scanned in or resulting from capture in the previous cycle. When simulating this event, FastScan provides the capture data for inputs to leading edge triggered flip-flops. The Set Clock_off Simulation command enables or disables the simulation of this event.

This command's usage is as follows:

SET CLock_off Simulation **ON** | **OFf**

If DRC flags C3 violations, you should run ATPG using the Set Split Capture_cycle command.

Event 2 corresponds to the default simulation performed by FastScan. It represents a point in the simulation cycle where the clocks have just been pulsed. State elements have not yet changed, although all combinational logic, including that connected to clocks, has been updated.

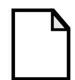
Event 3 corresponds to a time when level sensitive and leading edge state elements have updated as a result of the applied clocks. This simulation correctly calculates capture values for trailing edge and level sensitive state elements, even in the presence of C3 violations. The Set Split Capture_cycle enables or disables the simulation of this event. This command's usage is as follows:

SET SPlit Capture_cycle **ON** | **OFf**

If DRC flags C6 violations, you should run ATPG using the Set clock_off Simulation command.

All Zhold gates hold their value between events 1 and 2, even if the zhold is marked as having clock interaction. All latches maintain state between events 1 to 2 and 2 to 3, although state will not be held in TLAs between cycles.

If you issue both commands, each cycle of the clock results in up to 3 simulation passes with the leading and falling edges of the clock simulated separately.


Note

These are not available for RAM sequential simulations. Since clock sequential ATPG can test the same faults as RAM sequential, this is not a real limitation.

Setting the Checkpoint

Checkpointing lets you automatically save test patterns during the ATPG process. There are two checkpoint commands: Set Checkpoint, which turns the checkpoint functionality on or off, and Setup Checkpoint, which identifies the time period and the name of the pattern file.

Before turning checkpoint functionality on, you must first issue the Setup Checkpoint command. This command's usage is as follows:

SETUp CHeckpoint ***filename*** [*period*] [-Replace] [-Overwrite | -Sequence]
[-Faultlist *fault_file*]

To turn the checkpoint functionality on or off, use the Set Checkpoint command. This command's usage is as follows:

SET CHeckpoint **OFf** | **ON**

You must specify a filename in which to write the patterns (FastScan only). You can optionally specify the minutes of the checkpoint period, after which time the tool writes the patterns. You can replace or overwrite the file. Additionally, you could specify to write a sequence of separate pattern files--one for each checkpoint period. The
-Faultlist *fault_file* option also enables you to save a fault list.

Performing a Default ATPG Run

You execute the ATPG process by using the Run command while in the ATPG system mode, as follows:

```
ATPG> run
```

If the first ATPG run provides inadequate coverage, refer to "Approaches for Improving ATPG Efficiency" on page 6-78. To analyze the results of a failed run, use the Analyze Atpg Constraints command and the Analyze Restrictions command (FastScan Only).

Compressing Patterns

Because a tester requires a relatively long time to apply each scan pattern, it is important to create as small a test pattern set as possible while still maintaining the same test coverage. Static pattern compression minimizes the number of test patterns in a generated set.

Many patterns generated early on in the pattern set may no longer be necessary because later patterns also detect the faults detected by these earlier patterns. Thus, you can compress the pattern set by rerunning fault simulation on the same

patterns, first in reverse order and then in random order, keeping only those patterns necessary for fault detection. This method normally reduces the original test pattern set by 30 to 40 percent with very little effort.

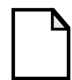
To compress test patterns, you use the Compress Patterns command. This commands usage is as follows:

COMpress PAtterns [*passes_integer*] [-Reset_au] [-MAx_useless_passes *integer*] [-MIn_elim_per_pass *number*]

Or, if you are using the graphical user interface, you can select the COMPRESS PATTERNS palette menu item.

- The integer option lets you specify how many compression passes the fault simulator should make. If you do not specify any number, it performs only one compression pass.
- The -MAx_useless_passes option lets you specify a maximum number of passes with no pattern elimination before the tool stops compression.
- The -MIn_elim_per_pass option lets you constrain the compression process by specifying that the tool stop compression when a single pass does not eliminate a minimum number of patterns.
- (FastScan only) The -Reset_au switch tells the simulator to simulate AU faults and if they are possible detected during the pattern, to label them as possible-detected ATPG untestable (PU) and to give them the appropriate test coverage credit. If the number of passes is greater than one, the tool resets AU faults for the first pass only.

For FastScan users, if after pattern compression the pattern set remains unacceptably large, you should run the entire ATPG process again with ATPG pattern compression turned on (see page 6-83). You can then use Compress Patterns in the normal manner to compress this new pattern set.


Note

The tools only perform pattern compression on independent test blocks; that is, for patterns generated for combinational or scan designs. Thus, FlexTest first does some checking of the test set to determine whether it can implement pattern compression.

Automatic ATPG Compression

FastScan supports the ability to execute good ATPG compression by combining the following sequence of events into one executable command.

CREate PAtterns [**-Compact**]

The Create Patterns command combines the following events:

- Error checking - where the Check Pattern Source is internal and Check Sequential ATPG is disabled.
- Add Faults -All (if no faults have been added).
- Turns off ATPG compression and turns on random patterns.
- Performs ATPG without saving patterns.
- Performs a Reset State.
- Turns on ATPG compression, turns off random patterns, and executes Set Decision Order -Random
- Performs ATPG saving patterns.
- Performs Compress Patterns with 1 compression pass.

Approaches for Improving ATPG Efficiency

If you are not satisfied with the test coverage after initially running ATPG or if the resulting pattern set is unacceptably large, you can make adjustments to several system defaults to improve results in another ATPG run. The following subsections provide helpful information and strategies for obtaining better results during the ATPG run.

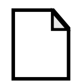
Understanding the Reasons for Low Test Coverage

There are two basic reasons for low test coverage: 1) constraints on the tool, and 2) abort conditions. A high number of faults in the AU or PU fault categories

indicates the problem lies with tool constraints. A high number of faults in the UO or UC categories indicates the problem lies with abort conditions. If you are unfamiliar with these fault categories, refer to "Fault Classes" on page 2-32.

When trying to establish the cause of low test coverage, you should examine the messages the tool prints during the deterministic test generation phase. These messages can alert you to what might be wrong with respect to redundant faults, ATPG untestable faults, and aborts. If you do not like the progress of the run, you can terminate the process with CTRL-C.

If a high number of aborted faults appears to cause the problem, you can set the abort limit to a higher number, or you can modify some command defaults to change the way the application makes decisions. The following subsections discuss several ways to handle aborted faults.



Note

changing the abort limit is not always a viable solution for a low coverage problem. The tool cannot detect ATPG untestable faults, the most common cause of low test coverage, even with an increased abort limit. Sometimes you may need to analyze why a fault, or set of faults, remain undetected to understand what you can do.

Also, if you have defined several ATPG constraints or have specified Set Contention Check On -Atpg, the tool may not abort because of the fault, but because it cannot satisfy the required conditions. In either of these cases, you should analyze the buses or ATPG constraints to ensure the tool *can* satisfy the specified requirements.

Analyzing a Specific Fault

You can report on all faults in a specific fault category with the Report Faults command. You can analyze each fault individually, using the pin pathnames and types listed by Report Faults, with the Analyze Fault command. This command's usage is as follows:

ANAlyze FAult ***pin_pathname*** {**-Stuck_at** {**0** | **1**}} [-Observe *gate_id#*] [-Boundary] [-Auto] [-Continue] [-Display]

This command runs ATPG on the specified fault, displaying information about the processing and the end results. The application displays different data depending on the circumstances. You can optionally display relevant circuitry in the DFTInsight schematic viewer using the -display option. Refer to the Analyze Fault command in the *FastScan and FlexTest Reference Manual* for more information. You can also report data from the ATPG run using the Report Testability Data command within FastScan or FlexTest, for a specific category of faults. This command displays information about connectivity surrounding the problem areas. This information can give you some ideas as to where the problem might lie, such as with RAM or clock PO circuitry. Refer to the Report Testability Data command in the *FastScan and FlexTest Reference Manual* for more information.

Reporting on ATPG Untestable Faults (FlexTest Only)

FlexTest has the capability to report the reasons why a fault is classified as ATPG untestable. This fault category includes AU, UI, PU, OU, and HU faults. For more information on this fault category, refer to "Fault Classes" on page 2-32. You can determine why these faults are undetected by using the Report AU Faults command. This command's usage is as follows:

REPort AU FAults [Summary | All | TRistate | TIed_constraint |
Blocked_constraint | Uninitialized | Clock | Wire | Others]

Fore more information on this command, refer to the Report AU Faults command page in the *FastScan and FlexTest Reference Manual*.

Reporting on Aborted Faults

During the ATPG process, FastScan or FlexTest may abort attempts to detect certain faults given the ATPG effort required. The tools place these types of faults, called *aborted faults*, in the undetected fault category. You can determine why these faults are undetected by using the Report Aborted Faults command. This command's usage is as follows:

REPort ABorted Faults [*format_type*]

The format type you specify gives you the flexibility to report on different types of aborted faults. The format types vary between FastScan and FlexTest. Refer to

the Report Aborted Faults command reference page in the *FastScan and FlexTest Reference Manual* for more information.

Setting the Abort Limit

If the fault list contains a number of aborted faults, the tools may be able to detect these faults if you change the abort limit. You can increase the abort limit for the number of backtracks, test cycles, or CPU time and rerun the ATPG process. To set the abort limit using FastScan, you use the Set Abort Limit command. This command's usage is as follows:

SET ABort Limit [*comb_abort_limit* [*seq_abort_limit*]]

The comb_abort_limit and seq_abort_limit arguments specify the number of conflicts allowed for each fault during the combinational and clock_sequential ATPG processes, respectively. The default for combinational ATPG is 30. The clock sequential abort limit defaults to the limit set for combinational. Both the Report Environment command and a message at the start of deterministic test generation indicate the combinational and sequential abort limits. The sequential limit follows the combinational abort limit, if they differ.

The Set Abort Limit command for FlexTest has the following usage:

SET ABort Limit [-Backtrack *integer*] [-Cycle *integer*] [-Time *integer*]

The initial defaults are 30 backtracks, 300 test cycles, and 300 seconds per target fault. If your fault coverage is too low, you may want to re-issue this command using a larger integer with the -Backtrack switch. A reasonable choice for a second pass would be 500. Use caution however, because if the numbers you specify are too high, test generation may take a long time to complete.

The application classifies any faults that remain undetected after reaching the limits as *aborted faults*--which it considers undetected faults.

Related Commands:

Report Aborted Faults - displays and identifies the cause of aborted faults.

Setting Random Pattern Usage

FastScan and FlexTest also let you specify whether to use random test generation processes to create patterns during ATPG (when the selected pattern source is internal). In general, the test generation process runs faster and the number of test patterns in the set is longer if you use random patterns. If not specified, the default is to use random patterns in addition to deterministic patterns. If you use random patterns exclusively, test coverage is typically very low. To set random pattern usage for the ATPG, you use the Set Random Atpg command, whose usage is as follows:

SET RAndom Atpg **ON** | **OFf**

Changing the Decision Order (FastScan Only)

Prior to ATPG, FastScan learns which inputs of multiple input gates it can most easily control. It then orders these inputs from easiest to most difficult to control. Likewise, FastScan learns which outputs can most easily observe a fault and orders these in a similar manner.Then during ATPG, the tool uses this information to generate patterns in the simplest way possible.

This facilitates the ATPG process, however it minimizes random pattern detection. This is not always desirable, as you typically want generated patterns to randomly detect as many faults as possible.To maximize random pattern detection, FastScan provides the Set Decision Order command to allow flexible selection of control inputs and observe outputs during pattern generation. Usage for the Set Decision Order command is as follows:

SET DEcision Order {{**-NORandom** | **-Random**} | {**-NOSIngle_observe** | **-SIngle_observe**} | {**-NOClock_equivalence** | **-Clock_equivalence**}}

The -Random switch specifies random order for selecting inputs of multiple input gates. The -Single_observe switch constrains ATPG to select a single observe point for a generated pattern. The -Clock_equivalence switch constrains ATPG to select a single observe point for the set of latches clocked by equivalent clocks.

This command works in conjunction with Set Atpg Compression to provide efficient ATPG compression runs. For more information, refer to the following section, "Dynamically Compressing Patterns (FastScan Only)".

Dynamically Compressing Patterns (FastScan Only)

You should utilize this feature only if your test size is still too large even after static pattern compression. If you have achieved your desired test coverage, but you desire a more compact test pattern set, you can turn on ATPG (dynamic) compression and re-run ATPG.

During ATPG with compression turned on, FastScan selects a target fault, determines the pattern conditions necessary to detect that fault, and then attempts to merge detection of a large number of additional faults with the same pattern. This type of ATPG process generates single patterns to detect a multitude of faults, which results in very compact test sets.

Helpful Hint: To minimize the runtime of a dynamic compression ATPG process, you should first perform a default run, and then use the Reset State command. This technique results in a quick ATPG process that classifies all the faults, resets all the detected faults to undetected—except for the AU faults, which remain classified as AU—and deletes the generated internal pattern set. You can then perform a dynamic compression ATPG run on the remaining (undetected) faults. Because large numbers of AU faults can hinder the performance of the dynamic compression process, screening them out prior to the run improves the run's efficiency.

To set the ATPG compression usage, use the Set Atpg Compression command as follows:

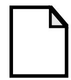
SET ATpg Compression [OFf | ON] [-Limit *number*] [-NOVerbose | -Verbose]
 [-Abort_limit *number*] [-CONsecutive_fails *number*]
 [-SEq_merge_limit *number*]

By default, ATPG compression is off, so you must issue this command and specify the ON option to utilize this feature. The -Limit switch, which by default is 200 and is used by the combinational compression algorithm only, sets the number of faults FastScan allows to fail to merge with the target fault for each generated pattern. The -Verbose option indicates compression results on a per pattern basis. The -Noverbose setting is the default, but if you want to obtain useful information about the progress of the ATPG run with dynamic compression turned on, you should use the -Verbose option. The -Abort_limit switch, which by

default is set to 10, indicates the number of conflicts allowed for subsequent merged faults for the same pattern.

The -Consecutive_fails switch, which by default is 40 and is used by the sequential compression algorithm only, specifies the maximum number of *consecutive* faults that the test pattern generator will unsuccessfully attempt to merge with the target fault pattern.

The -Seq_merge_limit switch, which by default is 5000 and is used by the sequential compressional algorithm only, specifies the maximum number of faults that the test pattern generator will *successfully* attempt to merge with the target fault pattern.


Note

Turning ATPG pattern compression on with default settings can result in the ATPG process taking 2-3 times longer than usual. Thus, you should only use this feature if your original pattern set is unacceptably large, or when you are running the final pass to produce actual production vectors. For most efficient operation, you should use this command in conjunction with Set Decision Order.

Related Commands:

Set AU Analysis - specifies whether the ATPG untestable information can be used during the ATPG process.
Set Decision Order - specifies how FastScan selects which inputs of multiple gates to control when building patterns.

Saving the Test Patterns

To save generated test patterns, at the Atpg mode prompt, enter the Save Patterns command using the following syntax:

For FastScan:

SAVe PAtterns ***filename*** [-Replace] [*format_switch*]
[*timing_filename* | *proc_filename*] [-Parallel | -Serial] [-EXternal]
[-NOInitialization] [-BEgin {*pattern_number* | *pattern_name*}]

[-END {*pattern_number* | *pattern_name*}] [-TAg *tag_name*]
[-CEll_placement {Bottom | Top | None}] [-ENVironment] [-One_setup]
[-ALl_test | -CHain_test | -SCan_test] [-NOPadding | -PAD0 | -PAD1] [-Noz]
[-Map *mapping_file*] [-TIMingfile | -PROcfile] [-PATtern_size *integer*]

For FlexTest:

SAVe PAtterns ***filename*** [-Replace] [*format_switch*]
[*timing_filename* | *proc_filename*] [-Parallel | -Serial] [-EXternal]
[-NOInitialization] [-BEgin *begin_number*] [-END *end_number*]
[-CEll_placement {Bottom | Top | None}] [-One_setup]
[-ALl_test | -CHain_test | -CYcle_test] [-NOPadding | -PAD0 | -PAD1] [-Noz]
[-TIMingfile | -PROcfile] [-PAttern_size *integer*]

You save patterns to a filename using one of the following format switches: -Ascii, -BInary, -Compass, -Fjtdl, -Mgcwdb, -MItdl, -Lsim, -TItdl, -TSsiwgl, -TSTl2, -Utic, -Verilog, -VHdl or -Zycad. For information on the remaining command options, refer to the Save Patterns in the *FastScan and FlexTest Reference Manual*. For more information on the test data formats, refer to "Saving the Patterns" on page 7-24.

Creating an IDDQ Test Set

FastScan and FlexTest support the pseudo stuck-at fault model for IDDQ testing. This fault model allows detection of most of the common defects in CMOS circuits (such as resistive shorts) without costly transistor level modeling. "IDDQ Test" on page 2-20 introduced IDDQ testing.

Additionally, FastScan and FlexTest support both selective and supplemental IDDQ test generation. The tool creates a selective IDDQ test set when it selects a set of IDDQ patterns from a pre-existing set of patterns originally generated for some purpose other than IDDQ test. The tool creates a supplemental IDDQ test set when it generates an original set of IDDQ patterns based on the pseudo stuck-at fault model. Before running either the supplemental or selective IDDQ process, you must first set the fault type to IDDQ using Set Fault Type.

Using FastScan and FlexTest, you can either select or generate IDDQ patterns using several user-specified checks. These checks can help ensure that the IDDQ

test vectors do not increase IDDQ in the good circuit. The following subsections describe IDDQ pattern selection, test generation, and user-specified checks in more detail.

Creating a Selective IDDQ Test Set

The following subsections discuss basic information concerning selecting IDDQ patterns from an existing set and provide an example of a typical IDDQ pattern selection run.

Setting the External Pattern Set

In order to create a selective IDDQ test set, you must have an existing set of test patterns. These patterns must reside in an external file and you must change the pattern source so the tool works from this external file. You specify the external pattern source using the Set Pattern Source command. This external file must be in one of the following formats: FastScan Text, FlexTest Text, or FastScan Binary.

Determining When to Perform the Measures

The pre-existing external test set may or may not target IDDQ faults. For example, you can run ATPG using the stuck-at fault type and then select patterns from this set for IDDQ testing. If the pattern set does not target IDDQ faults, it will not contain statements that specify IDDQ measurements. IDDQ test patterns must contain statements that tell the tester to make an IDDQ measure. In FastScan or FlexTest Text formats, this IDDQ measure statement, or *label,* appears as follows:

```
measure IDDQ ALL <time>;
```

By default, FastScan and FlexTest place these statements at the end of patterns (cycles) that can contain IDDQ measurements. You can manually add these statements to patterns (cycles) within the external pattern set.

When you want to select patterns from an external set, you must specify which patterns can contain an IDDQ measurement. If the pattern set contains no IDDQ measure statements, you can specify that the tools assume the tester can make a measurement at the end of each pattern or cycle. If the pattern set already contains IDDQ measure statements (if you manually added these statements), you can

specify that simulation should only occur for those patterns that already contain an IDDQ measure statement, or label. You set this measurement information using the Set Iddq Strobes command.

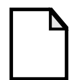
Selecting the Best IDDQ Patterns

Typically, ASIC vendors have restrictions on the number of IDDQ measurements they allow. The expensive nature of IDDQ measurements typically restricts a test set to a small number of patterns with IDDQ measure statements.

Additionally, you can set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. "Specifying IDDQ Checks and Constraints" on page 6-90 discusses these IDDQ restrictions. You specify the IDDQ pattern selection criteria and run the selection process using Select Iddq Patterns. This command's usage is as follows:

SELect IDdq Patterns [-Max_measures *number*] [-Threshold *number*] [-Eliminate | -Noeliminate]

The Select Iddq Patterns command fault simulates the current pattern source and determines the IDDQ patterns that best meet the selection criteria you specify, thus creating an IDDQ test pattern set. If working from an external pattern source, it reads the external patterns into the internal pattern set, and places IDDQ measure statements within the selected patterns or cycles of this test set based on the specified selection criteria.


Note

FlexTest supplies some additional arguments for this command. Refer to Select Iddq Patterns in the *FastScan and FlexTest Reference Manual* for details.

Selective IDDQ Example

The following list demonstrates a common situation in which you could select IDDQ test patterns using FastScan or FlexTest.

1. Invoke FastScan or FlexTest on the design, set up the appropriate parameters for ATPG run, pass rules checking, and enter the ATPG mode.

```
...
SETUP> set system mode atpg
```

This example assumes you set the fault type to stuck-at, or some fault type other than IDDQ.

2. Run ATPG.

   ATPG> **run**

3. Save generated test set to external file named *orig.pats*.

   ATPG> **save patterns orig.pats**

4. Change pattern source to the saved external file.

   ATPG> **set pattern source external orig.pats**

5. Set the fault type to IDDQ.

   ATPG> **set fault type iddq**

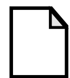
6. Add all IDDQ faults to the current fault list.

   ATPG> **add faults -all**

7. Assume IDDQ measurements can occur within each pattern or cycle in the external pattern set.

   ATPG> **set iddq strobe -all**

8. Specify to select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

   
   Note
   You could use the Add Iddq Constraints or Set Iddq Checks commands prior to the ATPG run to place restrictions on the selected patterns.

   ATPG> **select iddq patterns -max_measure 15 -threshold 10**

9. Save these IDDQ patterns into a file.

   ATPG> **save patterns iddq.pats**

Generating a Supplemental IDDQ Test Set

The following subsections discuss the basic IDDQ pattern generation process and provide an example of a typical IDDQ pattern generation run.

Generating the Patterns

Prior to pattern generation, you may want to set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. "Specifying IDDQ Checks and Constraints" on page 6-90 discusses these IDDQ restrictions. As with any other fault type, you issue the Run command within ATPG mode. This generates an internal pattern set targeting the IDDQ faults in the current list. If you are using FastScan, you can turn dynamic pattern compression on with the Set Atpg Compression On command, targeting multiple faults with a single pattern and resulting in a more compact test set.

Selecting the Best IDDQ Patterns

Issuing the Run command results in an internal IDDQ pattern set. Each pattern generated automatically contains a "measure IDDQ ALL" statement, or label. Thus, if you use FastScan or FlexTest to generate the IDDQ patterns, you do *not* need to use the Set Iddq Strobes command, because by default the tools only simulate IDDQ measures at each label.

The generated IDDQ pattern set may contain more patterns than you want for IDDQ testing. Thus, at this point, you just set up the IDDQ pattern selection criteria and run the selection process using Select Iddq Patterns.

Supplemental IDDQ Example

1. Invoke FastScan or FlexTest on design, set up appropriate parameters for ATPG run, pass rules checking, and enter ATPG mode.

   ...
   SETUP> **set system mode atpg**

2. Set the fault type to IDDQ.

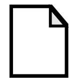
   ATPG> **set fault type iddq**

3. Add all IDDQ faults to the current fault list.

   ATPG> **add faults -all**

   Instead of creating a new fault list, you could load a previously-saved fault list. For example, you could write the undetected faults from a previous ATPG run and load them into the current session with Load Faults, using them as the basis for the IDDQ ATPG run.

4. Run ATPG, generating patterns that target the IDDQ faults in the current fault list.

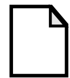

Note

You could use the Add Iddq Constraints or Set Iddq Checks commands prior to the ATPG run to place restrictions on the generated patterns.

   ATPG> **run**

5. Select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

   ATPG> **select iddq patterns -max_measure 15 -threshold 10**


Note

You did not need to specify which patterns could contain IDDQ measures with Set Iddq Strobes, as the generated internal pattern source already contains the appropriate measure statements.

6. Save these IDDQ patterns into a file.

   ATPG> **save patterns iddq.pats**

Specifying IDDQ Checks and Constraints

Because IDDQ testing uses current measurements for fault detection, you may want to ensure the patterns selected for the IDDQ test set do not produce high current measures in the good circuit. FastScan and FlexTest let you set up special IDDQ current checks and constraints to ensure careful IDDQ pattern generation or selection.

Related Commands:

> **Delete Iddq Constraints** - deletes internal and external pin constraints during IDDQ measurement.
> **Report Iddq Constraints** - reports internal and external pin constraints during IDDQ measurement.

Specifying Leakage Current Checks

For CMOS circuits with pull-up or pull-down resistors or tri-state buffers, the good circuit should have a nearly zero IDDQ current. FastScan and FlexTest allow you to specify various IDDQ measurement checks to ensure that the good circuit does not raise IDDQ current during the measurement.

The Set Iddq Checks command usage is:

SET IDdq Checks [-NONe | -ALl | {-Bus | -WEakbus | -Int_float | -EXt_float | -Pull | -Clock | -WRite | -REad | -WIre | -WEAKHigh | -WEAKLow | -VOLTGain | -VOLTLoss}...] [-WArning | -ERror] [-NOAtpg | -ATpg]

By default, neither tool performs IDDQ checks. Both ATPG and fault simulation processes consider the checks you specify. Refer to the Set Iddq Checks reference page in the *FastScan and FlexTest Reference Manual* for details on the various capabilities of this command.

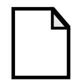
Preventing High IDDQ Current in the Good Circuit

CMOS models can have some states for which they draw a quiescent current. Some I/O pads that have internal pull-ups or pull-downs normally draw a quiescent current. You may be able to disable these pull-ups or pull-downs from another input pin during IDDQ testing. You can also specify pin constraints, if the pin is an external pin, or cell constraints, if the net connects to a scan cell. Constrained pins or cells retain the state you specify (that which produces low IDDQ current in the good circuit) only during IDDQ measurement.

With the following command, you can force a set of internal pins to a specific state during IDDQ measurement to prevent high IDDQ:

ADD IDdq Constraints {**C0** | **C1** | **CZ**} ***pinname...*** [-Model *modelname*]

The repeatable pinname argument lets you specify the constraint on multiple pins. The -Model option determines the meaning of the *pinname* argument. If you specify the -Model option, the tool assumes that *pinname* represents a library model pin, for which all instances of this model will constrain the specified pin. Otherwise, the tool assumes *pinname* represents any pin in the hierarchical netlist.



This command is similar to the Add Atpg Constraints command. However, ATPG constraints specify pin states for all ATPG generated test cycles, while IDDQ constraints specify values that pins must have only during IDDQ measurement. You can change both during ATPG or fault simulation to achieve higher coverage.

Creating a Path Delay Test Set (FastScan)

FastScan can generate patterns to detect path delay faults. These patterns determine if specific paths operate correctly at-speed. "At-Speed Testing and the Path Delay Fault Model" on page 2-29 introduced the path delay fault model.

Path Delay Fault Detection

Path delay testing requires an edge, which implies two events need to occur to detect a fault. These events include a launch event and a capture event. Figure 6-13 depicts the launch and capture events of a small circuit during path delay testing.

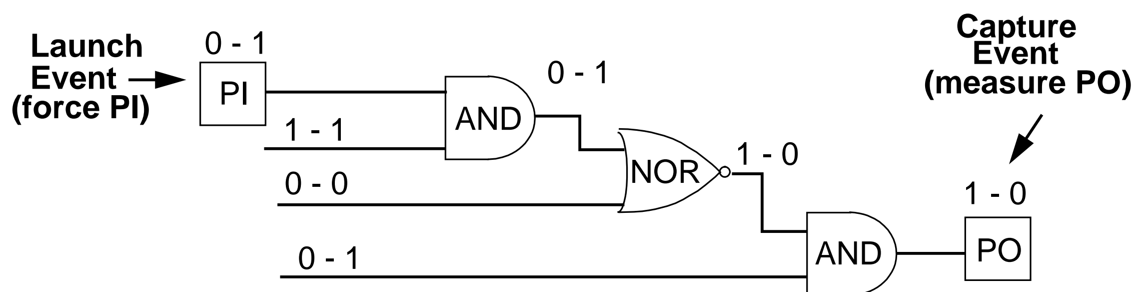
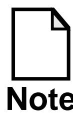


Figure 6-13. Path Delay Launch and Capture Events

Path delay patterns are a variant of clock-sequential patterns. A typical FastScan pattern to detect a path delay fault includes the following events:

1. Load scan chains
2. Force primary inputs
3. Pulse clock (to create a launch event for a launch point that is a state element)
4. Force primary inputs (to create a launch event for a launch point that is a primary input)
5. Measure primary outputs (to create a capture event for a capture point that is a primary output)
6. Pulse clock (to create a capture event for a capture point that is a state element)
7. Unload scan chains

The additional force_pi/pulse_clock cycles may occur before or after the launch or capture events. The cycles depend on the sequential depth required to set the launch conditions or sensitize the captured value to an observe point.



Path delay testing often requires greater depth than for stuck-at fault testing. The sequential depths that FastScan calculates and reports are the minimums for stuck-at testing.

To get maximum benefit from path delay testing, the launch and capture events must have accurate timing. The timing for all other events is not critical.

FastScan detects a path delay fault with either a *robust test* or a *transition test*. Robust detection occurs when the gating inputs used to sensitize the path are stable from the time of the launch event to the time of the capture event. Robust detection keeps the gating of the path constant during fault detection and thus, does not affect the path timing. Because it avoids any possible reconvergent timing effects, it is the most desirable type of detection. However, FastScan cannot use robust detection on many paths because of its restrictive nature. The

application places faults detected by robust detection in the DR (detected_robust) fault class.

Figure 6-14 gives an example of robust detection for a rising-edge transition within a simple path. Notice that, due to the circuitry, the gating value at the second OR gate was able to retain the proper value for detection during the entire time from launch to capture events.

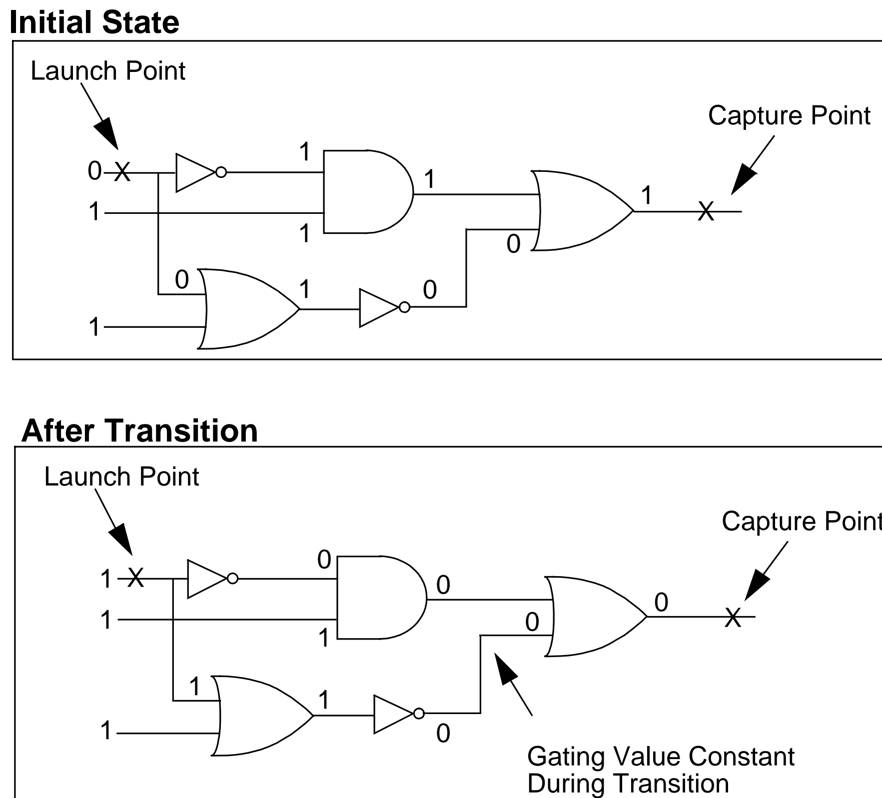


Figure 6-14. Robust Detection Example

Transition detection does not require constant values on the gating inputs used to sensitize the path. It only requires the proper gating values at the time of the capture event. FastScan places faults detected by transition detection in the DS (detected_simulation) fault class.

Figure 6-15 gives an example of transition detection for a rising-edge transition within a simple path.

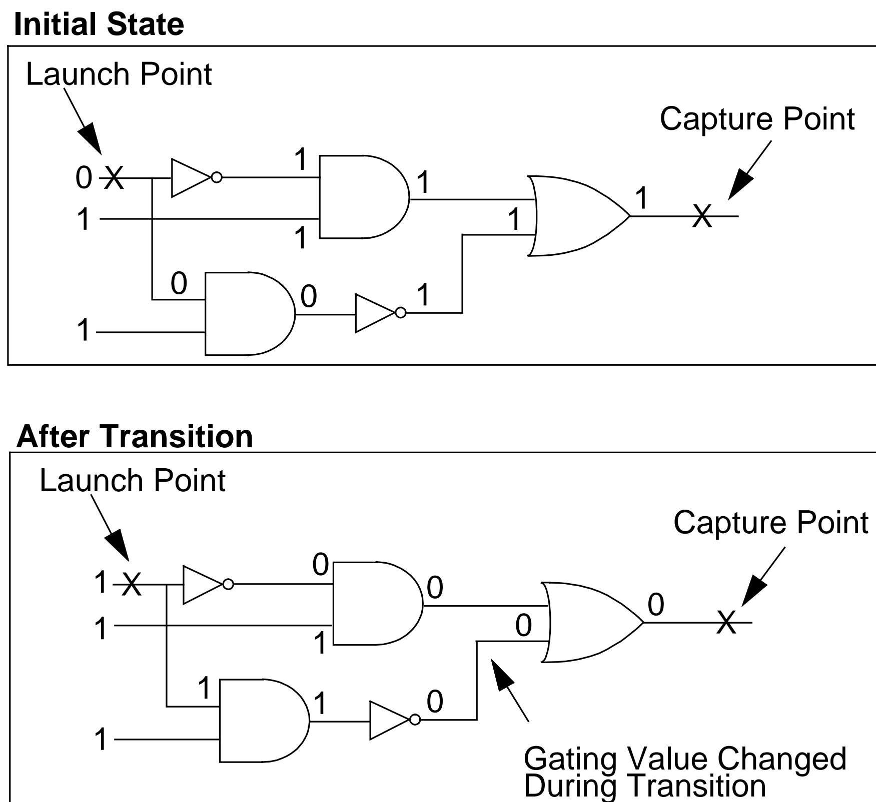


Figure 6-15. Transition Detection Example

Notice that due to the circuitry, the gating value on the second OR gate changed during the 0 to 1 transition placed at the launch point. Thus, the proper gating value was only at the OR gate at the capture event.

Related Commands:

Add Ambiguous Paths - specifies the number of paths FastScan should select when encountering an ambiguous path.
Analyze Fault - analyzes a fault, including path delay faults, to determine why it was not detected.
Delete Paths - deletes paths from the internal path list.
Load Paths - loads in a file of path definitions from an external file.
Report Paths - reports information on paths in the path list.
Set Pathdelay Holdpi - sets whether non-clock primary inputs can change after the first pattern force, during ATPG.
Write Paths - writes information on paths in the path list to an external file.

The Path Definition File

In an external ASCII file, you must define all paths that you want tested in the test set. For each path, you must specify:

- Path_name - a unique name you define to identify the path.
- Path_definition - the topology of the path from launch to capture point as defined by an ordered list of pin pathnames. Each path must be unique.

The ASCII path definition file has several syntax requirements. The tools ignore as a comment any line that begins with a double slash (//) or pound sign (#). Each statement must be on its own line. The four types of statements include:

- **Path** - A required statement that specifies the unique pathname of a path.
- **Condition** - An optional statement that specifies any conditions necessary for the launch and capture events. Each **condition** statement contains two arguments: a full pin pathname for either an internal or external pin, and a value for that pin. **Condition** statements must occur before the first pin statement.
- **Pin** - A required statement that identifies a pin in the path by its full pin pathname. **Pin** statements must be ordered from launch point to capture point. A “+” or “-” after the pin pathname indicates the inversion of the pin with respect to the launch point. A “+” indicates no inversion, while a “-” indicates inversion.

  You must specify a minimum of two **pin** statements, the first being a valid launch point (primary input, data input of a state element, or combinational pin) and the last being a valid capture point (primary output, data or clk input of a state element, or combinational pin). The current pin must have a combinational connectivity path to the previous pin and the edge parity must be consistent with the path circuitry. If a statement violates either of these conditions, the tool issues an error. If the path has edge or path ambiguity, it issues a warning.

Paths can include state elements (through data or clock inputs), but you must explicitly name the data or clock pins in the path. If you do not, FastScan does not recognize the path and issues a corresponding message.

- **End** - A required statement that signals the completion of data for the current path. Optionally, following the **end** statement you can specify the name of the path. However, if the name does not match the pathname specified with the **path** statement, the tool issues an error.

The following shows the path definition syntax:

```
PATH <pathname> =
   CONDition <pin_pathname> <0|1|Z>;
   PIN <pin_pathname> [+|-];
   PIN <pin_pathname> [+|-];
    . . .
   PIN <pin_pathname> [+|-];
END [pathname];
```

The following is an example of a path definition file:

```
PATH "path0" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$951/I$1/IN - ;
    PIN /I$1/I$951/I$1/OUT + ;
    PIN /A_EQ_B + ;
END ;
PATH "path1" =
    PIN /I$6/Q + ;
    PIN /I$35/B0 + ;
    PIN /I$35/C0 + ;
    PIN /I$1/I$650/IN + ;
    PIN /I$1/I$650/OUT - ;
    PIN /I$1/I$684/I1 - ;
    PIN /I$1/I$684/OUT - ;
    PIN /I$5/D - ;
END ;
```

```
PATH "path2" =
    PIN /I$5/Q + ;
    PIN /I$35/B1 + ;
    PIN /I$35/C1 + ;
    PIN /I$1/I$649/IN + ;
    PIN /I$1/I$649/OUT - ;
    PIN /I$1/I$622/I2 - ;
    PIN /I$1/I$622/OUT - ;
    PIN /A_EQ_B + ;
END ;
PATH "path3" =
    PIN /I$5/QB + ;
    PIN /I$6/TI + ;
END ;
```

You use the Load Paths command to read in the path definition file. The tool loads the paths from this file into an internal path list. You can add to this list by adding paths to a new file and re-issuing the Load Paths command with the new filename.

Path Definition Checking

FastScan checks the points along the defined path for proper connectivity and to determine if the path is ambiguous. *Path ambiguity* indicates there are several different paths from one defined point to the next. Figure 6-16 indicates a path definition that creates ambiguity.

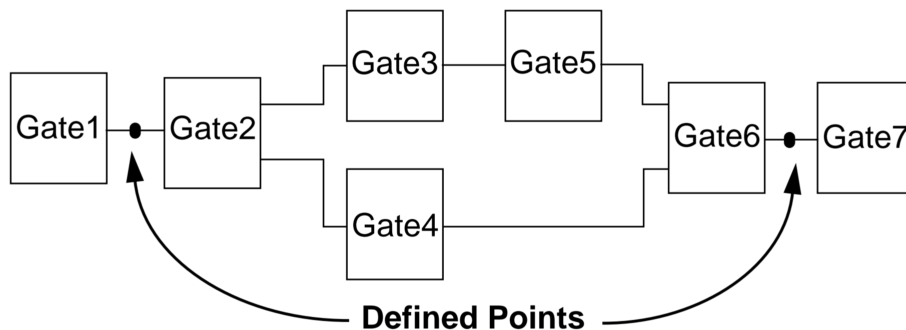


Figure 6-16. Example of Ambiguous Path Definition

In this example, the defined points are an input of Gate2 and an input of Gate7. Two paths exist between these points, thus creating path ambiguity. When FastScan encounters this situation, it issues a warning message and selects a path,

typically the first fanout of the ambiguity. If you want FastScan to select more than one path, you can specify this with the Add Ambiguous Path command.

During path checking, FastScan can also encounter *edge ambiguity*. Edge ambiguity occurs when a gate along the path has the ability to either keep or invert the path edge, depending on the value of another input of the gate. Figure 6-17 shows a path with edge ambiguity.

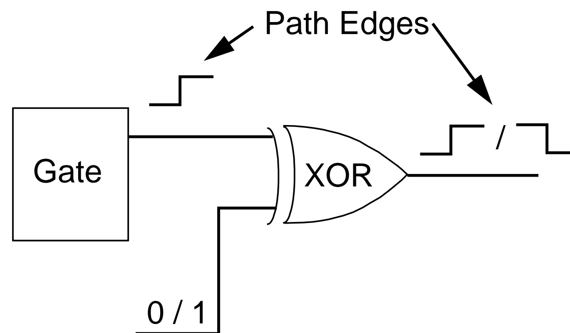


Figure 6-17. Example of Ambiguous Path Edges

The XOR gate in this path can act as an inverter or buffer of the input path edge, depending on the value at its other input. Thus, the edge at the output of the XOR is ambiguous. The path definition file lets you indicate edge relationships of the defined points in the path. You do this by specifying a "+" or "-" for each defined point, as was described previously in "The Path Definition File" on page 6-96.

Basic Path Delay Test Procedure

The basic procedure for generating a path delay test set is as follows:

1. Perform circuit setup, rules checking, and entry into Atpg mode.

2. Set the fault type to path delay:

   ATPG> **set fault type path_delay**

3. Set sequential depth to two or greater:

   ATPG> **set simulation mode combination -depth 2**

4. Write a path definition file with all the paths you want tested. You can do this prior to the session if you wish. You can only add faults based on the paths defined in this file.

5. Load the path definition file (path_file_1):

   ATPG> **load path path_file_1**

6. Specify ambiguous path selection limits (in this case 4), if desired.

   ATPG> **add ambiguous paths -all -max_paths 4**

7. Add faults to the fault list:

   ATPG> **add faults -all**

   This adds a rising edge and falling edge fault associated with each defined path.

8. Run test generation:

   ATPG> **run**

Path Delay Testing Limitations

Path delay testing does not support the following:

- RAMs within a specified path
- Paths through sequentially transparent latches (FastScan supports combinationally transparent latches, but not as launch or capture points)
- Compression of path delay patterns

Creating a Transition Test Set

Similar to path delay fault detection, FastScan and FlexTest can generate patterns to detect transition faults. These patterns determine if specific pins operate correctly at-speed. “At-Speed Testing and the Transition Fault Model” on page 2-28 introduced the transition fault model.

Transition Fault Detection

To detect transition faults, two conditions must be met:

- The corresponding stuck_at fault must be detected.
- Within a single previous cycle, the fault must be at the opposite value than the value detected in the current cycle.

Figure 6-18 depicts the launch and capture events of a small circuit during transition testing. Transition faults can be detected on any pin.

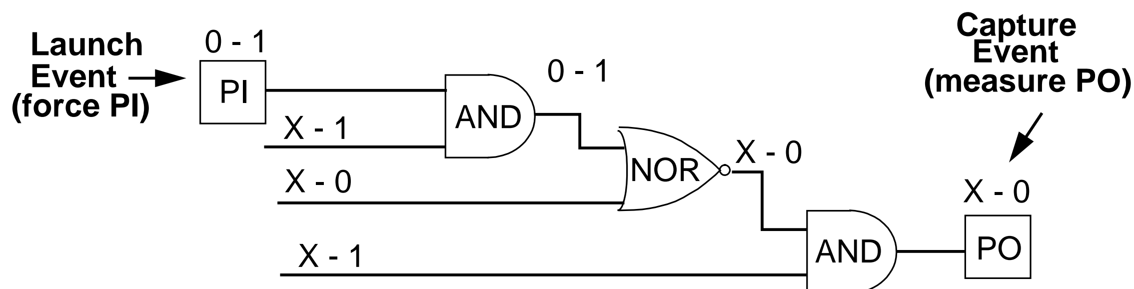


Figure 6-18. Transition Launch and Capture Events

To detect a transition fault, a typical FlexTest pattern includes the events in Figure 6-19.

Figure 6-19. Type One Pattern Set

1. Load scan chains
2. Force primary inputs
3. Pulse clock

Figure 6-19. Type One Pattern Set

4. Force primary inputs
5. Measure primary outputs
6. Pulse clock
7. Unload scan chains

In type one pattern sets, the transition occurs because of the two pulses of the clock. By default, FastScan always attempts to create a pattern set similar to the one produced by FlexTest. If it fails to do so, it produces a pattern set that includes the events show in Figure 6-20.

Figure 6-20. Type Two Pattern Set

1. Init_force primary inputs
2. Load scan chains
3. Force primary inputs
4. Measure primary outputs
5. Pulse clock
6. Unload scan chains

In the type two pattern set, the transition occurs because of the last shift in the load scan chains procedure (event #2) or the forcing of the primary inputs (event #3).

Random pattern simulation, in FastScan, always produces the type two patterns. When dynamic compression is on, FastScan always generates the type one patterns. In FastScan, to avoid creating type two patterns, set random ATPG off, set the combinational abort limit to 0, and set the sequential abort limit greater than 0.

Related Commands:

Set Abort Limit - specifies the abort limit for the test pattern generator.
Set Fault Type - specifies the fault model for which the tool develops or selects ATPG patterns.

Set Simulation Mode - specifies whether the ATPG simulation run uses combinational or sequential RAM test patterns.

Basic Transition Test Procedure

The basic procedure for generating a transition test set is as follows:

1. Perform circuit setup, rules checking, and entry into Atpg mode.
2. Set the fault type to transition:

   ATPG> **set fault type transition**

3. Set sequential depth to two or greater (optional, FastScan only):

   ATPG> **set simulation mode combination -depth 2**

4. Add faults to the fault list:

   ATPG> **add faults -all**

5. Run test generation:

   ATPG> **run**

Generating Patterns for a Boundary Scan Circuit

The following example shows how to create a test set for an 1EEE 1149.1 (boundary scan)-based circuit. The following subsections list and explain the FastScan dofile and test procedure file.

Dofile and Explanation

The following dofile shows the commands you could use to specify the scan data in FastScan:

```
add cl 0 tck
add sc g group1 proc_fscan
add sc ch chain1 group1 tdi tdo
add pin con tms c0
add pin con trstz c1
set capture cl TCK -ATPG
```

You must define the tck signal as a clock because it captures data. There is one scan group, group1, which uses the *proc_fscan* test procedure file (see page 6-106). There is one scan chain, chain1, that belongs to the scan group. The input and output of the scan chain are tdi and tdo, respectively.

The listed pin constraints only constrain the signals to the specified values during ATPG--not during the test procedures. Thus, the tool constrains tms to a 0 during ATPG (for proper pattern generation), but not within the test procedures, where the signal transitions the TAP controller state machine for testing. The basic scan testing process is:

1. Initialize scan chain.

2. Apply PI values.

3. Measure PO values.

4. Pulse capture clock.

5. Unload scan chain.

During Step 2, you must constrain tms to 0 so that the Tap controller's finite state machine (Figure 6-21) can go to the Shift-DR state when you pulse the capture clock (tck). You constrain the trstz signal to its off-state for the same reason. If you do not do this, the Tap controller goes to the Test-Logic-Reset state at the end of the Capture-DR sequence.

The Set Capture Clock TCK -ATPG command defines tck as the capture clock and that the capture clock must be utilized for each pattern (as FastScan is able to create patterns where the capture clock never gets pulsed). This ensures that the Capture-DR state properly transitions to the Shift-DR state.

TAP Controller State Machine

Figure 6-21 shows the finite state machine for the TAP controller of a IEEE 1149.1 circuit.

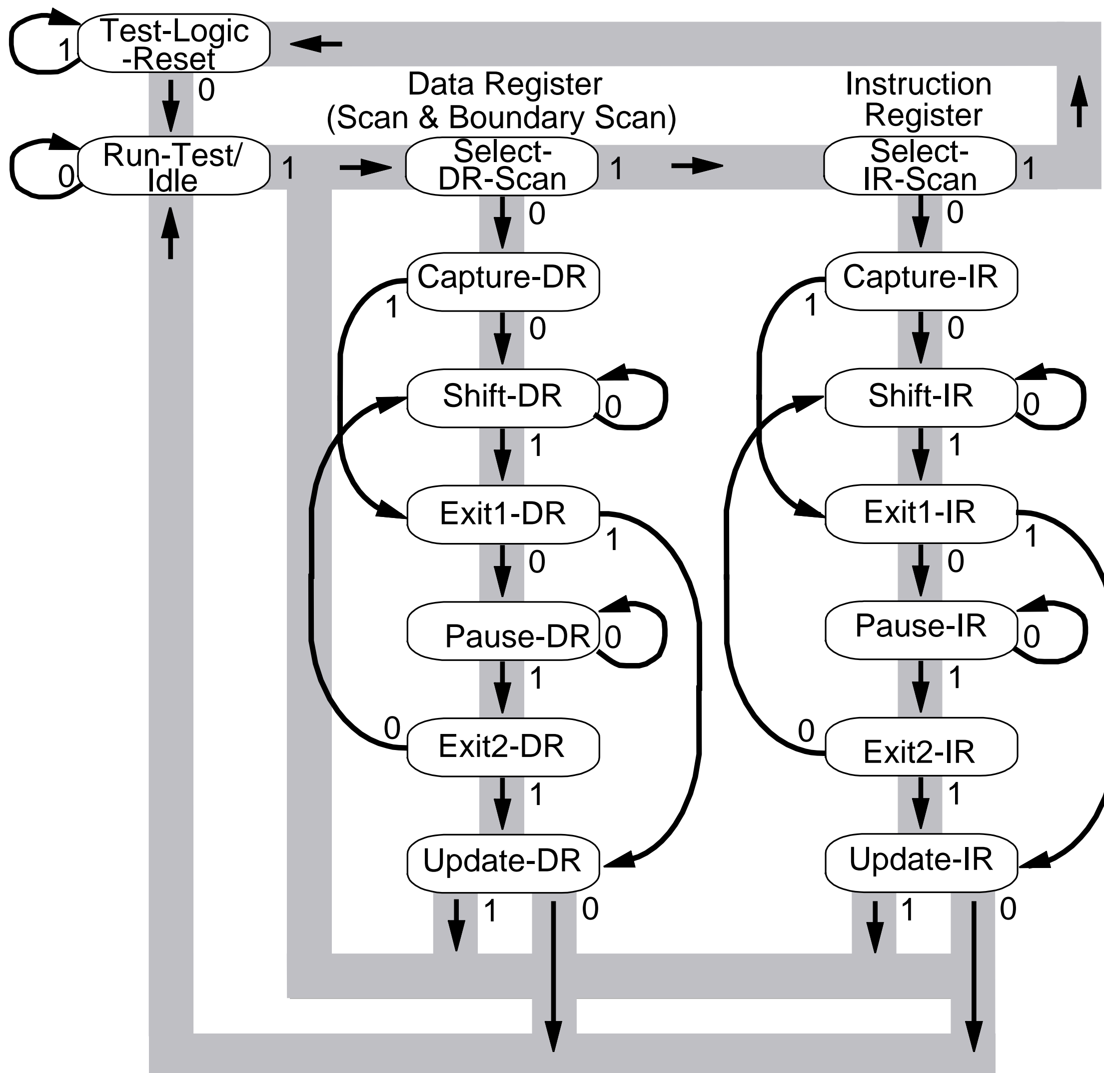


Figure 6-21. State Diagram of TAP Controller Circuitry

The TMS signal controls the state transitions. The rising edge of the TCK clock captures the TAP controller inputs. You may find this diagram useful when writing your own test procedure file or trying to understand the example test procedure file that the next subsection shows.

Test Procedure File and Explanation

The test procedure file *proc_fscan* follows:

```
proc test_setup =

    //apply reset procedure
    //test cycle one

    force "TMS" 1 1;
    force "TDI" 0 1;
    force "TRST" 0 1;
    force "TCK" 1 3;
    force "TCK" 0 4;

   //"TMS"=0 change to run-test-idle
   //test cycle two

    force "TMS" 0 6;
    force "TRST" 1 6;
    force "TCK" 1 8;
    force "TCK" 0 9;

    //"TMS"=1 change to select-DR
    //test cycle three

    force "TMS" 1 11;
    force "TCK" 1 13;
    force "TCK" 0 14;

    //"TMS"=1 change to select-IR
    //test cycle four

    force "TMS" 1 16;
    force "TCK" 1 18;
    force "TCK" 0 19;
```

```
//"TMS"=0 change to capture-IR
//test cycle five

force "TMS" 0 21;
force "TCK" 1 23;
force "TCK" 0 24;

//"TMS"=0 change to shift-IR
//test cycle six

force "TMS" 0 26;
force "TCK" 1 28;
force "TCK" 0 29;

//load MULT_SCAN instruction "1000" in IR
//test cycle seven

force "TMS" 0 31;
force "TCK" 1 33;
force "TCK" 0 34;

//test cycle eight

force "TMS" 0 36;
force "TCK" 1 38;
force "TCK" 0 39;

//test cycle nine

force "TMS" 0 41;
force "TCK" 1 43;
force "TCK" 0 44;

//Last shift in Exit-IR Stage
//test cycle ten

force "TMS" 1 46;
force "TDI" 1 46;
force "TCK" 1 48;
force "TCK" 0 49;

//change to shift-dr stage for shifting in data
//"TMS" = 11100
```

```
    //"TMS"=1 change to update-IR state
    //test cycle eleven

    force "TMS" 1 51;
    force "TDI" 1 51;
    force "TCK" 1 53;
    force "TCK" 0 54;

    //"TMS"=1 change to select-DR state
    //test cycle twelve

    force "TMS" 1 56;
    force "TCK" 1 58;
    force "TCK" 0 59;

    //"TMS"=0 change to capture-DR state
    //test cycle thirteen

    force "TMS" 0 61;
    force "TCK" 1 63;
    force "TCK" 0 64;

    //"TMS"=0 change to shift-DR state
    //test cycle fourteen

    force "TMS" 0 66;
    force "TEST_MODE" 1 66;
    force "RESETN" 1 66;
    force "TCK" 1 68;
    force "TCK" 0 69;
    period      70;
end;

proc shift =
    force_sci 1;
    measure_sco 2;
    force "TCK" 1 3;
    force "TCK" 0 4;
    period      5;
end;

proc load_unload =
    force "TMS" 0 1;
```

```
        force "CLK" 0 1;
        apply shift 77 5;

        //"TMS"=1 change to exit-1-DR state

        force "TMS" 1 6;
        apply shift 1 10;

        //"TMS"=1 change to update-DR state

        force "TMS" 1 11;
        force "TCK" 1 13;
        force "TCK" 0 14;

        //"TMS"=1 change to select-DR-scan state

        force "TMS" 1 16;
        force "TCK" 1 18;
        force "TCK" 0 19;

        //"TMS"=0 change to capture-DR state

        force "TMS" 0 21;
        force "TCK" 1 23;
        force "TCK" 0 24;

        period      25;
    end;
```

Upon completion of the test_setup procedure, the tap controller is in the shift-DR state in preparation for loading the scan chain(s). It is then placed back into the shift-DR state for the next scan cycle. This is achieved by the following:

- The items that result in the correct behavior are the pin constraint on tms of C1 and the fact that the capture clock has been specified as TCK.
- At then end of the load_unload procedure, FastScan asserts the pin constraint on TMS, which forces tms to 0.
- The capture clock (TCK) occurs for the cycle and this results in the tap controller cycling from the run-test-idle to the Select-DR-Scan state.

- The load_unload procedure is again applied. This will start the next load/unloading the scan chain.

The first procedure in the test procedure file is **test_setup**. This procedure's first resets the test circuitry by forcing trstz to 0. The next set of actions moves the state machine to the Shift-IR state to load the instruction register with the internal scan instruction code (1000) for the MULT_SCAN instruction. This is accomplished by shifting in 3 bits of data (tdi=0 for three cycles) with tms=0, and the 4th bit (tdi=1 for one cycle) when tms=1 (at the transition to the Exit1-IR state). The next move is to sequence the TAP to the Shift-DR state to prepare for internal scan testing.

The second procedure in the test procedure file is **shift**. This procedure forces the scan inputs, measures the scan outputs, and pulses the clock. Because the output data transitions on the falling edge of tck, the measure_sco command at time 0 occurs as tck is falling. The result is a rules violation unless you increase the period of the **shift** procedure so tck has adequate time to transition to 0 before repeating the shift. The **load_unload** procedure, which is next in the file, calls the **shift** procedure.

The basic flow of the **load_unload** procedure is to:

1. Force circuit stability (all clocks off, etc.).
2. Apply the **shift** procedure n-1 times with tms=0
3. Apply the shift procedure one more time with tms=1
4. Set the TAP controller to the Capture-DR state.

The **load_unload** procedure inactivates the reset mechanisms, because you cannot assume they hold their values from the **test_setup** procedure. It then applies the **shift** procedure 77 times with tms=0 and once more with tms=1 (one shift for each of the 77 scan registers within the design). The procedure then sequences through the states to return to the Capture-DR state. You must also set tck to 0 to meet the requirement that all clocks be off at the end of the procedure.

Creating Instruction-Based Test Sets (FlexTest)

FlexTest can generate a functional test pattern set based on the instruction set of a design. You would typically use this method of test generation for high-end, non-scan designs containing a block of logic, such as a microprocessor or ALU. Because this is embedded logic and not fully controllable or observable from the design level, testing this type of functional block is not a trivial task. In many such cases, the easiest way to approach test generation is through manipulation of the instruction set.

Given information on the instruction set of a design, FlexTest randomly combines these instructions and determines the best data values to generate a high test coverage functional pattern set. You enable this functionality by using the Set Instruction Atpg command, whose usage is as follows:

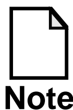
SET INstruction Atpg **OFf** | {**ON** ***filename***}

By default, FlexTest turns off instruction-based ATPG. If you choose to turn this capability on, you must specify a filename defining information on the design's input pins and instruction set. The following subsections discuss the fault detection method and instruction information requirements in more detail.

Instruction-Based Fault Detection

The instruction set of a design relates to a set of values on the control pins of a design. Given the set of control pin values that define the instruction set, FlexTest can determine the best data pin (and other non-constrained pin) values for fault detection.

For example, Table 6-2 shows the pin value requirements for an ADD instruction which completes in three test cycles.


Note

An N value indicates the pin may take on a new value, while an H indicates the pin must hold its current value.

Table 6-2. Pin Value Requirements for ADD Instruction

	Ctrl 1	Ctrl 2	Ctrl 3	Ctrl 4	Data1	Data2	Data3	Data4	Data5	Data6
Cycle1	1	0	1	0	N	N	N	N	N	N
Cycle2	H	H	H	H	H	H	H	H	H	H
Cycle3	H	H	H	H	H	H	H	H	H	H

As Table 6-2 indicates, the value 1010 on pins Ctrl1, Ctrl2, Ctrl3, and Ctrl4 defines the ADD instruction. Thus, a vector to test the functionality of the ADD instruction must contain this value on the control pins. However, the tool does not constrain the data pin values to any particular values. That is, FlexTest can test the ADD instruction with many different data values. Given the constraints on the control pins, FlexTest generates patterns for the data pin values, fault simulates the patterns, and keeps those that achieve the highest fault detection.

Instruction File Format

The following list describes the syntax rules for the instruction file format:

- The file consists of three sections, each defining a specific type of information: control inputs, data inputs, and instructions.
- You define control pins, with one pin name per line, following the “Control Input:” keyword.
- You define data pins, with one pin name per line, following the “Data Input:” keyword.
- You define instructions, with all pin values for one test cycle per line, following the “Instruction” keyword. The pin values for the defined instructions must abide by the following rules:
    - You must use the same order as defined in the “Control Input:” and “Data Input:” sections.

  - You can use values 0 (logic 0), 1 (logic 1), X (unknown), Z (high impedance), N (new binary value, 0 or 1, allowed), and H (hold previous value) in the pin value definitions.
  - You cannot use N or Z values for control pin values.
  - You cannot use H in the first test cycle.
- You define the time of the output strobe by placing the keyword "STROBE" after the pin definitions for the test cycle at the end of which the strobe occurs.
- You use "/" as the last character of a line to break long lines.
- You place comments after a "//" at any place within a line.
- All characters in the file, including keywords, are case insensitive.

During test generation, FlexTest determines the pin values most appropriate to achieve high test coverage. It does so for each pin that is not a control pin, or a constrained data pin, given the information you define in the instruction file.

Figure 6-22 shows an example instruction file for the ADD instruction defined in Table 6-2 on page 6-112, as well as a subtraction (SUB) and multiplication (MULT) instruction.

```
Control Input:
Ctrl1
Ctrl2
Ctrl3
Ctrl4
Data Input:
Data1
Data2
Data3
Data4
Data5
Data6
Instruction: ADD
1010NNNNNN   //start of 3 test cycle ADD Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE       //strobe after last test cycle
Instruction: SUB
1101NNNNNN   //start of 3 test cycle SUB Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE       //strobe after last test cycle
Instruction: MULT
1110NNNNNN   //start of 6 test cycle MULT Instruction
HHHHHHHHHH
1001NNNNNN   //next part of MULT Instruction
```

Figure 6-22. Example Instruction File

This instruction file defines four control pins, six data pins, and three instructions: ADD, SUB, and MULT. The ADD and SUB instructions each require three test cycles and strobe the outputs following the third test cycle. The MULT instruction requires six test cycles and strobes the outputs following the fifth test cycle. During the first test cycle, the ADD instruction requires the values 1010 on pins Ctrl1, Ctrl2, Ctrl3, Ctrl4, and allows FlexTest to place new values on any of the data pins. The ADD instruction then requires that all pins hold their values for the

remaining two test cycles. The resulting pattern set, if saved in ASCII format, contains comments specifying the cycles for testing the individual instructions.

Using FastScan's Macrotest Capability

Macrotest automates the testing of embedded RAMs or ROMs, embedded hierarchical instances, and embedded blocks of logic with unidirectional I/O. For example, a sequential block may be reused, with added combinational logic on its inputs, outputs, or both (to MUX in other functions, etc.). Or perhaps a RAM marching test is to be applied to a small embedded RAM or a register file. In these cases, the tests for the nonembedded logic are already known or generated, but they need to be converted for use in the embedded environment.

When to Use Macrotest

Macrotest is primarily used to test small memories (register file, cache, FIFO, etc.). Although FastScan can test the faults at the boundaries of such devices, and can propagate the fault effects through them (using the _ram or _cram primitives), it does not attempt to create a set of patterns to internally test them. This is consistent with how it treats all primitives. Since these devices are far more complex than a typical primitive (such as a NAND gate), you may prefer to augment FastScan patterns with patterns which you create to test the internals of the more complex memory primitives. Such complex primitives are usually packaged as models or modules, and these are called macros. Since you determine an appropriate test, you must also provide the inputs and expected outputs for the macro. Typically, you can select a set of tests, simulate these tests in some time based simulator, and use the results predicted by that simulator as the expected outputs of the macro.

Macrotest helps aid this process because these devices are often embedded in systems so that the inputs and outputs of the memory are not directly accessible. The inputs might have enables ANDed outside the macro, or the outputs might be MUXed to various places, etc. The tests must be converted so they can be applied to the inputs of the AND and will have the correct value at the macro's inputs. Macrotest converts the tests (provided in a file) so that the inputs are provided to the macro as specified in the file, and the outputs of the macro are observed.

Each row of a macrotest file is converted to a scan test. A scan chain load, PI assertion, output measure, clock pulse, and scan chain unload typically result for each row of the file. For example, the first few rows might apply known inputs to the device but have no known expected outputs. First, specify the values to apply at the inputs to the device and give X output values (Don't Care or Don't Measure). Then, specify a write with no expected known outputs. Again, only input values should be specified. Next, a read might be done, with expected known outputs. You should specify both the inputs to apply, and the outputs that are expected (as a result of those and all prior inputs applied in the file so far).

Defining the Macro Boundary

A macro is specified by its instance name. The macro is a top level model in the ATPG library, and one or more instances of that device occur in the netlist, with a unique instance name for each occurrence (instantiation) of the device in the netlist. This is all that is needed to define the boundary. The definition of the macro is accessed to determine the pin order as defined in the port list, and that pin order is expected to be used in the file specifying the I/O (input and expected output) values for the macro (the tests). For example, the command:

macrotest regfile_8 file_with_tests

would specify macrotest to find the instance "regfile_8", look up its model definition, and record the name and position of each pin in the port list. Given that the netlist was Verilog, with the command:

regfile_definition_name regfile_8 (net1, net2, ...);

the portlist of regfile_definition_name is used to get the pin names, directions, and the ordering expected in the test file, *file_with_tests*. If the library definition was:

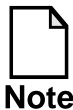
model "regfile_definition_name" ("Dout_0", "Dout_1", Addr_0", "Addr_1", "Write_enable",..) (input ("Addr_0") () ... output ("Dout_0") () ....)

then macrotest knows to expect the output Dout_0 as the first pin mentioned in each row (test) of the file, *file_with_tests*. The output Dout_1 should be the 2nd pin, input pin Addr_0 should be the 3rd pin value encountered, etc.

If it is inconvenient to use this ordering, the ordering can be changed at the top of the test file, *file_with_tests*. This can be done using the following syntax:

```
macro_inputs Addr_0  Addr_1
macro_output Dout_1
macro_inputs Write_enable
...
end
```

which would cause macrotest to expect the value for input Addr_0 to be the first value in each test, followed by the value for input Addr_1, the expected output value for Dout_1, the input value for Write_enable, etc.


Note

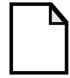
Only the pin names need be specified, because the instance name "regfile_8" was given on the macrotest command line.

Defining Test Values

The test file may consist of the following:

- Comments (a line starting with "//" or #)
- Blank lines
- An optional pin reordering section (which must come before any values) that begins with "MACRO_INPuts" or "MACRO_OUTputs" and ends with "END"
- The tests (one test per row of the file)

Normal (nonpulseable) input pin values include {0,1,X,Z}. Some macro inputs may be driven by PIs declared as pulseable pins (Add Clocks, Add Read Control, and Add Write Control specify these pins in FastScan). These pins can have values from {P,N} where P designates a positive pulse and N designates a negative pulse.


Note

It is the declaration of the pin driving the macro input, not any declaration of the input itself, which determines whether a pin can be pulsed in FastScan.

Output values include {L,H,X,F} which are analogous to {0,1,X,Z}. L represents LO (output 0), H for output 1, X for Don't Compare, and F is for Float (output Z). If you provide a file with these characters, a checking is done to ensure that an input pin gets an input value, and an output pin gets an output value. If an "L" is specified in an input pin position, an error message is issued. This helps detect ordering mismatches between the port list and the test file. If you prefer to use "0" and "1" for both inputs and outputs, then the -NO_L_H option should be used with the macrotest command:

macrotest regfile_8 file_with_tests -no_l_h

Assuming that the -L_H default is used, the following might be the testfile contests for our example register file, if the default port list pin order is used.

```
// Tests for regfile_definition_name.
//
//       W
//       r
//       i
//       t
//       e
//       _
// DD AA e
// oo dd n
// uu dd a
// tt rr b
// __ __ l
// 01 01 e

   XX 00 0
   XX 00 1
   HH 00 0
```

The example file above only has comments and data. Spaces are used to separate the data into fields for convenience. Each row must have exactly as many value

characters as pins mentioned in the original port list of the definition (or the exact number of pins in the reordering if one was given).

Specifying less than all pins can be done by simply omitting the pins from the header when reordering the pins. The omitted pins are ignored for purposes of macrotest. If the correct number of values do not exist on every row, an error occurs and a message is issued.

The following is an example where the address lines are exchanged, and only Dout_0 is to be tested:

```
// Tests for regfile_definition_name testing only Dout_0
macro_output Dout_0
macro_inputs Addr_1 Addr_0 write_enable ..
...
end
//       W
//       r
//       i
//       t
//       e
//       _
// D AA e
// o dd n
// u dd a
// t rr b
// _ __ l
// 0 10 e

   X 00 0
   X 00 1
   H 00 0
```

It is not necessary to have all macro_inputs together. You can repeat the direction designators as necessary:

```
macro_input write_enable
macro_output Dout_0
macro_inputs Addr_1 Addr_0
macro_outputs Dout_1 ...
...
end
```

Recommendations for Using Macrotest

When using macrotest, it is recommended that you begin early in the process. This is because the environment surrounding a regfile may prevent the successful delivery of the tests, and Design-for-Test may have to be used to allow the tests to be delivered. For example, if the write enable is the complement of the read enable due to a line which drives the read enable as well as an inverter which drives the write enable, and you specify that both pins should be 0 for some test, then macrotest is unable to satisfy this test requirement. It stops and reports the line of the test file, as well as the input pin(s) and value(s) that cannot be delivered.

Once macrotest is completed, you should simulate the resulting tests in a time based simulator (probably the one used to simulate the design). This verifies that the conversion was correct, and that no timing problems exist. FastScan cannot simulate the internals of primitives, and therefore relies on the fact that the inputs produced the expected outputs given in the test file. This final simulation ensures that no errors exist due to modeling or simulation details that might differ from one simulator to the next.

FastScan ATPG commands and options apply within macrotest. If macrotest fails, and reports that it aborted, you can use the Set Abort Limit command to try again. Also, the handling of bus contention (Set Bus Contention command) are determined by FastScan commands and work as they would for normal ATPG. As each row is converted to a test, that test is store internally, similar to a normal test. You can Save Patterns to write out the tests in the desired format (Verilog to allow simulation and TSSI WGL for a tester). It is recommended that you give a relatively high abort limit to FastScan, and also issue "Set Contention Check On -ATPG" prior to issuing any macrotest command.

Macros are typically small compared to the design that they are in. It is possible to get coverage of normal faults while testing the macro. The default is for macrotest to randomly fill any scan chain or PI inputs not needed for a particular test so that fortuitous detection of other faults occurs. If you add faults using the Add Faults -all command (or create a fault list by some other mechanism) before invoking macrotest, then the random fill and fault simulation of the patterns occurs, and any faults detected by the simulation will be marked as DS.

A Macrotest Example

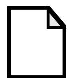
Verilog Contents:

```
RAM  mem1 (.Dout ({ \Dout[7] , \Dout[6] , \Dout[5] ,
\Dout[4] , \Dout[3] , \Dout[2] , \Dout[1] , \Dout[0] }) ,
.RdAddr ({ \RdAddr[1] , \RdAddr[0] }) , .RdEn ( RdEn ) ,
.Din ({ \Din[7] , \Din[6] , \Din[5] , \Din[4] , \Din[3] ,
\Din[2] , \Din[1] , \Din[0] }) , .WrAddr ({ \WrAddr[1] ,
\WrAddr[0] }) , .WrEn ( WrEn ) , .Clock ( clk ));
```

ATPG Library Contents:

```
model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn, Clock) (
  input (RdAddr,WrAddr) (array = 1 : 0;)
  input (RdEn,WrEn,Clock) ()
  input (Din) (array = 7 : 0;)

   output (Dout) (
    array = 7 : 0;
    data_size = 8;
    address_size = 2;
    read_write_conflict = XW;
    primitive = _cram(,,
      _write {,,} (WrEn,,WrAddr,Din),
      _read {,,,} (,RdEn,,RdAddr,Dout)
    );
  )
)
```


Note

Vectors are treated as expanded scalars.

So, because Dout is array 7:0, the string "Dout" in the port list is equivalent to "Dout<7> Dout<6> Dout<5> Dout<4> Dout<3> Dout<2> Dout<1> Dout<0>". If the declaration had been Dout is array 0:7, then the string "Dout" would be the reverse of the above expansion. Vectors are always allowed in the model definitions. Currently, vectors are not allowed in the test file, so if you redefine the

pin order in the test file, scalars must be used. Either “Dout<7>”, “Dout(7)”, or “Dout[7]” can be used if reordering is necessary.

Dofile Contents:

```
set system mode atpg
macrotest mem1 ram_patts2.pat
save patterns results/pattern2.f -replace
```

Test File Input (ram_patts2.pat) Contents:

```
// Testing the following RAM :
//model RAM (Dout, RdAddr, RdEn, Din, WrAddr, WrEn, Clock) (
//  input (RdAddr,WrAddr) (array = 1 : 0;)
//  input (RdEn,WrEn,Clock) ()
//  input (Din) (array = 7 : 0;)
//
//  output (Dout) (
//    array = 7 : 0;
//    .....
XLHLHLHL 00 1 X0101010 11 1 P
LXLHLHLH 00 1 0x010101 11 1 P
LHxHLHLH 00 1 01X10101 11 1 P
LHLxXHLH 00 1 010Xx101 11 1 P
```

Converted Test File Output (results/pattern2.f) Contents:

```
... skipping some header information ....
SETUP =
  declare input bus "PI" = "/clk", "/Datsel",
                  "/scanen_early", "/scan_in1", "/scan_en";
  declare output bus "PO" = "/scan_out1";
.... skipping some declarations ....

CHAIN_TEST =
  pattern = 0;
  apply "grp1_load" 0 =
      chain "chain1" = "0011001100110011001100";
  end;
  apply "grp1_unload" 1 =
      chain "chain1" = "0011001100110011001100";
  end;
end;
```

```
SCAN_TEST =
  pattern = 0  macrotest ;
  apply "grp1_load" 0 =
      chain "chain1" = "11001010100011011010l0";
  end;
  force   "PI" "00110" 1;
  pulse "/scanen_early" 2;
  measure "PO" "1" 3;
  pulse "/clk" 4;
  apply "grp1_unload" 5 =
      chain "chain1" = "XXXXXXXXXXXXXXX0101010";
  end;

  pattern = 1  macrotest ;
  apply "grp1_load" 0 =
      chain "chain1" = "1101010101001111100001";
  end;
  force   "PI" "00110" 1;
  pulse "/scanen_early" 2;
  measure "PO" "1" 3;
  pulse "/clk" 4;
  apply "grp1_unload" 5 =
      chain "chain1" = "XXXXXXXXXXXXXX0X010101";
  end;

  pattern = 2  macrotest ;
  apply "grp1_load" 0 =
      chain "chain1" = "1101010101001111101111";
  end;
  force   "PI" "00110" 1;
  pulse "/scanen_early" 2;
  measure "PO" "1" 3;
  pulse "/clk" 4;
  apply "grp1_unload" 5 =
      chain "chain1" = "XXXXXXXXXXXXXX01X10101";
  end;

  pattern = 3  macrotest ;
  apply "grp1_load" 0 =
      chain "chain1" = "1101001101001100100100";
  end;
  force   "PI" "00100" 1;
```

```
      pulse "/scanen_early" 2;
      measure "PO" "1" 3;
      pulse "/clk" 4;
      apply "grp1_unload" 5 =
          chain "chain1" = "XXXXXXXXXXXXXX010XX101";
      end;
    end;

  SCAN_CELLS =
      scan_group "grp1" =
          scan_chain "chain1" =
             scan_cell =    0  MASTER  FFFF  "/rden_reg/ffdpb0"
  ""  "sd"  "so";
             scan_cell =    1  MASTER  FFFF  "/wren_reg/ffdpb0"
  ""  "sd"  "so";
             scan_cell =    2  MASTER  FFFF  "/datreg1/ffdpb7"
  ""  "sd"  "so";
      ... skipping some scan cells ...
             scan_cell =   20  MASTER  FFFF  "/doutreg1/ffdpb1"
  ""  "sd"  "so";
             scan_cell =   21  MASTER  FFFF  "/doutreg1/ffdpb0"
  ""  "sd"  "so";
          end;
      end;
    end;
```

Example Multiple Macro Invocation

Dofile command:

Test macros in file simultaneously. Default to -random_observe for all macros in file.

macrotest -mult macro_file_3 -random_observe

Multiple Macro File (macro_file_3) Contents:

```
// Produces 2 tests.
macrotest test0/mem1 ram_patts0.pat -no_L_H
// override -rand default. Produces 4 tests.
macrotest test1/mem1 ram_patts2.pat -det_observe
```

The above command and file cause macrotest to try to test two different macros simultaneously. It is not necessary that they have the same test set length (same number of tests in their respective test files). One has 2 tests, while the other has 4. It is possible to test each macro individually, discard the tests it creates, move its command into a file, and attempt to test it with other individually successful macros at the same time by referencing that file in a -multiple_macros run.

In the above example, an instance named “test0” has an instance named “mem1” inside it that is a macro to test using file ram_patts0.pat, while an instance named “test1” has an instance named “mem1” inside it that is another macro to test using fileram_patts2.pat as the test file.

Verifying Design and Test Pattern Timing

After testing the functionality of the circuit with QuickSim, and generating the test vectors with FastScan or FlexTest, you should run the test vectors in QuickSim and compare the results with predicted behavior from the ATPG tools. This run will point out any functionality discrepancies between the two tools, and also show timing differences that may cause different results. The following subsections further discuss the verification you should perform.

Simulating the Design with Timing

At this point in the design process, you should run a full timing verification to ensure a match between the results of golden simulation and ATPG. This verification is especially crucial for designs containing asynchronous circuitry. This section describes how you can accomplish that task.

You should have already saved the generated test patterns with the Save Patterns command in FastScan or FlexTest. If you selected -Mgcwdb as the format in which to save the patterns, the application automatically creates a dofile that you can use in QuickSim II for automatic vector comparison with simulation.

For example, assume you saved the patterns generated in FastScan or Flextest with the options as follows:

ATPG> **save patterns pat_file timing -serial -Mgcwdb**

The tool writes the test pattern out as a “forces” MGC WDB. The command also creates an “assert” MGC WDB. This contains the expected data for comparison with the results from QuickSim II. In addition, the application creates a dofile. This dofile loads the appropriate WDBs, defines input and output pins, sets up the assert signals, and runs the simulation. Additionally, the tool creates an error file containing the discrepancies. The following shows an example of the dofile:

```
// Quicksim dofile for parallel pattern simulation
// Load waveform databases and comparison function
$$load_wdb("pattern_in","forces");
$$load_wdb("pattern_out","asserts");
// Define buses
add bus pi_bus_000 /RST \
                   /CLK \
                   /SCAN_EN \
                   /SC2_I \
                   /SC1_I \
                   /SCAN_IN1 \
                   /RW_IN \
                   /D_IN(2) \
                   /D_IN(1) \
                   /D_IN(0) \
                   -replace
add bus po_bus_000 /D_OUT(0) \
                   /D_OUT(1) \
                   /D_OUT(2) \
                   /SCAN_OUT1 \
                   /SC1_O \
                   /SC2_O \
                   -replace
add bus si_bus_000 /U1/INST__565_FF_D_0__DFF/SDI \
                   /U1/INST__565_FF_D_3__13/SDI \
                   /U1/INST__565_FF_D_2__13/SDI \
                   /U1/INST__565_FF_D_1__13/SDI \
                   /U2/INST__302_FF_D_2__DFF/SDI \
                   /U2/INST__302_FF_D_1__DFF/SDI \
                   -replace
add bus so_bus_000 /U1/INST__565_FF_D_0__DFF/QB \
```

```
                       /U1/INST__565_FF_D_3__13/Q \
                       /U1/INST__565_FF_D_2__13/Q \
                       /U1/INST__565_FF_D_1__13/QB \
                       /U2/INST__302_FF_D_2__DFF/QB \
                       /U2/INST__302_FF_D_1__DFF/QB \
                       /U2/INST__302_FF_D_0__DFF/QB \
                       -replace
// Define keeps
add keeps po_bus_000
add keeps so_bus_000
// Define traces
//add trace forces@@pi_bus_000
//add trace asserts@@po_bus_000
//add trace results@@po_bus_000
// Define clocks, scan-in, scan-out pins
//add trace forces@@RST
//add trace forces@@CLK
//add trace forces@@SCAN_IN1
//add trace results@@SCAN_OUT1
//add trace asserts@@SCAN_OUT1
//add trace forces@@si_bus_000
//add trace asserts@@so_bus_000
//add trace results@@so_bus_000
// Define lists
//add list forces@@pi_bus_000
//add list asserts@@po_bus_000
//add list results@@po_bus_000
// Define clocks, scan-in, scan-out pins
//add list forces@@RST
//add list forces@@CLK
//add list forces@@SCAN_IN1
//add list results@@SCAN_OUT1
//add list asserts@@SCAN_OUT1
//add list forces@@si_bus_000
//add list asserts@@so_bus_000
//add list results@@so_bus_000
// Run the simulation and compare waveforms
// Define asserts on each primary output pin
$assert("asserts@@D_OUT(0)", "Xr", 0, void, void, void, \
         @relative, "");
$assert("asserts@@D_OUT(1)", "Xr", 0, void, void, void, \
         @relative, "");
$assert("asserts@@D_OUT(2)", "Xr", 0, void, void, void, \
```

```
        @relative, "");
$assert("asserts@@SCAN_OUT1", "Xr", 0, void, void, void, \
        @relative, "");
$assert("asserts@@SC1_O", "Xr", 0, void, void, void, \
        @relative, "");
$assert("asserts@@SC2_O", "Xr", 0, void, void, void,  \
       @relative, "");
//$assert("asserts@@po_bus_000", "XrXrXrXrXrXr", 0, void, \
          void, void,@relative, "");
// Define asserts on each scan output pin
$assert("asserts@@/U1/INST__565_FF_D_0__DFF/QB", "Xr", 0, \
         void, void, void, @relative, "");
$assert("asserts@@/U1/INST__565_FF_D_3__13/Q", "Xr", 0, void,
\
          void, void,@relative, "");
$assert("asserts@@/U1/INST__565_FF_D_2__13/Q", "Xr", 0, void,
\
          void, void,@relative, "");
$assert("asserts@@/U1/INST__565_FF_D_1__13/QB", "Xr", 0,  \
         void, void, void,@relative, "");
$assert("asserts@@/U2/INST__302_FF_D_2__DFF/QB", "Xr", 0,  \
         void, void, void,@relative, "");
$assert("asserts@@/U2/INST__302_FF_D_1__DFF/QB", "Xr", 0,  \
         void, void, void,@relative, "");
$assert("asserts@@/U2/INST__302_FF_D_0__DFF/QB", "Xr", 0,  \
         void, void, void,@relative, "");
//$assert("asserts@@so_bus_000", "XrXrXrXrXrXrXr", 0,  \
         void, void, void,@relative, "");
$setup_assertion_generic(@other, "01Z", @all, void);
$setup_assertion_report(@file, "pattern.error", @replace, \
          @binary);
run
```

Checking for Clock-Skew Problems with Mux-DFF Designs

If you have mux-DFF scan circuitry in your design, you should be aware of, and thus test for, a common timing problem involving clock skew. Figure 6-23 depicts the possible clock-skew problem with the mux-DFF architecture.

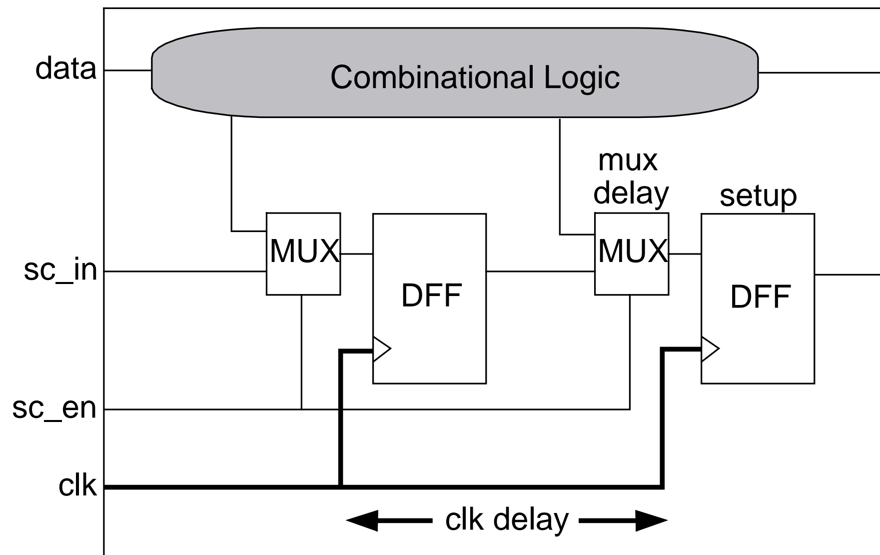


Figure 6-23. Clock-Skew Example

You can run into problems if the clock delay due to routing is greater than the mux delay minus the flip-flop setup time. In this situation, the data does not get captured correctly from the previous cell in the scan chain and therefore, the scan chain does not shift data properly.

To detect this problem, you should run both critical timing analysis and functional simulation of the scan load/unload procedure. In the Mentor Graphics environment, you can use QuickSim II for the functional simulation and QuickPath for the timing analysis. Refer to the *QuickSim II User's Manual* or the *QuickPath User's and Reference Manual* for details on performing timing verification.

Chapter 7
Test Pattern Formatting and Timing

Figure 7-1 shows a basic process flow for defining test pattern timing.

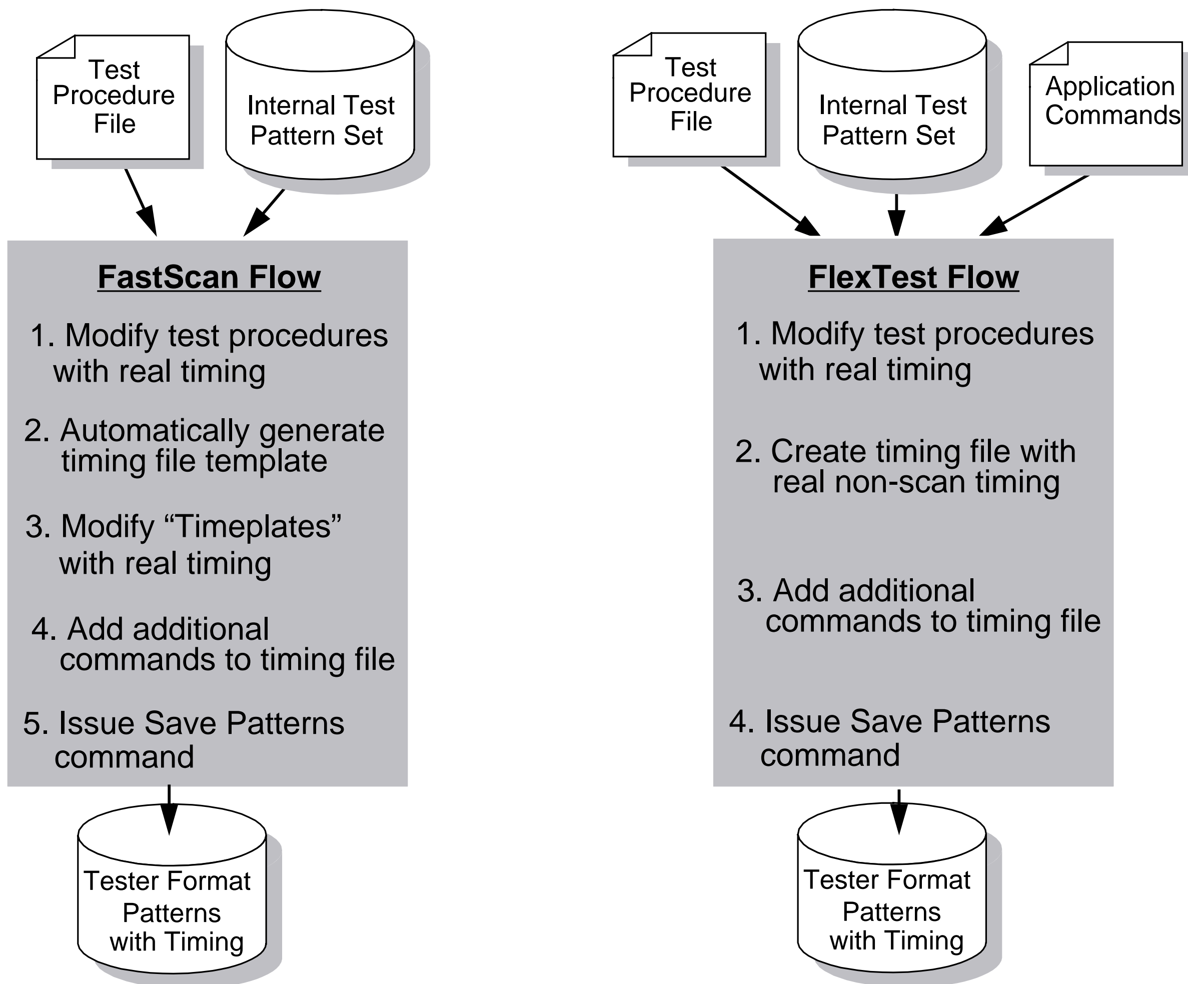


Figure 7-1. Defining Timing Process Flow

The subsections of this chapter describe each step in more detail.

Test Pattern Timing Overview

Test procedure files define scan operations. Thus, *scan-related events* refer to those scan events (or operations) defined in test procedure files. *Non-scan-related events* include the remaining test pattern events not defined in the test procedure files. Test procedure files only support timing information for scan-related events.

While the ATPG process itself does not require test procedure files to contain real timing information, automatic test equipment (ATE) and some simulators do require this information. Therefore, you must modify the test procedure files you use for ATPG to include real timing information. "Defining Scan-Related Event Timing" on page 7-3 discusses how you add timing information to existing test procedures.

Because test procedure files do not support timing for non-scan-related events, FastScan and FlexTest require an external timing file to define this timing information. "Defining Non-Scan Related Event Timing" on page 7-13 discusses defining timing for non-scan-related events.

If you want the timing checker to check for timing restrictions required by certain test formats, you can add special commands to the timing file. "Performing Timing Checks for Tester Formats" on page 7-21 discusses this task.

After creating real timing for the test procedures and an external timing file for non-scan events, you are ready to save the patterns. You use the Save Patterns command with the proper format and timing file name to create a test pattern set with timing information. "Saving the Patterns" on page 7-24 discusses this in more detail.

Timing Terminology

The following list defines some timing-related terms:

- **ATPG capture cycle** - non-scan event test cycle.
- **Constant values** - pins that stay at a specific value (0, 1, X, or Z) during non-scan operation.

- **Non-return timing** - primary inputs that change, at most, once during a test cycle.
- **Offset** - the timeframe in a test cycle in which pin values change.
- **Period** - the duration of pin timing—one or more test cycles.
- **Return timing** - primary inputs, typically clocks, that pulse high or low during every test cycle. Return timing indicates that the pin starts at one logic level, changes, and returns to the original logic level before the cycle ends.
- **Suppressible return timing** - primary inputs that can exhibit return timing during a test cycle, although not necessarily.

Defining Scan-Related Event Timing

ATE require test data in a cycle-based format. Thus, the patterns you apply to such equipment must specify the waveforms of each input, output, or bidirectional pin, for each test cycle.

Within a test cycle, a device under test must abide by the following restrictions:

- At most, each non-clock input pin changes once in a test cycle. However, different input pins can change at different times.
- Each clock input pin is at its off-state at both the start and end of a test cycle.
- At most, each clock input pin changes twice in a test cycle. However, different clock pins can change at different times.
- Each output pin has only one expected value during a test cycle. However, the equipment can measure different output pin values at different times.
- A bidirectional pin acts as either an input or an output, but not both, during a single test cycle.

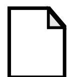
Converting Test Procedures to Test Cycles

Test procedures contain groups of statements that define scan related events. “Test Procedure Files” on page 3-11 introduces test procedures and statements.

The sequence of test procedure events must convert to a series of tester cycles. These tester cycles apply stimuli and observe responses from the circuit under test.

Both FastScan and FlexTest use a test pattern data formatter which, following the previously mentioned restrictions, converts test procedures to test cycles. The formatter algorithm groups events into test cycles by performing the following steps:

1. Group test procedure events into event groups based on the sequence of these events and the specified **break** or **break_repeat** statements. The algorithm creates a new test cycle whenever an input pin with non-return timing changes state, or whenever an input pin with return timing (a clock) goes active for a second time.
2. Calculate the procedure cycle time by dividing the test procedure period by the number of test cycles found in step 1.
3. Ensure **break** and **break_repeat** statements occur at multiples of the test procedure cycle time.
4. Ensure that each input pin keeps the same offset when changing states in different test cycles. For pins with return timing, ensure that the pin retains the same pulse width in each of the test cycles.


Note

A pin can only have one timing definition within each test procedure, and the pin timing should be consistent in all test cycles of the procedure. The **shift** procedure duration is always a single test cycle.

Test Procedure Timing Examples

The following example depicts how the test pattern formatter converts a **test_setup** procedure to test cycles.

```
procedure test_setup =
   force TE    0 0;
   force NCLK  0 0;
   force NCLK  1 1;
   force NCLK  0 2;
   force TE    1 2;
   period        4;
end;
```

Figure 7-2 shows the resulting timing diagram generated for input pin TE and clock pin NCLK.

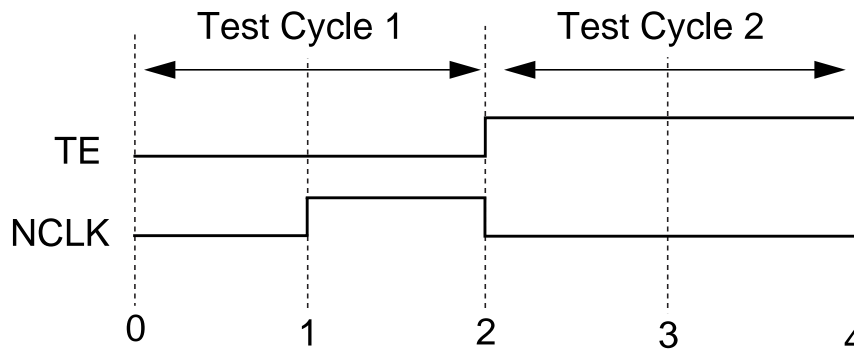
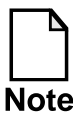


Figure 7-2. Test Cycle Timing for Test_Setup Procedure

The input pin TE undergoes a transition at time 2. This initiates the second test cycle in the procedure. Clock pin NCLK changes twice, but makes only one active transition in the first test cycle.



Note

The number of test cycles (2) evenly divides into the period (4). Also, the TE pin changes state at time 0 in both test cycles. If the TE pin force occurred at time 3 instead of 2, the tool would issue an error indicating the pin had incompatible offsets of 0 and 1 (3 mod 2).

Use care when defining test procedures for tester environments that allow only a single timing definition. "Saving the Patterns" on page 7-24 discusses each of the format types and the timing requirements, such as single timing definition restrictions, for each. In this situation, the test procedure timing must coincide with the timing of the non-scan cycle.

FastScan applies capture or RAM clocks after the primary input pins change state. Additionally, it measures output pins before the capture clock pulses in the non-scan cycle. The events in each test cycle in a test procedure should follow this sequence to coincide with the non-scan cycle timing. If the test procedure file violates this condition, you may have to regenerate test patterns after running the design rules checker on the modified test procedure.

The following example illustrates this issue:

```
PROC TEST_SETUP =
   FORCE nclk     0    0;
   FORCE nclk     1    1;
   FORCE nclk     0    2;
   FORCE te       1    3;
   PERIOD              4;
END;
```

This **test_setup** procedure needs only one test cycle. However, the timeplate for this test procedure does not coincide with FastScan's non-scan cycle because the input pin changes after the clock pin NCLK pulses. Thus, you could not use this test procedure to generate test patterns in a tester format that allows only one timing definition.

If you modify this test procedure after FastScan produces the pattern set, you will encounter problems. This is because you cannot change the sequence of test procedure events after pattern generation and then save patterns with the modified test procedure file. You can only change the specified times in the test procedures after pattern generation. In this case, if you modify the test procedure to ensure consistent timing, you would have to run pattern generation again using the following modified test procedure:

```
PROC TEST_SETUP =
   FORCE nclk     0    0;
   FORCE te       0    1;
   FORCE nclk     1    1;
   FORCE nclk     0    2;
   FORCE te       1    3;
   PERIOD              4;
END;
```

Some procedures require a more complex conversion process. For most scan styles, each test procedure maps to a test cycle. However, with more complex scan styles (like boundary scan, which uses a sequential scan controller), you should write the test procedures with timing issues in mind.

The following example shows a test procedure file, which either FastScan or FlexTest can use, that requires a 400ns test cycle:

```
PROC TEST_SETUP =
  FORCE trstz  0 0;
  FORCE clearz 0 0;
  FORCE clk    0 0;
  FORCE tms    1 0;
  FORCE tck    0 0;
  FORCE tck    1 200;
  FORCE trstz  1 300;
  FORCE clearz 1 300;
  FORCE tck  0 300;
// change to run-test/idle
  FORCE tms  0 400;
  FORCE tck  1 600;
  FORCE tck  0 700;
// tms=1  change to select DR scan state
  FORCE tms  1 800;
  FORCE tck  1 1000;
  FORCE tck  0 1100;
// tms=1  change to select-IR scan state
  FORCE tms  1 1200;
  FORCE tck  1 1400;
  FORCE tck  0 1500;
// tms=0  change to capture-IR state
  FORCE tms  0 1600;
  FORCE tck  1 1800;
```

```
  FORCE tck  0 1900;
// tms=0  change to shift-IR state
  FORCE tms  0 2000;
  FORCE tck  1 2200;
  FORCE tck  0 2300;
// load instruction register with SAMPLE using opcode
// 00 HEX==00001 BIN
// tdi-1
  FORCE tms  0 2400;
  FORCE tdi  1 2500;
  FORCE tck  1 2600;
  FORCE tck  0 2700;
// tdi-2
  FORCE tms  0 2800;
  FORCE tdi  0 2900;
  FORCE tck  1 3000;
  FORCE tck  0 3100;
// tdi-3
  FORCE tms  0 3200;
  FORCE tdi  0 3300;
  FORCE tck  1 3400;
  FORCE tck  0 3500;
// tdi-4
  FORCE tms  0 3600;
  FORCE tdi  0 3700;
  FORCE tck  1 3800;
  FORCE tck  0 3900;
// tdi-5, tms=1 to change to exit(1)-IR state
  FORCE tms  1 4000;
  FORCE tdi  0 4100;
  FORCE tck  1 4200;
  FORCE tck  0 4300;
// change to shift-DR state
// tms=1  change to update-IR state
  FORCE tms  1 4400;
  FORCE tck  1 4600;
  FORCE tck  0 4700;
// tms=1  change to select-DR-scan state
  FORCE tms  1 4800;
  FORCE tck  1 5000;
  FORCE tck  0 5100;
// tms=0  change to capture-DR state
  FORCE tms  0 5200;
```

```
    FORCE tck  1 5400;
    FORCE tck  0 5500;
  // tms=1 to change to exit(1)-DR state
    FORCE tms  1 5600;
    FORCE tck  1 5800;
    FORCE tck  0 5900;
  // tms=1  change to update-DR state
    FORCE tms  1 6000;
    FORCE tck  1 6200;
    FORCE tck  0 6300;
  // tms=1  change to select-DR-scan state
    FORCE tms  1 6400;
    FORCE tck  1 6600;
    FORCE tck  0 6700;
  // tms=1  change to select-IR-scan state
    FORCE tms  1 6800;
    FORCE tck  1 7000;
    FORCE tck  0 7100;
  // tms=0  change to Capture-IR state
    FORCE tms  0 7200;
    FORCE tck  1 7400;
    FORCE tck  0 7500;
  // tms=0  change to Shift-IR state
    FORCE tms  0 7600;
    FORCE tck  1 7800;
    FORCE tck  0 7900;
  // load instruction register with fullscan using opcode
  // 00 HEX==10001 BIN
  // tdi-1
    FORCE tms  0 8000;
    FORCE tdi  1 8100;
    FORCE tck  1 8200;
    FORCE tck  0 8300;
  // tdi-2
    FORCE tms  0 8400;
    FORCE tdi  0 8500;
    FORCE tck  1 8600;
    FORCE tck  0 8700;
  // tdi-3
    FORCE tms  0 8800;
    FORCE tdi  0 8900;
    FORCE tck  1 9000;
    FORCE tck  0 9100;
```

```
// tdi-4
  FORCE tms  0 9200;
  FORCE tdi  0 9300;
  FORCE tck  1 9400;
  FORCE tck  0 9500;
// tdi-5, tms=1 to change to exit(1)-IR state
  FORCE tms  1 9600;
  FORCE tdi  1 9700;
  FORCE tck  1 9800;
  FORCE tck  0 9900;
// change to shift-DR state
// tms=1  change to update-IR state
  FORCE tms  1 10000;
  FORCE tck  1 10200;
  FORCE tck  0 10300;
// tms=1  change to select-DR-scan state
  FORCE tms  1 10400;
  FORCE tck  1 10600;
  FORCE tck  0 10700;
// tms=0  change to capture-DR state
  FORCE tms  0 10800;
  FORCE tck  1 11000;
  FORCE tck  0 11100;
// tms=0  change to shift-DR state & execute data capture
  FORCE tms  0 11200;
  FORCE tck  1 11400;
  FORCE tck  0 11500;
  PERIOD  11600;
END;

PROC SHIFT =
  FORCE_SCI     0;
  MEASURE_SCO   0;
  FORCE tck   1 200;
  FORCE tck   0 300;
  PERIOD 400;
END;

PROC LOAD_UNLOAD =
  FORCE tck    0 0;
  FORCE trstz  1 0;
  FORCE clearz 1 0;
  FORCE clk    0 0;
```

```
    FORCE tms    0 0;
  // 26 cells in scan path
  APPLY SHIFT 25 400;
  // tms=1 to change to exit(1)-DR state
    FORCE tms   1 800;
    APPLY SHIFT 1 1200;
  // change state to capture-DR
  // tms=1  change to update-DR state
    FORCE tms  1 1200;
    FORCE tck  1 1000;
    FORCE tck  0 1100;
  // tms=1  change to select-DR-scan state
    FORCE tms  1 1200;
    FORCE tck  1 1400;
    FORCE tck  0 1500;
  // tms=0  change to capture-DR state
    FORCE tms  0 1600;
    FORCE tck  1 1800;
    FORCE tck  0 1900;
    PERIOD  2000;
  END;
```

The **test_setup** procedure applies two instructions in sequence and places the TAP controller in the shift-DR state. The **load_unload** procedure applies the main shift sequence and puts the controller back in the capture-DR state. The ATPG non-scan (capture) cycle should apply TCK exactly once to put the TAP controller back in the shift-DR state.

From this example, you should note the following:

- The TMS pin changes at multiples of 400 nanoseconds, making an offset of 0 in each test cycle.
- You should define the TCK, TRSTZ and CLEARZ pins as clocks such that they have return timing in the test procedures.
- A change in an input pin, either TMS or TDI, triggers each new test cycle.
- TCK pulses after an input pin changes in each test cycle. This ensures compatibility with the FastScan non-scan cycle timing.

- The **load_unload** procedure applies the **shift** procedure once after the main shift cycles, such that the last shift occurs in the exit(1)-DR state of the TAP controller.

The **test_setup** allows the use of a restricted measure statement. You must provide measured values and these measured values are verified by DRC through simulation. The observation of that value does not contribute to fault coverage in any way. This is useful in cases where you may want a certain set of force and measure statements to appear in their final patterns and you already know the values which will be present on the output pins. This can be helpful with parametric testing.

Test Procedure Timing Issues

To avoid adverse timing problems, the following timing requirements satisfy some ATE timing constraints:

- **Unused outputs.**
  By default, test procedures without measure events (all procedures except **shift**) strobe unused outputs at a time of cycle/2, and end the strobe at 3*cycle/4. The **shift** procedure strobes unused outputs at the same time as the scan output pin.

- **Unused inputs.**
  By default, all unused input pins in a test procedure have a force offset of 0.

- **Unused clock pins.**
  By default, unused clock pins in a test procedure have an offset of cycle/4 and a width of cycle/2, where cycle is the duration of each cycle in the test procedure.

- **Pattern loading and unloading.**
  During the **load_unload** procedure, when one pattern loads, the result from the previous pattern unloads. When the tool loads the first pattern, the unload values are X. After the tool loads the last pattern, it loads a pattern of X’s so it can simultaneously unload the values resulting from the final pattern.

- **Events between loading and unloading (FastScan only).**
  If other events occur between the current unloading and the next loading, in order to load and unload the scan chain simultaneously, FastScan performs the events in the following order:
  - Observe procedure only: FastScan performs the observe procedure before loading and unloading.
  - Initial force only: FastScan performs the initial force before loading and unloading.
  - Both observe procedure and initial force: FastScan performs the observe procedures followed by the initial force before loading and unloading.

Defining Non-Scan Related Event Timing

Non-scan events include all events not related to scan operation and not described in the test procedure file. FlexTest, by its very nature, handles non-scan designs and provides flexible handling of non-scan event timing through its application commands. For example, you can set the length of the test cycle and specify when to strobe inputs and measure outputs.

FastScan, on the other hand, contains an algorithm optimized for scan-based designs. It does not contain a user-specifiable method for defining non-scan event timing in its application commands. Thus, FastScan and FlexTest handle non-scan related event timing differently.

The following subsections describe the different ways in which you specify non-scan event timing for FastScan and FlexTest.

FastScan Non-Scan Event Timing

FastScan patterns include a number of non-scan related events. “FastScan Pattern Types” on page 6-9 describes the different types of patterns that FastScan generates. Different pattern types require different combinations of non-scan

events. Each combination of events defines a unique *event group*. Patterns with different event groups require different timing.

For example, assume that pattern 1 is a standard scan pattern that contains force_pi, measure_po, capture_clock_on, and capture_clock_off events. Also assume that pattern 2 is a transition pattern that contains init_force_pi, force_pi, measure_po, capture_clock_on, and capture_clock_off events. Because their events differ, patterns 1 and 2 require different timing definitions.

Often, different patterns share the same event group, in which case the patterns share the same timing information. However, regardless of whether or not patterns share event groups, you must define timing for all events in all patterns. You achieve this through a timing file containing *timeplate* commands.

Timing Files and Timeplate Commands

A timeplate command consists of a sequence of non-scan events and timing for each event. Timeplates define the timing component of waveforms for non-scan related event groups. Each event group requires its own timeplate. You define and name a timeplate for each event group within an external timing file.

After you construct the timing file, complete with the necessary timeplates, FastScan associates the proper timing with the patterns using the timeplates defined in this file. When you issue the Save Patterns command with the timing file argument, FastScan matches the patterns to the event groups specified in the timeplates and applies the proper timing.

FastScan tries to match the exact timeplate to an event group for a particular pattern. If such a timeplate does not exist, FastScan chooses another timeplate in the timing file that contains all events in the current pattern. A *super timeplate* contains a superset of the events of all other timeplates for the pattern set.

In the previous example, pattern 1 and 2 use different timeplates, although pattern 1 is a subset of pattern 2. If the timing file did not contain a timeplate for the pattern 1 event group, FastScan would use the timeplate defined for the pattern 2 event group because it contains all the events of pattern 1. Thus, the pattern 2 timeplate would be a super timeplate for the test pattern set of pattern 1 and pattern 2.

At a minimum, you need only specify the super timeplate for all non-scan event groups. FastScan requires a super timeplate when the test format you wish to write allows only a single timing definition.

Timeplate Syntax

The basic syntax of a timeplate is as follows:

TIMEPLATE "***timeplate_name***" =
    timeplate_statement...
  END;

The timeplate statements may include the following:

INIT_FORCE_PI ***time***

FORCE_PI ***time***

BIDI_FORCE_PI ***time***

SKEW_FORCE_PI ***{"pin_name"...} time***

WRITE_RAM_CLOCK_ON ***time***

WRITE_RAM_CLOCK_OFF ***time***

SKEW_WRITE_RAM_CLOCK_ON ***time***

SKEW_WRITE_RAM_CLOCK_OFF ***time***

MEASURE_PO ***time***

CAPTURE_CLOCK_ON ***time***

CAPTURE_CLOCK_OFF ***time***

SKEW_CAPTURE_CLOCK_ON ***pin_name time***

SKEW_CAPTURE_CLOCK_OFF ***pin_name time***

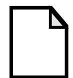
DUMMY_CLOCK_ON ***time***

DUMMY_CLOCK_OFF ***time***

SKEW_DUMMY_CLOCK_ON ***pin_name time***

SKEW_DUMMY_CLOCK_OFF ***pin_name time***

PERIOD ***time***


Note

The keywords in each timeplate statement must appear in uppercase. Also, the time argument must be either 0 or a positive integer.

Refer to the Timeplate timing command reference page in the *FastScan and FlexTest Reference Manual* for more information on this command and the statements it uses.

Timeplate Example

The following timing file example contains two timeplates:

```
TIMEPLATE "tp1" =
   FORCE_PI               0;
   MEASURE_PO           200;
   CAPTURE_CLOCK_ON     300;
   CAPTURE_CLOCK_OFF    400;
   PERIOD               500;
END;

TIMEPLATE "tp2" =
   FORCE_PI               0;
   MEASURE_PO           200;
   PERIOD               500;
END;
```

The first timeplate, “tp1”, defines timing for a basic pattern. The second timeplate, “tp2”, defines timing for a clock_po pattern. If you did not specify “tp2” in the timing file, FastScan would use the “tp1” timing, because “tp1” covers timing for all the required events.

Writing Default Timeplates

FastScan usually needs multiple timeplates to construct proper timing waveforms for the patterns it generates. And because FastScan automatically generates the pattern set, you may not know what kinds of timeplates you must provide. FastScan provides this information, specifying how many timeplates the generated patterns require, as well as what event groups must reside inside each timeplate. You can access this information, after running ATPG, by issuing the Write Timeplate command. The command's format is as follows:

WRIte TImeplate ***timing_filename*** [-Replace]

When this command executes, FastScan creates a default timing file—a file containing default timing values in each required timeplate. You can then change the default timing values to the real timing values, based on the requirements of your environment.

Editing Default Timeplate Values

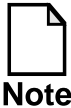
Default timing assigns the value of 0 to the first event, the value of 1 to the second event, and so on. In addition, the period value equals the number of event statements inside the timeplate.

For example, the following example shows the default timing FastScan generates for pattern 1 (discussed previously):

```
TIMEPLATE "tp1" =
   FORCE_PI             0;
   MEASURE_PO           1;
   CAPTURE_CLOCK_ON     2;
   CAPTURE_CLOCK_OFF    3;
   PERIOD               4;
END;
```

To edit the timeplate, you replace the default values with real timing values. For example, your edited timeplate may appear as follows:

```
TIMEPLATE “tp1” =
   FORCE_PI              0;
   MEASURE_PO          200;
   CAPTURE_CLOCK_ON    300;
   CAPTURE_CLOCK_OFF   500;
   PERIOD              600;
END;
```


Note

If any required timeplate is missing or incorrect, FastScan cannot generate the timing waveforms and issues an error message.

FlexTest Non-Scan Event Timing

In FlexTest, all primary inputs and primary outputs in non-scan related events (force_pi and measure_po) exhibit cycle-based behavior. Within the FlexTest ATPG session, you use the Set Test Cycle, Add Pin Constraints, and Add Pin Strobes commands to define this cycle behavior.

The Set Test Cycle command lets you specify the number of timeframes needed per test cycle. The Add Pin Constraints command lets you specify when in the test cycle the forces can occur and the waveform values allowed for each primary input. The Add Pin Strobes command lets you specify the strobe time for the primary outputs. For more information on these commands, refer to the *FastScan and FlexTest Reference Manual*.

The FlexTest application commands define basic timing for the events in the test cycle, so the tool can properly simulate the order of the events. However, the timing information you specify with the application commands does not include the real timing values that the tester requires. Thus, in conjunction with the application commands, you must specify the real timing information using an external *timing file*. The external timing file contains a number of statements that FlexTest reads and utilizes when it saves patterns with timing information.

For example, within a timing file you can define real timing values for input pin forces and output pin strobes by using the SET FORCE TIME and SET MEASURE TIME commands. Their usage lines are:

SET FORCE TIME ***time_value_list*****;**

SET MEASURE TIME ***time_value_list*****;**

The *time_value_list* argument consists of a set of time values indicating when in the test cycle the force or measure occurs. The number of time values in this list must equal the number of timeframes you set with the Set Test Cycle command.

Assume one test cycle contains four timeframes and the timing information file includes the following:

```
SET FORCE TIME 20 40 70 150;
SET MEASURE TIME 15 38 65 135;
```

Figure 7-3 shows the corresponding timing diagram.

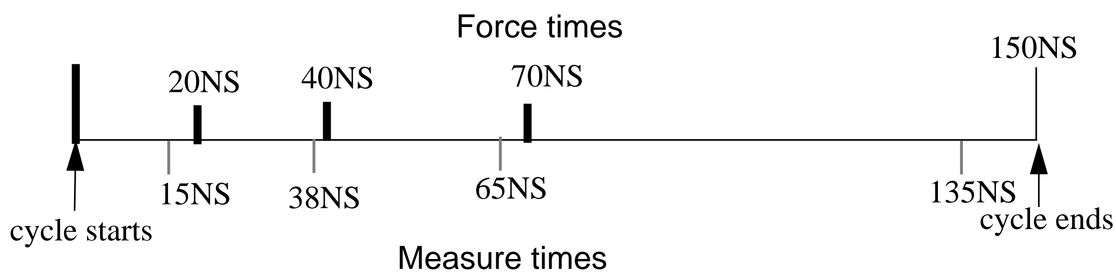


Figure 7-3. Timing for Non-Scan Events

The timing file can contain a number of additional timing-related commands. The Timing Command Dictionary within the *FastScan and FlexTest Reference Manual* summarizes and describes each of the timing-related commands you can use in this file.

Global Timing Issues in the Timing File

Regardless of which tool you use, either FastScan or FlexTest, you must specify the timing unit, scale, and test procedure file to use for pattern saving. The following subsections describe the timing commands you add to the timing file to accomplish these tasks.

Setting the Time Scale

You set the timing scale and unit by placing the SET TIME SCALE command in the timing file. Its usage line is as follows:

SET TIME SCALE ***number unit***;

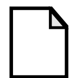
Number is the multiplying factor or scale for all times values. This number can be any real number value. The *unit* can be ns, ps, ms, or us. Once defined in the timing file, the tool applies the scale and unit to all time values in both the test procedure file and timing file.

Specifying the Timing-Modified Test Procedure File

If, after the ATPG process, you change the timing values in the test procedure file, you must specify the new test procedure file to use for pattern saving. You do this by placing the following SET PROCEDURE FILE command in the timing file. The command's usage line is as follows:

SET PROCEDURE FILE {***"scan_group_name" "test_proc_file"***}...

You must enclose both the scan_group_name and test_proc_file in double quotes. You can specify multiple test procedure files for multiple groups by repeatedly listing scan group names and their respective test procedure filenames. You can also specify multiple SET PROCEDURE FILE commands in the timing file. If you do not specify a scan group or test procedure file name, the tool uses the original test procedure file timing for all scan groups.


Note

If the tool encounters any mismatches (such as different event order or missing events) between the modified and original test procedure files, it issues an error and fails to generate the tester format patterns with timing.

Performing Timing Checks for Tester Formats

FastScan and FlexTest provide flexibility in specifying timing for the patterns they generate. For example, each input pin can have its own force times and each output pin its own strobe time.

However, most tester formats allow only one timing definition for each pin in one tester cycle. Moreover, certain tester formats impose other restrictions. "Saving the Patterns" on page 7-24 discusses the restrictions imposed by each of the different simulation and tester formats.

The test pattern formatter that FastScan and FlexTest use contains a timing rules checker to ensure that the timing definition you specified adheres to the constraints of the pattern format you wish to write.

For both FastScan and FlexTest, you create a timing file consisting of commands for this timing definition. Within this timing file, you can place a number of additional commands that enable the pattern formatter to perform specific rules checking. The timing rules checker ensures that the specified timing information meets certain tester format restrictions.

The following commands cause the timing rules checker to perform various timing checks:

- SET SINGLE_CYCLE TIME
- SET SPLIT_BIDI_CYCLE TIME
- SET END_MEASURE_CYCLE TIME
- SET SPLIT_MEASURE_CYCLE TIME
- SET STROBE_WINDOW TIME

For more information on these timing file commands, refer to the "Timing Command Dictionary" chapter in the *FastScan and FlexTest Reference Manual*.

Tester Format Restrictions for FastScan

Most tester formats supported by FastScan allow only a single timing definition for all non-scan event groups. Thus, FastScan pattern timing must adhere to the following rule:

- If there is a super timeplate in the timing file, the pattern formatter uses it. If not, the formatter tries to construct one. If it cannot construct a super timeplate for all the event groups in the pattern set, it will issue an error.

For example, assume a design contains RAMs and bidirectional pins. A super timeplate can specify timing that meets the previous rule. The following timing file contains four timeplates, timeplate "tp1", timeplate "tp2", timeplate "tp3", and super timeplate "tp4".

```
TIMEPLATE "tp1" =
   FORCE_PI                0;
   BIDI_FORCE_PI         100;
   WRITE_RAM_CLOCK_ON    200;
   WRITE_RAM_CLOCK_OFF   300;
   PERIOD               1000;
END;

TIMEPLATE "tp2" =
   FORCE_PI                0;
   BIDI_FORCE_PI         100;
   MEASURE_PO            400;
   CAPTURE_CLOCK_ON      500;
   CAPTURE_CLOCK_OFF     600;
   PERIOD               1000;
END;
TIMEPLATE "tp3" =
   FORCE_PI                0;
   BIDI_FORCE_PI         100;
   MEASURE_PO            400;
   PERIOD               1000;
END;

TIMEPLATE "tp4" =
   FORCE_PI                0;
   BIDI_FORCE_PI         100;
   WRITE_RAM_CLOCK_ON    200;
   WRITE_RAM_CLOCK_OFF   300;
```

```
    MEASURE_PO              400;
    CAPTURE_CLOCK_ON        500;
    CAPTURE_CLOCK_OFF       600;
    PERIOD                 1000;
END;
```

If the tester format you wish to write the pattern in requires a single timing definition, you need only specify “tp4” in the timing file. If you specified all of the timeplates, the formatter would pick the appropriate one to use as the single timing definition.

Tester Format Restrictions for FlexTest

Most tester formats supported by FlexTest allow only a single timing definition for all non-scan (primary) test cycles. Thus, FlexTest pattern timing must adhere to the following rule:

- The pulse width of return-type input pins (pins with R0, R1, SR0, or SR1 constraints) must be either less than or an integral multiple of the test cycle.

When the pulse width of the return-type pin exceeds the test cycle, the pattern formatter internally constructs the proper pin timing and reassigns timing to the return-type pin when it writes the patterns. This ensures that the pin displays non-return timing on the tester. The timing definition that follows illustrates this rule:

```
SET TEst Cycle 2;
ADD PIn Constraints CLK_A SR0 1 1 1;
ADD PIn Constraints CLK_B SR1 3 2 2;
```

The clock pin CLK_B has a period of 3 test cycles, an offset of 2 timeframes, and a pulse width of 2 timeframes. The test pattern formatter assigns this pin non-return timing that has a period of 1 and an offset of 0. The timing transformation that the formatter produces is called a *modified timing definition*. The test pattern formatter then writes this modified timing definition in the vendor-specific test pattern format.

Saving the Patterns

You can save patterns generated during the ATPG process both for timing simulation and use on the ATE. Once you create the proper timing file (as described in the preceding sections), FastScan and FlexTest use an internal test pattern data formatter to generate the patterns in the following formats:

- FastScan text format (ASCII)
- FlexTest text format (ASCII)
- FastScan binary format (FastScan only)
- Mentor Graphics Waveform DataBase (MGC WDB)
- Lsim test vectors
- TSSI Wave Generation Language (WGL)
- Binary TSSI WGL
- Verilog
- VHDL
- Zycad
- Compass Scan
- Texas Instruments Test Description Language (TDL 91)
- Fujitsu Test data Description Language (FTDL-E)
- Motorola Universal Test Interface Code (UTIC)
- Mitsubishi Test Description Language (MITDL)
- Toshiba Standard Tester interface Language 2 (TSTL2)
- LSI Logic Test Description Language (LSITDL)

Features of the Formatter

The main features of the test pattern data formatter include:

- Generating basic test pattern data formats: FastScan Text, FlexTest Text, MGC WDB, Lsim, Verilog, VHDL, TSSI WGL (ASCII and binary), and Zycad.
- Generating ASIC Vendor test data formats (with the purchase of the ASIC Vector Interfaces option): TDL 91, Compass, FTDL-E, UTIC, MITDL, TSTL2, and LSITDL.
- Supporting parallel load of scan cells (in MGC WDB and Verilog formats).
- Using a common timing definition file for all formats.
- Performing user-specified timing checks for many tester environments.
- Reading in external input patterns and output responses, and directly translating to one of the formats.
- Reading in external input patterns, performing good or faulty machine simulation to generate output responses, and then translating to any of the formats.
- Writing out just a subset of patterns in any test data format.
- Facilitating failure analysis by having the test data files cross-reference information between tester cycle numbers and FastScan/FlexTest pattern numbers.
- Supporting differential scan input pins for each simulation data format.

Pattern Formatting Issues

The following subsections describe issues you should understand regarding the test pattern formatter and pattern saving process.

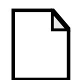
Parallel Scan Chain Loading

When you simulate test patterns, most of the time is spent loading and unloading the scan chain, as opposed to actually simulating the circuit response to a test pattern. To greatly reduce simulation time, you can directly (in parallel) load the simulation model with the necessary test pattern values. Parallel loading makes it practical for you to perform timing simulations for the entire pattern set in a reasonable time using popular simulators like QuickSim II and Verilog. Thus, you can use this method of parallel scan chain loading with the MGC WDB and Verilog formats.

You accomplish parallel loading through the scan input and scan output pins of scan *sub-chains* (a chain of one or more scan cells, modeled as a single library model) because these pins are unique to both the timing simulator model and the FastScan and FlexTest internal models. You can parallel load the scan chain by using force events in QuickSim II or Verilog to change the value of the scan input pin of each sub-chain.

After the parallel load, you apply the **shift** procedure a few times (depending on the number of scan cells in the longest subchain, but usually only once) to load the scan-in value into the sub-chains. Simulating the **shift** procedure only a few times can dramatically improve timing simulation performance. You can then observe the scan-out value at the scan output pin of each sub-chain.

Parallel loading ensures that all memory elements in the scan sub-chains achieve the same states as when serially loaded. Also, this technique is independent of the scan design style or type of scan cells the design uses. Moreover, when writing patterns using parallel loading, you do not have to specify the mapping of the memory elements in a sub-chain between the timing simulator and FastScan or FlexTest. And, this method does not constrain library model development for scan cells.


Note

When your design contains at least one stable-high scan cell, the **shift** procedure period must exceed the shift clock off time. If the **shift** procedure period is less than or equal to the shift clock off time, you may encounter timing violations during simulation. The test pattern formatter checks for this condition and issues an appropriate error message when it encounters a violation.

For example, the test pattern timing checker would issue an error message when reading in the following **shift** procedure:

```
proc shift =
   force_sci    0;
   measure_sco 1;
   force clk    1 2;  //force shift clock on
   force clk    0 3;  //force shift clock off
   period         3;  //period same as shift clock off time
end;
```

The error message would state:

```
// Error:  There is at least one stable high scan cell in the
design.  The shift procedure period must be greater than the
shift clock off time to avoid simulation timing violations.
```

The following modified **shift** procedure would pass timing rules checks:

```
proc shift =
   force_sci    0;
   measure_sco 1;
   force clk    1 2;  //force shift clock on
   force clk    0 3;  //force shift clock off
   period        4;  //period greater than shift clock off time
end;
```

Test Pattern Data Support for IDDQ

For best results, you should measure current after each non-scan cycle if doing so catches additional IDDQ faults. However, you can only measure current at specific places in the test pattern sequence, typically at the end of the test cycle boundary. To identify when IDDQ current measurement can occur, FastScan and FlexTest pattern files add the following command at the appropriate places:

```
measure IDDQ ALL;
```

Several ASIC test pattern data formats support IDDQ testing. There are special IDDQ measurement constructs in TDL 91(Texas Instruments), MITDL (Mitsubishi), UTIC (Motorola), TSTL2 (Toshiba), and FTDL-E (Fujitsu). The tools add these constructs to the test data files. All other formats (TSSI, Verilog,

VHDL, Compass, Lsim, MGC WDB, and LSITDL) represent these statements as comments.

Saving Patterns in Basic Test Data Formats

The Save Patterns usage lines for FastScan and FlexTest are as follows:

For FastScan

SAVe PAtterns ***filename*** [-Replace] [*format_switch*] [*timing_filename*] [-Parallel | -Serial] [-EXternal] [-BEgin {*pattern_number* | *pattern_name*}] [-END {*pattern_number* | *pattern_name*}] [-CEll_placement {Bottom | Top | None}] [-ENVironment] [-ALl_test | -CHain_test | -SCan_test] [-NOPadding] [-Noz] [-Map *mapping_file*]

For FlexTest

SAVe PAtterns ***filename*** [-Replace] [*format_switch*] [*timing_filename*] [-Parallel | -Serial] [-EXternal] [-BEgin *begin_number*] [-END *end_number*] [-CEll_placement {Bottom | Top | None}] [-ALl_test | -CHain_test | -CYcle_test] [-NOPadding] [-Noz]

For more information on this command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

The basic test data formats include FastScan text, FlexTest text, FastScan binary, MGC WDB, Verilog, VHDL, Lsim, TSSI WGL (ASCII and binary), and Zycad. The test pattern formatter can write any of these formats as part of the standard FastScan and FlexTest packages—you do not have to buy a separate option. You can use these formats for timing simulation.

FastScan Text

This is the default format that FastScan generates when you run the Save Patterns command. This is one of only two formats (the other being FastScan binary format) that FastScan can read back in, so you should generate a pattern file in either this or binary format to save intermediate results.

This format contains test pattern data in a text-based parallel format, along with pattern boundary specifications. The main pattern block calls the appropriate test

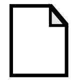
procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a basic FastScan text format file, enter the following at the application command line:

ATPG> **save patterns *filename* -ascii**

The formatter writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.


Note

This pattern format *does not* contain explicit timing information. Refer to the "Test Pattern File Formats" chapter in the *FastScan and FlexTest Reference Manual* for more information on this test pattern format.

FlexTest Text

This is the default format that FlexTest generates when you run the Save Patterns command. This is one of only two formats (the other being FlexTest table format) that FlexTest can read back in, so you should always generate a pattern file in this format to save intermediate results.

This format contains test pattern data in a text-based parallel format, along with cycle boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a FlexTest text format file, enter the following at the application command line:

ATPG> **save patterns *filename* -ascii**

The formatter writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Note

This pattern format *does not* contain explicit timing information. Refer to the “Test Pattern File Formats” chapter in the *FastScan and FlexTest Reference Manual* for more information on this test pattern format.

Comparing FastScan and FlexTest Text Formats with Other Test Data Formats

The FastScan and FlexTest text formats describe the contents of the test set in a human readable form. In many cases, you may find it useful to compare the contents of a simulation or test data format with that of the text format for debugging purposes. This section provides detailed information necessary for this task.

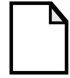
Often, the first cycle in a test set must perform certain tasks. The first test cycle in all test data formats turns off the clocks at all clock pins, drives Z on all bidirectional pins, drives an X on all other input pins, and disables measurement at any primary output pins.

The FastScan and FlexTest test pattern sets can contain two main parts: the *chain test* block, to detect faults in the scan chain, and the *scan test* or *cycle test* block, to detect other system faults.

The Chain Test Block

The chain test applies the **test_setup** procedure, followed by the **load_unload** procedure for loading scan chains, and the **load_unload** procedure again for unloading scan chains. Each **load_unload** procedure in turn calls the **shift** procedure. This operation typically loads a repeating pattern of “0011” into the chains. However, if scan chains with less than four cells exist, then the operation loads and unloads a repeating “01” pattern followed by a repeating “10” pattern. Also, when multiple scan chains in a group share a common scan input pin, the chain test process separately loads and unloads each of the scan chains with the repeating pattern to test them in sequence.

The test procedure file applies each event in a test procedure at the specified time. Each test procedure corresponds to one or more test cycles. Each test procedure can have a test cycle with a different timing definition. By default, all events use a timescale of 1000 ns.


Note

If you specify a capture clock with the FastScan Set Capture Clock command, the test pattern formatter does not produce the chain test block. For example, the formatter does not produce a chain test block for IEEE 1149.1 devices in which you specify a capture clock during FastScan setup.

The Scan Test Block (FastScan Only)

The scan test block in the FastScan pattern set starts with an application of the **test_setup** procedure. The scan test block contains several test patterns, each of which typically applies the **load_unload** procedure, forces the primary inputs, measures the primary outputs, and pulses a capture clock. The **load_unload** procedure translates to one or more test cycles. The force, measure, and clock pulse events in the pattern translate to the ATPG-generated capture cycle.

Each event has a sequence number within the test cycle. The sequence number's default time scale is 1000 ns. You can change the timing of the test cycle using the timing file.

You can split the ATPG cycle into two cycles to satisfy ASIC vendor timing constraints. You accomplish this by using the SET SPLIT_MEASURE_CYCLE TIME, and SET SPLIT_BIDI_CYCLE TIME commands in the timing file.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. Therefore the last pattern in the test set contains an extra application of the **load_unload** sequence.

More complex scan styles, like LSSD, use **master_observe** and **skewed_load** procedures in the pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may have several test cycles in it to operate the sequential scan controller. Some pattern types, like RAM sequential and clock sequential types, are more complex than the basic patterns. RAM sequential patterns involve multiple loads of the scan chains and multiple applications of the

RAM write clock. Clock sequential patterns involve multiple capture cycles after loading the scan chains. Another special type of pattern is the clock_po pattern. In these patterns, clocks may be held active throughout the test cycle and without applying capture clocks.

If the test data format supports only a single timing definition, FastScan cannot save both clock_po and non-clock_po patterns in one pattern set. This is so because the tester cannot reproduce one clock waveform that meets the requirements of both types of patterns. Each pattern type (combinational, clock_po, ram_sequential, clock_sequential) can have a separate timing definition.

The Cycle Test Block (FlexTest Only)

The cycle test block in the FlexTest pattern set also starts with an application of the **test_setup** procedure. This test pattern set consists of a sequence of scan operations and test cycles. The number of test cycles between scan operations can vary within the same test pattern set. A FlexTest pattern can be just a scan operation along with the subsequent test cycle, or a test cycle without a preceding scan operation. The scan operations use the **load_unload** procedure and the **master_observe** procedure for LSSD designs. The **load_unload** procedure translates to one or more test cycles.

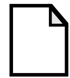
Using FlexTest, you can completely define the number of timeframes and the sequence of events in each test cycle. Each timeframe in a test cycle has a force event and a measure event. Therefore, each event in a test cycle has a sequence number associated with it. The sequence number's default time scale is 1000 ns. You can change the timing of the test cycle using the timing file.

You can split the ATPG cycle into two cycles to satisfy certain ASIC vendor timing constraints. You accomplish this by using the SET SPLIT_MEASURE_CYCLE TIME and SET SPLIT_BIDI_CYCLE TIME commands in the timing file.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may contain several test cycles that operate the sequential scan controller.

General Considerations

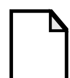
During a test procedure, you may leave many pins unspecified. Unspecified primary input pins retain their previous state. FlexTest does not measure unspecified primary output pins, nor does it drive (drive Z) or measure unspecified bidirectional pins. This prevents bus contention at bidirectional pins.


Note

If you run ATPG after setting pin constraints, you should also ensure that you set these pins to their constrained states at the end of the **test_setup** procedure. The Add Pin Constraints command constrains pins for the non-scan cycles, not the test procedures. If you do not properly constrain the pins within the **test_setup** procedure, the tool will do it for you, internally adding the extra force events after the **test_setup** procedure. This increases the period of the **test_setup** procedure by one time unit. This increased period can conflict with the test cycle period, potentially forcing you to re-run ATPG with the modified test procedure file.

All test data formats contain comment lines that indicate the beginning of each test block and each test pattern. You can use these comments to correlate the test data in the FastScan and FlexTest text formats with other test data formats.

These comment lines also contain the *cycle count* and the *loop count*, which help correlate tester pattern data with the original test pattern data. The cycle count represents the number of test cycles, with the shift sequence counted as one cycle. The loop count represents the number of all test cycles, including the shift cycles. The cycle count is useful if the tester has a separate memory buffer for scan patterns, otherwise the loop count is more relevant.


Note

The cycle count and loop count contain information for all test cycles—including the test cycles corresponding to test procedures. You can use this information to correlate tester failures to a FastScan pattern or FlexTest cycle for fault diagnosis.

FastScan Binary (FastScan Only)

This format contains test pattern data in a binary parallel format, which is the only format (other than FastScan text) that FastScan can read. A file generated in this

format contains the same information as FastScan text, but uses a condensed form. You should use this format for archival purposes or when storing intermediate results for very large designs.

To create a FastScan binary format file, enter the following at the FastScan command line:

ATPG> **save patterns *filename* -binary**

FastScan writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Mentor Graphics WDB

The Mentor Graphics Waveform Database (MGC WDB) format contains test pattern data and timing information in a binary waveform database format, which QuickSim II, QuickFault, and other Mentor Graphics design analysis tools can read. In this format, you can write the patterns to load scan cells either serially or in parallel. You can also specify timing information in a timing file, otherwise the tools use default timing.

To create a basic file set in MGC WDB format, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] [-Parallel | -Serial] -MGcwdb

FastScan and FlexTest write test data as input (*filename*_in) and expected output (*filename*_out) waveform databases. Each database consists of three files: a pattern data file, a header file, and an attribute file. In addition, the tools generate a QuickSim II dofile (*filename.do*) which loads appropriate waveform databases, defines input and output pins, runs the simulator, compares the output waveforms with the expected output waveforms, and prints out a report containing information about miscompares. The last generated file is an index file (*filename.index*) used to correlate the beginning of each pattern with a simulation time. Each waveform database contains waveforms, which are time-ordered sequences of events. MGC WDB, because it is event-based, supports all timing definitions that FastScan and FlexTest support.

You must specify a name for the WDB file set into which FastScan or FlexTest writes the complete test data, in either serial or parallel format, using timing data from the specified timing file.

For example, to save your patterns in parallel format to a file called *pat_wdb* in the directory *wdb.test,* using a timing file called *timefile*, and printing out a directory listing of the resulting files, you would enter the following:

ATPG> **save patterns wdb.test/pat_wdb timefile -parallel -mgcwdb**

ATPG> system ls ./wdb.test

```
pat_wdb.do                   pat_wdb_in.wdb_1
pat_wdb.index                pat_wdb_out.Svdm_svdb.attr
pat_wdb_in.Svdm_svdb.attrpat_wdb_out.dat_1
pat_wdb_in.dat_1             pat_wdb_out.wdb_1
```

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the MGC WDB format, refer to the *Waveform Dataport Programmer's Guide*, available through Mentor Graphics.

Verilog

This format contains test pattern data and timing information in a text-based format readable by both the Verilog and Verifault simulators. This format also supports both serial and parallel loading of scan cells. You can specify timing information in a timing file, otherwise the tools use default timing. The Verilog format supports all FastScan and FlexTest timing definitions, because Verilog stimulus is a sequence of timed events.

To generate a basic Verilog format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] [-Parallel | -Serial] -Verilog

The Verilog pattern file contains procedures to apply the test patterns, compare expected output with simulated output, and print out a report containing information about failing comparisons. The tools write all patterns and comparison functions into one main file (*filename*), while writing the primary

output names in another file (*filename.po.name*). If you choose parallel loading, they also write the names of the scan output pins of each scan sub-chain of each scan chain in separate files, for example, *filename.chain1.name*. This allows the tools to report output pins that have discrepancies between the expected and simulated outputs. You can enhance the Verilog testbench with Standard Delay Format (SDF) back-annotation.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the Verilog format, refer to the *Verilog-XL Reference Manual*, available through Cadence Design Systems.

VHDL

The VHDL interface supports both a serial and parallel test bench.

SAVe PAtterns ***filename*** [*timing_filename*] [-Parallel | -Serial] -Vhdl

The serial test bench uses only the VHDL language in a single test bench file, and therefore should be simulator independent. The parallel test bench consists of two files, one being a VHDL language test bench, and one being a QuickHDL dofile containing QuickHDL and TCL commands. The QuickHDL dofile is used to force and examine values on the internal scan cells. Because of this, the parallel test bench is not simulator independent.

The serial test bench is almost identical to the Verilog serial test bench. It consists of a a top level module which declares an input bus, an output bus, and an expected output bus. The module also instantiates the device under test and connects these buses to the device. The rest of the test bench then consists of assignment statements to the input bus, and calls to a compare procedure to check the results of the output bus.

The parallel test bench is similar to the serial test bench in how it applies patterns to the primary inputs and observes results from the primary outputs. However, the VHDL language does not support at this time anyway to force and observe values on internal nodes below the top level of hierarchy. Because of this, it is necessary to create a second file which is a simulator specific dofile which uses simulator commands to force and observe values on the internal scan cell. This dofile runs in

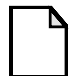
sync with the test bench file by using run commands to simulate the test bench and device under test for certain time periods.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Lsim

Lsim is a popular Mentor Graphics simulator, commonly used to analyze custom-designed integrated circuits. The Lsim test vector format consists of a simulation trace file that contains all the input and output pin values for each time at which a pin changes. Currently, FastScan and Flextest only support the Lsim serial test vector format, which, for large designs, can lead to large test data files.

The test pattern data files contain timing information. You can either specify timing using a timing file, or use default timing. You can use the Verify command in Lsim to read in the test vector file and compare the expected output values with the simulated output values.


Note

Lsim does not allow two traces corresponding to the same timestamp. Instead, Lsim test vectors are a sequence of traces at each timestamp. Thus, Lsim test pattern format supports all the timing definitions that FastScan and FlexTest support. Other simulators, such as Powermill, also use the Lsim trace format.

To generate a basic Lsim format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -Serial -LSIM

FastScan or FlexTest writes the complete test data to the file named *filename*, in serial format, using timing data from the specified timing file.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the Lsim test vector format, refer to the Mentor Graphics *Explorer Lsim Reference Manual*.

TSSI Wave Generation Language (ASCII)

The TSSI WGL format contains test pattern data and timing information in a structured text-based format. You can translate this format into a variety of simulation and tester environments, but you must first read it into the TSSI Waveform database and use the appropriate TSSI translator. This format supports both serial and parallel loading of scan cells.

You can either specify timing information in a timing file, or use default timing. The TSSI WGL format supports all FastScan and FlexTest timing definitions, because this format represents test patterns as sequences of cycles, with each cycle having its own timing definition. By default, they use a separate timing definition for each test procedure and for the capture cycle. However, it is possible to produce a TSSI WGL file containing a single timing definition by using the SET SINGLE_CYCLE TIME, SET SPLIT_MEASURE_CYCLE TIME, or the SET SPLIT_BIDI_CYCLE TIME timing commands.

Some test data flows verify patterns by translating TSSI WGL (via Summit Design WGL-simulation translators) to stimulus and response files for use by the chip foundry's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure the following: 1) there is only one scan cell for each DFT library model (also called a scan subchain), 2) the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator (because the scan cell names in the ATPG model appear in the scan section of the parallel TSSI WGL output), 3) the scan-in and scan-out pin names of all scan cells are the same.

To generate a basic TSSI WGL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] [-Parallel | -Serial] -TSSIWgl

FastScan or FlexTest writes the complete test data to the file *filename*, in either serial or parallel format, using timing data from the specified timing file.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the TSSI WGL format, refer to the *TDS Software System WDB Tool Kit*, available through Summit Design, Inc.

TSSI Wave Generation Language (Binary)

The TSSI WGL binary format contains the same test pattern data and timing information as ASCII TSSI WGL format. However, the binary format has the following advantages:

- Compact parallel and scan pattern descriptions
- Platform-independent binary coding
- Faster writing/parsing times
- No scan state definition block
- Scan "in-line" with parallel vectors rather than indirectly pre-declared
- Upwardly compatible

To generate a basic TSSI WGL binary format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] [-Parallel | -Serial] -TSSIBinwgl

When you specify the -tssibinwgl switch, FastScan or FlexTest writes the entire "pattern" section of the WGL file in both a structured text-based format named *filename* and in binary format in a separate file named *filename.patternbin.*

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the TSSI WGL format, refer to the *Binary Waveform Generation Language External Specification*, available through Summit Design, Inc.

Zycad

You can use Zycad format patterns to verify ATPG patterns on the Zycad hardware-accelerated timing and fault simulator. Zycad patterns do not have any special constructs for scan. You can either specify timing information in a timing file, or use default timing. Currently, the test pattern formatter creates only serial format Zycad patterns.

Zycad patterns consist of two sections: the first section defines all design pins, and the second section defines all pin values at any time in which at least one pin changes.

To generate a basic Zycad format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -serial **-**Zycad

FastScan and FlexTest produce two files in the Zycad format, one for the fault simulator (*filename.fault.sen*) and the other for the timing simulator (*filename.assert.sen*).

A comment line in Zycad format includes the pattern number, cycle number, and loop number information of a pattern. At the user's request, the simulation time is also provided in the comment line:

    # Pattern 0 Cycle 1 Loop 1 Simulation time 500

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Saving in ASIC Vendor Data Formats

The ASIC vendor test data formats include Texas Instruments TDL 91, Compass Scan, Fujitsu FTDL-E, Motorola UTIC, Mitsubishi MITDL, Toshiba TSTL2, and LSI Logic LSITDL. The ASIC vendor's chip testers use these formats. If you purchased the ASICVector Interfaces option to FastScan or FlexTest, you have access to these formats.

All the ASIC vendor data formats are text-based and load data into scan cells in a parallel manner. Also, ASIC vendors usually impose several restrictions on pattern timing. Most ASIC vendor pattern formats support only a single timing definition. Refer to your ASIC vendor for test pattern formatting and other requirements.

The following subsections briefly describe the ASIC vendor pattern formats and give sample timing files for each.

TI TDL 91

This format contains test pattern data in a text-based format. You can either specify timing information in a timing file, or use default timing.

Currently, FastScan and FlexTest support features of TDL 91 version 3.0 only. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. TI's ASIC division imposes the additional restriction that comparison should always be done at the end of a tester cycle.

You must ensure that all the non-scan cycle timing and test procedures have compatible timing. The SET SINGLE_CYCLE TIME command ensures that one timing definition represents all non-scan and scan cycle timing. It does this by splitting the non-scan cycle into two pieces at measurement time. The SET SPLIT_MEASURE_CYCLE TIME and SET END_MEASURE_CYCLE TIME commands ensure that output measurements occur only at the end of a tester cycle. If you do not check for compatible timing, the resulting test data may have incorrect timing.

To generate a basic TI TDL 91 format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -TItdl

The formatter writes the complete test data to the file *filename*, using timing data from the specified timing file. It also writes the chain test to another file (*filename.chain*) for separate use during the TI ASIC flow.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Example TI TDL 91 Timing Definition File

The following is a typical FastScan timing definition file that creates a tester cycle of 500ns. In this example, the default period is 1000ns, but the SET SPLIT_MEASURE_CYCLE TIME command splits the non-scan cycle in two at 500ns to ensure output measurement at the end of the test cycle.

```
set time scale 1 ns;
Timeplate "tp0" =
   force_pi          2;
   bidi_force_pi     100;
   measure_po        490;
   capture_clock_on  600;
   capture_clock_off 700;
   period            1000;
end;
set split_measure_cycle time 500;
set procedure file "g1" "split_measure.g1";
```

The following example shows equivalent FlexTest timing commands and timing definition file.

```
set test cycle 2;
setup pin constraints NR 1 0;
add pin constraints SR0 CLK 1 1 1;
setup pin strobes 1;

set time scale 1 ns;
set split_measure_cycle time 500;
set force time 600 700;
set bidi_force time 100 625;
set measure time 490 650;
set first_force time 2;
set cycle time 1000;
set procedure file "g1" "split_measure.g1";
```

The following example shows the compatible “split_measure.g1” file.

```
proc shift =
```

```
      measure_sco     0;
      force_sci       2;
      force CLK       1 100;
      force CLK       0 200;
      period          500;
   end;
   proc load_unload =
      force SE        1 2;
      force CLEAR     1 100;
      force CLK       0 100;
      apply shift     10 500;
      period          500;
   end;
```

Compass Scan

This format contains test pattern data in a text-based format. You can either specify timing information in a timing file, or use default timing.

This format supports only single scan chains and a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. You must ensure that all the non-scan cycle timing and the test procedures have compatible timing. The SET SINGLE_CYCLE TIME command ensures that one timing definition represents all non-scan and scan cycle timing. If you do not check for compatible timing, the resulting test data may have incorrect timing.

To generate a basic Compass format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -Compass

The formatter writes test pattern data into the following files:

- The block map file (*filename.tbm*).
- The entry file (*filename_entry.vif*), to denote the **load** procedure.
- The exit file (*filename_exit.vif*), for specifying the **unload** procedure.
- The scan I/O file (*filename_sio.vif*), to denote non-scan vectors.

- The scan in file (*filename_si.trc*), to denote scan in patterns.
- The scan out file (*_so.trc*), to denote scan out patterns.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on the Compass Scan format, refer to the *Vector Reference Manual*, available through Compass Design Automation.

Example Compass Timing Definition File

The following is a typical FastScan timing definition file that creates a tester cycle of 1000ns.

```
set time scale 1 ns;
Timeplate "tp0" =
   force_pi          2;
   bidi_force_pi     100;
   measure_po        490;
   capture_clock_on  600;
   capture_clock_off 700;
   period            1000;
end;
set single_cycle time 1000;
set procedure file "g1" "one.g1";
```

The following example shows equivalent FlexTest timing commands and timing definition file.

```
set test cycle 2;
setup pin constraints NR 1 0;
add pin constraints SR0 CLK 1 1 1;
setup pin strobes 1;

set time scale 1 ns;
set single_cycle time 1000;
set force time 600 700;
set bidi_force time 100 625;
set measure time 490 650;
set first_force time 2;
set cycle time 1000;
set procedure file "g1" "one.g1";
```

The following example shows the compatible “one.g1” file.

```
proc shift =
   force_sci        2;
   measure_sco      490;
   force CLK        1 600;
   force CLK        0 700;
   period           1000;
end;
proc load_unload =
   force SE         1 2;
   force CLEAR      1 600;
   force CLK        0 600;
   apply shift      10 1000;
   period           1000;
end;
```

Fujitsu FTDL-E

This format contains test pattern data in a text-based format. You can either specify timing information in a timing file, or use default timing.

The Fujitsu FTDL-E format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. You must ensure that all the non-scan cycle timing and

test procedures have compatible timing. The SET SINGLE_CYCLE TIME command ensures that one timing definition represents all non-scan and scan cycle timing. If you do not check for compatible timing, the resulting test data may have incorrect timing.

The FTDL-E format splits test data into patterns that measures 1 or 0 values, and patterns that measures Z values. The test patterns divide into test blocks that each contain 64K tester cycles.

To generate a basic FTDL-E format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -Fjtdl

The formatter writes the complete test data to the file named *filename.fjtdl.func,* using timing data from the specified timing file. If the test pattern set contains IDDQ measurements, the formatter creates a separate DC parametric test block in a file named *filename.ftjtl.dc*.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

You can also use the Compass Scan or TI TDL 91 format timing definition files to generate MITDL patterns. Refer to the “Compass Scan” section for more details.

For more information on the Fujitsu FTDL-E format, refer to the *FTDL-E User's Manual for CMOS Channel-less Gate Array*, available through Fujitsu Microelectronics.

Motorola UTIC

This format contains test pattern data in a text-based format. You can either specify timing information in a timing file, or use default timing.

This format supports multiple scan chains, but allows only two timing definitions. One timing definition is for scan shift cycles and one is for all other cycles. When saving patterns, the formatter does not check the **shift** procedure for timing rules. You must ensure that all the non-scan cycle timing and the test procedures (except for the **shift** procedure) have compatible timing. This format also supports the use of differential scan pins.

Additionally, Motorola's ASIC division requires that you force bidirectional pins in a tester cycle after forcing other non-return input pins. The SET SPLIT_BIDI_CYCLE TIME command ensures the force of all non-return input pins before the split_bidi_cycle time and the force of all bidirectional pins after this time. This command also ensures one timing definition represents all scan and non-scan cycle timing. Motorola ASIC also requires that all outputs be stable for at least 30ns. You can ensure this is the case by using the Set Strobe_window check.

Because UTIC supports only two timing definitions, one for the shift cycle and one for all other test cycles, all test cycles except the shift cycle must use the timing of the main capture cycle. If you do not check for compatible timing, the resulting test data may have incorrect timing.

To generate a basic Motorola UTIC format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -Utic

The formatter writes the complete test data to the file named *filename* using timing data from the specified timing file.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

Some test data verification flows do pattern verification by translating UTIC (via Motorola ASIC tools) into stimulus and response files for use by the chip factory's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure that the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator. This is because the scan cell names in the ATPG model appear in the scan section of the parallel UTIC output.

For more information on the Motorola UTIC format, refer to the *Universal Test Interface Code Language Description*, available through Motorola Semiconductor Products Sector.

Example UTIC Timing Definition File

The following is a typical FastScan timing definition file that creates a tester cycle of 500ns. In this case, the non-scan cycle is split into two at 500ns to ensure output measurement at the end of the test cycle.

```
set time scale 1 ns;
Timeplate "tp0" =
   force_pi          2;
   bidi_force_pi     525;
   measure_po        550;
   capture_clock_on  600;
   capture_clock_off 700;
   period            1000;
end;
set split_bidi_cycle time 500;
set strobe_window time 30;
set procedure file "g1" "split_bidi.g1";
```

The following example shows equivalent FlexTest timing commands and timing definition file.

```
set test cycle 2;
setup pin constraints NR 1 0;
add pin constraints SR0 CLK 1 1 1;
setup pin strobes 1;

set time scale 1 ns;
set split_bidi_cycle time 500;
set force time 600 700;
set bidi_force time 525 625;
set measure time 550 650;
set first_force time 2;
set cycle time 1000;
set procedure file "g1" "split_bidi.g1";
```

The following example shows the compatible “split_measure.g1” file.

```
proc shift =
   force_sci       2;
   measure_sco     50;
   force CLK       1 100;
```

```
      force CLK         0 200;
      period            500;
   end;
   proc load_unload =
      force SE          1 2;
      force CLEAR       1 100;
      force CLK         0 100;
      apply shift       10 500;
      period            500;
   end;
```

Mitsubishi TDL

This format contains test pattern data in a text-based format. You can either specify timing information in a timing file, or use default timing.

This format supports multiple scan chains, as well as multiple timing definitions. You can use the SET SINGLE_CYCLE TIME or the SET SPLIT_MEASURE_CYCLE TIME command to create a MITDL format file that uses only a single timing definition.

To generate a basic Mitsubishi TDL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -MItdl

The formatter represents all scan data in a parallel format. It writes the test data into two files: the program file (*filename.td0*), which contains all pin definitions, timing definitions, and scan chain definitions; and the test data file (*filename.td1*), which contains the actual test vector data in a parallel format. You can also use the Compass Scan or TI TDL 91 format timing definition files to generate MITDL patterns. Refer to the “Compass Scan” section for more details.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information on Mitsubishi's TDL format, refer to the *TD File Format* document, which Hiroshi Tanaka produces at Mitsubishi Electric Corporation.

Toshiba TSTL2

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. You can either specify timing information in a timing file, or use default timing.

This format supports multiple scan chains, but allows only a single timing definition for all test cycles. TSTL2 represents all scan data in a parallel format.

You can use the SET SINGLE_CYCLE TIME or the SET SPLIT_BIDI_CYCLE TIME command to create a TSTL2 format file which uses only a single timing definition. The SET SPLIT_BIDI_CYCLE TIME command ensures that bidirectional pins and input pins change in different cycles to prevent transient bus contention.

To generate a basic Toshiba TSTL2 format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -TSTl2

The formatter writes the complete test data to the file named *filename* using timing data from the specified timing file. You can use the Compass Scan format or Motorola UTIC timing definition files for generating Toshiba TSTL2 patterns.

For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

For more information about the Toshiba TSTL2 format, refer to *Toshiba ASIC Design Manual TDL, TSTL2, ROM data*, (document ID: EJFB2AA), available through the Toshiba Corporation.

LSI Logic LSITDL

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. You can either specify timing information in a timing file, or use default timing. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. LSITDL represents all scan data in a parallel format.

To generate an basic LSITDL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PAtterns ***filename*** [*timing_filename*] -LSITdl -map [mapping_file]

The LSITDL format generates 7 files: *filename.apat1s* (primary input data), *filename.bpat1s* (parallel scan chain loading data for master memory elements), *filename.cpatls* (parallel scan chain loading data for non-master memory elements), *filename.vpats000* (expected primary output data), *filename.vpats001* (expected scan output data), *filename.tifends* (scan chain and cell inversion data), and *filename.scl1s* (simulation control file for parallel loading). Because the LSITDL format requires a fixed number of cycles between consecutive scan loads, the formatter automatically pads the test data such that the number of cycles between two consecutive scan loads is always the same. FastScan uses only one cycle to measure the primary output. FlexTest uses all cycles to measure the primary output.

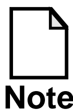
For more information on the Save Patterns command and its options, see Save Patterns in the *FastScan and FlexTest Reference Manual*.

The LSITDL design flow for verification and translation into ATE patterns is as follows: First, the LSI Logic LSIM golden simulator simulates the LSITDL patterns. Next the Simulation_Comparator compares the actual outputs with the expected outputs. Then the Test_Extractor translates the LSIM trace outputs into ATE patterns. These tools always compare at the end of a cycle, so you should use the SET SPLIT_MEASURE_CYCLE TIME command in this flow.

Some test data verification flows perform pattern verification by translating UTIC (via Motorola ASIC tools) into stimulus and response files for use by the chip factory's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure that the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator. This is because the scan cell names in the ATPG model appear in the scan section of the parallel UTIC output.

During translation, the tool uses a parallel loading scheme that also uses memory-to-memory mapping at the scan cell level. For this reason, you should have only one scan cell in your scan library models. The tool implements parallel loading using special internal pins with special names in the LSI Logic LSIM simulation model. If you wish to create user-specific scan models, you must name the internal node used for parallel loading with the default pin name used for other scan cells.

You should be also careful in defining timing for LSITDL patterns, so as to prevent bus contention. You should adopt the LSI Logic design approach to preventing transient bus contention at pins by disabling tri-state drivers until all other pins and scan cells change. The example shown in this section illustrates this approach.


Note

The Simulation_Comparator may give false warnings of bus contention when multiple drivers drive a bus with the same value.

On the other hand, the scan design rules checker in the Mentor Graphics ATPG tools performs a simulation that is more accurate than the LSI Logic parallel loading scheme. In particular, the LSI Logic LSIM may not accurately simulate non-scan memory elements that behave as constant 0 or 1 generators or transparent latches during scan loading. Typically, the flattened model FastScan creates for rules checking contains these types of gates. To work around the simulation limitation, you can set the pattern type to scan sequential with a depth of 2 (Set Simulation Mode combinational -depth 2) prior to rules checking. Doing so removes these gate types from the simulation model. After rules checking you can then set the pattern type back to combinational if you desire. The example at the end of this section demonstrates this technique.

Another limitation of the Test_Extractor tool is that if the overall inversion polarity of a scan chain from scan input pin to the scan output pin is odd, the result is incorrect final ATE scan patterns. You can work around this problem by adding a +INVERT statement to the scan input SCANPORT statement in the *pattern_name.tifends* file. The Test_Extractor tool has another limitation if there are multiple scan chains operating in parallel with separate scan clocks. The tool generates extra shift cycles while generating serial patterns for ATE. You can

work around this problem by specifying only one scan clock with each scan chain in the *<pattern_name>.tifends* file.

Handling Parallel Load in the C-MDE Environment

DFT ATPG tools group memory elements on a scan chain into scan cells according to the shift procedure provided by the user. This "grouping" can result in a scan cell with multiple memory elements.

DFT's method of parallel loading of a scan chain is to apply appropriate values at the scan subchain input and then apply one or more shift procedures. All of the memory elements on a scan chain can be loaded with desired values after the parallel loading.

LSI Logic parallel loading uses a different approach. In the C-MDE environment, no shift clock is applied for parallel loading. A desired logic value is loaded directly into the output of a memory element of a scan chain (by using set point, the s2(a), s3(a), etc. internal pins of the logic model). In the current DFT LSITDL implementation, a desired logic value is always loaded into the last memory element of a scan cell, if there is more than one memory elements in a scan cell. If a scan cell has more than one memory elements, only the last memory element of the scan cell will be loaded with desired logic value while the logic values on the other memory elements of the scan cell will be unknown after the parallel loading in C-MDE environment. This is the source of mismatches of DFT LSITDL patterns in C-MDE simulation.

To alleviate this problem, desired logic values can be loaded directly to the output of all memory elements of a design by force appropriate set points of these memory element library cells in C-MDE simulation to achieve the same logic state of the design as serial scan chain loading.

The desired values of master gates of scan cells can be provided in *.bpat* file while the desired values of other memory elements of scan cells (copies, slaves, shadows, and extras) can be provided in a *.cpat* file.

For illustration purpose, the concept of observable gate of a scan cell is introduced here. If a scan cell has a slave gate, the observable gate of that scan cell is the slave gate. If a scan cell doesn't have a slave gate, but has a copy gate, the copy

gate is the observable gate of the scan cell; Otherwise, the observable gate is the master gate of the scan cell.

Expected logic values on the outputs of all observable gates after capturing (and application of observe procedures) will be provided in a *.vpat001* file for simulation comparison.

In the C-MDE simulation control file, *.scl* file, instructions will be given to save logic values on all observable gates during C-MDE simulation for generating ATE program.

For each scan chain, inversion information between scan input and first observable gate, adjacent observable gates, the last observable gate and the scan output will be provided in a *.tifend* file for generating ATE program.

A mapping file is required to save LSITDL format pattern file from DFT ATPG tools. The mapping file provides names of set point(s) and observe point associated with each memory library cells used in the design. LSI Logic should provide mapping files to their customers.

The command for saving LSITDL format pattern file from Mentor ATPG tools will be enhanced to:

save patterns <filename> [timing-file] -lsitdl **-map <mapping-file>**

The syntax for mapping file is provided below:

LSITDL Mapping File Syntax

A line starting with a "#" character is a comment line.

One line can hold at most one library cell mapping information.

For a edge triggered memory element library cell, first field is the cell name, second field is the name of the observe point, third field is the name of the set point associated with low clock level (0), and the fourth field of the name of the set point associated with the high clock level (1).

For a level sensitive memory element library cell, first field is the cell name, second field is the name of the observe point, and the third field is the name of the set point.

The syntax of a mapping file looks like this:

```
#cell-name   observe-point   set-point1   set-point2
fd1          q(z)            s2(a)        m3(a)    <return>
fd2          q(z)            s2(a)        m3(a)     <return>
latch        q(z)            s2(a)                  <return>
<EOF>
```

After scan chain loading, DFT ATPG tools identify some nonscan memory elements as conditional/unconditional transparent latches, tie0s, or tie1s. The states of all other nonscan memory elements are considered unknown for ATPG. To achieve the same state in CMDE simulation, tie0 and tie0 nonscan memory elements will be set to strong tie1 and tie0 in *.scl* file while all other nonscan memory elements will be set to initial tieX in *.scl* file.

In order for Mentor ATPG tools to provide correct logic values to be loaded and to be observed on memory elements, following requirements must satisfy.

1. No library cell can have more than one ATPG memory element primitive. For example, in the CMDE lcb300k.lib, library cell fd1x4 has 4 of fd1s. This library cell should be replaced by four individual fd1s when using Mentor ATPG tools.

2. A library cell which has a level sensitive memory element primitive must have exact one set point.

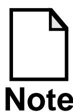
3. A library cell which has a edge triggered memory element should have two set points associated with the two clock levels (0 and 1). When loading logic value to a memory element library cell, appropriate set point will be used according to the current clock logic level.

4. There should be no inversion between the output of a ATPG memory element primitive and the state of its library cell which is determined by it's set point.

For more information on the LSITDL format, refer to *LSI Logic Chip Level Full Scan Design Methodology Guide* or the *CMDE TestBuilder Reference Manual*, available from LSI Logic Corporation.

Example LSITDL Timing Definition

The following example illustrates the FastScan and FlexTest dofiles, test procedure files, test procedure files with timing, and the timing files that are compatible with the LSI Logic design approach.


Note

The P_SCANTRIN pin disables the tri-state-capable output pins.

The following is the FastScan dofile:

```
add clocks 0 p_clk
add clocks 0 p_resetn
add write controls 0 p_clk
add scan group g1 l1a6760.g1
add scan chain c1 g1 p_scanin p_scanout
set clockpo patterns off
set contention check capture_clock -atpg
set simulation mode combinational -depth 2
set sys mode atpg
set atpg compression on
set simulation mode combinational
add fault -all
run
compress pattern 16
save pat patterns_fst/l1a6760.pat -re
save pat patterns_fst/l1a6760.lsitdl l1a6760.fst.time -lsitdl
save pat patterns_fst/l1a6760.tssi.par l1a6760.fst.time -tssiw
save pat patterns_fst/l1a6760.vp l1a6760.fst.time -verilog
save pat patterns_fst/l1a6760.wdb.par l1a6760.fst.tim -mgcwdb
```

The following is the FlexTest dofile:

```
add clocks 0 p_clk
add clocks 0 p_resetn
add write controls 0 p_clk
add scan group g1 l1a6760.g1
add scan chain c1 g1 p_scanin p_scanout
set test cycle 2
add pin constraints p_clk sr0 1 1 1
add pin constraints p_resetn sr0 1 1 1
set contention check -bus -atpg
set sys mode atpg
add fault -all
run
save pat patterns_flx/l1a6760.pat -re
save pat patterns_flx/l1a6760.lsitdl l1a6760.flx.time -lsitdl
save pat patterns_flx/l1a6760.tssi.par l1a6760.flx.time -tssiw
save pat patterns_flx/l1a6760.vp l1a6760.flx.time -verilog
```

The following is the test procedure file with timing:

```
proc shift =
   measure_sco             0;
   force_sci               0;
   force p_clk          1  200;
   force p_clk          0  400;
   period                  1000;
end;
proc load_unload =
   force p_scanen     0    0;
   force p_resetn     0    0;
   force p_clk        0    0;
   force p_scantrin   0    500;
   apply shift        8    1000;
end;
```

The following is the FastScan timing file:

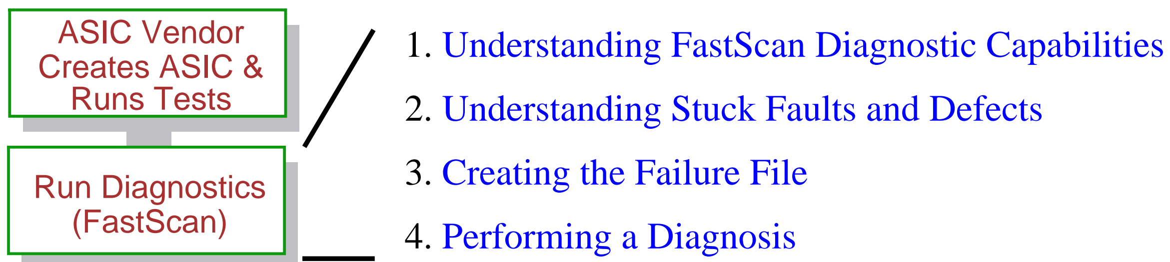
```
set time scale       1nS;
Timeplate "tp0" =
    force_pi                      0;
    skew_force_pi "P_SCANTRIN"  500;
    measure_po                  950;
    capture_clock_on            1200;
    capture_clock_off           1400;
    period                      2000;
end;
set split_measure_cycle time 1000;
set procedure file "g1" "l1a6760.g1.time";
```

The following is the equivalent FlexTest timing file:

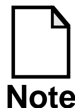
```
set time scale       1nS;
set force time                 1200 1400;
set skew_force time "P_SCANTRIN" 500  1300;
set measure time               950  1350;
set cycle time                      2000;
set split_measure_cycle time        1000;
set procedure file "g1" "l1a6760.g1.time";
```

Chapter 8
Running Diagnostics

This chapter discusses running chip failure diagnostics, as shown in the following outline:



You can use FastScan to diagnose chip failures during the ASIC testing process.



FlexTest does *not* provide this capability.

Understanding FastScan Diagnostic Capabilities

In the test process, you run FastScan on a design to create a test pattern set. You then use ATE to run the same patterns on the fabricated chip. If the chip is good, it passes the test set. If the chip is faulty, it fails one or more patterns in the test set, and you will probably want to know why. Although these chips are not repairable, the information that fault diagnosis provides could help you find manufacturing yield and quality problems and prevent their recurrence.

You can use fault diagnosis on chips that fail during the application of the scan test patterns to identify the precise location of a fault, given the actual response of a faulty circuit to a test pattern set.

You perform a diagnosis by first collecting the full set of failing pattern data from the tester. FastScan utilizes this data during fault simulation to determine the set of faults whose simulated failures most closely match the actual failures. The more data (failing patterns) it has to draw from, the more accurate the diagnosis. Thus, if you intend to perform fault diagnosis, you should not compress the pattern set when you run ATPG with FastScan.

Compared to the standard fault dictionary approach, post-test fault simulation (which considers all failing patterns) not only improves precision but also provides the capability to diagnose non-stuck fault defects and multiple defects. The ability to precisely identify a fault site depends on the faults associated with a single fault equivalence class. FastScan achieves this level of precision for most defects that behave as stuck-at faults.

FastScan does not perform its “normal” diagnosis if the chain test fails. However, there is a special diagnosis mode for chain test fails. Instead of reporting a fault site, chain diagnosis reports the last scan cell in each chain that appears to unload in a plausible way.

If the failures given to FastScan include a chain fail, or if the *-chain* option is given to the diagnose failures command, a chain diagnosis is performed.

Chain diagnosis uses fail information from the scan test section. The chain test failures are ignored except to indicate that chain diagnosis is to be performed.

Diagnosis is performed by looking at the actual values unloaded from the scan cells. This is achieved by XOR-ing the fail data with the expected data. It is assumed that a chain failure will cause constant data to be shifted out past the fault site. The diagnosis is performed by looking for the scan cell nearest scan out that unloads constant data. Assuming that over a few patterns every cell at some time will capture both a zero and one, this give a way to localize the fault site.

Understanding Stuck Faults and Defects

A *diagnosis* simulates stuck-at faults to identify the defects that cause test failures. Unfortunately, many defects (such as shorts and AC defects) do not behave as stuck-at faults. However, it is generally true that when defects cause circuit failures during testing, the defect site briefly behaves as a stuck-at fault.

Depending on the degree to which the defect behaves like a stuck-at fault, the diagnosis categorizes it into one of the following three defect classes:

- **Single Stuck Faults (SSF)**
  Defects in this class behave precisely the same as a stuck-at fault. In addition to the failing pattern data, FastScan uses passing pattern data to narrow down the list of fault candidates.

  Diagnosis for this fault class identifies a single defect that fully explains both failing and passing pattern results. Examples of defects in this class include open lines in bipolar chips and cell defects that cause an output to remain at a constant value.

- **Non-SSF Single Site Defects**
  Defects in this class do not always behave like stuck-at faults, but the source of all failures is a single defect site. The stuck-at fault associated with the defect site explains all failing patterns, but can cause some passing patterns to fail. FastScan cannot use passing patterns to resolve between fault candidates because this degrades the precision of the diagnosis.

  Diagnosis for this fault class identifies a single defect that fully explains all of the failing patterns. However, FastScan issues a warning message indicating the fault candidate causes passing patterns to fail. Examples of defects in this class include AC defects, CMOS opens, and intermittent defects.

- **Non-SSF Multiple Site Defects**
  Defects in this class require more than one stuck-at fault to explain all failures. In diagnosing these defects, FastScan assumes that a single fault explains all single pattern failures. The diagnosis identifies faults that explain the first failing pattern and, in addition, provide the best match for

all of the failures. FastScan then eliminates the explained failing patterns from further consideration and repeats the process for the remaining failures. FastScan records patterns that it cannot explain by any one stuck fault and then continues diagnosis on the next unexplained failure.

Diagnosis for this fault class identifies multiple defects, however, it may not explain all failing patterns. Examples of defects in this class include shorts and any combination of defects in the first two classes.

Creating the Failure File

The failure file contains a list of failing responses that result from applying the scan test patterns to a defective chip via ATE. You then capture the failing pattern data and ensure it is in the proper file format. You can also create this failure file by simulating a fault and writing all the failures that could result from that fault, using the Write Failures command. The Write Failures command works as a training or experimentation aid for understanding fault diagnosis.

You can use this failure file as input to the Diagnose Failures command, which identifies the most likely cause of the failures.

If the file does not include all failing patterns, you must identify the last pattern applied. The file must include the failing output measurements of all failing patterns up to that point.

It is important that this file contain all observed failures for a given pattern. Because of the scan output’s serial nature, you can easily truncate the list of failures not on a pattern boundary, which hinders diagnostic resolution. Providing the tool with as many failures as possible allows maximum resolution of the diagnosis.

Failure File Format

The failure file format rules are as follows:

- All data for a single failing response is on a single line.
- For a failing response that occurs during the parallel measure of the primary outputs, each entry contains the pattern number followed by the pin name of the failing primary output.
- For a failing response that occurs during the unloading of a scan chain, each entry contains the pattern number followed by the scan chain name followed by the failing scan cell's position in the scan chain. Positions start at 0, with position 0 being the scan cell closest to the scanout pin.
- The pattern number for an entry must not be smaller than the pattern number of a preceding entry.
- FastScan assumes an entry that begins with a double slash (//) is a comment and ignores it.
- The failure file must contain all the failing responses for all patterns up to and including the last failing pattern.

The following shows a failure file example:

```
10   output17
10   output29
10   chain1 314
10   chain3 75
195   output29
311   chain2 0
```

Performing a Diagnosis

Figure 8-1 gives a pictorial representation of the chip testing and diagnostic process.

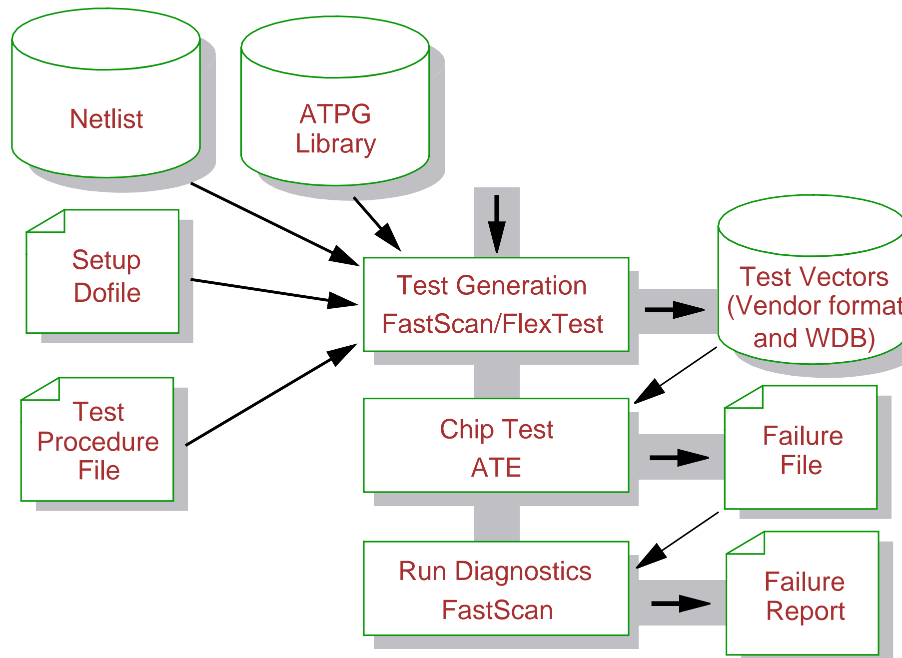


Figure 8-1. Diagnostics Process Flow

The following list provides a basic process for performing failure diagnosis within a FastScan session (from either the Atpg, Fault, or Good system mode):

1. Prior to running a diagnosis, you must store the failing pattern data in a file in the proper format. “Creating the Failure File” on page 8-4 describes the format of this file.

2. Set the pattern source to external and specify the test pattern file name (pattern_file).

   ATPG> **SET PAttern Source external pattern_file**

3. Enter the Diagnose Failures command, identifying the failure file (fails_file), and the last pattern used from the pattern file (in this case, pattern number 284), if you did not wish to apply all patterns.

   ATPG> **DIAgnose FAilures fails_file -last 284**

   This command generates a diagnostics report--either displayed or written to a file. The first line of the report is a summary of the diagnosis, which identifies the number of failing patterns, the number of different defects diagnosed, and the number of unexplained failing patterns. The tool lists any unexplained failures following the summary.

   For each defect it diagnoses, it gives the following information:

   - The number of failing patterns explained by the defect.
   - A warning if the fault candidates for the defect caused passing patterns to fail.
   - A list of the failing patterns explained by the defect.
   - A list of the possible fault candidates for the defect. For each fault candidate, the standard fault data, which includes fault type, fault code, pin pathname, and cell name, are displayed. The tool uses the fault code DS (detected by simulation) for the non-equivalent faults. The cell name identifies the type of cell that connects to the faulted pin. The cell name is “primary_input” for primary inputs, “primary_output” for primary outputs, and “unknown” for unresolvable instances.
   - CPU time the diagnosis uses.

INDEX

A

Abort limit, 6-81
Aborted faults, 6-80
  changing the limits, 6-81
  reporting, 6-80
Acronyms, xxiii
Ambiguity
  edge, 6-99
  path, 6-98
Apply statement, 3-13
ASCII WGL format, 7-38
ASIC Vector Interfaces, 7-25, 7-40 *through* 7-58
ATPG
  applications, 2-16
  basic procedure, 6-1
  default run, 6-76
  defined, 2-14
  for IDDQ, 6-85 *through* 6-91
  for path delay, 6-92 *through* 6-100
  for transition fault, 6-101 *through* 6-103
  full scan, 2-16
  increasing test coverage, 6-78 *through* 6-84
  instruction-based, 6-14, 6-111 *through* 6-115
  partial scan, 2-17
  process, 6-69
  scan identification, 5-26
  setting up faults, 6-46, 6-63
  with FastScan, 6-8 *through* 6-14
  with FlexTest, 6-14
ATPG constraints, 6-70
ATPG function, 6-71
At-speed test, 2-19
Automatic scan identification, 5-26
Automatic test equipment, 1-8, 6-16

B

BACK algorithm, 6-14
Batch mode, 1-18
Binary WGL format, 7-39
BIST
  optimum coverage, 6-56 *through* 6-59
  pattern simulation, 6-53, 6-55 *through* 6-56
  setup, 6-42
  troubleshooting simulation, 6-55
Blocks, functional or process flow, 1-14
Boundary scan
  defined, 2-2
Break statement, 3-14
Break_repeat statement, 3-15
Bus
  dominant, 3-34
  float, 4-18
Bus contention, 4-18
  checking during ATPG, 6-28
  fault effects, 6-29
Button Pane, 1-14

C

Capture handling, 6-32
Capture point, 2-29
Chain test, 7-30
Clock
  capture, 6-38, 6-105
  list, 6-38
  off-state, 6-38
  scan, 6-38
Clock groups, 5-43
Clock PO patterns, 6-10
Clock procedure, 3-25 *through* 3-26, 6-10, 6-12, 6-43
clock procedure, 3-14, 3-25
Clock procedures, 3-12
Clock sequential patterns, 6-11
Clocked sequential test generation, 4-23
Clocks, merging chains with different, 5-43
Combinational loop, 4-5, 4-6, 4-7, 4-8, 4-9, 4-10
  cutting, 4-6

INDEX [continued]

Command Line window, 1-10
Command usage, help, 1-16
Commands
- command line entry, 1-12
- command transcript, 1-12
- interrupting, 1-20
- running UNIX system, 1-20
- transcript, session, 1-11

Compass Scan format, 7-43 *through* 7-45
Compressing pattern set, 6-76
Condition statement, 3-14
Constant value loops, 4-6
Constraints
- ATPG, 6-70
- IDDQ, 6-91
- pin, 6-26, 6-36
- scan cell, 6-40

Contention, bus, 3-41
Continuation character, 1-13
Control Panel window, 1-14
Control points
- automatic identification, 5-30
- manual identification, 5-29

Controllability, 1-1
Controllability test coverage, 6-56
Copy, scan cell element, 3-5
Coupling loops, 4-9
Customizing
- help topics, 1-21, 1-23, 1-25
- menus, 1-21, 1-23, 1-25

Cycle count, 7-33
Cycle test, 7-30
Cycle-based timing, 6-16

D

Data capture simulation, 6-32
Data_capture gate, 4-25
Defect, 2-18
Design flattening, 3-29 *through* 3-36
Design rules checking
- BIST rules, 3-46
- blocked values, 3-46
- bus keeper analysis, 3-45
- bus mutual-exclusivity, 3-41
- clock rules, 3-44
- constrained values, 3-46
- data rules, 3-44
- extra rules, 3-46
- forbidden values, 3-46
- general rules, 3-40
- introduction, 3-40
- procedure rules, 3-40
- RAM rules, 3-44
- scan chain tracing, 3-42
- scannability rules, 3-46
- shadow latch identification, 3-43
- transparent latch identification, 3-44

Design-for-Test, defined, 1-1
Deterministic test generation, 2-15
DFTAdvisor
- as a point tool, ?? *through* 5-47
- block-by-block scan insertion, 5-47 *through* ??
- features, 2-12
- help topics, customizing, 1-21
- inputs and outputs, 5-5
- invocation, 5-10
- menus, customizing, 1-21
- process flow, 5-3
- supported test structures, 5-7
- user interface, 1-21

DFTAdvisor commands
- add buffer insertion, 5-41
- add cell models, 5-14
- add clock groups, 5-43
- add clocks, 5-15
- add nonscan instance, 5-32
- add nonscan models, 5-32
- add pin constraints, 5-23

INDEX [continued]

add scan chains, 5-17
add scan groups, 5-17
add scan instance, 5-33
add scan models, 5-33
add scan pins, 5-37, 5-40
add sequential constraints, 5-22
add test points, 5-29
analyze input control, 5-24
analyze output observe, 5-25
analyze testability, 5-30
delete buffer insertion, 5-41, 5-43
delete cell models, 5-15
delete clock groups, 5-45
delete clocks, 5-16
delete nonscan instances, 5-33
delete nonscan models, 5-33
delete scan instances, 5-33
delete scan models, 5-34
delete scan pins, 5-38
delete test points, 5-29
exit, 5-47
insert test logic, 5-42
report buffer insertion, 5-41
report cell models, 5-15
report clock groups, 5-45
report clocks, 5-16
report control signals, 5-35
report dft check, 5-34, 5-45
report nonscan models, 5-34
report primary inputs, 5-16
report scan cells, 5-45
report scan chains, 5-45
report scan groups, 5-45
report scan identification, 5-36
report scan instances, 5-34
report scan models, 5-34
report scan pins, 5-38
report statistics, 5-35, 5-36
report test logic, 5-15
report test points, 5-29
report testability analysis, 5-30
ripup scan chains, 5-18
run, 5-36
set system mode, 5-19
set test logic, 5-13
setup scan identification, 5-20
setup scan insertion, 5-38
setup scan pins, 5-38
setup test_point identification, 5-28
write atpg setup, 5-46
write netlist, 5-45
write primary inputs, 5-16
write scan identification, 5-36

Differential scan input pins, 7-28
Distributed FlexTest, 6-21
Dofiles, 1-18
dofiles, 1-18
Dominant bus, 3-34
Dont_touch property, 5-32

E

Edge ambiguity, 6-99
End statement, 3-12
Event group, 7-14
Exiting the tool, 1-20
External pattern generation, 2-15
Extra, scan cell element, 3-6

F

FastScan
ATPG method, 6-8 *through* 6-14
basic operations, 6-19
diagnostics-only version, 6-21
features, 2-16
help topics, customizing, 1-23
inputs and outputs, 6-6
introduced, 2-16
menus, customizing, 1-23
non-scan cell handling, 4-20 *through* 4-25
pattern types, 6-9 *through* 6-14

INDEX [continued]

test cycles, 6-9
timing model, 6-9
tool flow, 6-3
user interface, 1-23
FastScan commands
add ambiguous paths, 6-95
add atpg functions, 6-71
add capture handling, 6-33
add cell constraints, 6-40
add clocks, 6-38
add control points, 6-58
add faults, 6-46, 6-63, 6-64
add iddq constraints, 6-91
add lfsr connections, 6-43
add lfsr taps, 6-43
add lfsrs, 6-43
add lists, 6-48, 6-52
add nofaults, 6-41
add notest points, 6-58
add observe points, 6-58
add pin equivalences, 6-24
add primary inputs, 6-25, 6-42
add primary outputs, 6-25, 6-42
add scan chains, 6-39
add scan groups, 6-39
add slow pad, 6-27
add tied signals, 6-26
analyze atpg constraints, 6-73
analyze bus, 6-29
analyze control, 6-56
analyze fault, 6-79, 6-95
analyze observe, 6-57
analyze restrictions, 6-73
compress patterns, 6-77
delete atpg constraints, 6-73
delete atpg functions, 6-73
delete cell constraints, 6-41
delete clocks, 6-38
delete faults, 6-64
delete iddq constraint, 6-91
delete lfsr connections, 6-44
delete lfsr taps, 6-44
delete lfsrs, 6-44
delete nofaults, 6-41
delete paths, 6-95
delete pin equivalences, 6-24
delete primary inputs, 6-25
delete primary outputs, 6-25
delete scan chains, 6-39
delete scan groups, 6-39
delete slow pad, 6-28
delete tied signals, 6-26
diagnose failures, 8-7
flatten model, 3-29
insert testability, 6-58
load faults, 6-64
report aborted faults, 6-80, 6-81
report atpg constraints, 6-73
report atpg functions, 6-73
report bus data, 6-29
report cell constraints, 6-41
report clocks, 6-38
report control data, 6-56
report control points, 6-58
report environment, 6-34
report faults, 6-48, 6-64, 6-79
report gates, 6-29
report iddq constraints, 6-91
report lfsr connections, 6-44
report lfsrs, 6-44, 6-55
report nofaults, 6-41
report observe data, 6-57
report observe points, 6-58
report paths, 6-95
report pin equivalences, 6-24
report primary inputs, 6-25
report primary outputs, 6-25
report scan chains, 6-39
report scan groups, 6-39
report slow pads, 6-28

INDEX [continued]

report statistics, 6-48
report testability data, 6-80
report tied signals, 6-26
reset state, 6-49, 6-52
run, 6-47, 6-51, 6-76
save patterns, 6-84
set abort limit, 6-81, 6-102
set atpg compression, 6-83
set au analysis, 6-84
set bus handling, 6-29
set capture clock, 6-43, 6-47
set capture handling, 6-33
set checkpoint, 6-76
set clock restriction, 6-40
set clock_off simulation, 6-74
set contention check, 6-28
set decision order, 6-82, 6-84
set dofile abort, 1-19
set drc handling, 6-34
set driver restriction, 6-29
set fault mode, 6-67
set fault type, 6-46, 6-63, 6-102
set iddq checks, 6-91
set learn report, 6-28
set list file, 6-48, 6-52
set net dominance, 6-29
set net resolution, 6-30
set observation point, 6-43
set observe threshold, 6-57
set pattern source, 6-47, 8-6
set possible credit, 6-28, 6-68
set pulse generators, 6-28
set random atpg, 6-82
set random clocks, 6-47
set random patterns, 6-42, 6-47
set sensitization checking, 6-34
set simulation mode, 6-103
set split capture_cycle, 6-75
set static learning, 6-31
set system mode, 6-23, 6-44
set z handling, 6-30
setup checkpoint, 6-76
setup lfsrs, 6-44
setup tied signals, 6-26
write environment, 6-28
write faults, 6-65
write paths, 6-95
write primary inputs, 6-25
write primary outputs, 6-25

Fault
- aborted, 6-81
- classes, 2-32 *through* 2-40
- collapsing, 2-23
- detection, 2-31
- internal, 2-23
- no fault setting, 6-41
- simulation, 6-45
- undetected, 6-81

Fault models, 2-22 *through* 2-30
- path delay, 2-29
- psuedo stuck-at, 2-25
- stuck-at, 2-24
- toggle, 2-25
- transition, 2-28

Feedback loops, 4-5 *through* 4-16

Flattening, design, 3-29 *through* 3-36

FlexTest
- ATPG library verification, 6-65
- ATPG method, 6-14
- basic operations, 6-19
- Distributed FlexTest, 6-21
- fault simulation version, 6-21
- help topics, customizing, 1-25
- inputs and outputs, 6-6
- introduced, 2-16, 2-17
- menus, customizing, 1-25
- non-scan cell handling, 4-25
- pattern types, 6-18
- timing model, 6-16
- tool flow, 6-3

INDEX [continued]

- user interface, 1-25
- FlexTest commands
  - abort interrupted process, 6-22
  - add cell constraints, 6-40
  - add clocks, 6-38
  - add faults, 6-46
  - add iddq constraints, 6-91
  - add lists, 6-48, 6-52
  - add nofaults, 6-41
  - add nonscan handling, 4-26
  - add pin constraints, 6-26, 6-36
  - add pin equivalences, 6-24
  - add pin strobes, 6-37
  - add primary inputs, 6-25
  - add primary outputs, 6-25
  - add scan chains, 6-39
  - add scan groups, 6-39
  - add tied signals, 6-26
  - compress patterns, 6-77
  - delete cell constraints, 6-41
  - delete clocks, 6-38
  - delete faults, 6-64
  - delete iddq constraint, 6-91
  - delete nofaults, 6-41
  - delete pin constraints, 6-37
  - delete pin equivalences, 6-24
  - delete pin strobes, 6-38
  - delete primary inputs, 6-25
  - delete primary outputs, 6-25
  - delete scan chains, 6-39
  - delete scan groups, 6-39
  - delete tied signals, 6-26
  - flatten model, 3-29
  - load faults, 6-64
  - report aborted faults, 6-80
  - report AU faults, 6-48
  - report bus data, 6-29
  - report cell constraints, 6-41
  - report clocks, 6-38
  - report environment, 6-34
  - Report Faults, 6-79
  - report faults, 6-48, 6-64
  - report gates, 6-29
  - report iddq constraints, 6-91
  - report nofaults, 6-41
  - report nonscan handling, 4-26
  - report pin constraints, 6-37
  - report pin equivalences, 6-24
  - report pin strobes, 6-38
  - report primary inputs, 6-25
  - report primary outputs, 6-25
  - report scan chains, 6-39
  - report scan groups, 6-39
  - report statistics, 6-48
  - report tied signals, 6-26
  - reset state, 6-49, 6-52
  - resume interrupted process, 6-22
  - run, 6-47, 6-51, 6-76
  - save patterns, 6-84
  - set abort limit, 6-81
  - set bus handling, 6-29
  - set checkpoint, 6-76
  - set clock restriction, 6-40
  - set contention check, 6-28
  - set dofile abort, 1-19
  - set driver restriction, 6-29
  - set fault mode, 6-67
  - set fault sampling, 6-67
  - set fault type, 6-46, 6-63, 6-102
  - set hypertrophic limit, 6-67
  - set iddq checks, 6-91
  - set interrupt handling, 6-22
  - set list file, 6-48, 6-52
  - set loop handling, 6-28
  - set net dominance, 6-29
  - set net resolution, 6-30
  - set output comparison, 6-51
  - set pattern source, 6-47
  - set possible credit, 6-28, 6-68
  - set pulse generators, 6-28

INDEX [continued]

set race data, 6-28
set random atpg, 6-82
set redundancy identification, 6-28
set self initialization, 6-66
set state learning, 6-32
set system mode, 6-44
set test cycle, 6-35
set z handling, 6-30
setup checkpoint, 6-76
setup pin constraints, 6-36
setup pin strobes, 6-37
setup tied signals, 6-26
write environment, 6-28
write faults, 6-65
write library_verification setup, 6-65
write primary inputs, 6-25
write primary outputs, 6-25
Force statement, 3-13
Force_sci statement, 3-13
Force_sci_equiv statement, 3-13
Fujitsu FTDL-E format, 7-45 *through* 7-46
Full scan, 2-4, 5-7
Functional blocks, 1-14
Functional test, 2-19

G

Gate duplication, 4-8
Good simulation, 6-50
Graphic Pane, 1-14

H

Head register, attaching, 5-39
Help
command usage, 1-16
dialog box help, 1-15
functional block, 1-15
Help menu, 1-17
online manuals, 1-17
popup in Control Panel, 1-15
process block, 1-15
query help in dialogs, 1-15
Help topics, customizing, 1-21, 1-23, 1-25
Hierarchical instance, definition, 3-30
Hold gate, 4-25

I

IDDQ testing, 6-85 *through* 6-91
creating the test set, 6-85 *through* 6-91
defined, 2-19
methodologies, 2-20
performing checks, 6-91
psuedo stuck-at fault model, 2-25
setting constraints, 6-91
test pattern formats, 7-27
vector types, 2-21
Init0 gate, 4-25
Init1 gate, 4-25
Initialize statement, 3-13
InitX gate, 4-25
Instance, definition, 3-30
Instruction-based ATPG, 6-14, 6-111 *through* 6-115
Internal faulting, 2-23
Internal scan, 2-1, 2-2
Interrupting commands, 1-20

L

Latches
handling as non-scan cells, 4-19
lockup, 5-43
scannability checking of, 4-4
Launch point, 2-29
Layout-sensitive scan insertion, 5-42
Learning analysis, 3-36 *through* 3-39
dominance relationships, 3-39
equivalence relationships, 3-36
forbidden relationships, 3-38
implied relationships, 3-38
logic behavior, 3-37
LFSR, 4-29

INDEX [continued]

Line continuation character, 1-13
Line holds, 2-33
Lockup latches, 5-43
Log files, 1-19
Loop count, 7-33
Loop cutting, 4-6
    by constant value, 4-6
    by gate duplication, 4-8
    for coupling loops, 4-9
    single multiple fanout, 4-7
Loop handling, 4-5 *through* 4-16
LSI Logic LSITDL format, 7-50 *through* 7-58
    C-MDE Environment, 7-53
LSSD, 3-10

M

Macro, 2-7
Macros, 2-21
Manuals, viewing, 1-17
Manufacturing defect, 2-18
Mapping scan cells, 5-11
Masking primary outputs, 6-26, 6-27
Master, scan cell element, 3-3
MBISTArchitect commands
    system, 1-20
Measure_sco statement, 3-13
Menus
    pulldown, 1-10
Menus, customizing, 1-21, 1-23, 1-25
Merging scan chains, 5-43
MISR, 4-29
Mitsubishi TDL format, 7-49
Modified timing definition, 7-23
Module, definition, 3-30
Motorola UTIC format, 7-46 *through* 7-49

N

No fault setting, 6-41
Non-scan cell handling, 4-19 *through* 4-25
    clocked sequential, 4-23
    data_capture, 4-25
    FastScan, 4-20
    FlexTest, 4-25
    hold, 4-25
    init0, 4-25
    init1, 4-25
    initx, 4-25
    sequential transparent, 4-21
    tie-0, 4-20, 4-25
    tie-1, 4-20, 4-25
    tie-X, 4-20
    transparent, 4-20
Non-Scan Related Events, 7-13
Non-scan sequential instances
    reporting, 5-34

O

Observability, 1-1
Observability test coverage, 6-57
Observe points
    automatic identification, 5-30
    manual identification, 5-29
Offset, 6-17
Off-state, 3-8, 5-15, 6-38
Online
    help available, 1-15
    manuals, 1-17

P

Panes
    button, 1-14
    graphic, 1-14
    process, 1-21, 1-23, 1-25
Parallel scan chain loading, 7-26
Partial scan
    defined, 2-5
    types, 5-7
Partition scan, 2-8, 5-8

INDEX [continued]

Path ambiguity, 6-98
Path definition file, 6-96
Path delay testing, 2-29, 6-92 *through* 6-100
  basic procedure, 6-99
  limitations, 6-100
  path amibiguity, 6-98
  path definition checking, 6-98
  path definition file, 6-96
  patterns, 6-93
  robust detection, 6-93
  transition detection, 6-93
Path sensitization, 2-31
Pattern compression
  dynamic, 6-77
  static, 6-76
Pattern formats
  FastScan binary, 7-33
  FastScan text, 7-28, 7-29
  FlexTest text, 7-28, 7-29
  Lsim, 7-37
  MGCWDB, 7-34
  TSSI WGL (ASCII), 7-38
  TSSI WGL (binary), 7-39
  Verilog, 7-35
  ZYCAD, 7-40
Pattern generation
  deterministic, 2-15
  external source, 2-15
  random, 2-14
Pattern types
  basic scan, 6-9
  clock PO, 6-10
  clock sequential, 6-11
  cycle-based, 6-18
  RAM sequential, 6-12
  sequential transparent, 6-13
Period, 6-16
Pin constraints, 6-26, 6-35, 6-36
Popup help, 1-15
Possible-detect credit, 2-36, 6-68
Possible-detected faults, 2-36
Primary inputs
  constraining, 6-26
  constraints, 5-23, 6-26, 6-35, 6-36, 6-91
  cycle behavior, 6-35
  cycle-based requirements, 6-18
Primary outputs
  masking, 5-24, 6-27
  strobe requirements, 6-18
  strobe times, 6-37
Primitives, simulation, 3-32
Procedure File, setting, 7-20
Procedure statement, 3-12
Process flow blocks, 1-14
Process pane, 1-21, 1-23, 1-25
PRPG, 4-29
Pulldown menus, 1-10
Pulse width, 6-17

Q

Query help, 1-15

R

RAM
  Common Read and Clock Lines, 4-39
  Common Write and Clock Lines, 4-39
  FastScan support, 4-35
  pass-through mode, 4-36
  RAM sequential mode, 4-37
  read-only mode, 4-36
  related commands, 4-41 *through* 4-42
  rules checking, 4-42 *through* 4-43
  testing, 4-34 *through* 4-43
RAM sequential patterns, 6-12
Random pattern generation, 2-14
Registers
  head, attaching, 5-39
  tail, attaching, 5-39
Related documentation, xviii
Restore_bidis statement, 3-14

INDEX [continued]

Restore_pis statement, 3-14
ROM
  FastScan support, 4-35
  related commands, 4-41 *through* 4-42
  rules checking, 4-42
  testing, 4-34 *through* 4-43
Running ATPG, 6-69 *through* 6-85

S

Scan
  basic operation, 2-4
  clock, 2-4
Scan cell
  concepts, 3-2
  constraints, 6-40
  mapping, 5-11
Scan cell elements
  copy, 3-5
  extra, 3-6
  master, 3-3
  shadow, 3-4
  slave, 3-4
Scan chains
  definition, 3-6
  head and tail registers, attaching, 5-39
  merging, 5-43
  parallel loading, 7-26
  specifying, 6-39
Scan clocks, 3-8, 5-15
  specifying, 5-15, 6-38
Scan design
  defined, 2-1, 2-2
  simple example, 2-3
Scan groups, 3-7, 6-39
Scan insertion
  layout-sensitive, 5-42
  process, 5-3
Scan output mapping, 5-11
Scan patterns, 6-9
Scan Related Events, 7-3
Scan sub-chain, 7-26
Scan test, 7-30
Scannability checks, 4-3
Scan-sequential ATPG, 2-8
SCOAP
  scan identification, 5-26
  test point insertion, 5-30
Scripts, 1-18
Sequential loop, 4-5, 4-14, 4-15
Sequential transparent latch handling, 4-21
Sequential transparent patterns, 6-13
Sequential_transparent procedure, 3-24 *through* 3-25
Session transcript, 1-11
Set Checkpoint, 6-76
Set Clock_off Simulation, 6-74
Set Split Capture_cycle, 6-75
Setting Test Procedure File Timing from the Timeplate File, 7-20
Shadow, 3-4
Shell commands, running UNIX commands, 1-20
Simulating captured data, 6-32
Simulation data formats, 7-28 *through* 7-40
Simulation formats, 7-25
Simulation primitives, 3-32 *through* 3-36
  AND, 3-33
  BUF, 3-32
  BUS, 3-34
  DFF, 3-33
  INV, 3-32
  LA, 3-33
  MUX, 3-33
  NAND, 3-33
  NMOS, 3-34
  NOR, 3-33
  OR, 3-33
  OUT, 3-36
  PBUS, 3-35
  PI, 3-32

INDEX [continued]

PO, 3-32
RAM, 3-36
ROM, 3-36
STFF, 3-34
STLA, 3-34
SW, 3-34
SWBUS, 3-35
TIE gates, 3-34
TLA, 3-34
TSD, 3-34
TSH, 3-34
WIRE, 3-35
XDET, 3-35
XNOR, 3-33
XOR, 3-33
ZDET, 3-35
ZHOLD, 3-35
ZVAL, 3-32
Single multiple fanout loops, 4-7
Sink gates, 6-33
Slave, 3-4
Source gates, 6-33
Structural loop, 4-5
combinational, 4-5
sequential, 4-5
Structured DFT, 1-2
Super timeplate, 7-14
Synchronization latches, 5-43
Synchronizing scan chain clocking, 5-43
System-class
non-scan instance, 5-32
non-scan instances, 5-31
scan instance, 5-32
scan instances, 5-31
test points, 5-28

T

Tail register, attaching, 5-39
Tap points, 4-29
Test clock, 4-26, 5-13
Test cycle
defined, 6-16
setting width, 6-35
Test logic, 4-4, 4-26, 5-12
Test Pattern Timing Information
Setting Time Scale in Timeplate File, 7-20
Test patterns, 2-14
chain test block, 7-30
cycle test block, 7-32
scan test block, 7-31
Test points
controlling the number of, 5-28
definition of, 5-9
locations not added by DFTAdvisor, 5-29
setting up identification, 5-28
understanding, 2-10
Test procedure file
defined, 3-11
DRC checking, 3-11
in DFTAdvisor, 5-6
scan chain checking, 3-28
statements, 3-12 *through* 3-15
syntax rules, 3-11 *through* 3-12
Test procedures
load_unload, 3-18
master_observe, 3-22
shadow_control, 3-21
shadow_observe, 3-23
shift, 3-16
skew_load, 3-26
test_setup, 3-15
Test structures
full scan, 2-4 *through* 2-5, 2-7 *through* 2-8, 5-7
identification interactions, 5-9
partial scan, 2-5 *through* 2-8, 5-7
partition scan, 2-8 *through* 2-10, 5-8
scan sequential ATPG-based partial scan, 5-8
sequential ATPG-based partial scan, 5-7

INDEX [continued]

sequential automatic partial scan, 5-8
sequential SCOAP-based partial scan, 5-8
sequential structure-based partial scan, 5-8
sequential transparent ATPG-based partial scan, 5-8
supported by DFTAdvisor, 5-7
test points, 2-10 *through* 2-11, 5-9

Test types
at-speed, 2-22
functional, 2-19
IDDQ, 2-20

Test vectors, 2-14
Testability, 1-1
TI TDL 91 format, 7-41 *through* 7-43
Tie-0 gate, 4-20, 4-25
TIE0, scannable, 4-4
Tie-1 gate, 4-20, 4-25
TIE1, scannable, 4-4
Tie-X gate, 4-20
Time frame, 6-17, 6-35
Time Scale, setting, 7-20
Timeplate, 7-14
Timing Checks for Tester Format Patterns, 7-21
Timing definition, 7-23
Timing file, 7-18
Toshiba TSTL2 format, 7-50

Transcript
command, 1-12
session, 1-11

Transition fault testing, 6-101 *through* 6-103

Transition testing
basic procedure, 6-103

Transiton testing
patterns, 6-101

Transparent latch handling, 4-20
Transparent slave, handling, 4-22
Transparent_capture cells, 3-26

U

Undetected faults, 6-81
UNIX commands, running within tool, 1-20
Usage, command, 1-16

User interface
button pane, 1-14
command line, 1-12
Command Line window, 1-10
command transcript, 1-12
common features, 1-9
Control Panel window, 1-14
DFTAdvisor, 1-21
dofiles, 1-18
exiting, 1-20
FastScan, 1-23
FlexTest, 1-25
functional or process flow blocks, 1-14
graphic pane, 1-14
interrupting commands, 1-20
log files, 1-19
menus, 1-10
process pane, 1-21, 1-23, 1-25
running UNIX system commands, 1-20
session transcript, 1-11

User-class
non-scan instances, 5-31
scan instances, 5-33
test points, 5-28

V

Verilog, 7-35 *through* 7-36
Viewing online manuals, 1-17

W

Windows
Command Line, 1-10
Control Panel, 1-14