

TetraMAX® ATPG

User Guide

Version A-2007.12-SP4, June 2008

Comments?

Send comments on the documentation by going to <http://solvnet.synopsys.com>, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

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

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____. "

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

Synopsys, AMPS, Astro, Cadabra, CATS, CSim, Design Compiler, DesignWare, Fablink, Formality, HSPICE, iN-Phase, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, Powermill, PrimeTime, PWA, SELID, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, TimeMill, VCS, Vera, WaveCalc, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

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

Service Marks (SM)

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

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

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

Contents

What's New in This Release.....	xxviii
About This Manual	xxviii
Customer Support.....	xxx
1. TetraMAX Overview	
TetraMAX ATPG Features	1-2
ATPG Modes	1-3
Basic-Scan ATPG.....	1-3
Fast-Sequential ATPG	1-3
Full-Sequential ATPG.....	1-3
Supported Fault Models.....	1-4
Stuck-At	1-4
Transition	1-4
Path Delay	1-4
IDDQ.....	1-4
Bridging.....	1-4
Design Flow Using DFT Compiler and TetraMAX	1-5
2. Running TetraMAX	
Installation.....	2-2
Specifying the Location for TetraMAX Installation.....	2-2
Licensing.....	2-3
Invoking TetraMAX	2-4
Methods for Invoking	2-6

Running Background Jobs.....	2-7
Predefined Aliases.....	2-7
64-Bit Mode on Solaris and HP-UX Platforms.....	2-7
64-Bit Support for GUI Mode.....	2-8
Crashes and What to Expect	2-8
Setup Files.....	2-8
Variables	2-9
Controlling TetraMAX Processes	2-10
Starting the TetraMAX GUI	2-10
Stopping the TetraMAX GUI	2-10
Interrupting a Long Process.....	2-11
Discarding Pending Output.....	2-12
Adjusting the Workspace Size.....	2-12
Saving Preferences	2-13
3. Command Interface	
TetraMAX Main Window.....	3-2
Command Entry	3-3
Menu Bar.....	3-3
Command Toolbar and GSV Toolbar	3-3
Command-Line Window	3-4
Command Mode Indicator	3-4
Command-Line Entry Field.....	3-4
Command Continuation.....	3-4
Command History	3-5
Stop Button.....	3-5
Commands From a Command File	3-5
Command Logging	3-5
Transcript Window	3-6
Setting the Keyboard Focus	3-6
Using the Transcript Text.....	3-6
Selecting Text in the Transcript.....	3-7
Copying Text From the Transcript.....	3-7
Finding Text in the Transcript	3-7
Saving or Printing the Transcript	3-7
Clearing the Transcript Window.....	3-8

Online Help	3-8
Text-Only Help	3-8
GUI-Based Help	3-9
Selecting a Topic From the Major Contents List	3-12
Selecting a Topic From the Detailed Contents List	3-14
Selecting a Topic From the Index	3-15
Selecting a Topic by Searching	3-16
Bookmarking a Topic	3-17
Invoking the Help System in StandAlone	3-19
4. ATPG Design Flow	
Basic ATPG Design Flow	4-2
Netlist Requirements	4-4
EDIF Netlist Requirements	4-4
Logic 1/0 Using Global Nets	4-4
Logic 1/0 by Special Library Cell	4-4
Verilog Netlist Requirements	4-5
VHDL Netlist Requirements	4-5
Reading the Netlist	4-5
Using the Read Netlist/Image Dialog Box	4-6
Using the read netlist Command	4-6
Read Netlist Usage Notes	4-6
Reading Library Models	4-7
Building the ATPG Model	4-8
Using the Run Build Model Dialog Box	4-8
Using the run build_model Command	4-8
Performing Test Design Rule Checking	4-9
Starting Test DRC	4-9
Using the Run DRC Dialog Box	4-10
Reviewing the DRC Results	4-11
Viewing Violations in the GSV	4-12
Test Design Rule Categories	4-14
Changing Test Design Rule Severity	4-15
Effects of Rule Violations	4-16
A Detailed View of the DRC Process	4-16
Scan Chain Tracing	4-17

Shadow Register Analysis	4-17
Procedure Simulation	4-18
Transparent Latches	4-18
Cells With Asynchronous Set/Reset Inputs	4-18
Sensitizable Feedback Paths	4-19
Save/Restore in TEST Mode.....	4-19
 Preparing for ATPG	4-20
Setting Up the Fault List	4-20
Selecting the Pattern Source.....	4-21
Using the Set Patterns Dialog Box	4-21
Using the set patterns Command	4-21
Choosing Settings for Contention Checking	4-22
Using the Set Contention Dialog Box	4-22
Using the set contention Command	4-22
Specifying ATPG Settings	4-23
Using the Run ATPG Dialog Box	4-23
Using the set atpg Command	4-23
 Running ATPG	4-23
Quickly Estimating Test Coverage.....	4-24
Using the Run ATPG Dialog Box	4-24
Using the set atpg Command.....	4-24
Increasing Effort Over Multiple Passes	4-27
Maximizing Test Coverage Using Fewer Patterns.....	4-28
Obtaining Target Test Coverage Using Fewer Patterns	4-28
Detecting Faults Multiple Times.....	4-29
Reviewing Test Coverage	4-29
Using the Report Summaries Dialog Box	4-29
Using the report summaries/fault Command	4-30
 Writing ATPG Patterns	4-31
Using the Write Patterns Dialog Box	4-31
Using the write patterns Command	4-32
Examples	4-32
 Writing Fault Lists	4-32
Using the Report Faults Dialog Box.....	4-32
Using the write faults Command	4-33
Examples	4-33

Using Command Files	4-34
Setting Up Advanced ATPG Features	4-35
Declaring Equivalent and Differential Input Ports	4-36
Using the Add PI Equivalences Dialog Box	4-36
Using the add pi equivalences Command	4-36
Declaring Primary Input Constraints	4-37
Using the Add PI Constraints Dialog Box	4-37
Using the add pi constraints Command	4-37
Masking Input and Output Ports	4-38
Using ATPG Constraints	4-38
Usage Example 1	4-38
Usage Example 2	4-40
Masking Scan Cell Inputs and Outputs	4-40
Using the Add Cell Constraints Dialog Box	4-41
Using the add cell constraints Command	4-41
Previewing Potential Scan Cells	4-41
Using the Set Scan Ability Dialog Box	4-42
Using the set scan ability Command	4-42
Deleting Top-Level Ports From Output Patterns	4-43
Creating Custom ATPG Models	4-43
Examples	4-43
Removing Unused Logic	4-44
Examples	4-45
Condensing ATPG Libraries	4-47
Working With Clock Groups	4-49
Pattern Count Reduction	4-49
Generating a Clock Group Report	4-50
Improving Test Coverage With Test Points	4-51
Specifying Test-Point Analysis	4-51
Running the Flow	4-53
Limitations	4-53
5. On-Chip Clocking Support	
Background	5-2
Supported Flows	5-2
Definitions	5-3
Pattern Support	5-3

Limitations	5-4
Design Flows	5-6
DFT Compiler-to-TetraMAX Flow	5-6
Non-DFT Compiler to TetraMAX Flows	5-9
OCC Support in TetraMAX	5-9
Scan ATPG Flow	5-9
Waveform and Capture Cycle Example	5-10
OCC-Specific DRC Rules	5-10
6. Working With Design Netlists and Libraries	
Using Wildcards to Read Netlists	6-2
Controlling Case-Sensitivity	6-3
Identifying Missing Modules	6-3
Using Black Box and Empty Box Models	6-5
Behavior of RAM Black Boxes	6-7
Troubleshooting Unexplained Behavior	6-11
Handling Duplicate Module Definitions	6-13
Memory Modeling	6-13
A Basic Template	6-14
Initializing RAM and ROM Contents	6-15
The Memory Initialization File	6-15
Default Initialization	6-15
Instance-Specific Initialization	6-16
Building the ATPG Design Model	6-17
Processes During run build_model Execution	6-17
Controlling the Build Process	6-19
Using the Set Build Dialog Box	6-19
Using the set build Command	6-19
ATPG-Specific Learning Processes	6-19
Using the Run Build Model Dialog Box	6-20
Using the set learning Command	6-20
Binary Image Files	6-20
Flow Example	6-22

7. Using the GSV For Review and Analysis

Getting Started With the GSV	7-2
Starting the GSV Using the SHOW Button	7-2
Starting the GSV From a DRC Violation or Specific Fault	7-3
Navigating Within the GSV	7-6
Selecting Objects in the GSV Schematic	7-6
Hiding Objects in the GSV Schematic	7-7
Using the Block ID Window	7-8
Expanding the Display From Net Connections	7-8
Hiding Buffers and Inverters in the GSV Schematic	7-10
ATPG Model Primitives	7-11
Tied Pins	7-11
Primary Inputs and Outputs	7-12
Basic Gate Primitives	7-13
Additional Visual Characteristics	7-15
RAM and ROM Primitives	7-15
Displaying Symbols in Primitive or Design View	7-17
Displaying Instance Path Names	7-17
Displaying Pin Data	7-17
Using the Setup Dialog Box	7-18
Using the set pindata Command	7-18
Pin Data Types	7-19
Displaying Clock Cone Data	7-20
Displaying Clock Off Data	7-21
Displaying Constrain Values	7-22
Displaying Load Data	7-23
Displaying Shift Data	7-25
Displaying Test Setup Data	7-25
Displaying Pattern Data	7-26
Displaying Logic Values for a Specific Pattern	7-26
Displaying Logic Values for Multiple Patterns	7-27
Displaying Tie Data	7-28
Displaying Other Design Information	7-29
Analyzing a Feedback Path	7-29
Checking Controllability and Observability	7-30
Using the Run Justification Dialog Box	7-31
Using the run justification Command	7-31

Analyzing DRC Violations.....	7-32
Scan Chain Blockage	7-32
Example: A Bidirectional Contention Problem.....	7-35
Analyzing Buses	7-37
Bus Contention Status.....	7-37
The Contention Checking Report	7-38
Reduction of Aborted Bus and Wire Gates	7-38
Using the Analyze Buses Dialog Box	7-39
Using the set atpg and analyze bus Commands	7-39
Causes of Bus Contention.....	7-40
Analyzing ATPG Problems	7-41
Example: Analyzing an AN Fault.....	7-41
Example: Analyzing a UB Fault.....	7-43
Example: Analyzing a NO Fault.....	7-45
Printing Schematic to a File	7-45
8. Using the Simulation Waveform Viewer	
Getting Started With the SWV	8-2
Supported Pin Data Types and Definitions.....	8-2
Invoking the SWV	8-4
Using the SWV Interface	8-5
Understanding the SWV Layout	8-6
Using the Signal List Pane.....	8-7
Manipulating Signals.....	8-7
Refreshing the View.....	8-10
Understanding the Simulation Waveform Viewer Color Codes	8-11
Identifying Signal Types in the Graphical Pane.....	8-12
Using the Time Scales.....	8-12
Upper Time Scale.....	8-12
Lower Time Scale.....	8-12
Using the Marker Header Area	8-13
Adding and Deleting Pointers.....	8-13
Moving a Marker Pointer.....	8-14
Measuring Between Two Pointers	8-14
Using the SWV With the GSV.....	8-15

Using the SWV Without the GSV	8-16
SWV Inputs and Outputs	8-17
Analyzing Violations	8-17
9. STIL Procedure Files	
STIL Usage in TetraMAX	9-2
Creating a New STIL Procedure File	9-2
Declaring Primary Input Constraints	9-3
Using the Add PI Constraints Dialog Box	9-4
Using the add pi constraints Command	9-4
Declaring Clocks	9-4
Using the Edit Clocks Dialog Box	9-5
Using the add clocks Command	9-5
Asynchronous Set and Reset Ports	9-5
Declaring Scan Chains and Scan Enables	9-6
Using the DRC Dialog Box	9-6
Using a Command	9-6
Writing the Initial STIL Template	9-7
Editing the STIL Procedure File	9-7
Defining the load_unload Procedure	9-8
Defining the Shift Procedure	9-8
Defining the test_setup Macro	9-9
Using Loop Statements	9-10
Controlling Bidirectional Ports	9-11
Defining Pulsed Ports	9-13
Defining Basic Signal Timing	9-14
Defining System Capture Procedures	9-17
Creating Generic Capture Procedures	9-18
Advantages of Generic Capture Procedures	9-19
Writing Out Generic Capture Procedures	9-19
Controlling Multiple Clock Capture	9-20
Using Allclock Procedures	9-22
Using load_unload for Last-Shift-Launch Transition	9-24
Example Post-Scan Protocol	9-25
Limitations	9-26
Defining a Sequential Capture Procedure	9-26
Using the Default Capture Procedures	9-27

Using a Sequential Capture Procedure	9-27
Sequential Capture Procedure Syntax	9-28
Defining the End-of-Cycle Measure	9-29
Defining Constrained Primary Inputs	9-30
Defining Equivalent Primary Inputs	9-31
Defining Reflective I/O Capture Procedures	9-32
Defining Other STIL Procedures	9-33
The master_observe Procedure	9-33
The shadow_observe Procedure	9-34
Using Quick STIL Commands with an On-Chip Clock Controller	9-35
Testing the STIL Procedure File	9-37
STIL Procedure File Guidelines	9-37
JTAG/TAP Controller Variations for the load_unload Procedure	9-38
Final Shift With TMS=1	9-38
Multiple Scan Groups	9-39
Limiting Clock Usage	9-46

10. Fault Lists and Faults

Fault Lists	10-2
Using Fault List Files	10-2
New Faults With -retain_code.....	10-3
Existing Faults With -retain_code.....	10-3
Collapsed and Uncollapsed Fault Lists	10-3
Random Fault Sampling	10-4
Fault Dictionary	10-5
Fault Categories and Classes.....	10-5
Detected Faults	10-6
Possibly Detected Faults	10-7
Undetectable Faults	10-7
ATPG Untestable Faults	10-8
Not Detected Faults	10-8
Fault Summary Reports	10-9
Test Coverage	10-10
Fault Coverage	10-11
ATPG Effectiveness	10-11

Reporting Clock Domain-Based Faults	10-12
Using Signals That Conflict With Reserved Keywords	10-19
Finding Particular Untested Faults Per Clock Domain.	10-19
11. Fault Simulation	
Fault Simulation Design Flow	11-2
Preparing the Functional Test Patterns for Fault Simulation	11-3
Acceptability to ATE	11-4
Checking for Timing Insensitivity	11-5
Examples That Cause Timing Sensitivity	11-5
Recognizable Format.	11-6
Requirements for Scan Functional Patterns.	11-6
Requirements for Nonscan Functional Patterns.	11-7
VCDE Input Patterns	11-9
Preparing the Design for Fault Simulation	11-11
Preprocessing the Netlist.	11-11
Reading the Design and Libraries	11-11
Building the ATPG Design Model	11-11
Declaring Clocks	11-12
Running DRC.	11-12
DRC for Nonscan Functional Test Patterns	11-12
DRC for Scan Functional Test Patterns	11-14
Reading the Functional Test Patterns	11-14
Using the Set Patterns Dialog Box	11-14
Using the set patterns Command	11-15
Specifying Strobes for VCDE Pattern Input.	11-15
Initializing the Fault List	11-18
Using the Add Faults Dialog Box	11-18
Using the add faults Command	11-18
Performing Good Machine Simulation	11-19
Setting Up the Good Machine Simulation	11-19
Using the Run Simulation Dialog Box.	11-19
Using the set/run simulation Commands	11-19
Performing Fault Simulation	11-20
Setting Up for Fault Simulation	11-20

Using the Run Fault Simulation Dialog Box	11-20
Using the run fault_sim Command	11-20
Writing the Fault List	11-21
Combining ATPG and Functional Test Patterns.....	11-21
Creating Independent Functional and ATPG Patterns	11-22
Creating ATPG Patterns After Functional Patterns	11-22
Creating Functional Patterns After ATPG Patterns	11-23
12. Test Pattern Data	
Controlling ATPG	12-2
ATPG Modes	12-2
Basic-Scan ATPG.....	12-2
Fast-Sequential ATPG	12-2
Full-Sequential ATPG	12-3
ATPG Algorithm Sequence.....	12-3
Full-Sequential ATPG Limitations.....	12-4
Random Decision Option.....	12-5
Specifying a Test Coverage Target Value.....	12-5
Limiting the Number of Patterns	12-5
Limiting the Number of Aborted Decisions	12-6
Using Multiple-Session Test Generation	12-6
Splitting Patterns	12-7
Extracting a Pattern Subrange.....	12-7
Merging Multiple Pattern Files	12-8
Using Pattern Files Generated Separately	12-8
Compressing Patterns	12-10
Balancing Pattern Compaction and CPU Runtime	12-10
Compression Reports	12-11
Pattern Output	12-13
Pattern Input	12-13
STIL Functional Pattern Input	12-14
Verilog Functional Pattern Input	12-15
WGL Functional Pattern Input	12-19
Running Logic Simulation	12-21
Comparing Simulated and Expected Values	12-21

Patterns in the Simulation Buffer	12-22
Sequential Simulation Data	12-23
Single-Point Failure Simulation	12-23
GSV Display of a Single-Point Failure	12-24
Per-Cycle Pattern Masking	12-25
Masks File	12-25
Running the Flow	12-26
Limitations	12-27
13. PatternMap	
Overview of PatternMap	13-2
PatternMap Requirements	13-2
Embedded Module Requirements	13-2
Functional Test Pattern Requirements	13-3
About Pattern Mapping	13-6
Independent Pattern Mapping	13-7
Dependent Pattern Mapping	13-8
Preparing the Module	13-9
Running PatternMap	13-9
Working Example	13-14
The Embedded Module	13-14
Running TetraMAX PatternMap	13-15
Analyzing the Report	13-16
Report Description	13-17
Enhancing the Quality of Results	13-18
Modifying the PatternMap Results	13-18
Fault Grading Your Vectors from TetraMAX PatternMap	13-19
Tips for Using PatternMap	13-20
Applying a Set of Vectors to Several Instances	13-20
Using Clock As Scan Chain and Memory Write Clocks	13-21
14. Transition Delay Fault ATPG	
Transition Delay Fault Model	14-2
Transition Fault ATPG Modes	14-4
STIL Protocol for Transition Faults	14-6

Creating Transition Fault Waveform Tables	14-6
DRC for Transition Faults	14-8
Limitations of Transition Delay Fault ATPG.....	14-9
Specifying Transition Delay Faults	14-9
Selecting the Fault Model	14-10
Adding Faults to the Fault List.....	14-10
Reading a Fault List File	14-10
Pattern Generation for Transition Delay Faults	14-11
Set ATPG Command.....	14-11
Set Delay Command	14-11
Run ATPG Command	14-12
Pattern Compression for Transition Faults	14-13
Report Faults Command	14-13
Write Faults Command	14-13
Pattern Formatting for Transition Delay Faults.....	14-14
MUXClock Support for Transition Patterns	14-16
Specifying Setup Timing Exceptions From an SDC File.....	14-16
Reading an SDC File.....	14-17
Interpreting an SDC File	14-18
How TetraMAX Interprets SDC Commands	14-20
Controlling Clock Timing	14-21
Controlling ATPG Interpretation.....	14-21
Controlling Timing Exceptions Simulation for Stuck-at Faults	14-21
Reporting SDC Results	14-22
Limitations	14-23
Small Delay Defect Testing	14-25
Basic Usage Flow	14-25
Extracting Slack Data from PrimeTime.....	14-25
Understanding the Slack Data File Format.....	14-26
Utilizing Slack Data in the TetraMAX Flow.....	14-26
Command Support	14-27
Special Elements of Small Delay Defect Testing.....	14-29
Allowing Variation From the Minimum-Slack Path	14-29
Defining Faults of Interest.....	14-30
Reporting Faults	14-30
Limitations	14-31

Engine and Flow Limitations	14-31
Algorithmic Limitations	14-31
Limitations in Support for TetraMAX Primitives	14-32

15. Path Delay Fault Testing

Path Delay Fault Theory	15-2
Definitions	15-2
Models for Manufacturing Tests	15-4
Models for Characterization Tests	15-5
Testing I/O Paths	15-6
Path Delay Testing Flow	15-7
Obtaining Delay Paths	15-9
Importing PrimeTime Path Lists	15-9
Path Definition Syntax	15-10
Translating Timing Exceptions	15-12
Generating Path Delay Tests	15-12
Set Delay Options	15-13
Reading and Reporting Path Lists	15-13
Analyzing Path Rule Violations	15-14
Viewing Delay Paths	15-14
Path Delay ATPG Options	15-14
Internal Loopback and False/Multicycle Paths	15-14
Creating DSMTTest WaveformTables	15-15
Maintaining DSMTTest Waveform Table Information	15-18
Limitations of Waveform Table Support for Path Delay Patterns	15-18
MUXClock Support for Path Delay Patterns	15-19
Enabling MUXClock Functionality	15-19
Delay Test Vector Format	15-19
Limitations of MUXClock Support for Path Delay Patterns	15-22
ATPG Requirements to Support MUXClock	15-22
Untested Paths	15-22
False Paths	15-22
Untestable Paths	15-23
Reporting Untestable Paths	15-24
Analyzing Untestable Faults	15-25
TetraMAX Commands for Path Delay Fault Testing Example	15-26

16. Quiescence Test Pattern Generation

Why Do IDDQ Testing?	16-2
CMOS Circuit Characteristics	16-2
IDDQ Testing Methodology	16-4
Types of Defects Detected	16-5
Number of IDDQ Strobes.....	16-6
About IDDQ Pattern Generation	16-6
Limitations.....	16-8
Fault Models	16-8
Pseudo-Stuck-At Fault Model	16-8
Toggle Fault Model	16-9
DRC Rule Violations	16-9
Generating IDDQ Test Patterns	16-11
iddq_capture	16-11
Off-Chip IDDQ Monitor Support.....	16-12
Specifying Additional Signals in the Netlist.....	16-12
Defining iddq_capture to Support Additional Signals	16-13
IDDQ Commands	16-17
Set Faults.....	16-17
Set IDDQ	16-17
Add ATPG Constraints	16-18
Design Principles for IDDQ Testability	16-19
I/O Pads.....	16-19
Buses.....	16-19
RAMs and Analog Blocks	16-20
Free-Running Oscillators.....	16-20
Circuit Design	16-20
Power and Ground.....	16-21
Models With Switch/FET Primitives.....	16-21
Connections.....	16-22
IDDQ Design-for-Test Rule Summary	16-22
Additional System-on-a-Chip Rules	16-23

17. Bridging Fault ATPG

Understanding Bridging Defects	17-2
Detecting Bridging Faults	17-2
Bridge Locations	17-2
Strength-Based Patterns	17-3
Bridging Fault Flows	17-4
Bridging Faults and the Overall TetraMAX Flow	17-4
Bridging Fault Flow in TetraMAX.....	17-5
Setup	17-5
Input Faults.....	17-6
Manipulating the Fault List	17-6
Examining the Fault List.....	17-7
Fault Simulation	17-7
Running ATPG.....	17-7
Analysis	17-8
Example Script	17-8
Using Star-RCXT to Generate a Bridge Fault List	17-9
TCAD Characterization	17-10
Extracting Capacitance	17-11
Running Star-RCXT in GUI or Batch Mode	17-11
Coupling Capacitance Report.....	17-13
Running TetraMAX	17-13
Bridging Fault Model Limitations.....	17-14

18. ATPG Distributed Processing

Command Summary.....	18-2
Identifying a Work Directory.....	18-2
Adding Machines to the Distributed Processor List.....	18-2
Removing a Machine From the Distributed Processor List	18-2
Controlling Timeouts	18-3
Reporting Current Slave Machines	18-3
Starting Distributed ATPG	18-3
Distributed Process Flow	18-3
Verifying Your Environment	18-4
Remote Shell Considerations	18-5

Tuning Your .cshrc File	18-5
Checking the Load Sharing Setup.....	18-6
Using Distributed Processing: Step By Step.....	18-6
Building the Design and Running DRC	18-6
Sample Script	18-6
Selecting the Fault Model and Creating the Fault List.....	18-6
Distributed Fault Simulation	18-6
Distributed ATPG	18-6
Setting Up the Distributed Environment	18-7
Setting Up a Distributed Environment With Load Sharing.....	18-9
Starting Distributed Fault Simulation	18-10
Events After Starting A Distributed Run	18-11
Interpreting Distributed Fault Simulation Results	18-12
Starting Distributed ATPG	18-14
Saving Results.....	18-16
Distributed Processor Log Files.....	18-16
Licensing Schemes.....	18-16
Licensing Examples.....	18-17
Limitations.....	18-17
19. Diagnosing Manufacturing Test Failures	
Providing Tester Failure Log Files	19-2
Providing a Pattern-Based Failure Log File.....	19-2
Providing a Cycle-Based Failure Log File	19-3
Cycle-based Failure Log File Format.....	19-4
Limitations	19-5
Pattern File	19-5
Split Patterns File.....	19-6
Translating Adaptive Scan Patterns Into Normal Scan Patterns	19-6
Example Flow	19-7
Limitations	19-8
Diagnosing Tester Failure.....	19-8
Using the Run Diagnosis Dialog Box.....	19-9
Using the run diagnosis Command	19-9
Performing Scan Chain Diagnosis.....	19-10

Understanding the Diagnosis Summary	19-11
Displaying Composite Fault Types	19-12
Reading Physical Data	19-13
Understanding the read layout Command Output.....	19-14
Viewing Physical Data in the Diagnosis Report.....	19-15
Adding Netlist Data In Image Files	19-16
20. Troubleshooting	
Reporting Port Names	20-2
Reviewing a Module Representation	20-2
Rerunning Design Rule Checking.....	20-4
Troubleshooting Netlists	20-4
Troubleshooting STIL Procedures	20-5
STIL load_unload Procedure.....	20-5
STIL Shift Procedure	20-6
STIL test_setup Macro.....	20-7
Correcting DRC Violations by Changing the Design	20-8
Analyzing the Cause of Low Test Coverage.....	20-8
Where Are the Faults Located?.....	20-9
Why Are the Faults Untestable or Difficult to Test?.....	20-10
Using Justification	20-11
Completing an Aborted Bus Analysis	20-12
21. Using Tcl With TetraMAX	
Converting TetraMAX Command Files to Tcl	21-2
Translating Native-mode Scripts	21-2
Tcl Syntax and TetraMAX Commands	21-2
Abbreviating Commands and Options.....	21-3
Using Tcl Special Characters	21-4
Using the Result of a Command	21-4
Using Built-In Commands	21-5
TetraMAX Extensions and Restrictions.....	21-5
Redirecting Output	21-6

Using the redirect Command	21-6
Getting the Result of Redirected Commands	21-7
Using the Redirection Operators	21-8
Using Command Aliases	21-8
Interrupting Commands	21-9
Using Command Files	21-9
Adding Comments	21-10
Controlling Command Processing When Errors Occur	21-10
Using a Setup Command File	21-10
Cross-Reference List	21-11

Appendix A. Test Concepts

Why Perform Manufacturing Testing?	A-2
What Are Fault Models?	A-2
Stuck-At Fault Models	A-2
Detecting Stuck-At Faults	A-3
Using Fault Models to Determine Test Coverage	A-5
IDQ Fault Model	A-5
Fault Simulation	A-6
Automatic Test Pattern Generation	A-6
Translation for the Manufacturing Test Environment	A-7
What Is Internal Scan?	A-7
Applying Test Patterns	A-9
Scan Design Requirements	A-10
Controllability of Sequential Cells	A-10
Observability of Sequential Cells	A-10
Full-Scan Design	A-11
Partial-Scan ATPG Design	A-12
What Is Boundary Scan?	A-12

Appendix B. ATPG Design Guidelines

ATPG Design Guidelines	B-2
Internally Generated Pulsed Signals	B-2
Guideline 1	B-2
Clock Control	B-6

Guideline 2	B-6
Pulsed Signals to Sequential Devices	B-9
Guideline 3	B-9
Multidriver Nets	B-11
Guideline 4	B-11
Bidirectional Port Controls	B-13
Guideline 5	B-13
Guideline 6	B-14
Clocking Scan Chains: Clock Sources, Trees, and Edges	B-14
Guideline 7	B-14
Guideline 8	B-16
Guideline 9	B-18
Guideline 10	B-19
Protection of RAMs During Scan Shifting	B-21
Guideline 11	B-21
RAM and ROM Controllability During ATPG	B-21
Guideline 12	B-21
Pulsed Signal to RAMs and ROMs	B-23
Guideline 13	B-23
Bus Keepers	B-25
Guideline 14	B-25
Guideline 15	B-26
Guideline 16	B-26
Ports for Test I/O	B-27
Existing Ports for Scan Chain Input and Output	B-27
Existing Ports for Any Test Usage	B-27
Checklists for Quick Reference	B-28
ATPG Design Guideline Checklist	B-28
Ports for Test I/O Checklist	B-29

Appendix C. Importing Designs From DFT Compiler

Importing a Design From DFT Compiler	C-2
--	-----

Appendix D. Utilities

Ltran Translation Utility	D-2
Ltran in the Shell Mode	D-2
FTDL/TDL91/TSTL2 Configuration Files	D-2

Understanding the Configuration File	D-4
Customizing the FTDL Configuration File.....	D-4
Customizing the TDL91 Configuration File.....	D-5
Customizing the TSTL2 Configuration File.....	D-6
Additional Controls	D-6
Support for Other Formats	D-7
Configuration File Syntax.....	D-8
OVF_BLOCK Statements.....	D-8
PROC_BLOCK Statements	D-9
TVF_BLOCK Statements	D-11
Translating PrimeTime Timing Exceptions.....	D-13
Write Timing Exceptions Flow	D-14
Syntax	D-15
Example.....	D-16
Summary of Translations.....	D-16
Generating PrimeTime Constraints	D-17
Input Requirements	D-17
Start Tcl Command Parser Mode	D-18
Set Up TetraMAX.....	D-18
Perform an Analysis for Each Mode	D-20
Implementation	D-22

Appendix E. STIL Language Support

STIL Overview	E-2
IEEE Std. 1450-1999.....	E-2
IEEE Std. 1450.1 Design Extensions to STIL	E-3
TetraMAX ATPG and STIL.....	E-3
STIL Conventions in TetraMAX	E-4
Use of STIL Procedures	E-4
Context of Partial Signal Sets in Procedure Definitions.....	E-5
Use of STIL SignalGroups	E-6
WaveformCharacter Interpretation	E-7
IEEE Std. 1450.1 Extensions Used in TetraMAX.....	E-8
Vector Data Mapping Using \m	E-8
Vector Data Mapping Using \j	E-11
Syntax.....	E-11

General Example	E-11
Signal Constraints Using Fixed and Equivalent.....	E-15
ScanStructures Block	E-15
Elements of STIL Not Used by TetraMAX	E-16
TetraMAX STIL Output	E-16
TetraMAX STIL Input.....	E-18

Appendix F. STIL99 vs. STIL

Index

Preface

This preface includes the following sections:

- [What's New in This Release](#)
- [About This Manual](#)
- [Customer Support](#)

What's New in This Release

Information about new features, enhancements, and changes, along with known problems and limitations and resolved Synopsys Technical Action Requests (STARs), is available in the *TetraMAX ATPG Release Notes* in SolvNet.

To see the *TetraMAX ATPG Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

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

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

2. Select TetraMAX ATPG, and then select a release in the list that appears.

About This Manual

The *TetraMAX ATPG User Guide* describes TetraMAX ATPG usage and methodology. The TetraMAX product is used to check testability design rules and to automatically generate manufacturing test vectors for a logic design.

This guide provides some background material on design-for-test (DFT) concepts, especially test terminology and scan design techniques. You can obtain more information on TetraMAX features and commands by using the TetraMAX help commands.

Audience

This manual is intended for design engineers who have ASIC design experience and some exposure to testability concepts and strategies.

This manual is also useful for test engineers who incorporate the test vectors produced by TetraMAX ATPG into test programs for a particular tester or who work with DFT netlists.

Related Publications

For additional information about TetraMAX ATPG, see Documentation on the Web, which is available through SolvNet at the following address:

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

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

- DFT Compiler
- DC Compiler

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Courier	Indicates command syntax.
<i>Courier italic</i>	Indicates a user-defined value in Synopsys syntax, such as <i>object_name</i> . (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
Courier bold	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[]	Denotes optional parameters, such as <i>pin1 [pin2 ... pinN]</i>
	Indicates a choice among alternatives, such as <i>low medium high</i> (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
—	Connects terms that are read as a single term by the system, such as <i>set_annotated_delay</i>
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and "Enter a Call to the Support Center."

To access SolvNet, go to the SolvNet Web page at the following address:

<http://solvnet.synopsys.com>

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar or in the footer.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to <http://solvnet.synopsys.com> (Synopsys user name and password required), and then clicking "Enter a Call to the Support Center."
- Send an e-mail message to your local support center.
 - E-mail support_center@synopsys.com from within North America.
 - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr
- Telephone your local support center.
 - Call (800) 245-8005 from within the continental United States.
 - Call (650) 584-4200 from Canada.
 - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr

1

TetraMAX Overview

TetraMAX is a high-speed, high-capacity automatic test pattern generation (ATPG) tool. It can generate test patterns that maximize test coverage while using a minimum number of test vectors for a wide variety of design types and design flows. It is well suited for designs of all sizes up to millions of gates.

This chapter contains the following sections:

- [TetraMAX ATPG Features](#)
- [Design Flow Using DFT Compiler and TetraMAX](#)

TetraMAX ATPG Features

Functional testing and stuck-at testing are the traditional circuit testing methods. With functional testing, the tester applies a sequence of input data and detects the resulting sequence of output data. The output sequence is compared against the expected behavior of the device. Functional testing exercises the device as it would actually be used in the target application. However, this type of testing has only a limited ability to test the integrity of the device's internal nodes.

With scan testing, the sequential elements of the device are connected into chains and used as primary inputs and primary outputs for testing purposes. Using ATPG techniques, you can test a much larger number of internal faults than with functional testing alone. The goal of ATPG is to set all nodes of the circuit to both 0 and 1, and to propagate any defects to nodes where they can be detected by the test equipment.

TetraMAX has the following major ATPG capabilities:

- Can read design netlists in Verilog, VHDL, and EDIF formats; and test protocol information in STIL format
- Can write test pattern files in a variety of standard and proprietary formats: WGL, STIL, Verilog, VHDL, Fujitsu TDL, TI TDL91, and Toshiba TSTL2
- Offers a choice of ATPG modes:
 - Basic-Scan ATPG, an efficient combinational-only mode for full-scan designs
 - Fast-Sequential ATPG for limited support of partial-scan designs
 - Full-Sequential ATPG for maximum test coverage in partial-scan designs
- Supports the following design-for-test (DFT) styles:
 - Various scan flip-flop types (multiplexed flip-flop, master, slave, transparent latch, and so on)
 - Internal, nondecoded three-state buses
 - Bus keepers
 - RAM and ROM models
 - Proprietary and standard test controllers (such as IEEE 1149.1-compliant boundary scan)
- Produces and verifies ATPG patterns that avoid bus contention and float conditions
- Offers interactive analysis and debugging with the Graphical Schematic Viewer (GSV), for easy analysis of design rule violations and other conditions found in the design

- Provides links to Verilog and VHDL simulators
 - Provides an integrated fault simulator that supports fault simulation of functional patterns
 - Can perform direct automated test equipment (ATE) diagnostics, allowing you to quickly map a test failure to a fault site in the design
-

ATPG Modes

TetraMAX offers three different ATPG modes: Basic-Scan, Fast-Sequential, and Full-Sequential.

Basic-Scan ATPG

In Basic-Scan mode, TetraMAX operates as a full-scan, combinational-only ATPG tool. To get high test coverage, the sequential elements need to be scan elements. Combinational ROMs can be used to gain coverage of circuitry in their shadows in this mode.

Fast-Sequential ATPG

Fast-Sequential ATPG provides limited support for partial-scan designs. In this mode, multiple capture procedures are allowed between scan load and scan unload, allowing data to be propagated through nonscan sequential elements in the design such as functional latches, non-scan flops, and RAMs and ROMs. However, all clock and reset signals to these nonscan elements must still be directly controllable at the primary inputs of the device. You enable the Fast-Sequential mode and specify its effort level by using the `-capture_cycles` option of the `set atpg` command.

Full-Sequential ATPG

Full-Sequential ATPG, like Fast-Sequential ATPG, supports multiple capture cycles between scan load and unload, thus increasing test coverage in partial-scan designs. Clock and reset signals to the nonscan elements do not need to be controllable at the primary inputs; and there is no specific limit on the number of capture cycles used between scan load and unload. You enable the Full-Sequential mode by using the `-full_seq_atpg` option of the `set atpg` command. The Full-Sequential mode supports an optional feature called Sequential Capture. Defining a sequential capture procedure in the STIL file lets you compose a customized capture clock sequence applied to the device during Full-Sequential ATPG. For example, you can define the clocking sequence for a two-phase latch design, where CLKP1 is followed by CLKP2. This feature is enabled by the `-clock -seq_capture` option of the `set drc` command. Otherwise, the tool will create its own sequence of clocks and other signals in order to target the as-yet-undetected faults in the design.

Supported Fault Models

TetraMAX supports test pattern generation for five types of fault models: stuck-at faults, IDDQ faults, transition delay faults, path delay faults, and bridging faults.

Stuck-At

The stuck-at fault model is the standard model for test pattern generation. This model assumes that a circuit defect behaves as a node stuck at either 0 or 1. The test pattern generator attempts to propagate the effects of these faults to the primary outputs and scan cells of the device, where they can be observed at a device output or captured in a scan chain.

Transition

The transition delay fault model is used to generate test patterns to detect single-node slow-to-rise and slow-to-fall faults. For this model, TetraMAX launches a logical transition upon completion of a scan load operation and uses a capture clock procedure to observe the transition results. (This feature is licensed separately. For more information, see [Chapter 14, “Transition Delay Fault ATPG.”](#))

Path Delay

The path delay fault model tests and characterizes critical timing paths in a design. Path delay fault tests exercise the critical paths at-speed (the full operating speed of the chip) to detect whether the path is too slow because of manufacturing defects or variations. For more information, see [Chapter 15, “Path Delay Fault Testing.”](#)

IDDQ

The IDDQ fault model assumes that a circuit defect will cause excessive current drain due to an internal short circuit from a node to ground or to a power supply. For this model, TetraMAX does not attempt to observe the logical results at the device outputs. Instead, it tries to toggle as many nodes as possible into both states while avoiding conditions that violate quiescence, so that defects can be detected by the excessive current drain that they cause.

Bridging

The bridging fault model tests for shorts between two normally unconnected instance pins or net names. This model reports candidate defects as types bridging fault at 0 (ba0) or bridging fault at 1 (ba1), and victim and aggressor node sets. For more information, see [Chapter 17, “Bridging Fault ATPG.”](#)

Design Flow Using DFT Compiler and TetraMAX

TetraMAX is compatible with a wide range of design-for-test tools such as DFT Compiler. The design flow using DFT Compiler and TetraMAX ATPG is recommended for maximum ease of use and quality of results.

[Figure 1-1](#) shows how TetraMAX ATPG fits into the DFT Compiler design-for-test flow for a module or a medium-sized design of less than 750K gates.

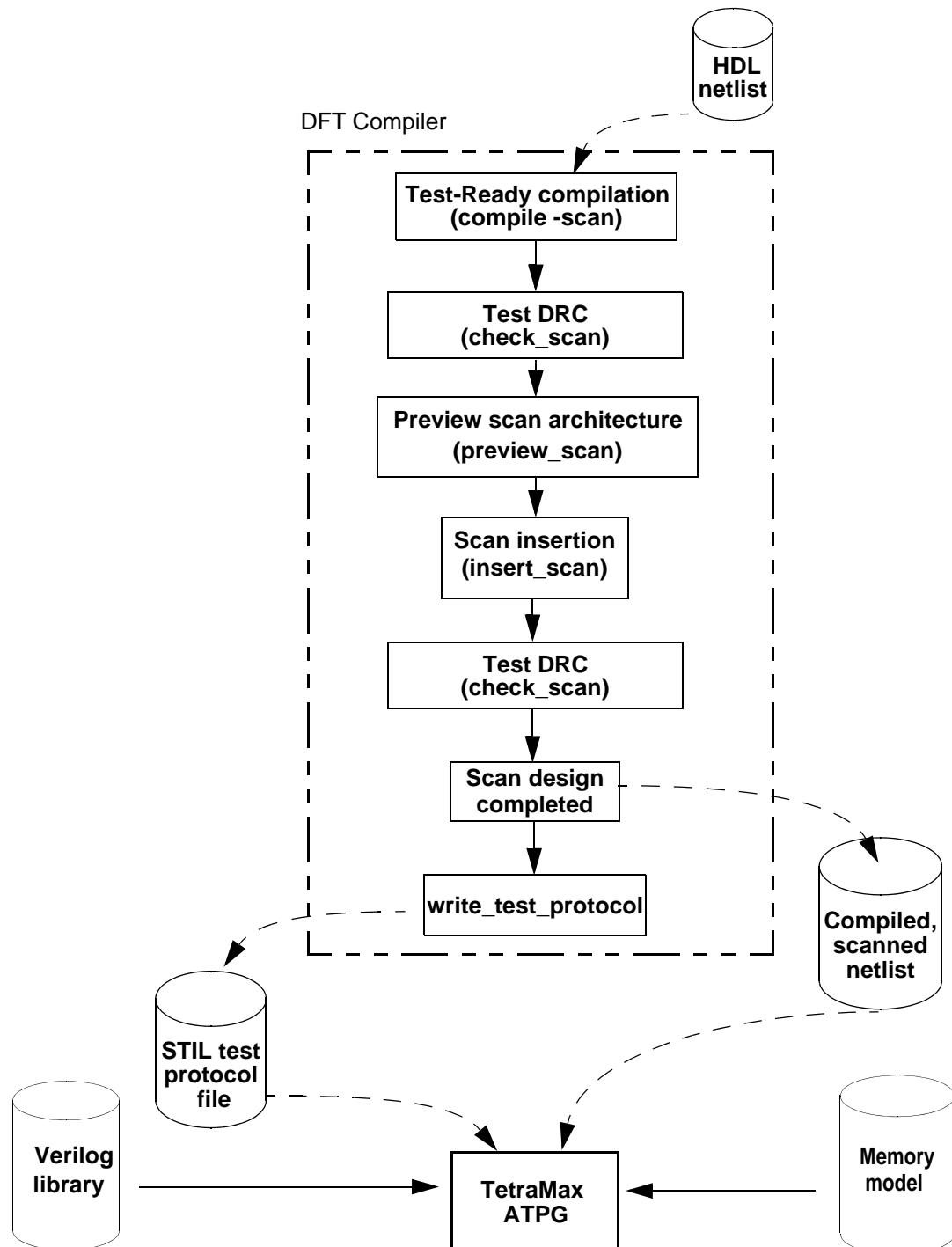
Starting with an HDL netlist at the register transfer level (RTL), within DFT Compiler you first run a Test-Ready compilation, which integrates logic optimization and scan replacement.

The `compile -scan` command maps all sequential cells directly to their scan equivalents. At this point, you still don't know whether the sequential cells meet the test design rules.

Next, you perform test design rule checking; the `check_scan` command reports any sequential cells that violate test design rules.

After you resolve the DRC violations, you run the `preview_scan` command to examine the scan architecture that will be synthesized by the `insert_scan` command. You can repeat this procedure until you are satisfied with the scan architecture, then run the `insert_scan` command, which implements the scan architecture.

Figure 1-1 Design Flow for a Module or Medium-Sized Design



Finally, you rerun the `check_scan` command to identify any remaining DRC violations and to infer a test protocol. For details about the DFT Compiler design flow through completion of the scan design, see the *DFT Compiler Scan Synthesis User Guide*.

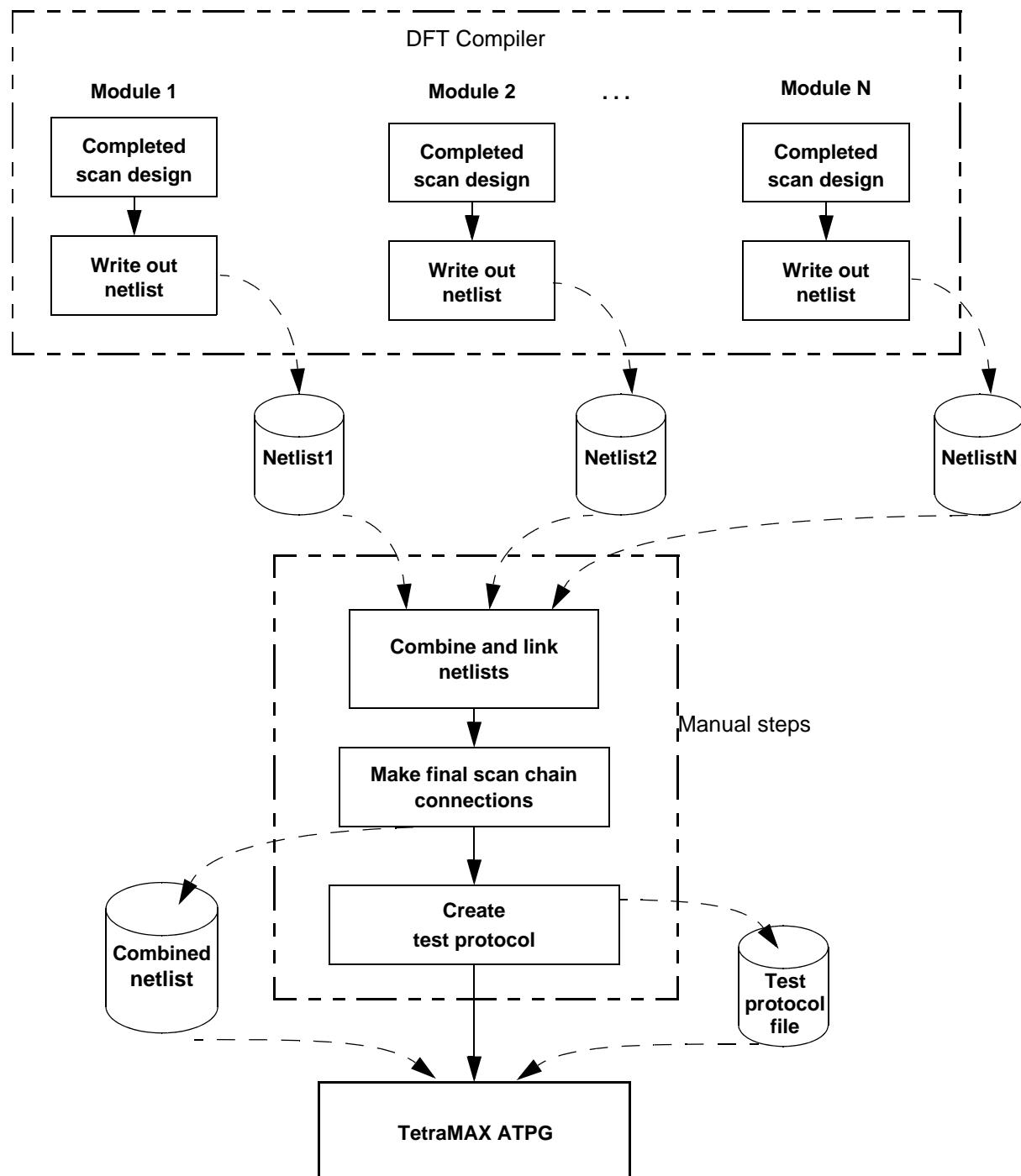
When your netlist is free of DRC violations, it is then ready for ATPG. For medium-sized and smaller designs, DFT Compiler provides the `write_test_protocol` command, which allows you to write out a STIL protocol file. TetraMAX then reads the STIL protocol file and design netlist.

Details of the TetraMAX portion of the design flow are provided in [Chapter 4, “ATPG Design Flow.”](#)

[Figure 1-2](#) shows the design flow for a design that is too large for test protocol file generation from a single netlist (about 750K gates or larger). In this case, you follow the design flow shown in [Figure 1-1](#) at the module level, using modules of 200K gates or fewer, to get the completed scan design for each module.

Then, as shown in [Figure 1-2](#), you start with the completed scan design for each module. You write the netlists, combine and link the netlists, and make the final scan chain connections, thus generating a combined netlist for the entire design. A test protocol file is created automatically. For information on creating a test protocol file, see [Chapter 9, “STIL Procedure Files.”](#) Finally, you use the combined netlist and the manually generated test protocol file as inputs to TetraMAX.

Figure 1-2 Design Flow for a Very Large Design



2

Running TetraMAX

This chapter describes the basic steps for starting, and operating TetraMAX. It contains the following sections:

- [Installation](#)
- [Licensing](#)
- [Invoking TetraMAX](#)
- [Setup Files](#)
- [Variables](#)
- [Controlling TetraMAX Processes](#)

Installation

You can obtain the TetraMAX installation files by downloading them from Synopsys using electronic software transfer (EST) or File Transfer Protocol (FTP).

TetraMAX can be installed as a stand-alone product or over an existing Synopsys product installation (an “overlay” installation). An overlay installation shares certain support and licensing files with other Synopsys tools, whereas a stand-alone installation has its own independent set of support files. You specify the type of installation you want when you install the product.

An environment variable called `SYNOPSYS` specifies the location for the TetraMAX installation. You need to set this environment variable explicitly.

Complete installation instructions are provided in the *Installation Guide* that comes with each release of TetraMAX.

Specifying the Location for TetraMAX Installation

TetraMAX now requires the `SYNOPSYS` environment variable, a variable typically used with all Synopsys products. For backward compatibility, `SYNOPSYS_TMAX` can be used instead of the `SYNOPSYS` variable. However, TetraMAX looks for `SYNOPSYS` and if not found, then looks for `SYNOPSYS_TMAX`. If `SYNOPSYS_TMAX` is found, then it overrides `SYNOPSYS` and issues a warning that there are differences between them.

The conditions and rules are

- `SYNOPSYS` is set and `SYNOPSYS_TMAX` is not set. This is the preferred and recommended condition.
- `SYNOPSYS_TMAX` is set and `SYNOPSYS` is not set. The tool will set `SYNOPSYS` using the value of `SYNOPSYS_TMAX` and continue.
- Both `SYNOPSYS` and `SYNOPSYS_TMAX` are set. `SYNOPSYS_TMAX` will take precedence and `SYNOPSYS` is set to match before invoking the kernel.
- Both `SYNOPSYS` and `SYNOPSYS_TMAX` are set, and are of different values, then a warning message is generated similar to the following:

```
WARNING:  
WARNING: $SYNOPSYS and $SYNOPSYS_TMAX are set differently,  
using  
$SYNOPSYS_TMAX  
WARNING:      SYNOPSYS_TMAX = /mount/groucho/joeuser/tmax  
WARNING:      SYNOPSYS      = /mount/harpo/production/synopsys  
WARNING:  
WARNING: Use of SYNOPSYS_TMAX is outdated and support for  
this  
will be removed  
WARNING: in a future release. Please use SYNOPSYS instead.  
WARNING:
```

Licensing

To use TetraMAX, you must have a valid license key file on your machine. The license key file authorizes you to run TetraMAX (and other licensed Synopsys tools) for a specific calendar time period. The full path name of the file is specified by the environment variable `SYNOPSYS_KEY_FILE`.

The license file is keyed to the host ID of the machine, so the key is valid only on that machine.

TetraMAX automatically checks out a required license each time you invoke the program or execute a command that requires a specific license. It automatically releases all licenses when you exit from the program. Therefore, you do not need to be concerned about checking out or checking in licenses unless there are conflicting needs between multiple users.

The standard capabilities licensed for TetraMAX are

- Design compilation, design rule checking, and vector formatting
- Test vector generation and vector diagnosis
- Functional vector fault simulation

You can generate a report showing the licenses currently in use by TetraMAX using the `report licenses` command, as follows:

```
report licenses
```

To check out a specific license manually (to reserve the license for yourself), use the following command:

```
get licenses license_name
```

To release a specific license without exiting from TetraMAX (to make the license available to others), use the following command:

```
remove licenses license_name
```

Note:

You must retain at least one TetraMAX license when running the tool.

TetraMAX can check out a DFT Compiler license if you set the one these environment variables:

`TMAX_USE_DFT_LICENSE` - will always check out Test-Compiler

or

`TMAX_USE_ATPG_LICENSE` - always check out Test-Compile

For information on the individual licenses required for different TetraMAX features, see online Help for the `get licenses` command.

Invoking TetraMAX

Before you invoke TetraMAX, ensure that your `PATH` environment variable includes the path to the Synopsys tools, typically `$SYNOPSYS/bin`. Then invoke TetraMAX as follows:

```
tmax [ command_file ] [-env SYM value]... [-shell | -gui]  
[-iconify] [-nostartup] [-64] [-tcl] [-man | -help | -usage |  
-version] [-root <path_to_install_dir>] [-debug]
```

Note:

You can also abbreviate most command options using the first character or first three characters; for example, `-v` or `-ver` for `-version`.

Argument	Definition
<code>command_file</code>	Specifies the path name to the file that contains the commands to be run. If specified with a "<" redirection symbol, reads commands from <code>command_file</code> one command at a time.
<code>-env SYM <i>value</i></code>	Allows for the definition of an environment variable that can be used in commands. Such variables can be recognized only where file pathnames are used in commands, where "SYM" is the symbol name and <i>value</i> is the string value to use wherever \$SYM is found in a file path name. You must use a separate entry for every variable you define.

Argument	Definition
-shell -gui	<p>For <code>-shell</code> when used with <i>command_file</i>, specifies to execute the command file when you invoke TetraMAX. Only a command-line interface is provided. You can also define an environment variable <code>TMSHELL=1</code>, which is equivalent to using <code>-shell</code>. For example, to run a batch script as a background process that is also immune to hangups, enter:</p> <pre>% tmax -shell run.scr >& /dev/null &</pre> <p>For <code>-gui</code> (the default), forces use of the windows version of the tool. Overrides the environment variable <code>TMSHELL</code> if it is defined. Launches two executables—one for the kernel and one for the graphical interface.</p>
-man -help -usage -version	<p>For <code>-help</code>, prints the description of this command.</p> <p>For <code>-man</code>, invokes online Help to view the online man pages (does not consume a license).</p> <p>For <code>-usage</code>, prints a summary of command switches and arguments.</p> <p>For <code>-version</code>, reports the version.</p>
-iconify	For the graphical version of the tool, iconify the window after invoking. Has no effect on the shell version. You can also define the environment variable <code>TMAX_ICONIFY</code> to any nonnull string to always start the window process in an iconified mode.
-nostartup	Disables the automatic execution of startup files. Startup files are searched for in a number of locations including the install area of TetraMAX, the user's home directory, the current working directory, and as specified by the <code>TMAXRC</code> environment variable.
-64	Runs 64-bit address enabled executable. This implies <code>-shell</code> . You can also define the environment variable <code>TMAX_64BIT</code> to any nonnull string to always invoke the 64-bit address enabled executable. See " "64-Bit Mode on Solaris and HP-UX Platforms" on page 2-7 " for additional information.
-tcl	Selects the Tcl command-line parser as the default command processor. You can also define the <code>TMAX_TCL</code> environment variable to any nonnull string to always start in the Tcl command parser mode.
-debug	Print setup and internal information prior to invoking the executable.

Methods for Invoking

The following methods can be used for invoking TetraMAX

Invoke with ¹	You get	Process list
tmax	32-bit kernel plus GUI	tmax ² , tmax32, tmaxgui ³
tmax -64 [bit]	64-bit kernel plus GUI	tmax, tmax64, tmaxgui
tmax -shell	32-bit kernel	tmax, tmax32
tmax -shell -64	64-bit kernel	tmax, tmax64
tmax -man	Online help	tmax, netscape
tmax -help	List of options	--

1. *The order of switches is not important.*
2. *The tmax process is the invoking shell. It will be CPU idle once the kernel launches but remains open so that the kernel has a transcript window in which to display output.*
3. *The tmaxgui process is always a 32-bit process.*

There are two environment variables that can be set in the user's environment to avoid having to constantly specify switches:

- TMAX_SHELL if defined as nonnull string, implies “-shell”
- TMAX_64BIT if defined as nonnull string, implies “64-bit”

If you have either of these environment variables defined, you can still override them by invoking

```
% tmax -32bit      // overrides TMAX_64BIT=1  
% tmax -gui        // overrides TMAX_SHELL=1
```

Specifying a command file containing TetraMAX commands can be done in a number of ways:

```
% tmax command_file [other_args]...
```

OR

```
% tmax [other_args]... < command_file
```

OR

within a script as a “here” document:

```
#!/bin/sh
tmax [other_args] -shell <<!
source command_file
exit -force
!
```

OR

```
% tmax [other_args]
BUILD> source command_file
```

Running Background Jobs

When TetraMAX is running in background mode, such as

```
% tmax command_file [other_args]... -shell &
```

it is recommended that the `set commands noabort` command be placed at the top of the command file. The command file should also end with an `exit -force` command. If you fail to do this and a problem occurs during the processing of the command file causing the command file to abort, then TetraMAX will stop and wait for more input. Because there is no standard-in connected to the background process, TetraMAX will hang indefinitely and you will have to manually kill the process.

Predefined Aliases

A number of aliases are predefined automatically when TetraMAX is invoked by the presence of the Synopsys-supplied startup file:

```
$SYNOPSYS/admin/setup/tmax.rc
```

This startup file is similar in operation to the `.synopsys_dc.setup` file for Design Compiler.

64-Bit Mode on Solaris and HP-UX Platforms

TetraMAX ATPG supports 64-bit operation on 64-bit Solaris and HP-UX platforms. When you run TetraMAX ATPG in 64-bit mode, the upper limit for virtual address space is extended beyond the usual limit imposed by the 32-bit mode, allowing you to process larger designs without running out of memory.

The 64-bit version is automatically installed together with the 32-bit version when you specify sparcOS5 or HP-UX 11.0 as the platform during installation. If you attempt to use the 64-bit mode on a 32-bit platform, TetraMAX returns an error message.

To invoke TetraMAX ATPG in 64-bit mode, use the `-64bit` switch:

```
% tmax -64bit [other options]
```

An alternative method is to set the environment variable `TMAX_64BIT` to the string `true` (or to any other string other than null):

```
% setenv TMAX_64BIT true  
% tmax [other options]
```

64-Bit Support for GUI Mode

You can run the GUI in either 32- or 64-bit mode. The 64-bit kernel enables you to make use of the Graphical Schematic Viewer (GSV) window to analyze the supported violations happening during the BUILD, DRC, or TEST modes.

Crashes and What to Expect

In the new dual-process (or split-GUI) model, there are two independent processes running that communicate with each other. Therefore this mode creates the possibility of

- kernel crashing, but the GUI is still running
- GUI crashing, but the kernel is still running

Both processes have been written to detect when the other process is missing and to try to do the safe thing. So, when the kernel crashes, the GUI will notify you that no communication with the kernel is possible. You should then exit the GUI process. If the GUI crashes, the kernel should also exit. You can check the process list with a command such as `ps` and give a helping hand to killing off a kernel that was started with a GUI, and then the GUI process terminates prematurely.

Setup Files

If you have a command file that you always want TetraMAX to execute at startup, you can cause TetraMAX to execute the file automatically by making it a setup command file.

When you invoke TetraMAX, it searches for a setup command file named `.tmaxrc` or `tmax.rc` in the current directory and executes it automatically (Note: in Tcl mode, TetraMAX searches for `.tmaxtclrc` or `tmaxtc.rc`). If you specify another command file in the command line, TetraMAX first executes the setup file and then the command file.

TetraMAX executes command files from multiple locations. There are a total of four possible startup files. They are listed here in order of execution:

- \$SYNOPSYS/admin/setup/tmax.rc
- \$TMAXRC (if defined)
- \$HOME/.tmaxrc or \$HOME/.tmaxtclrc
- .tmaxrc, tmax.rc, .tmaxtclrc, or tmaxtcl.rc (placed in the current working directory)

Note the following:

- TetraMAX executes all existing startup files in the previous list in the order shown, not just the first one found.
- TetraMAX does not automatically include a global setup file for Tcl mode in the \$SYNOPSYS/admin/setup directory. If you are running TetraMAX in Tcl mode (using the -tcl option) and want to use a command setup file, you will need to name it either .tmaxtclrc or tmaxtcl.rc, and then place it in the directory where TetraMAX was started or in your home directory.
- By default, a startup command file does not echo commands to the transcript. To see the commands as they are executed, use a set messages display command at the beginning of the startup file.

Variables

TetraMAX supports limited use of variables in commands and command files. Variables are accepted only as the prefix (or first) string of a file path name argument. No other arguments or options of commands support the use of variables.

Variable usage is recognized by the leading dollar sign “\$” followed by the variable name. Here are some examples:

```
set message log $MYLOG -replace  
read netlist $LIBDIR/cmos/verilog/*.v  
write patterns $tmp/testbench.v -format verilog -replace
```

Two types of variables are supported:

- UNIX environment variables
- User-defined environment variables

UNIX environment variables are defined in the standard method for the shell in use, typically with the `setenv` or `set` and `export` commands. These must be defined in the shell environment prior to invoking TetraMAX.

User-defined environment variables are defined in the TetraMAX invocation line as follows:

```
% tmax -env MYLOG save/tmax.log -env tmp /tmp
```

Multiple variable/value pairs can be defined by repeating the `-env` argument. The current setting of any user-defined environment variable is visible with the `report settings` command. A variable defined with `-env` will override any existing environment variable with the same name.

Controlling TetraMAX Processes

This section contains information on starting and stopping the TetraMAX graphical user interface (GUI) in the shell mode, interrupting long processes, setting the workspace size, and saving GUI preferences.

Starting the TetraMAX GUI

In the shell mode, enter the following command to display the TetraMAX GUI in its current state:

```
gui_start
```

The TetraMAX GUI appears. The context is switched to listen only to the GUI console. To exit the GUI, use the `gui_stop` command.

Stopping the TetraMAX GUI

After you have started displaying the TetraMAX GUI (using the `gui_start` command), enter the following command to exit the GUI:

```
gui_stop
```

This command stops the TetraMAX GUI session and reverts to the TetraMAX shell command prompt. If you did not use the `gui_start` command to start the GUI, the `gui_stop` command exits the TetraMAX application. You can also use the `gui_stop` command from the pull-down menu: File > Exit GUI. If you use the `gui_stop` command before invoking TetraMAX using the `gui_start` command, TetraMAX displays an error message.

Interrupting a Long Process

While TetraMAX is processing commands, the Submit button in the TetraMAX window changes to a Stop button. To stop a process, click the Stop button. (Depending on the type of platform you are using, Control-c and Control-Break might also perform the Stop function.)

Note:

The Stop button usually works within a few seconds. However, interrupting a large file I/O process might take longer.

You can use the Stop function to interrupt these types of processes:

- Reading netlists
- Running ATPG, simulation, or fault simulation
- Reporting faults to the screen
- Running design rule checking (DRC)
- Building the design-level ATPG model
- Learning following an ATPG build
- Compressing patterns
- Writing patterns to a file
- Reading or writing fault lists from files
- Reporting scan cells
- Executing command files

When you use the Stop function to stop execution of a command file, TetraMAX normally stops execution of the entire file, unless the file contains the following line:

```
set command noabort
```

In that case, TetraMAX stops execution of only the current command in the file and continues execution of any commands following the stopped command. The `set command noabort` command is useful when you want a file to continue command file execution even though an error might occur.

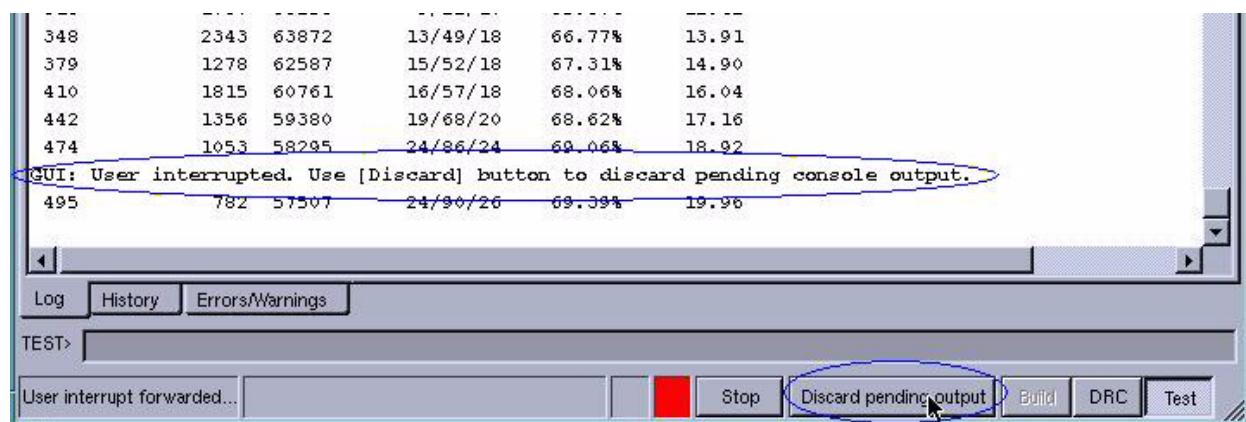
Discarding Pending Output

There are times when the Stop button doesn't interrupt lengthy output from TetraMAX. This occurs, for example, if you enter the following:

```
rep atpg con -all
```

To discard pending output, you can use the "Discard pending output" button located at the bottom of the TMAX console. This button is visible only after an interrupt (via the Stop button or ESC key) is detected, as shown in [Figure 2-1](#).

Figure 2-1 Discard Pending Output button



Note also that the Stop button will always be enabled and operational as long as the kernel is still processing the current command.

Adjusting the Workspace Size

In TetraMAX, you can set the maximum line length, maximum string length, and maximum number of decisions allowed during test pattern generation. If you encounter messages indicating that the limits for the line or string lengths have been reached, on the menu bar, choose Edit > Environment to display the Environment dialog box. Then click Kernel and increase the limits.

For information about the other options available to this command, see online Help for the `set workspace sizes` command.

As an alternative to the Environment dialog box, you can use the `set workspace sizes` command to change the workspace size settings.

Saving Preferences

TetraMAX enables you to save these GUI preferences so that the settings persist the next time you invoke TetraMAX.

When you invoke the TetraMAX GUI, it reads some of the default GSV preferences from the `tmax.rc` file. The Tetramax GUI has a Preferences dialog box to change the default settings to control the appearance and behavior of the GUI. These default settings control the size of main window, window geometry, application font, size, GSV preferences and other preferences. If you change the appearance and behavior of the GUI using the Preferences dialog box, TetraMAX saves your changes in the `$(HOME)/tmaxgui.rc` file before it exits. The next time you invoke TetraMAX, it

- Reads the default preferences from `tmax.rc`.
- Reads the preferences from the `$(HOME)/tmaxgui.rc` file. For the preferences that are listed in the `tmax.rc` file, the `$(HOME)/tmaxgui.rc` file has precedence over the `tmax.rc` file. For all other GUI preferences, TetraMAX uses the values from the `tmaxgui.rc` file to define the appearance and behavior of the GUI.

3

Command Interface

TetraMAX provides an easy-to-use command interface for interactive control of TetraMAX tasks, and a powerful command language that lets you execute command sequences in batch mode. Detailed online Help is available on commands, error messages, design flows, and many other TetraMAX topics.

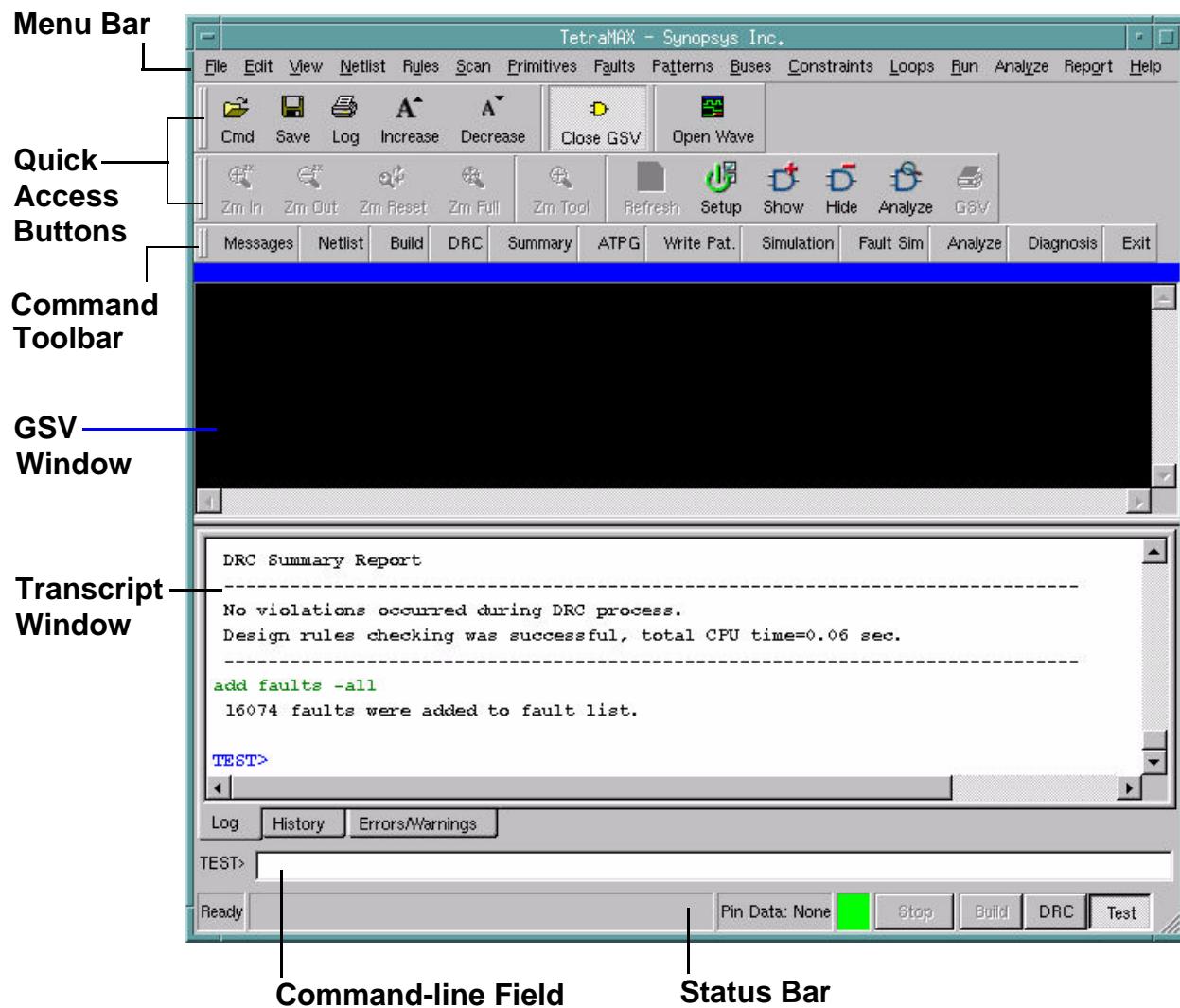
This chapter contains the following sections:

- [TetraMAX Main Window](#)
- [Command Entry](#)
- [Transcript Window](#)
- [Online Help](#)

TetraMAX Main Window

Figure 3-1 shows the main window of the TetraMAX graphical user interface (GUI). The major components in this window are (from top to bottom): the menu bar, the quick access buttons, the command toolbar, the Graphical Schematic Viewer (GSV) toolbar and window, the transcript window, the command-line window, and the status bar.

Figure 3-1 TetraMAX Main Window



Note:

The GSV window is not displayed when you first start TetraMAX. It first appears when you execute a command that requests a schematic display. For more information on the GSV window, see [Chapter 7, “Using the GSV For Review and Analysis.”](#)

The status bar, located at the very bottom of the main window, contains the STOP button and displays the state of TetraMAX (Kernel Busy/Ready), pin/block reference data, Pin Data details, red/green signal indicating the kernel busy/ready status, and the command mode indicator. For more information on the command mode indicator, see [“Command Mode Indicator” on page 3-4.](#)

Command Entry

The main window provides three different ways to interactively enter commands: by selecting a pull-down menu command from the menu bar at the top, by clicking a command button in the command toolbar or GSV toolbar, or by typing a command in the command-line window.

The pull-down menus and command buttons let you specify the command options in dialog boxes, whereas the command-line window is an entirely command-line-based entry method.

Menu Bar

The menu bar consists of a set of pull-down menus you use to select a desired action. This set of menus provides the most comprehensive set of command selections.

Command Toolbar and GSV Toolbar

The command toolbar is a collection of buttons you use to run TetraMAX commands. These buttons provide a fast and convenient alternative to using the pull-down menus or command-line window. Similarly, the GSV toolbar provides a fast way to control the contents of the GSV window.

By default, the command toolbar is displayed along the top of the main window, just below the menu bar; and the GSV toolbar is displayed along the left side of the GSV window. Both toolbars are “dockable”—that is, you can move and “dock” them to any of the four sides of the GSV window, or have them operate as free-standing windows.

To move the toolbar, position the pointer on the border of the toolbar (outside any of the buttons), press and hold the mouse button, drag the toolbar to the desired location, and release the mouse button.

Command-Line Window

The command-line window is located between the transcript window and the status bar.

Command Mode Indicator

The command mode indicator, located to the left of the status bar, displays either BUILD>, DRC>, or TEST>, depending on the operating mode currently enabled.

To change the command mode, use the Build, DRC, and Test buttons, located to the far right. The buttons are dimmed if you cannot change to that mode in the current context. To change to one of these modes, click the corresponding button. If an attempt to change the current mode fails, the command-line window remains unchanged and an error message appears in the transcript window.

Command-Line Entry Field

You type TetraMAX commands in the command-line text field at the bottom of the screen. To enter a command, click in the text field, type the command, and either click Submit or press Enter. After it has been entered, the command is echoed to the transcript, stored in the command history, and sent to TetraMAX for execution.

You can use the editing features Cut (Control-x), Copy (Control-c), and Paste (Control-v) in the command-line text field. If the command is too long for the text field, the text field automatically scrolls so that you can continue to see the end of the command entry.

The command line supports multiple commands. You can enter more than one command on the command line by separating commands with a semicolon.

You can enter two exclamation characters (! !) to repeat the last command. Entering ! ! xyz repeats the most recent command that begins with the string xyz.

Command Continuation

To continue a long command line over multiple lines, place at least one space followed by a backslash character (\) at the end of each line.

[Example 3-1](#) shows an example of the add atpg primitives command, which defines an ATPG primitive connected to multiple pins. Each pin path name is presented on a separate line using the backslash character. All five lines are treated as a single command.

Example 3-1 Command Continuation Across Multiple Lines

```
BUILD> add atpg primitives my_atpg_primi1 equiv \
/BLASTER/MAIN/CPU/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U1936/in1 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U16/in2 \
/BLASTER/MAIN/ALU_CORE/TP/CYCL/CDEC/U13/in0
```

Command History

The command history contains commands you have entered at the command line. To run a previous command, use the arrow keys to highlight the desired command, and then press Enter.

Another way to view a list of recent commands is to use the `report commands -history` command.

Stop Button

The Stop button is located to the right of the status bar. If TetraMAX is idle, the Stop button is dimmed. If TetraMAX is busy processing a command (and the command mode indicator displays <Busy>), the button is labeled Stop. Click this button to halt processing of the current command. TetraMAX might take several seconds to halt the activity.

Commands From a Command File

You can submit a list of commands as a file and have TetraMAX execute those commands in batch mode. Any line starting with `#` or `//` is treated as a comment and is ignored.

Although a command file can have any legal file name, for easy identification, you might want to use the standard extension `.cmd` (for example, `myfile.cmd`).

To run a command file, click the Cmd File button in the command toolbar, or enter the following in the command-line window:

```
> source filename
```

Command files can be nested. In other words, a command file can contain a `source` command that invokes another command file.

For an example of a command file, see “[Using Command Files](#)” on page 4-34.

Note:

The history list shows only the `source filename` command, not the commands executed in the command file.

Command Logging

Commands that you enter through menus, buttons, and the command-line window can be logged to a file along with all information reported to the transcript. By default, the command log contains the same information as the saved transcript. In addition, the command log contains comments from any command files that were used.

To turn on command logging (also called message logging), click the Set Msg button in the command toolbar to open the Set Messages dialog box, or type the following command:

```
> set messages log my_logfile.log
```

If the log file already exists, an error message is displayed unless you add the optional -replace or -append options, as follows:

```
> set messages log my_logfile.log -replace  
> set messages log my_logfile.log -append
```

If you intend to use the log file as an executable command file, use the -double_slash option in the set messages command. In that case, TetraMAX writes out the comment lines starting with a double slash, so that those lines are ignored when you use the log file as a command file.

Transcript Window

The transcript window is a read-only, scrollable window that displays the session transcript, including text produced by TetraMAX and commands entered in the command line and from the GUI. The transcript provides a record of all activities carried out in the TetraMAX session.

Setting the Keyboard Focus

Setting the keyboard focus in the transcript window allows you to use keyboard shortcuts and some keypad keys in the transcript window. You set the keyboard focus by clicking anywhere in the transcript window. A blinking vertical-bar cursor appears in the text where you have set the focus.

Using the Transcript Text

The transcript window has the editing features Copy, Find, Find Next, Save, Print, and Clear. If the cursor is in the transcript window, you can use keyboard shortcuts for these editing features. Otherwise, open the transcript window pop-up menu by clicking anywhere in the transcript window with the right mouse button.

You can look at any part of the entire transcript by using the horizontal and vertical scroll bars. Notice that if you scroll up, you will not be able to see new text being added to the bottom of the transcript.

If the cursor is in the transcript window, you can use the following keypad keys:

- *Up / Down arrow*: Moves the cursor up or down one line.

- *Left / Right arrow*: Moves the cursor left or right one character position.
 - *Page Up / Page Down*: Scrolls the transcript up or down one page. A “page” is the amount of text that can be displayed in the transcript window at one time.
 - *Home / End*: Moves the cursor to the beginning or end of the current line.
 - *Control-Page Up / Control-Page Down*: Moves the cursor to the top or bottom of the current transcript page.
 - *Control-Home*: Moves the cursor to the beginning of the first line in the transcript.
 - *Control-End*: Moves the cursor to the end of the last line in the transcript.
-

Selecting Text in the Transcript

To select part or all of the text in the transcript window, press the left mouse button at the beginning of the desired text, drag to the end of the desired text, and release the mouse button. The selected text is highlighted.

Copying Text From the Transcript

You can copy selected text from the transcript window to the Clipboard. Use the keyboard shortcut Control-c, or choose Copy from the pop-up menu that appears when you press the right mouse button. The Copy command is disabled if no text has been selected.

Finding Text in the Transcript

To find specified text in the transcript window, choose Find from the Transcript pop-up menu. If the cursor is in the transcript window, you can use the keyboard shortcut Control-f. The Find dialog box appears.

Type the search text in the “Find what” text field, and set the options as desired. Click “Find Next” to find the next occurrence of the string in the transcript after the current cursor position. The Direction setting determines the direction of the search from the current cursor position.

Saving or Printing the Transcript

To save selected text from the transcript window or to save the entire transcript, choose Save Selected or Save Transcript from the transcript pop-up menu. If the cursor is in the transcript window, you can use the keyboard shortcut Control-s. To print the transcript, use the keyboard shortcut Control-p.

Clearing the Transcript Window

To clear the transcript window, choose Clear in the transcript pop-up menu. If the cursor is in the transcript window, you can use the keyboard shortcut Control-Delete. This removes all of the existing text from the window.

Online Help

TetraMAX provides online Help in the following forms:

- Text-only help on TetraMAX commands, displayed in the transcript window. Use the `help` command in the command-line window.
 - GUI-based, hyperlinked help on commands, design flows, error messages, design rules, fault classes, and many other topics. Choose from the Help pull-down menu.
 - The *TetraMAX ATPG User Guide* (this manual), available in online form. Use the pull-down menu command Help > User's Guide.
 - The *TetraMAX ATPG Quick Reference*, available in online form. Use the pull-down menu command Help > Quick Reference.
 - The *Test Pattern Validation User Guide*, available in online form. Use the pull-down menu command Help > Test Pattern Validation User's Guide.
 - The TetraMAX release notes, available in online form. Use the pull-down menu command Help > Release Notes.
-

Text-Only Help

To obtain text-only help on a command, use the `help` command in the command-line window.

For a list of available help topics, type the following command in the command-line window:

```
> help
```

To display a list of all commands and a summary of their supported syntax, type the following command in the command-line window:

```
> help -usage
```

To display the syntax of a specific command, type the following command in the command-line window:

```
> help command_name
```

where *command_name* is the command whose syntax you want to view. The following example shows a request to display the syntax of the `report primitives` command, followed by the TetraMAX response:

```
> help report primitives
REPort PRimitives
  < id | instance_name | net_name | pin_pathname |
    -PORTs | -PIS | -POS | -PIOs | -Type type | -Summary
>
  [-Max d]
```

The capital letters in the command syntax indicate the minimum characters necessary to enter the command. For example, the `report primitives -ports` command can be entered as `rep pr -por`.

GUI-Based Help

You can view detailed help on a wide range of TetraMAX topics by using the GUI-based help utility. To open the window, do one of the following:

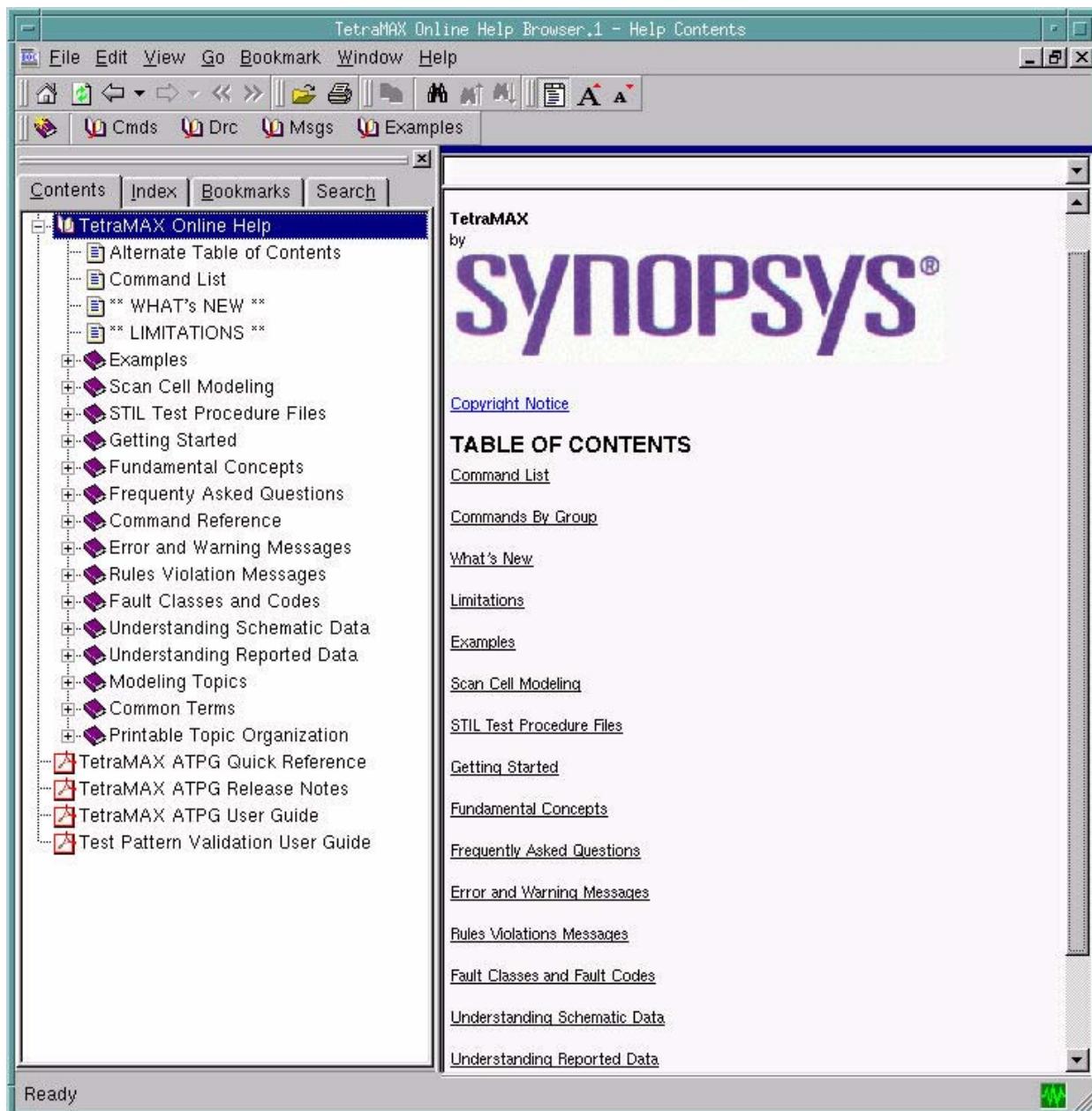
- From the pull-down menu bar, choose Help > Table of Contents. The Help system is displayed in a frameset that consists of three frames (see [Figure 3-2](#)):
 - The menu and button bars at the top
 - The navigation frame (for the table of contents, index, bookmarks, and full-text search) below the button bar on the left
 - The contents frame (where Help topics are displayed) below the button bar on the right

You can use the slide bar between the navigation and contents frames to adjust the relative widths of the two frames.

- In the command-line text field, enter the following:

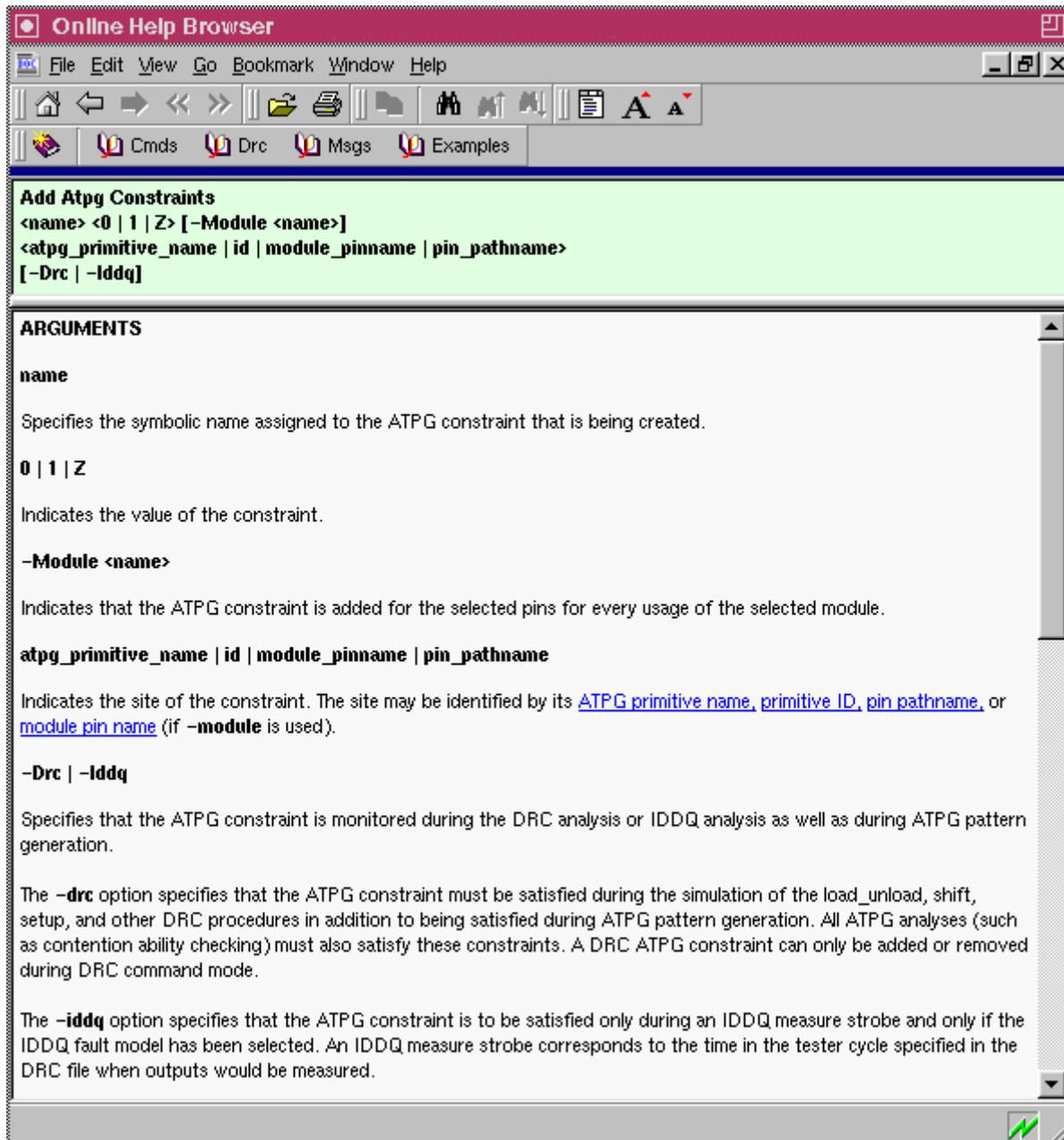
```
> man command
```

Figure 3-2 Online Help Browser Window



Using any of these methods displays a help page like the example shown in Figure 3-3.

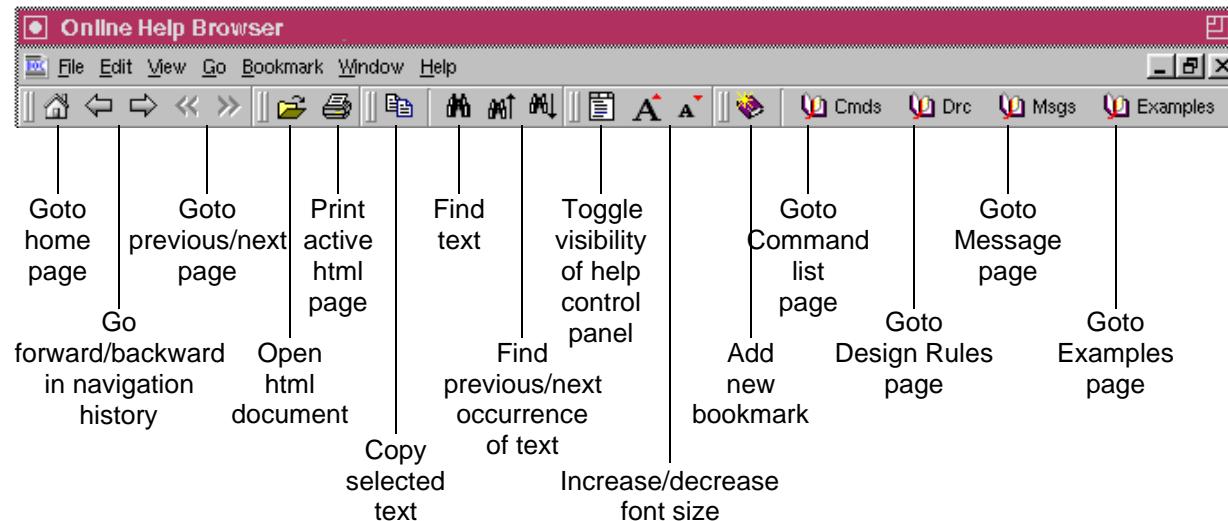
Figure 3-3 Typical Help Window



A help page may contain hyperlinks to related topics. Hyperlinks are underlined and displayed in a distinctive color. To jump to the related topic, click the hyperlink.

The buttons along the top of the help window provide easy access of most frequently used functions (see [Figure 3-4](#)).

Figure 3-4 Help Toolbar

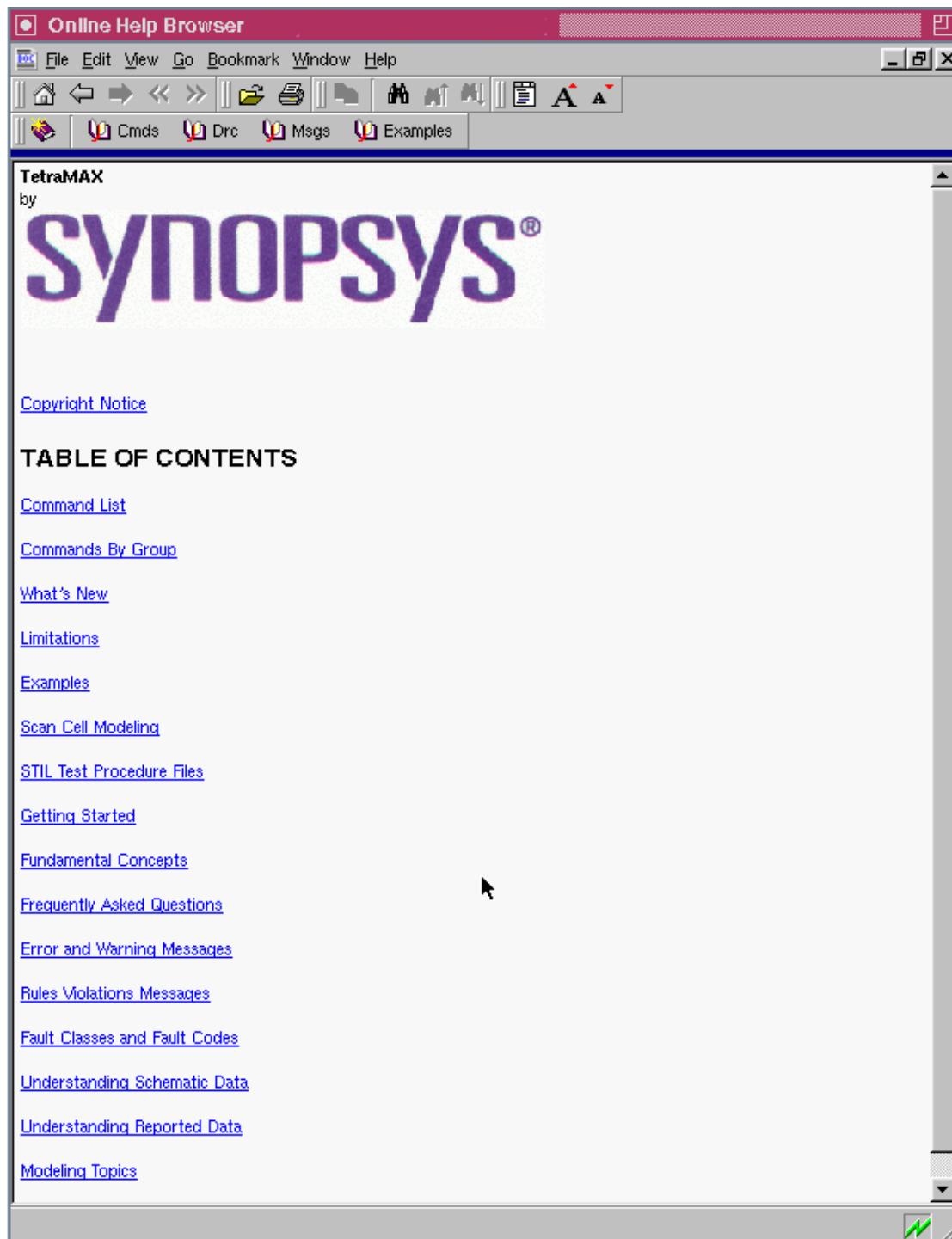


Selecting a Topic From the Major Contents List

To select from a list of TetraMAX topics organized hierarchically, choose the Help > Help Table of Contents command from the pull-down menu. The top-level topics appear, as shown in [Figure 3-5](#).

Selecting a topic of interest takes you to a hyperlinked list of subtopics. Continue the selection process and traverse the hierarchy of topics until you reach the desired help page.

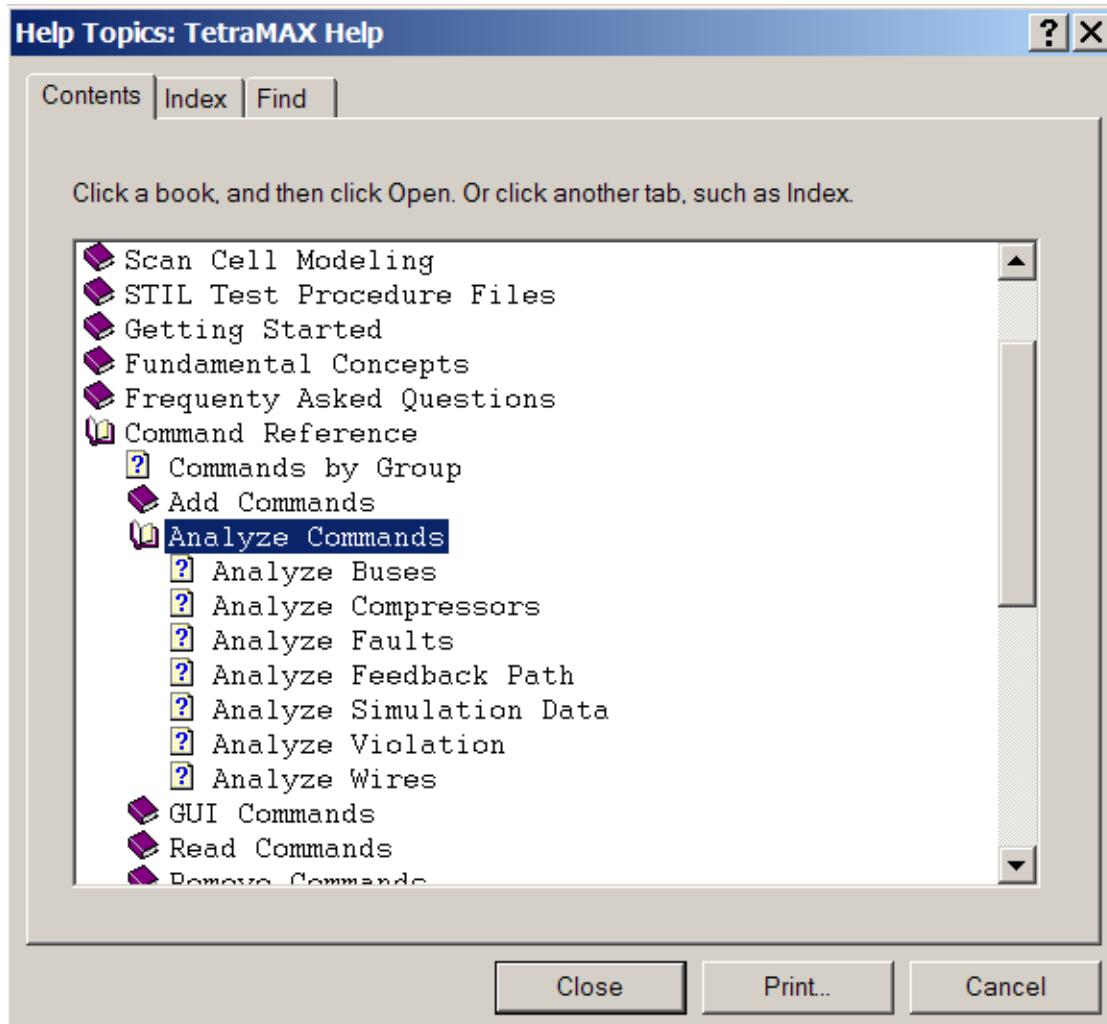
Figure 3-5 Help Window's Top-Level Contents



Selecting a Topic From the Detailed Contents List

To select a help topic from a detailed table of contents, click the Contents tab in the navigation pane. The list of contents appears, as shown in [Figure 3-6](#).

Figure 3-6 Detailed Contents Dialog Box



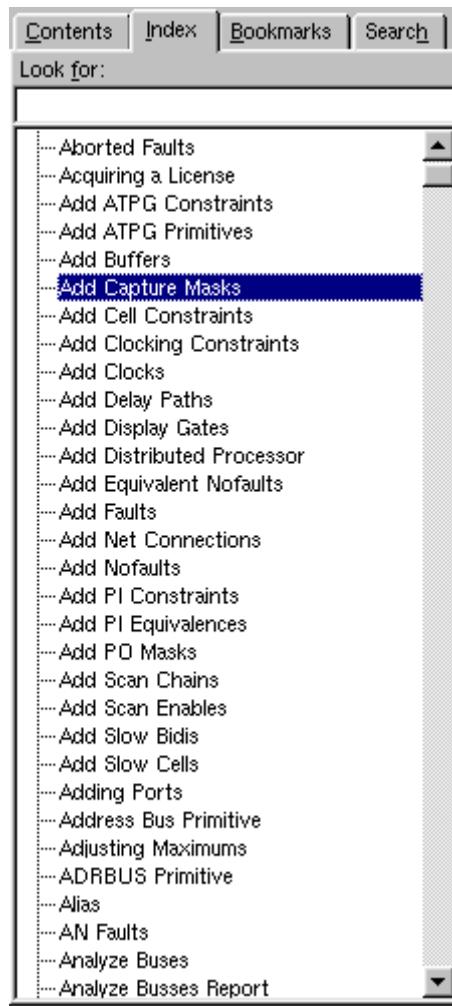
Selecting a closed-book icon opens the book and displays a list of subcontents under the book. Selecting an open-book icon closes the book and collapses its subcontents back.

Select the item of interest. When you select a sheet-of-paper icon, it opens a help page. In some cases, this help page contains a set of hyperlinked subtopics.

Selecting a Topic From the Index

To select a help topic from the index, click the Index tab in the navigation pane. A list of index entries appears, as shown in [Figure 3-7](#).

Figure 3-7 Index Dialog Box



To display the help page for a particular topic, type the topic in the text entry field and press the Return key. In most cases, it is not necessary to type the complete topic; you only need to type until the topic appears highlighted in the list box beneath the text entry field. For example, to get help on the topic Add Clocks, you only need to type **add c** followed by the Return key.

Selecting a Topic by Searching

You can usually find a topic of interest in the index or table of contents, or by following hyperlinks from a related topic. However, if you cannot find a topic using these methods, you can search through the entire database for any text string.

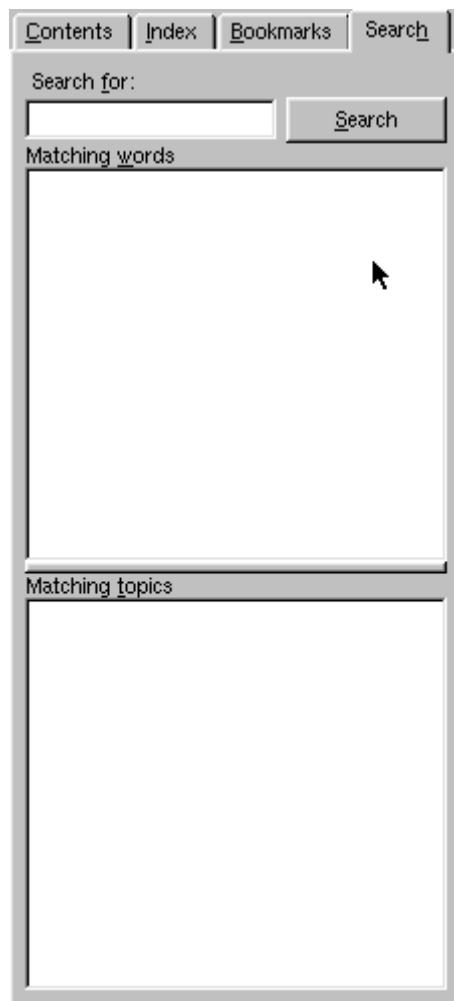
To search the database, click the Search tab in the navigation pane. When the Contents or Index list is displayed.

After the database has been built, clicking the Search tab displays the search dialog box, like the one shown in [Figure 3-8](#).

To perform a search, do the following:

1. In the text entry field at the top, type the word that you want to find.
2. As you type, words similar to your entry are displayed in the list box. Select the word from the list that you want to find.
3. Topics containing the selected word are displayed in the bottom list box. Double-click the topic of interest to display the help page for that topic.

Figure 3-8 Find Dialog Box

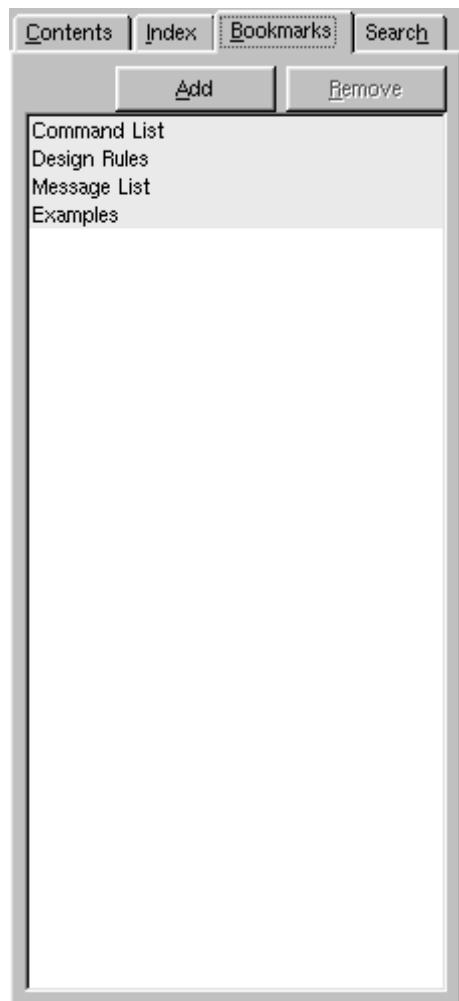


Bookmarking a Topic

You can bookmark a topic of interest for revisiting later. Add and Delete buttons enable you to modify your list of bookmarks.

To access the list of bookmarks, click the Bookmarks tab in the navigation pane. This displays the Bookmarks dialog box, like the one shown in [Figure 3-9](#).

Figure 3-9 Bookmarks Dialog Box



To add a bookmark, do the following:

1. View the topic you want to bookmark.
2. Click Add.

To delete a bookmark from the list, do the following:

1. Select the desired bookmark in the list.
2. Click Remove.

Invoking the Help System in StandAlone

You can invoke the GUI-based help utility independent of TetraMAX with the following command-line entry and options:

```
olh_tmax <config_file> [-html <html_file>] [-pid <page_id>] [-r <snps_root>]  
[-help] [-hidepanel]
```

Where:

Argument	Description
<config_file>	Path name of the configuration file
-html <html_file>	Specifies a topic by its HTML file name
-pid <page_id>	Specifies a topic by its context ID
-r <snps_root>	Specifies the root directory of the TetraMAX software
-help	Display this syntax
-hidepanel	Hides the control panel on startup

4

ATPG Design Flow

This chapter describes the sequence of operations for basic test pattern generation. It contains the following sections:

- Basic ATPG Design Flow
- Netlist Requirements
- Reading the Netlist
- Reading Library Models
- Building the ATPG Model
- Performing Test Design Rule Checking
- Preparing for ATPG
- Running ATPG
- Writing ATPG Patterns
- Writing Fault Lists
- Using Command Files
- Setting Up Advanced ATPG Features

Basic ATPG Design Flow

The basic ATPG flow applies to relatively straightforward designs – designs that you have developed using test design rules and for which you have generated a test protocol file from the Synopsys DFT Compiler or a similar tool. (For more information, see [Chapter 1, “TetraMAX Overview.”](#)) If you encounter problems with your design, see [Chapter 7, “Using the GSV For Review and Analysis,”](#) which provides information on graphical analysis and troubleshooting.

To successfully set up TetraMAX for test pattern generation, you must provide the following information on your design:

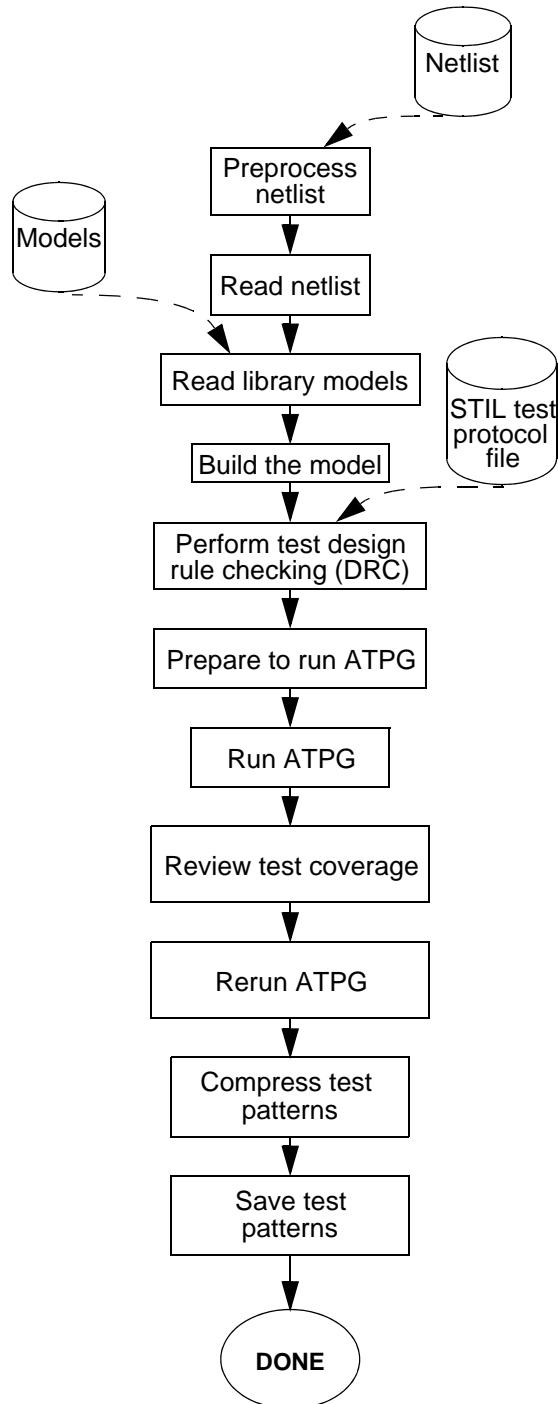
- Clock ports
- Asynchronous set and reset ports
- Scan chain input and output ports
- Any ports that place the design in test mode and their active states
- Any ports that enable shifting of scan chains and their active states
- Any ports that globally control bidirectional drive and their active states

If your design requires manual generation of the test protocol file, see [Chapter 9, “STIL Procedure Files.”](#) [Figure 4-1](#) illustrates the basic ATPG design flow. This flow consists of the following steps:

1. If necessary, preprocess your netlist to meet the requirements of TetraMAX.
2. Read in the netlist.
3. Read in the library models.
4. Build the ATPG design model.
5. Read in the STIL test protocol file, generated by DFT Compiler or another source. Perform test DRC and make any necessary corrections.
6. To prepare the design for ATPG, set up the fault list, analyze buses for contention, and set the ATPG options.
7. Run automatic test pattern generation.
8. Review the test coverage and rerun ATPG if necessary.
9. Rerun ATPG.
10. Save the test patterns and fault list.

The individual steps in this flow are described in detail in the following sections of this chapter.

Figure 4-1 TetraMAX Design Flow



Netlist Requirements

TetraMAX can read designs in EDIF, Verilog, and VHDL formats. In some cases, some minimal preprocessing is required to make the netlist compatible with TetraMAX.

EDIF Netlist Requirements

Depending on how logic connections to power and ground are represented in your EDIF netlist, you must take different actions to make this aspect of the netlist compatible with TetraMAX.

Logic 1/0 Using Global Nets

In EDIF, a design can reference two or more global nets, which represent the tie to logic 1 or logic 0 connections. Because there is no driver for these nets, TetraMAX issues “floating internal net” warnings as it analyzes the design.

If your design uses this global net approach and you are using Synopsys tools to create your netlist, set the EDIF environment variables as shown in [Example 4-1](#) before writing the EDIF netlist. The global net names used in the example are logic0 and logic1, but you can use any legal net names.

Example 4-1 EDIF Variable Settings for Global Logic 1/0 Nets

```
edifout_netlist_only = true
edifout_power_and_ground_representation = net
edifout_ground_net_name = "logic0"
edifout_power_net_name = "logic1"
write options
```

Logic 1/0 by Special Library Cell

The EDIF library can contain special tie_to_low and tie_to_high cells. Every logic connection to power or ground is then connected by a net to one of these cells. If your design uses this library cell approach, you must define an ATPG model for each cell to supply the proper function of TIE1 or TIE0; otherwise, the missing model definition is translated to a TIEX primitive, and the logic 1/0 connections are all tied to X instead of to the desired logic value.

If you do not yet have models describing the logic functions of the special cells, you might have to add some module definitions to your library. Normally, your ASIC vendor provides these; if not, see [Example 4-2](#), which shows a Verilog module description for modules called POWER and GROUND. You can use this module description by changing the name of the module to match the library cell names referenced by your EDIF design netlist.

Example 4-2 ATPG Model Definition for Logic 1/0 Library Cells

```
module POWER (pin);
output pin;
_TIE1(pin);
endmodule

module GROUND (pin);
output pin;
_TIE0(pin);
endmodule
```

To provide an ATPG functional model for each EDIF cell description, place the module definition in a separate file to be referenced during the process flow when it is time to read in library definitions.

Verilog Netlist Requirements

Verilog netlist style, syntax, and instance and net naming conventions vary greatly. The following are some guidelines:

- Do not use a period (.) within the name of any net, instance, pin, port, or module without enclosing it with the standard Verilog backslash mechanism.
 - Verify that your Verilog modules are structural and not behavioral, except for modules used to define ATPG RAM/ROM functions.
 - Remember that Verilog is case-sensitive, although many tools ignore case and treat “MyNet” and “mynet” as the same item.
 - If you are using Synopsys tools to create your Verilog netlist, review the `define_name_rules` command to find options for adjusting the naming conventions used in your design.
-

VHDL Netlist Requirements

Designs provided in VHDL format must be completely structural in nature. Bits and vectors must only use `std_logic` types. Other types such as `SIGNED` are not supported. Conversion functions are not supported.

Reading the Netlist

You can read a netlist using the Read Netlist/Image dialog box, or you can enter the `read netlist` command from the command line.

Note:

Before reading in the netlist or the library models, check to see if any of the modules in your netlist have the same names as the library models. In the case of duplicate module definitions, the last one encountered is the one used; therefore, if you read in your netlist and then read in library models, modules in your netlist are overwritten by any library models having the same names.

Using the Read Netlist/Image Dialog Box

To read a netlist by using the Read Netlist/Image dialog box,

1. Click the NetList button in the command toolbar at the top of the TetraMAX main window. The Read Netlist/Image dialog box appears.
2. In the Netlist File Name(s) window, type the path to your netlist file, or use the Browse button to navigate and select the file.
3. If you require settings other than the defaults, choose them now.

The default settings are the ones used most often. For more details on the Read Netlist/ Image dialog box, see online Help for the `set netlist` and `read netlist` commands.

4. Click Run.
-

Using the `read netlist` Command

You can also read a netlist using the `read netlist` command. For example:

```
BUILD> read netlist filename
```

For the complete syntax and option descriptions, see online Help for the `read netlist` command.

Read Netlist Usage Notes

Netlists can be in standard ASCII format or GZIP format. GZIP is a commonly available compression utility on UNIX and PC platforms. TetraMAX automatically detects compressed files and decompresses them during the read operation.

TetraMAX automatically determines what type of netlist is referenced and reads the file to collect design and module descriptions. By default, Verilog netlists are treated as case-sensitive; EDIF and VHDL netlists are treated as case-insensitive. If you want to override the default, add `-sensitive` or `-insensitive` at the end of the command.

Your design can be in one file or in multiple files. Use as many `read netlist` commands as necessary to read in all portions of the design. You can read multiple files from the same directory using wildcards (for example, `*.v`).

By default, if a module is defined more than once, TetraMAX issues a warning and uses the module definition found in the last netlist read. However, when you read in a module with the `-master_modules` option, that module is marked as a Master Module and will not be replaced if another module with the same name is encountered (unless the later module is also read in with the `-master_modules` option).

If you need to start over or you want to clear the in-memory design and switch to a new design, use one of the `-delete` option alternatives shown in the following example. The first alternative deletes all netlist designs from memory; the second deletes all designs and then reads a specified netlist. The `-delete` option is the same as the Clear Previous Netlist option in the Read Netlist/Image dialog box.

```
BUILD> read netlist -delete  
BUILD> read netlist filename -delete
```

Reading Library Models

You must read in all Verilog library models referenced by your design; you can read in one model at a time, or you can read in the entire library with one command.

Note:

If your design contains a module that has the same name as one of the library models, when you read in the library, the library model overwrites your module.

To read in one model at a time, you use the Read Netlist/Image dialog box or the `read netlist` command in the same way as you read in your netlist. You can use wildcards to read multiple definition files, as in the following example:

```
/proj1234/shared_verilog/*.v
```

You can also use wildcards to read in multiple definition files with one `read netlist` command, as in the following example:

```
BUILD> read netlist/proj1234/shared_verilog/*.v -noabort
```

Building the ATPG Model

Building the ATPG design model takes those parts of the design that are to be part of the ATPG process, removes the hierarchy, and puts them into an in-memory image that TetraMAX can use.

You can build the ATPG model for your design using the Run Build Model dialog box, or you can use the `run build_model` command from the command line.

Using the Run Build Model Dialog Box

To use the Run Build Model dialog box to build your design or a module within your design,

1. Click the Build button in the command toolbar at the top of the TetraMAX main window. The Run Build Model dialog box appears.
2. In the Top Module Name field, type the name of the design or module you want to build.
If you don't provide a name, TetraMAX searches for a module not referenced by any other module by default. If there is more than one possibility, or if you want to work on a lower-level module, specify the name of the desired top-level module to build.
3. To set the less commonly used build options, click the Set Build Options button and fill in the dialog box that appears.
For information about the items in the Set Learning portion of the Run Build Model dialog box, see the TetraMAX online documentation topic "Set Learning."
For more details on the Read Netlist/Image dialog box, see online Help for the `run build_model` command.
4. Click Run.

Using the `run build_model` Command

To use the command line to build the ATPG model for the entire design or for a module within the design, run this command:

```
BUILD> run build_model top_module_name
```

The optional argument `top_module_name` is the name of the design or module for which you want to build the ATPG model. As with the Run Build Model dialog box, specify the name unless you want TetraMAX to search for a module that is not referenced by another module.

For the complete syntax and option descriptions, see online Help for the `run build_model` command.

[Example 4-3](#) shows a typical `run build_model` transcript.

Example 4-3 run build_model Transcript

```
BUILD> run build_model my_asic
-----
Begin build model for topcut = my_asic ...

-----
End build model: #primitives=101004, CPU_time=13.90 sec,
Memory=34702381

-----
Begin learning analyses...
End learning analyses, total learning CPU time=33.02
-----
```

Performing Test Design Rule Checking

Test DRC involves analysis of many aspects of the design. Among other things, DRC checks the following:

- Whether the scan chains inputs and outputs are logically connected
- Whether all the clocks and asynchronous set/reset pins connected to scan chain flip-flops are controlled only by primary input ports
- Whether the clocks/sets/resets are off when you switch from normal mode to scan shift mode and again when you switch back to normal mode
- Whether any internal multiple-driver nets can be in contention

So that TetraMAX can perform these and other DRC checks, you must provide information about clock ports, scan chains, and other controls by means of a STIL test protocol file. The STIL file can be generated from DFT Compiler, or you can create one manually as described in [Chapter 9, “STIL Procedure Files.”](#)

After you read in the test protocol, the next step is to perform test design rule checking.

Starting Test DRC

You can perform DRC using the Run DRC dialog box, or you can execute the `run drc` command from the command line.

Using the Run DRC Dialog Box

To use the Run DRC dialog box to perform DRC,

1. Click the DRC button in the command toolbar at the top of the TetraMAX main window.
The DRC dialog box appears. Click the Run tab if it is not already active.
2. In the Test protocol file name field, type the path name of the STIL procedure file previously created, or use the Browse button to navigate and select the file.
3. For the basic design flow, accept the default settings.
For details about these and other settings, see online Help for the `run drc` command.
4. Click Run.

Using the `run drc` Command. To perform DRC from the command line, use the `run drc` command. For example:

```
BUILD> run drc filename
```

The *filename* argument is the name of the STIL procedure file previously created. After starting the DRC checks, TetraMAX produces a status report and lists the DRC violations, as shown in [Example 4-4](#). For a list of all test DRC rules, see online Help. For the complete syntax and option descriptions, see online Help for the `run drc` command.

Example 4-4 Typical DRC Run

```
BUILD> run drc my_stil_file.spf
-----
Begin scan design rules checking...
-----
Begin reading test protocol file lander.spf...
End parsing STIL file lander.spf with 0 errors.
Test protocol file reading completed, CPU time=0.10 sec.
-----
Begin Bus/Wire contention ability checking...
Bus summary: #bus_gates=40, #bidi=40, #weak=0, #pull=0,
#keepers=0
    Contention status: #pass=0, #bidi=40, #fail=0, #abort=0,
#not_analyzed=0
    Z-state status   : #pass=0, #bidi=40, #fail=0, #abort=0,
#not_analyzed=0
Bus/Wire contention ability checking completed, CPU time=0.04
sec.
-----
Begin simulating test protocol procedures...
Nonscan cell constant value results: #constant0 = 4, #constant1 = 7
Nonscan cell load value results   : #load0 = 4, #load1 = 7
Warning: Rule Z4 (bus contention in test procedure) failed 48
times.
Test protocol simulation completed, CPU time=0.14 sec.
```

```

Begin scan chain operation checking...
Chain c1 successfully traced with 31 scan_cells.
Chain c2 successfully traced with 31 scan_cells.
Scan chain operation checking completed, CPU time=0.34 sec.
-----
Begin clock rules checking...
Warning: Rule C17 (clock connected to PO) failed 16 times.
Warning: Rule C19 (clock connected to non-contention-free BUS)
failed 1 times.
Clock rules checking completed, CPU time=0.14 sec.
-----
Begin nonscan rules checking...
Nonscan cell summary: #DFF=201 #DLAT=0 tla_usage_type=none
Nonscan behavior: #C0=4 #C1=7 #LE=11 #TE=179
Nonscan rules checking completed, CPU time=0.05 sec.
-----
Begin contention prevention rules checking...
26 scan cells are connected to bidirectional BUS gates.
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)
failed 24 times.
Contention prevention checking completed, CPU time=0.02 sec.
-----
Begin DRC dependent learning...
DRC dependent learning completed, CPU time=0.97 sec.
-----
DRC Summary Report
-----
Warning: Rule C17 (clock connected to PO) failed 16 times.
Warning: Rule Z4 (bus contention in test procedure) failed 48
times.
Warning: Rule Z9 (bidi bus driver enable affected by scan cell)
failed 24 times.
There were 72 violations that occurred during DRC process.
Design rules checking was successful, total CPU time=2.01 sec.
-----
```

Reviewing the DRC Results

The DRC summary report at the end of the DRC run provides a starting point for reviewing the DRC results. The DRC Summary Report in [Example 4-5](#) shows one class of clock rule warnings (C17) and two classes of bus rule warnings (Z4 and Z9).

Note:

Correct all DRC violations that are errors. Ignoring or overlooking these might result in ATPG patterns that fail in simulation or on the real device.

For more information about specific rule violations, use the `man` command, as follows:

```
DRC> man z4
```

For a summary of failing rule messages, enter the following:

```
DRC> report rules -fail
```

An example of the `report rules -fail` output is shown in [Example 4-5](#).

Example 4-5 Reporting Rules That Fail

```
TEST> report rules -fail
// C16: #fails=190 severity=warning
// C17: #fails=16 severity=warning
// C19: #fails=1 severity=warning
// Z4: #fails=128 severity=warning
// Z9: #fails=24 severity=warning
```

For more detailed information about specific DRC violations in the design, use the `report violations` command. You can identify a single violation, all violations of a single type, all violations within a class, or all violations, as in the following examples:

```
DRC> report violations c17-2
DRC> report violations c17
DRC> report violations c
DRC> report violations -all
```

Viewing Violations in the GSV

You can visually inspect many of the rule violations using the graphical schematic viewer (GSV). The GSV displays a subset of the design showing the logic gates involved in the DRC violation, along with appropriate diagnostic data such as logic values, constrained ports, or clock cones.

To follow up one of the warning messages and view a graphic representation of the problem,

1. Click the Analyze button in the command toolbar at the top of the TetraMAX main window.

The Analyze dialog box appears.

Click the Rules tab if not already active.

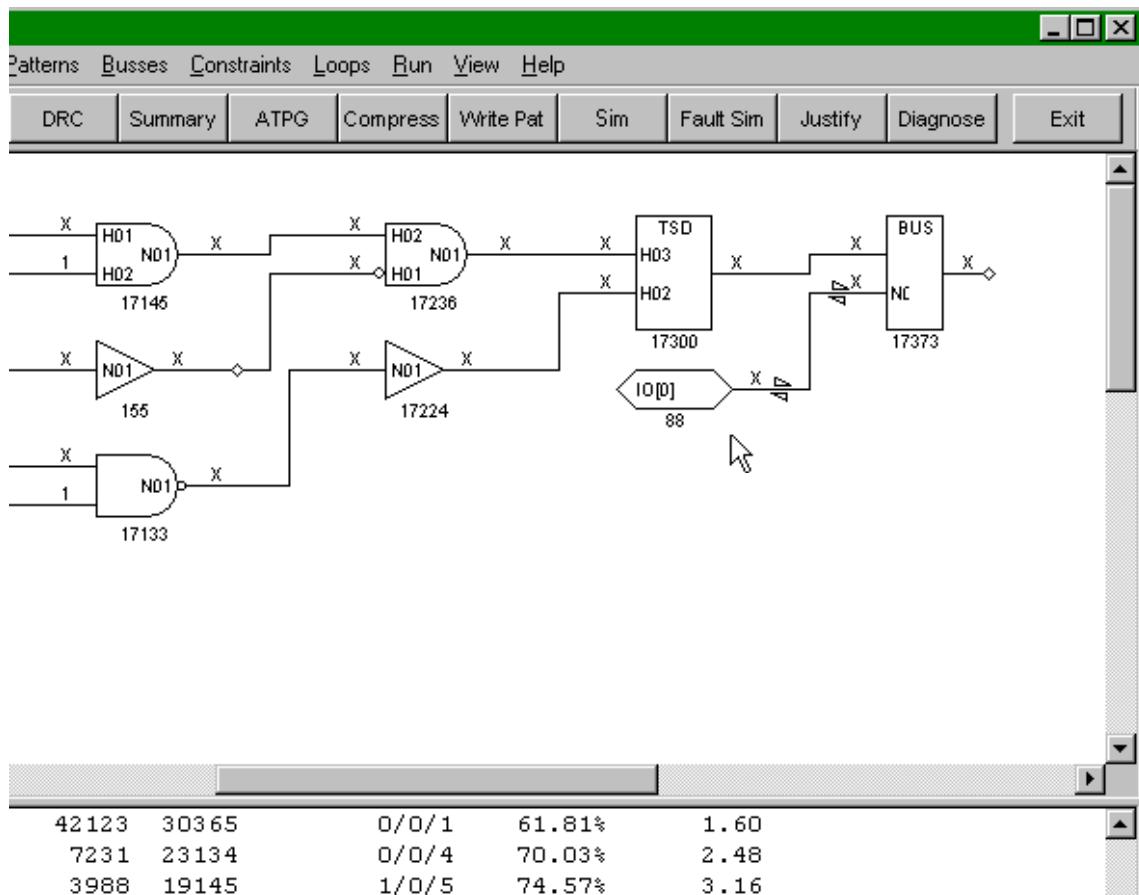
The window lists all the most recent violations. All violations are numbered. For example, Z4-1:12 means there are 12 violations of rule Z4, designated Z4-1 through Z4-12.

2. Select a violation from the list or type a specific violation occurrence number in the Rule Violation field.

3. Click OK.

The GSV opens and displays the violation. At the same time, the transcript window displays the error message.

Figure 4-2 Schematic Display of DRC Violation



The Z4 violation message that would accompany the GSV window is shown in [Example 4-6](#).

Example 4-6 Z4 Violation Message Accompanying GSV Window

Warning: Bus contention on /bixcr (17373) occurred at time 0 of test_setup procedure. (Z4-1)

A simple and fast way to view the schematic for a violation message is to point to the red-highlighted error message in the transcript window, click the right mouse button, and select Analyze in the pop-up menu.

[Figure 4-2](#) shows the gates involved in the Z4 violation, along with the logic values resulting from simulation of the `test_setup` macro. The `test_setup` macro is described in more detail in the section “[Defining the test_setup Macro](#)” on page 9-9.

In this example, most of the logic values are X (unknown). The violation might be caused by failing to force a Z on a bidirectional port called IO[0] in the `test_setup` procedure. You can choose to ignore or correct this violation. If you choose to ignore it, fault coverage will be lowered because the ATPG algorithm will not generate any pattern that would cause contention.

Some messages can be safely ignored; some can be resolved through adjustment of a procedure definition; and others require a change to the design. For more information on using the GSV, see [Chapter 7, “Using the GSV For Review and Analysis.”](#)

Test Design Rule Categories

The test design rules are organized by category. Each rule has an identification code (rule ID) consisting of a single character followed by a number. The first character defines the major category of the rule.

The rules are organized functionally into 10 major categories:

- B (Build rules)
- C (Clock rules)
- L (LBIST rules)
- N (Netlist rules)
- P (Path Delay rules)
- S (Scan Chain rules)
- T (Tester rules)
- V (Vector rules)
- X (X-state rules)
- Z (Tristate rules)

An example of a specific rule is Z4, “bus contention in test procedure.”

When a rule is violated, each violation incident is assigned a violation ID, which is the rule ID followed by a dash and then a sequence number. Thus, the rule violation ID for the 24th violation of a Z4 rule is Z4-24.

Some violation IDs show an abort indicator suffix, which appears as Z7-12.A or Z6-3 (Abort). This means that the ATPG analysis of the violation was aborted. In such cases, you might want to increase the ATPG abort limit.

All of the test design rules are described in detail in online Help. To see the online Help window, click Contents and select the Error and Warning Messages icon.

Changing Test Design Rule Severity

Each design rule is assigned one of four severity levels:

- Ignore: The rule is not checked and no messages are issued.
- Warning: Violation of the rule produces a warning message, and the current process continues.
- Error: Violation of the rule produces an error message, and the current processing step is halted.
- Fatal: The same as an Error, except that the severity level cannot be changed.

You can change the rule severity level by using the Set Rules dialog box, or you can execute the `set rules` command from the command line.

Additionally, you can determine a rule's severity level setting and the number of violations that have occurred by using the `report rules` command or by choosing Rules > Report Rules.

Using the Set Rules Dialog Box. To change the rule severity level by using the Set Rules dialog box,

1. From the menu bar, choose Rules > Set Rule Options. The Set Rules dialog box appears.
2. Enter a rule ID and select a severity level.

For additional information about the available options, see online Help for the `set rules` command.

3. Click OK.

Using the set rules Command. Alternatively, you can also change the rule severity level of any rule (except those that are Fatal) by using the `set rules` command. For example:

```
BUILD> set rules B5 warning
```

When running DRC (prior to ATPG) on circuits which include blocks that have both default and high X-tolerant architectures, specify the following command:

```
set rules R22 -warn
```

This will downgrade a check done for fully X-tolerant designs built by DFT MAX which were actually built with blocks that include both default X-tolerant architectures and high X-tolerant architecture

For the complete syntax and option descriptions, see online Help for the `set rules` command.

Effects of Rule Violations

When a design violates a design rule, the effects vary depending on the rule's severity level. For example, rule N5, "redefined module," has a default Warning severity level and in most cases notifies you that a module was defined and then redefined, and that the last definition encountered is being used. In contrast, the rule S1, "scan chain blockage," has a default Fatal severity level. The scan chain is not usable in its current state, and you must correct the problem before trying further pattern generation.

The default severity level for each rule reflects a conservative approach to ATPG efforts. When an error or warning is produced, review the potential problem and determine whether you need to change the design or the ATPG procedures. You might be able to adjust the severity level downward and continue ATPG.

The online Help for each rule suggests an action you can take to analyze the cause of the rule violation and determine whether the violation might be fixed by changing a procedure or setup, or whether the design might have to be changed.

A Detailed View of the DRC Process

In performing design rule checking, TetraMAX does the following:

1. Reads the STIL procedure file (SPF) to gather information and to check for syntax and consistency errors.
2. Performs contention ability checks on buses and wired logic. This step identifies drivers that could potentially be placed in a conflicting state and cause internal device contention (Z rules).
3. Simulates the test procedures in the SPF to determine whether certain conditions have been met involving the state of clocks and the sequencing of procedural events.
4. Simulates each scan chain under the direction of the defined test procedures, to guarantee that the scan path is operational and complies with all scan chain rules (S rules).
5. Analyzes all clocks and clocked devices against the ATPG rules for clock usage (C rules).

6. Analyzes all nonscan devices, including latches, RAMs, ROMs, and bus keepers (S rules). Nonscan devices that hold state are identified and used for ATPG purposes. Latches that can be made transparent are identified, and latches that cannot be made transparent are replaced with TIEX logic.
 7. Analyzes the multidriver nets identified in step 2 as potentially causing conflict to determine which drivers actually cause conflict.
 8. Performs some additional circuit learning that depends on the results of the previous steps. After identifying scan, nonscan, transparent and nontransparent devices, and sequential devices at a constant state, TetraMAX propagates the effects of PI constraints, ATPG constraints, and TIEX effects throughout the design.
 9. Produces a summary report listing the types and totals of DRC violations encountered.
-

Scan Chain Tracing

In performing scan chain tracing, TetraMAX does the following:

1. Initializes constrained ports to their constrained states.
2. Simulates the events in the `test_setup` macro.
3. Simulates the events in the `load_unload` procedure.
4. Simulates the events in the `Shift` procedure, and monitors the elements in the scan chain to ensure that the scan data path is valid, the scan cells are clocked, and any asynchronous set/clear pins are stable in their off positions.

To see a verbose report on the scan chain tracing, execute the following command:

```
BUILD> set drc -trace
```

The default is to not show the verbose tracing of scan chains.

Shadow Register Analysis

A shadow register is not in the scan chain, but is loaded when its master register in the scan chain is loaded, by the same clock or by a separate clock. A shadow register is considered a control point but not an observe point. During the DRC analysis, TetraMAX searches for nonscan cells that can be considered shadow registers.

If the shadow register's state can be observed at the shadow's master, TetraMAX classifies the register as an observable shadow. This usually requires defining a `shadow_observe` procedure in the SPF.

The default is to search for shadow registers. You can disable the default by executing the following command:

```
BUILD> set drc -noshadow
```

Procedure Simulation

In addition to the `test_setup`, `load_unload`, and `Shift` procedures, there are other procedures in the SPF or implied by the definition of clock ports. TetraMAX simulates all of these procedures as part of the design rule checking process to guarantee that they accomplish their intended purposes.

Transparent Latches

A transparent latch is a latch in which the enable line can be asserted so that data passes through it without activating any of the design's defined clocks. During the rule checking process, TetraMAX automatically determines the location of all latches in the design and checks to see whether the latches can be made transparent. For ATPG, you must be able to disconnect the latch control from any clock ports.

When latches are transparent, it is easier for TetraMAX to detect faults around those latches. When latches are not transparent, you might need to use a Full-Sequential ATPG run to get good fault coverage around those latches.

Cells With Asynchronous Set/Reset Inputs

You can use the `set drc` command to specify the treatment of latches and flip-flops whose set and reset lines are not off when all clocks are at their off state. By default, these latches and flip-flops are treated as unstable cells, which prevents them from being used during test pattern generation.

To have these latches and flip-flops treated as stable cells, use the `set drc -allow_unstable_set_resets` command. Then the ATPG algorithm can use the cells with unstable set/reset inputs to improve test coverage. In that case, it is not necessary to define the set/reset inputs as clocks.

In certain cases, the `remove false clocks` option of the `set drc` command automatically invokes the "allow unstable set/reset" behavior. When a primary input port has been defined as a clock and a DRC analysis determines that the port cannot capture data into a sequential device, the input port is determined to be a "false clock." In the default DRC

configuration, the result is a C4 violation. However, using the `set drc -remove_false_clocks` command causes automatic removal of the clock declaration for each false clock, instead of a C4 violation.

If a primary input port declared to be a clock is connected to the set/reset inputs of sequential gates, and also to the D inputs of other sequential gates, it is considered a false clock. As a result, the algorithm removes the clock declaration for that port and then enables unstable set/reset cells, just like executing the `set drc -allow_unstable_set_resets` command.

The `-allow_unstable_set_resets` option can be useful if a scan-enable signal is used to disable the set/reset inputs of scan cells during load. Using this option means that the scan-enable signal does not have to be defined as a clock, which can greatly improve test coverage.

Sensitizable Feedback Paths

During initial processing, TetraMAX identifies feedback paths within the design and assigns each path a unique feedback path ID.

During DRC, TetraMAX analyzes the feedback paths to ensure that the loop of logic gates can be broken at some combinational gate within the loop. If the logic loop does not have a blocking point, simulations performed during ATPG will oscillate without resolving to a final value. If DRC analysis cannot find a set of inputs and scan chain load values that can break the loop and still maintain any other constraints in effect, TetraMAX issues an X1 rule violation.

Save/Restore in TEST Mode

You can use the save/restore feature to reduce the time needed to read the netlists, build the design, and run the design rule checker (DRC) for subsequent ATPG runs. This feature is implemented through the `write image` and `read image` commands.

After a successful DRC run, use the `write image` command while in TEST mode to save the in-memory TetraMAX database (gates) to a file. You can optionally save the DRC violations for the C, D, L, S, X, and Z rules with the `-violations` option. When you later decide to do more runs, issue a `read image` command to read the database file and proceed.

Preparing for ATPG

To prepare for ATPG, you initialize the fault list, select the pattern source, choose settings for bus contention checking, and specify the pattern generation effort.

Setting Up the Fault List

Before generating test patterns, you must initialize the fault list. TetraMAX attempts to generate a test pattern to test all faults contained in the fault list.

To generate a new fault list that includes all possible fault sites in the ATPG design model, use the `add faults` command,

```
TEST> add faults -all
```

To use an existing fault list from another tool or from a previous run, use the `read faults` command,

```
TEST> read faults fault_list_filename
```

You can exclude specific blocks, instances, gates, or pins from the fault list by using any of these methods:

- Specify objects to be excluded by using the `addnofault` command before executing the `add faults -all` command. For example:

```
TEST> addnofault /sub_block_A/adder
TEST> addnofault /io/demux/alu
TEST> add faults -all
```

- Remove faults based on fault locations in a fault list file after executing the `add faults -all` command. For example:

```
TEST> add faults -all
TEST> read faults fault_list_file -delete
```

- Remove faults using the `remove faults` command after executing the `add faults -all` command. For example:

```
TEST> add faults -all
TEST> remove faults /sub_block_A/adder
TEST> remove faults /io/demux/alu
```

Or, if you have a small number of faults, you can add them explicitly using `add faults`:

```
TEST> remove faults -all
TEST> add faults /proc/io
TEST> add faults /demux
TEST> add faults /reg_bank/bank2/reg5/Q
```

Selecting the Pattern Source

TetraMAX can read and write pattern files in a variety of formats, including Verilog, VHDL, STIL, and WGL.

By default, TetraMAX generates new internal patterns. You can configure TetraMAX to use external patterns (for example, patterns from a previous session) by using the Set Patterns dialog box, or you can enter the `set patterns` command at the command line.

Note:

Do not assume that TetraMAX can correctly read Verilog, STIL, WGL, or VHDL patterns created by tools other than TetraMAX.

Using the Set Patterns Dialog Box

To use the Set Patterns dialog box to configure TetraMAX to use external patterns,

1. From the menu bar, choose Patterns > Set Pattern Options. The Set Patterns dialog box appears.
2. Enter the name of the file in the Pattern File Name field, or use the Browse button to navigate and select the file. The file formats accepted are Binary, Extended VCD, limited-syntax STIL, Verilog, VHDL, and WGL.
3. If you require settings, enter them now.

For the complete syntax and option descriptions, see online Help for the `set patterns` command.

4. Click OK.

Another way to read in external patterns is to use the Run ATPG dialog box. Click the ATPG button in the command toolbar to open the dialog box, select `External` as the Pattern Source, and enter the file name in the Pattern File Name field.

Using the `set patterns` Command

You can also use the command line to configure TetraMAX to use an external pattern file. For example:

```
TEST> set patterns external b010.vil
```

The command options are generally the same as for the Set Patterns dialog box. You can specify any one of three types of pattern sources: Internal, Random, or External. Specifying Internal uses the internally generated patterns. Specifying Random generates a random pattern, as defined by the `set random_patterns` command or the Set Random Patterns dialog box (Patterns > Set Random Patterns). Specifying External uses an external pattern, as described previously.

For some pattern file examples in STIL, Verilog, and WGL format, see “[Pattern Input](#)” on [page 12-13](#).

For further information, see online Help for the `set patterns` command.

Choosing Settings for Contention Checking

When TetraMAX checks bus contention, it discards patterns that can potentially cause contention and generates additional patterns to avoid contention. You can select optional bus contention checks using the Set Contention dialog box, or you can enter the `set contention` command from the command line.

Using the Set Contention Dialog Box

To use the Set Contention dialog box to set contention options,

1. From the menu bar, choose Buses > Set Contention Options. The Set Contention dialog box appears.
2. If you require settings other than the defaults, choose them now.

The default settings are the ones used most often. For details about these and other settings, see online Help for the `set contention` command.

3. Click OK.

Using the `set contention` Command

You can also select bus contention options using the `set contention` command. For example, the following command is conservative without being overly restrictive to TetraMAX:

```
DRC> set contention bidi bus ram -capture -atpg -multiple_on
```

For the complete syntax and option descriptions, see online Help for the `set contention` command.

Specifying ATPG Settings

You can specify the pattern generation effort and other ATPG settings using the Run ATPG dialog box, or you can use the `set atpg` command at the command line.

Using the Run ATPG Dialog Box

To use the Run ATPG dialog box to specify the pattern generation effort,

1. Click the ATPG button in the command toolbar at the top of the TetraMAX main window.
The Run ATPG dialog box appears.
2. If you require settings other than the defaults, choose them now.

The dialog box contains four tabs: General ATPG Settings, Basic-Scan Settings, Fast-Sequential Settings, and Full-Sequential Settings. You select a set of options by clicking its labeled tab.

The default settings are the ones used most often. For details about these and other settings, see online Help for the `set atpg` and `run atpg` commands.

3. Click Set.

Using the `set atpg` Command

You can also specify the pattern generation using the `set atpg` command. For example:

```
DRC> set atpg -patterns 400 -abort_limit 5
```

For the complete syntax and option descriptions, see online Help for the `set atpg` command.

Running ATPG

You can run ATPG using the Run ATPG dialog box, or you can enter the `run atpg` command in the command line.

To access the Run ATPG dialog box, see “[Specifying ATPG Settings](#)” on page 4-23. To start the ATPG process, click the Run button in the Run ATPG dialog box.

To run ATPG from the command line, enter the `run atpg` command. For example:

```
TEST> run atpg -random
```

By default, TetraMAX performs Basic-Scan ATPG first, followed by Fast-Sequential ATPG (if enabled), and Full-Sequential ATPG last (if enabled).

To obtain a good balance between pattern compaction and execution speed, you can adjust the abort limit and merge effort settings of the `set atpg` command.

Quickly Estimating Test Coverage

To quickly estimate the final test coverage, you can use a low abort limit and low merge effort.

Using the Run ATPG Dialog Box

To use the Run ATPG dialog box to get the estimate:

1. Click the ATPG button in the command toolbar at the top of the TetraMAX main window. The Run ATPG dialog box appears.
2. Set the Abort Limit to 5.
3. Set the Merge Effort to Off.
4. Click Set.
5. For details about these and other settings, see TetraMAX On-line Help for the `set atpg` and `run atpg` commands.
6. Click Run.

Using the `set atpg` Command

You can also get the estimate by using the `set atpg` command:

```
TEST> set atpg -abort_limit 5 -merge off
TEST> run atpg
```

For the complete syntax and option descriptions, see online Help for the `set atpg` command.

Examples. [Example 4-7](#) shows a transcript produced by these commands. The reported test coverage is usually within 1 percent of the final answer, and the number of patterns with merge effort turned off is usually two to three times the number of patterns produced by a final pattern generation run with the merge effort set to high.

Example 4-7 Run ATPG Transcript, Merge Effort Turned Off

```
TEST> set atpg -abort 5 -merge off
TEST> run atpg
ATPG performed for 71800 faults using internal pattern source.
-----
#patterns      #faults      #ATPG faults    test      process
stored       detect/active   red/au/abort  coverage    CPU time
-----  -----  -----  -----  -----
Begin deterministic ATPG: abort_limit = 5...
32          41288     30512      0/0/2    60.92%     1.35
64          7135      23377      0/0/3    69.04%     2.17
96          3231      20146      0/0/6    72.73%     2.81
128         2643      17503      0/0/7    75.74%     3.33
160         1976      15527      0/0/11   78.00%     3.91
192         1977      13550      0/0/13   80.26%     4.43
224         1450      12100      0/0/16   81.92%     4.85
256         1246      10854      0/0/21   83.35%     5.32
288         1101      9753       0/0/24   84.61%     5.77
319         683       9070       0/0/26   85.39%     6.13
351         748       8322       0/0/27   86.24%     6.46
383         620       7702       0/0/29   86.95%     6.77
:           :           :           :           :
1617        41         348       0/0/170  95.37%    22.02
1648        51         297       0/0/171  95.43%    22.34
1652        12         285       0/0/171  95.45%    22.43
TEST>
```

For comparison, [Example 4-8](#) shows a transcript from an ATPG run on the same design with the merge effort set to high.

Example 4-8 Run ATPG Transcript, Merge Effort Set to High

```
TEST> set atpg -abort 5 -merge high
TEST> run atpg
ATPG performed for 71800 faults using internal pattern source.
-----
#patterns      #faults      #ATPG faults    test      process
stored        detect/active  red/au/abort  coverage   CPU time
-----  -----
Begin deterministic ATPG: abort_limit = 5...
32          52694     19106      0/0/2    73.93%    39.05
64          6363      12743      0/0/6    81.21%    58.29
96          3200      9543       0/0/10   84.88%    74.35
128         2082      7461       0/0/13   87.26%    91.86
160         1234      6227       0/0/15   88.65%   105.62
192         1182      5045       0/0/17   90.00%   117.14
224         849       4196       0/0/21   90.97%   127.18
256         610       3586       0/0/25   91.67%   136.52
288         572       3014       0/0/29   92.32%   145.44
320         514       2500       0/0/34   92.91%   154.06
352         420       2080       0/0/37   93.39%   161.81
383         327       1753       0/0/43   93.77%   169.07
415         320       1433       0/0/49   94.13%   176.13
447         253       1180       0/0/72   94.42%   183.10
479         212       968        0/0/80   94.67%   189.54
511         176       792        0/0/90   94.87%   195.15
543         110       682        0/0/111  94.99%   200.98
575          97       585        0/0/133  95.11%   205.85
607          60       525        0/0/145  95.17%   210.38
639          90       435        0/0/175  95.28%   214.81
671          84       351        0/0/177  95.37%   218.10
695          46       305        0/0/177  95.43%   220.55
TEST>
```

[Table 4-1](#) explains the columns in the Run ATPG transcripts.

Table 4-1 Columns in the Run ATPG Transcript

Column Title	Contents
#patterns stored	The total cumulative number of stored patterns (patterns that TetraMAX keeps)
#faults detect	The number of faults detected by the current group of 32 patterns
#faults active	The number of faults remaining active
#ATPG faults red/au/abort	The cumulative number of faults found to be redundant, ATPG untestable, or aborted
test coverage	The cumulative test coverage
process CPU time	The cumulative CPU runtime, in seconds

With merge effort turned off, the design example produced the following results:

- Test coverage = 95.45 percent
- Number of patterns stored = 1652
- CPU time = 22 seconds

With merge effort set to high, the same design produced the following results:

- Test coverage = 95.43 percent
- Number of patterns stored = 695
- CPU time = 221 seconds

For a compromise between pattern compactness and CPU runtime, you can use the `-auto_compression` option in the `run atpg` command. This option selects an automatic algorithm designed to produce reasonably compact patterns and high test coverage, with very little user effort and a reasonable amount of CPU time. To use this option, the fault source must be internal.

Increasing Effort Over Multiple Passes

Determining an appropriate setting for the abort limit is an iterative process. The following multipass approach produces reasonable results without using excessive CPU time:

1. Set the abort limit to 10 or less.
2. Set the merge effort to Off.
3. Generate test patterns (`run atpg`).
4. Examine the results. If there are too many ND (not detected) faults remaining, increase the abort limit and generate test patterns again.
5. Repeat as necessary to determine the minimum abort limit necessary to achieve the desired results.

The command-line form of this sequence might look like this:

```
TEST> set atpg -abort_limit 10 -merge off
TEST> run atpg
TEST> set atpg -abort 50
TEST> run atpg
TEST> set atpg -abort 250
TEST> run atpg
```

Note:

Increasing the abort limit might decrease the number of ND faults, but it will not decrease the number of AU (ATPG untestable) faults.

Maximizing Test Coverage Using Fewer Patterns

To obtain the maximum test coverage while minimizing the number of patterns,

1. Obtain an estimate of test coverage using the Quick Test Coverage technique. If you are not satisfied with the estimate, determine the cause of the problem and obtain satisfactory test coverage before you attempt to achieve minimum patterns.
 2. Set the abort limit to 100–300.
 3. Set the merge effort to High.
 4. Execute `run atpg -auto_compression`.
 5. Examine the results. If there are still too many NC or NO faults remaining, increase the Abort Limit by a factor of 2 and execute `run atpg` again.
-

Obtaining Target Test Coverage Using Fewer Patterns

To obtain a target test coverage value while minimizing the number of patterns, follow the procedure for obtaining maximum test coverage and set the coverage percentage (`-coverage` option) to a number between 1 and 99 that represents your target test coverage.

Note:

TetraMAX creates patterns in groups of 32 and checks this limit at each 32-pattern boundary, so the patterns generated might exceed the target test coverage.

Review the transcript. If you find that your target is met with the first few patterns of the last group of 32 and you do not want to include all of the last group of patterns, use the `write patterns -last` command to truncate the patterns written as output at the point at which the target was met.

The target coverage is affected by your use of the `set fault -report` command. If fault reporting is set to collapsed, the target percentage is in collapsed fault numbers. If fault reporting is set to uncollapsed, the target percentage is in uncollapsed numbers. The test coverage obtained through the uncollapsed fault list is usually higher and within a few percentage points of the test coverage obtained through the collapsed fault list (note, however, test coverage can slightly more with the fault report set to collapsed compared to the test coverage with fault coverage set to uncollapsed). To be conservative, set fault

reporting to collapsed before you generate patterns for a specific target coverage. When you have finished, display the test coverage using the uncollapsed fault list numbers. Often, the actual test coverage achieved will be higher than your target.

Detecting Faults Multiple Times

N-detection attempts to detect faults *n* times in ATPG. The default is one fault detection. During fault simulation, the fault is kept in the active list until it is detected *n* times. Studies have shown that detecting faults with multiple patterns helps catch defects that cannot be modeled with standard fault models. Examples include transistor stuck-open or cell-level faults.

Pattern size, memory consumption, and runtime will be larger than with the default one fault detection.

With the exception of the IDDQ and path delay fault models, all other fault models are supported. Distributed processing, Full-Sequential ATPG, and fault simulation are not supported.

The n-detection capability is implemented with options of the `run atpg`, `run fault_sim`, and `report faults`. See Online Help for each of these commands for descriptions of the n-detect options.

N-detect ATPG should be used in conjunction with the `set atpg -decision random` command to increase the probability of detecting the faults in different ways. Note that TetraMAX does not guarantee that each fault will be detected in different ways.

Reviewing Test Coverage

You can view the results of the test coverage and the number of patterns generated using the Report Summary dialog box, or you can enter the `report summaries` command at the command line.

Using the Report Summaries Dialog Box

To use the Report Summaries dialog box to generate a fault summary report,

1. Click the Summary button in the command toolbar at the top of the TetraMAX main window. The Report Summaries dialog box appears.
2. Select the summaries you want.

For details about available settings, see online Help for the `report summaries` command.

3. Click OK.

Using the report summaries/fault Command

You can also generate a fault summary report using the `report summaries` or `report fault` command. For example:

```
TEST> report summaries  
TEST> report fault -summary
```

For the complete syntax and option descriptions, see online Help for the `report summaries` and `report fault` commands.

Examples. An example output report showing the fault counts and the test coverage obtained by using the uncollapsed fault list is shown in [Example 4-9](#). A detailed description of each fault class is presented in [Chapter 10, “Fault Lists and Faults.”](#)

Example 4-9 Uncollapsed Fault Summary Report

```
TEST> report summaries  
Uncollapsed Fault Summary Report  
-----  
fault class           code   #faults  
-----  
Detected             DT     83348  
Possibly detected    PT     324  
Undetectable         UD     1071  
ATPG untestable     AU     3453  
Not detected         ND     212  
-----  
total faults          88408  
test coverage        95.62%  
-----  
      Pattern Summary Report  
-----  
#internal patterns    1636  
-----
```

[Example 4-10](#) shows the same report with collapsed fault reporting. Notice that the total faults are fewer, as are the individual fault categories.

Example 4-10 Collapsed Fault Summary Report

```
TEST> set fault -rep collapsed
TEST> report summaries
Collapsed Fault Summary Report
-----
fault class           code #faults
-----
Detected             DT    50993
Possibly detected   PT     214
Undetectable        UD     1035
ATPG untestable     AU     2370
Not detected        ND     122
-----
total faults         54734
test coverage        95.16%
-----
Pattern Summary Report
-----
#internal patterns  1636
```

To find out where the faults are located in the design, see “[Where Are the Faults Located?](#)” on page 20-9.

Writing ATPG Patterns

You can format and save the test patterns using the Write Patterns dialog box, or you can enter the `write patterns` command at the command line.

Note:

For information on translating adaptive scan patterns into normal scan-mode patterns, see “[Pattern File](#)” on page 19-5.

Using the Write Patterns Dialog Box

To use the Write Patterns dialog box to format and save test patterns,

1. Click the Write Pat button in the command toolbar at the top of the TetraMAX main window. The Write Patterns dialog box appears
2. In the Pattern File Name field, enter the name of the pattern file to be written. Use the Browse button to find the directory you want to use or to view a list of existing files.
3. Accept the default settings unless you require more.

The explanation of the options is the same as for the Write Patterns dialog box. For additional information about the available options, see online Help for the `write patterns` command.

4. Click OK.
-

Using the write patterns Command

You can also write test patterns from the command line with the `write patterns` command. For example:

```
TEST> write patterns pat.bin -format binary -last 49 -replace
```

For the complete syntax and option descriptions, see online Help for the `write patterns` command.

Examples

[Example 4-11](#) shows some commonly used `write patterns` command variations.

Example 4-11 Common write patterns Command Variations

```
TEST> write patterns patterns.bin -format binary -replace
End writing 645 patterns, CPU_time = 0.51 sec, File_size = 593960

TEST> write patterns patterns.stil -format stil -replace
End writing 645 patterns, CPU_time = 4.35 sec, File_size = 3073641

TEST> write patterns patterns.ver -format verilog -parallel 1 -
replace
End writing 645 patterns, CPU_time = 6.21 sec, File_size = 5039321
```

Writing Fault Lists

TetraMAX maintains a list of potential faults for a design. Using the Report Faults dialog box or the `write faults` command, you can write the fault list to a file for analysis or to read back in for future ATPG sessions. For more information, see “[Fault Lists](#)” on page 10-2.

Using the Report Faults Dialog Box

To use the Report Faults dialog box to generate a fault list,

1. From the menu bar, choose Faults > Report Faults. The Report Faults dialog box appears.
2. Use the Report Type list box to select the type of fault report that you want. A set of additional options may appear to the right of the Report Type list box, depending on your selection.
3. Select the options you want.

The options are generally the same as for the Report Faults dialog box. For additional information about the available options, see online Help for the `report faults` command.

4. Click OK.
-

Using the write faults Command

You can also generate a fault list using the `write faults` command. For example:

```
TEST> write faults faults.AU -class au -replace
```

For the complete syntax and option descriptions, see online Help for the `write patterns` command.

Examples

The following examples demonstrate how the command is used.

The following command writes all faults:

```
TEST> write faults filename -all -replace
```

The following command writes only the undetectable blocked (UB) and undetectable redundant (UR) fault classes:

```
TEST> write faults filename -class UB -class UR -replace
```

The following command writes only the faults down one hierarchical path:

```
TEST> write faults filename /top/demux/core/mul8x8 -replace
```

By default, the list of faults is either collapsed or uncollapsed as determined by the last set `fault -report` command. The following command overrides the default by using the `-collapsed` option:

```
TEST> write faults filename -all -replace -collapsed
```

[Example 4-12](#) shows a typical uncollapsed fault list. The equivalent faults always immediately follow the primary fault and are identified by two dashes (--) in the second column.

Example 4-12 Uncollapsed Fault List

```
sa0  NP  /moby/bus/Logic0206/N01
sa0  --  /moby/bus/Logic0206/H01
sa0  --  /xyz_nwr
sa0  NP  /moby/i278/N01
sa0  --  /moby/i278/H01
sa0  --  /moby/i337/N01
sa0  --  /moby/i337/H02
sa1  --  /moby/i337/H01
sa0  --  /moby/i222/N01
sa0  --  /moby/i222/H01
sa0  --  /moby/i222/H02
sa0  NP  /moby/core/PER/PRT_1/POUTMUX_1/i411/N01
sa0  --  /moby/core/PER/PRT_1/POUTMUX_1/i411/H03
sa0  --  /moby/core/PER/PRT_1/POUTMUX_1/i411/H04
sa1  --  /moby/core/PER/PRT_1/POUTMUX_1/i411/H01
sa1  --  /moby/core/PER/PRT_1/POUTMUX_1/i411/H02
```

For comparison, [Example 4-13](#) shows the same fault list written with `-collapsed` specified.

Example 4-13 Collapsed Fault List

```
sa0  NP  /moby/bus/Logic0206/N01
sa0  NP  /moby/i278/N01
sa0  NP  /moby/core/PER/PRT_1/POUTMUX_1/i411/N01
```

Using Command Files

You can accomplish all the types of interactive tasks described in this chapter using a command file. [Example 4-14](#) shows a typical command file, which reads in a design that has been debugged to eliminate DRC problems. It creates and stores ATPG patterns and fault lists while saving the execution log.

Example 4-14 A Typical Command File

```
// --- basic ATPG command sequence
//
set message log last_run.log -replace
//
// --- read design and libraries
//
read netlist my_design.v -delete
read netlist /home/vendor_A/tech_B/verilog/*.v -noabort
report modules -summary
report modules -error
//
// --- build design model
//
run build_model my_top_level_name
report rules -fail
//
// --- define clocks and pin constraints
```

```

//  

add clock 1 CLK MCLK SCLK  

add clock 0 resetn ioscl4m  

add pi constraint 1 testmode  

//  

// --- define scan chains & STIL procedures, perform DRC  

checks  

//  

run drc my_design.spf  

report rules -fail  

report nonscan cells -summary  

report bus -summary  

report feedback paths -summary  

//  

// --- create patterns  

//  

set atpg -abort 20 -pat 1500 -merge high  

add faults -all  

run atpg -auto_compression  

report summaries  

//  

// --- save fault list and patterns  

//  

report faults -level 5 64 -class au -collapse -verbose  

write faults faults.all -all -replace  

write patterns patterns.v -format verilog -parallel 2 -replace  

//  

exit  

//
```

Setting Up Advanced ATPG Features

This section covers advanced features available for setting up the ATPG environment for your design. The following topics are covered:

- [Declaring Equivalent and Differential Input Ports](#)
- [Declaring Primary Input Constraints](#)
- [Masking Input and Output Ports](#)
- [Using ATPG Constraints](#)
- [Masking Scan Cell Inputs and Outputs](#)
- [Previewing Potential Scan Cells](#)
- [Deleting Top-Level Ports From Output Patterns](#)
- [Creating Custom ATPG Models](#)
- [Removing Unused Logic](#)
- [Condensing ATPG Libraries](#)

- Working With Clock Groups
 - Improving Test Coverage With Test Points
-

Declaring Equivalent and Differential Input Ports

You can declare two primary input ports to be equivalent or differential. During ATPG, equivalent ports are always driven with the same values and differential ports are always driven with complementary values.

You can use the Add PI Equivalences dialog box to make this kind of declaration, or you can enter the `add pi equivalences` command at the command line.

Using the Add PI Equivalences Dialog Box

To use the Add PI Equivalences dialog box to make two primary input ports to be equivalent or differential,

1. From the menu bar, choose Constraints > PI Equivalences > Add PI Equivalences. The Add PI Equivalences dialog box appears.
2. Select the ports and logic relationships.
For additional information about the available options, see online Help for the `add pi equivalences` command.
3. Click OK.

Using the add pi equivalences Command

You can also declare equivalent or differential input ports by using the `add pi equivalences` command. For example:

```
DRC> add pi equivalences ENA_P -inv ENA_N
```

For the complete syntax and option descriptions, see TetraMAX Online Help for the `add pi equivalences` command.

Examples. In the following example, the first line defines the two input ports `my_port1` and `my_port2` as equivalent; the second line defines that the following ports is constrained to be at an inverted value relative to the first port in the list.

```
DRC> add pi equivalences my_port1 my_port2
DRC> add pi equivalences my_port1 -invert my_port2
```

When differential inputs are also clocks, you must first define each port as a clock and then define the equivalence relationship, as in the following example:

```
DRC> add clocks 0 clock_pos  
DRC> add clocks 1 clock_neg  
DRC> add pi equiv clock_pos -differential clock_neg
```

The third line defines them as differential. This is similar in function to the `-invert` option with two differences. The first difference is that only two pins are accepted. The second difference is that pins declared as having a `-differential` relationship that are also clocks retain that relationship when clock grouping is enabled. A differential clock relationship formed with the `-invert` option may be ignored by clock grouping. Pins declared as having a differential relationship are driven to opposite values by generated patterns.

Declaring Primary Input Constraints

You can define constant logic values on top-level ports (primary inputs or bidirectional). Thus defined, the ATPG will generate only patterns that satisfy a defined list of constraints.

You can use the Add PI Constraints dialog box to define this list, or you can enter the `add pi constraints` command at the command line.

Using the Add PI Constraints Dialog Box

To use the Add PI Constraints dialog box to declare a PI constraint,

1. From the menu bar, choose Constraints > PI Constraints > Add PI Constraints. The Add PI Constraints dialog box appears.
2. Enter the name of the port you want to constrain and choose the value to which you want to constrain the port.

For additional information about the available options, see online Help for the `add pi constraints` command.

3. Click OK.

Using the add pi constraints Command

You can also define PI constraints by using the `add pi constraints` command. For example:

For the complete syntax and option descriptions, see online Help for the `add pi constraints` command.

```
DRC> add pi equivalences ENA_P -inv ENA_N
```

Masking Input and Output Ports

You can mask an input port or output port to isolate it from the design during debugging. For example, if a lower-level module you are testing appears to have full controllability and observability of all of its input and output ports in stand-alone configuration but loses this control when placed in the higher-level module, you might want to mask those inputs and outputs that are not controllable or observable.

You mask an input port by defining a primary input constraint in which the input port is held to an X value. You can define the constraint by using the Add PI Constraints dialog box (see “[Declaring Primary Input Constraints](#)” on page 4-37) or by using the `add pi constraints` command:

```
DRC> add pi constraints X port_name
```

You mask an output port by listing it in the Add PO Masks dialog box (opened by choosing Constraints > PO Masks > Add PO Masks menu command) or by using the `add po masks` command:

```
DRC> add po masks port_name
```

Using ATPG Constraints

You can use ATPG constraints to define internal restrictions that you cannot define with the `add pi constraints` command. ATPG constraints are in effect during ATPG and optionally during test design rule checking (DRC). The use of ATPG constraints is illustrated in the following examples.

Usage Example 1

In this example, a library module called FIFO has two control inputs, push and pop. Under normal operation, the control logic for push and pop ensures that both are never asserted at the same time. However, under the random conditions of ATPG, this control is not guaranteed. To ensure that push and pop are never asserted at the same time, you can define an ATPG constraint at the module level by first adding a temporary gate to facilitate the ATPG constraint and then defining the constraint itself.

Adding a Temporary Gate to Facilitate the ATPG Constraint. You want a logic function with a single output that can be monitored to determine that the push and pop pins are at the desired logic states. To define an ATPG primitive to implement this logic function,

1. Choose Constraints > ATPG Primitives > Add ATPG Primitives. The Add ATPG Primitives dialog box appears.
2. In the Type list, select the ATPG primitive SEL01. (For a list of all available ATPG primitives, see the online reference for the `add atpg primitives` command.) The SEL01 function produces a 1 as its output if all inputs are 0 or if only one input is 1 and the rest are 0. For the example 2-input implementation, SEL01 produces a 0 only if both inputs are 1.
3. In the ATPG Primitive Name field, type the name you want to give this primitive.
4. In the Module field, type the name of the module in which you want this primitive to be.
5. In the Input Constraints field, enter the inputs that are to be constrained (in this case, push and pop). Click Add after each entry. The inputs are added to the list in the Input Constraints window
6. Click OK.
7. Click OK.

Alternatively, you can add the primitive using the `add atpg primitives` command, as follows:

```
DRC> add atpg primi FIFO_CTRL sel01 -module FIFO push pop
```

The new gate, FIFO_CTRL, is added to the module FIFO and uses the module-level pins named push and pop as input to the SEL01 function. The output pin of the function is referenced by the name FIFO_CTRL.

If necessary, you can add more primitives and cascade the logic to build more complex logic functions.

Defining the ATPG Constraint. To apply a constraint to the output of the newly added primitive, you can use the `add atpg constraints` command, as follows:

```
DRC> add atpg constraints MY_LABEL 1 -module FIFO FIFO_CTRL
```

This command defines a constraint, referenced by MY_LABEL, that holds the output FIFO_CTRL to a 1 value. SEL01 cannot have an output of 1 if both of its inputs are 1, so this constraint ensures that the pins push and pop are never asserted at the same time.

Usage Example 2

In this example, a combinational gate is buried within the design hierarchy. Under random conditions, there is a timing-sensitive path causing attempts to generate ATPG patterns to fail simulation. Your analysis concludes that if you could hold two of the pins of a 4-input NAND gate at a high value, you could block the use of this timing-sensitive path.

The instance path name of the NAND gate is `asic_top/BRL/regbank2/u1`, and the input pins you want to control are A and C.

To add the desired constraints,

1. Choose Constraints > ATPG Constraints > Add ATPG Constraints. The Add ATPG Constraints dialog box appears.
2. For each constraint, you specify a constraint name, the constraint site, and value.
3. You can apply the constraint to a single site or to selected pins of all instances of a module.
4. Click OK.

You can also add the constraints using the `add atpg constraints` command. For example:

```
BUILD> add atpg con NAND_BLK2 1 /asic_top/BRL/regbank2/u1/C
```

Masking Scan Cell Inputs and Outputs

TetraMAX supports a number of scan cell controls. You can define these controls by using the Add Cell Constraints dialog box, or you can enter the `add cell constraints` command at the command line.

You specify the location of the cell constraint by either

- Using the name of the scan chain and the bit position, with bit 0 as the bit closest to the scan chain output
- Using an instance path name to the scan chain element

The five scan cell controls are

- 0: The scan cell is always loaded with a 0 during the scan chain load.
- 1: The scan cell is always loaded with a 1.
- X: The scan cell is always loaded with an X.

- OX: No restrictions exist on the loaded value, but any data captured by the regular system clock is considered to be observed as X. That is, the scan cell can be loaded to control logic connected to its outputs, but its data input is always considered X.
- XX: The load is always X, and the observe is always X.

Note:

The loading of a scan cell with an X value for the X or XX cell constraint provides an X for simulation. However, on a device tester, the X is translated into a 0 or a 1 because you cannot drive an X on a tester.

Using the Add Cell Constraints Dialog Box

To use the Add Cell Constraints dialog box to define scan cell controls,

1. From the menu bar, choose Constraints > Cell Constraints > Add Cell Constraints. The Add Cell Constraints dialog box appears.
2. Specify the location of the cell constraint by entering the name of a scan chain or instance.
3. Enter a bit position for the scan chain and scan cell control values for the scan chain and instance.
4. For additional information about the available options, see online Help for the `add cell constraints` command.
5. Click OK.

Using the `add cell constraints` Command

You can also define scan cell controls using the `add cell constraints` command. For example:

```
BUILD> add cell constraints 0 /TOP/U1/sifter/reg42
```

For the complete syntax and option descriptions, see online Help for the `add cell constraints` command.

Previewing Potential Scan Cells

You can preview the effect on your design of changing flip-flops and latches from nonscan elements to scan elements in scan chains without actually changing your design. To do this, you place one or more nonscan sequential devices in a virtual scan chain. TetraMAX treats the virtual scan chain as a true scan chain. Remember to set up the clocks, and when you run ATPG, you see the potential effect on test coverage.

Sequential devices in the Set Scan Ability list must meet all DRC rule checks for scan chain elements. Some of the devices might fail DRC because of uncontrolled asynchronous set/reset connections. (TetraMAX converts the devices into a scan chain but does not change set/reset pins.)

Using the Set Scan Ability Dialog Box

You can place the nonscan devices in a virtual scan chain by listing them in the Set Scan Ability dialog box. To use the Set Scan Ability dialog box to list the nonscan devices in a virtual scan chain,

1. From the menu bar, choose Scan > Set Scan Ability. The Set Scan Ability dialog box appears.
2. Select the method and add DLAT/DFF gates from the list.

For more information about the controls in this dialog box, see online Help for the `set scan ability` command.

3. Click OK.

Using the `set scan ability` Command

You can also place nonscan sequential devices in a virtual scan chain using the `set scan ability` command. For example:

```
DRC> set scan ability on core/host/status
```

For the complete syntax and option descriptions, see online Help for the `set scan ability` command.

The following example adds four devices to the virtual scan chains:

```
DRC> set scan ability on /top/U1/U2/reg1
DRC> set scan ability on /top/U1/U2/reg2
DRC> set scan ability on /top/U1/U2/reg3
DRC> set scan ability on /top/U1/U2/reg4
```

When you use a `set scan ability` list, you might not be able to write patterns because the patterns include the virtual scan chain. Any patterns that are written will fail simulation unless the design is modified to convert the virtual scan chain into a real scan chain.

Deleting Top-Level Ports From Output Patterns

Some netlist formats include nonlogic top-level ports (for example, power and ground). ATPG patterns that include power and ground can create problems with simulation. You can eliminate these and other unwanted top-level ports from the generated patterns using the add net connections command.

The following example removes the top-level input ports pwr1, pwr2, and pwr3 from the generated patterns:

```
BUILD> add net connect pwr1 pwr2 pwr3 -remove
```

Note:

This command modifies only the in-memory image of the design. These changes do not appear in the output from the write netlist command.

Creating Custom ATPG Models

You can create custom models specifically for ATPG use by constructing a Verilog gate-level representation of the logic function using a combination of Verilog primitives, TetraMAX primitives, and other defined modules. For a list of TetraMAX primitives, see the online documentation on “ATPG Modeling Primitives.”

Use only Verilog primitives or instances of other Verilog modules when possible. Because Verilog understands these devices, you can simulate these modules to validate that they function as expected.

Examples

[Example 4-15](#) uses TetraMAX primitives to model the test mode of a particular device. The model provides a constant 1 on the output lock, and a constant 0 on the outputs ref_out, div2, and div4 when test is asserted. Otherwise, these outputs are X.

Example 4-15 Custom ATPG Model Using ATPG Primitives

```
module phase_lock1 (test, ref_in, delayed_in, ref_out, div2, div4,
lock);
    input test, ref_in, delayed_in;
    output ref_out, div2, div4, lock;
    wire xval;

    _TIEX u1 (delayed_in, xval);
    _MUX u2 (test, ref_in, 1'b0, ref_out);
    _MUX u3 (test, xval, 1'b0, div2);
    _MUX u4 (test, xval, 1'b0, div4);
    _MUX u5 (test, xval, 1'b1, lock);
endmodule
```

[Example 4-16](#) uses Verilog primitives to implement the same functions.

Example 4-16 Custom ATPG Model Using Verilog Primitives

```
module mux (sel,d0,d1, out);
    input d0,d1,sel;
    output out;
    wire n1,n2,n3;
    not u1 (selb, sel);
    and u2 (n2, d1,sel);
    and u3 (n3, d0,selb);
    or  u4 (out, n1,n2);
endmodule

module phase_lock2 (test, ref_in, delayed_in, \
                     ref_out, div2, div4, lock);
    input test, ref_in, delayed_in;
    output ref_out, div2, div4, lock;
    wire xval;

    buf   u1  (xval, 1'bx);
    mux  u2  (test, ref_in, 1'b0, ref_out);
    mux  u3  (test, xval, 1'b0, div2);
    mux  u4  (test, xval, 1'b0, div4);
    mux  u5  (test, xval, 1'b1, lock);
endmodule
```

[Example 4-17](#) shows a custom ATPG model of a D flip-flop with a rising-edge clock, asynchronous active-high set, asynchronous active-low resetn, and scan input sdi enabled when scan is asserted. The flip-flop has true and complementary outputs q and qn, and an output sdo, a buffered replica of output q used for scan.

Example 4-17 Custom ATPG Model of a D Flip-Flop

```
module DFWSRB (clk, data, sdi, scan, set, resetn, q, qn, sdo);
    input clk, data, sdi, scan, set, resetn;
    output q, qn, sdo; wire din;

    _MUX u1 (scan, data, sdi, din);
    _DFF u2 (set, !resetn, clock, din, q);
    _INV u3 (q, qb);
    _BUF u3 (q, sdo);
endmodule
```

Removing Unused Logic

Designs can contain unused logic for several reasons, including:

- Existing modules are reused and some sections of the original module are not used in the new design.
- Synthesis optimization has not yet been performed to remove unused logic.

- Gates are created as a side effect to support timing checks in the defining modules.

Examples

[Example 4-18](#) shows a module definition for a scan D flip-flop with asynchronous reset. Because of timing check side effects, the module contains extra gates, with instance names timing_check_1, timing_check_2, and so on. These gates form outputs that are referenced exclusively in the specify section. This is a common technique for developing logic terms used in timing checks, such as setup and hold.

Example 4-18 Example Module With Extra Logic

```
module sdffr (Q, D, CLK, SDI, SE, RN);
    input D, CLK, SDI, SE, RN;
    output Q;
    reg notify;

    // input mux
    not mux_u1 (ckb, CLK);
    and mux_u2 (n1, ckb, D);
    and mux_u3 (n2, CLK, SDI);
    or mux_u4 (data, n1, n2);

    // D-flop
    DFF_UDP dff (Q, data, CLK, RN, notify);

    // timing checks
    not timing_check_1 (seb, SE);
    and timing_check_2 (rn_and_SE, RN, SE);
    and timing_check_3 (rn_and_seb, RN, seb);

    specify
        if (RN && !SE) (posedge CLK => (Q +: D)) = (1, 1);
        if (RN && SE) (posedge CLK => (Q +: SDI)) = (1, 1);
        (negedge RN => (Q +: 1'b0)) = (1, 1);
        $setup (D, posedge CLK && rn_and_seb, 0, notify);
        $hold (posedge CLK, D && rn_and_seb, 0, notify);
        $setup (SDI, posedge CLK && rn_and_SE, 0, notify);
        $hold (posedge CLK, SDI && rn_and_SE, 0, notify);
        $setup (SE, posedge CLK && RN, 0, notify);
        $hold (posedge CLK, SE && RN, 0, notify);
    endspecify
endmodule
```

When this module is converted into a gate-level representation, the timing_check gates in the internal module representation are retained. The output of `report modules -verbose` for module sdffr in [Example 4-19](#) shows each primitive in the TetraMAX model, with the timing check gates present.

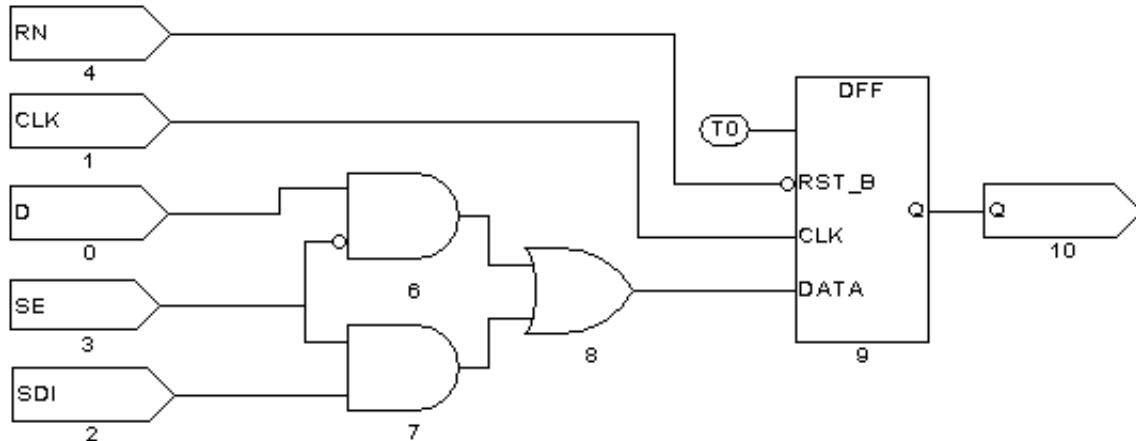
Example 4-19 Module Report Showing Unused Gates

```
BUILD> report module sdffr -verbose
          pins
module name      tot( i/ o/ io) inst refs(def'd) used
----- -----
sdffr           6( 5/ 1/ 0)   8     0 (Y)       1
Inputs : D ( ) CLK ( ) SDI ( ) SE ( ) RN ( )
Outputs: Q ( )
mux_u1 : not conn=( O:ckb I:CLK )
mux_u2 : and conn=( O:n1 I:ckb I:D )
mux_u3 : and conn=( O:n2 I:CLK I:SDI )
mux_u4 : or conn=( O:data I:n1 I:n2 )
dff   : DFF_UDP conn=( O:Q I:data I:CLK I:RN I:notify )
timing_check_1: not conn=( O:seb I:SE )
timing_check_2: and conn=( O:rn_and_SE I:RN I:SE )
timing_check_3: and conn=( O:rn_and_seb I:RN I:seb )
-----
```

By default, TetraMAX deletes unused gates when it builds the design. To specify whether unused gates are to be deleted or kept, choose Netlist > Set Build Options, which displays the Set Build dialog box. Notice that, in this case, the “Delete unused gates” box is checked, meaning that the deletion of unused gates is selected. To keep the extra gates, deselect the “Delete unused gates” box.

[Figure 4-3](#) shows the GSV display of the schematic created when the “Delete unused gates” option is selected. The extra gates do not appear in the schematic.

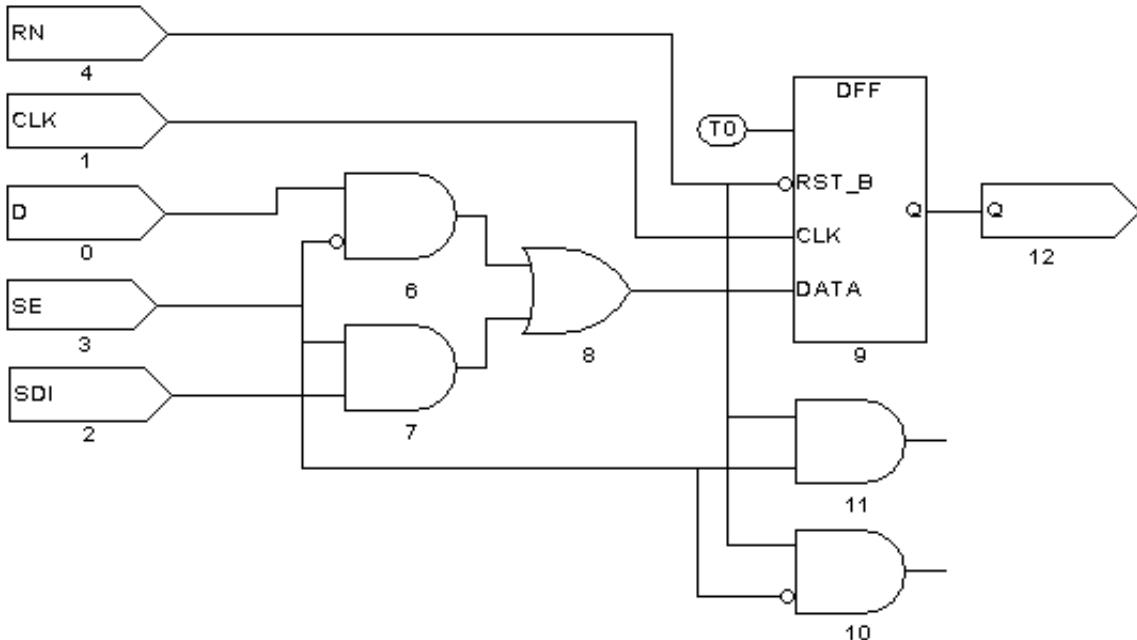
Figure 4-3 Design Schematic With Delete Unused Gates On



To keep the extra gates, deselect the “Delete unused gates” option in the Set Build dialog box. [Figure 4-4](#) shows the resulting schematic. The design retains the three extra timing check gates logically as two additional primitives with unused output pins. These extra gates can produce extra fault-site locations, increasing the total number of faults in the design and

therefore increasing the processing time. Any faults on these gates are categorized as UU (undetectable, unused). Although these UU faults do not lower the test coverage, they still cause an increase in memory usage and processing time.

Figure 4-4 Design Schematic With Delete Unused Gates Off



If you want to change the “Delete unused gates” setting, you must do so before executing the `run build_model` command on your design. If you build your design and then change the setting, you must return to build mode and rerun the `run build_model` command.

You can also change the unused gate deletion setting by using the `set build` command with the `-delete_unused_gates` or `-nodelete_unused_gates` option. The following command overrides the default and keeps unused gates:

```
BUILD> set build -nodelete_unused_gates
```

Condensing ATPG Libraries

TetraMAX attempts to condense each module’s functionality in a netlist into a gate-level representation using TetraMAX simulation primitives. This condensation task can be considerable and can produce some warning messages, which are typically unimportant and can be ignored.

You can create a file that has already been condensed into TetraMAX description form. Creating a condensed form of the library modules has the following benefits:

- Space economy. The modules are stripped of timing and other non-ATPG related information. In addition, the file can be created in compressed form.
- No error or warning messages. The modules are preprocessed and written using either ATPG modeling primitives or simple netlists instantiating other modules.
- Faster module reading. The modules require less time during analysis and are processed faster.
- Information protection. The file can be created in a compressed binary form that is unreadable by any other tool and partially protects the library information within. When you read in the library and write it out again, you see only a stripped-down functional gate version of the original module; no timing or other information remains.

The transcript in [Example 4-20](#) illustrates the creation of a condensed library file, which is a two-step process:

1. Read in all desired modules.

In [Example 4-20](#), 1,436 modules are initially found in 1,430 separate files. The read process took 21.5 seconds and reported 16 warnings.

2. Write out the modules as a single file in your choice of formats.

In the example, the modules are written out as a single GZIP compressed file.

Example 4-20 Creating a Condensed Library File

```
BUILD> read netlist lib/*.v
Begin reading netlists ( lib/*.v )...
Warning: Rule N12 (invalid UDP entry) failed 8 times.
Warning: Rule N13 (X_DETECTOR found) failed 8 times.
End reading netlists: #files=1430, #errors=0, #modules=1436,
#lines=157516,
    CPU_time=21.5 sec
BUILD> write netlist parts.lib.gz -compress gzip
End writing Verilog netlist, CPU_time = 1.13 sec,
    File_size = 47571
BUILD> read netlist parts.lib.gz -delete
Warning: All netlist and library module data are now deleted. (M41)
Begin reading netlist ( parts.lib )...
End parsing Verilog file parts.lib with 0 errors;
End reading netlist: #modules=1436, #lines=18929, CPU_time=0.84
sec
```

The next `read netlist` command processed the data in less than 1 second and produced the same 1,436 modules, this time without rule violation warnings.

Working With Clock Groups

Clock grouping enables TetraMAX to pulse clocks simultaneously as well as detect clocks that can be pulsed serially. It can also detect clocks that have a small amount of sequential effects and group them. In this case, TetraMAX takes all necessary steps in setting up the pattern generation environment to avoid generating vectors that would fail simulation.

Clock grouping has these limitations:

- Dynamic and disturbed clocking are only used by Basic-Scan ATPG.
- Disturbed clocking could result in a slightly lower test coverage because of disturbed cell masking.

Pattern Count Reduction

When you generate combinational vectors in Basic-Scan ATPG, TetraMAX normally uses only one clock pulse per pattern. However, it is sometimes possible to pulse several clocks in the same vector, thus observing more logic and reducing the need for additional patterns. If you have two clocks independent from one another (for example, when you pulse one clock, no logic driven by the other clock is affected), then you need two patterns to exercise the logic in the two clock domains. However, because the clocks are independent, you can pulse them at the same time, thereby saving one test vector. When you use static parallel clock grouping, the grouped clocks must always be pulsed together. None of the clocks in the group may be pulsed alone.

An enhanced version of clock grouping called *dynamic clock grouping* is available where the clocks pulsed for a given vector are selected dynamically during pattern generation, maximizing the fault detection and minimizing the pattern count. An additional clocking scheme to reduce pattern count called *disturbed clocking* is also available.

The distributed clocking scheme allows TetraMAX to group some clocks with a limited number of cells having sequential effects. In this case, even if there are sequential effects, it can be useful to group those clocks to further reduce pattern count. TetraMAX cannot use the disturbed cells. Potential disturbed grouping is done during DRC analysis.

The following procedure explains how to reduce the pattern count.

1. Read in the files and build your design in TetraMAX.
2. Choose your criteria for clock grouping. (See the `set drc` command.)

The static parallel grouping process groups some independent clocks together. To group clocks manually, use the `add pi` equivalence command. Once you have defined a group, any clock belonging to this group cannot be pulsed alone.

- Run the design rule checker.

DRC does an analysis for clock grouping capability.

- Generate Basic-Scan test vectors, for example,

```
run atpg -auto_compression
```

Generating a Clock Group Report

To report results of clock grouping analysis, use the following command:

```
report clocks [-Matrix] [-Verbose]
```

Clock Matrix Report. Use the `-Matrix` option of the `report clocks` command to display a matrix of clock pairs that can be grouped together. In the clock matrix, each row indicates the potential grouping relationships of a candidate clock with all of the other candidate clocks.

For example,

id#	clock_name	type	0	1	2	3	4	5	6	7	8	9	
0	clk	C	---	--A	--A	--A	--A	--A	--A	--A	--A	---	
1	iopclk11	C	B--	---	--A	BPA	BPA	BPA	BPA	BPA	BPA	B--	
2	iopclk12	C	B--	B--	---	--A	BPA	BPA	BPA	BPA	BPA	B--	
3	iopclk21	C	B--	BPA	B--	---	--A	BPA	BPA	BPA	BPA	B--	
4	iopclk22	C	B--	BPA	BPA	B--	---	BPA	BPA	BPA	BPA	B--	
5	iopclk31	C	B--	BPA	BPA	BPA	BPA	---	--A	BPA	BPA	B--	
6	iopclk32	C	B--	BPA	BPA	BPA	BPA	B--	---	BPA	BPA	B--	
7	iopclk41	C	B--	BPA	BPA	BPA	BPA	BPA	BPA	---	--A	B--	
8	iopclk42	C	B--	BPA	BPA	BPA	BPA	BPA	BPA	B--	---	B--	
9	tx_intf1_clk	C	---	--A	--A	--A	--A	--A	--A	--A	--A	---	
10	tx_intf2_clk	C	---	--A	BPA	--A	---	--A	BPA	--A	BPA	B--	
11	tx_intf3_clk	C	---	--A	BPA	--A	BPA	--A	---	--A	BPA	B--	
12	tx_intf4_clk	C	-D-	BPA	BPA	--A	BPA	--A	BPA	--A	---	BP-	
13	por	R	---	--A	--A	--A	--A	--A	--A	--A	--A	--A	
14	rst	SR	---	---	---	---	---	---	---	---	---	---	
id1	id2		C1	#masks C2 masked gates									

Improving Test Coverage With Test Points

In many cases, you can improve TetraMAX test coverage by adding control/observation points to specific areas with known low controllability and observability. TetraMAX then generates additional patterns for faults that are controlled or fed into these points. This process is particularly useful if you want to achieve very high test coverage targets — usually in the high 99% range.

You can use TetraMAX to further improve test coverage by inserting test points based on a user-specified population of undetected faults. In this case, TetraMAX, first performs an analysis to determine the relationship of a set of specified faults. It then determines the optimal location to insert observe points that specifically maximize test coverage. Several different parameters are used so that the number of selected test-points is minimized

Note: this is a general capability that can be applied to any class of faults.

This section includes the following topics:

- [Specifying Test-Point Analysis](#)
- [Running the Flow](#)
- [Limitations](#)

Specifying Test-Point Analysis

You control test point insertion using the `run_testpoint_analysis` command, which can only be specified in the TEST command mode. The syntax for this command is as follows:

```
run_testpoint_analysis
[-max_test_points <number>]
[-num_observe_points_per_cell <number>]
[-class <sub-class>]
[-test_point_file <file_name>]
[-replace]
[-dont_touch <instance>]
```

Argument	Description
<code>-max_test_points <number></code>	Specifies the largest number of test points to be inserted. TetraMAX will stop writing out test-points before reaching the maximum number specified with the <code>-max_test_points</code> option if no improvement is seen during analysis. If this option is not specified, the default maximum is 1000 test points.

Argument	Description
<code>-num_observe_points_per_cell <number></code>	Specifies the maximum number of observe points that should be connected through XOR (observation) to one observe scan flip-flop or primary output. The default is 8 observe points per observe flip-flop or primary output.
<code>-class <sub-class></code>	<p>Specifies the class of faults to be targeted. This switch can be specified multiple times if you need to specify multiple fault classes. For example, you can first generate test patterns, then execute the <code>run testpoint analysis</code> command using the <code>-class NO</code> and <code>-class AN</code> options. If the <code>-class</code> switch is not specified, all undetected faults will be considered for test-point insertion.</p> <p>Note the following fault classes and sub-classes (Subclasses are listed in bold, and are specified in conjunction with the <code>-class</code> option; you cannot specify a major class with the <code>-class</code> option.):</p> <ul style="list-style-type: none"> DT - Detected <ul style="list-style-type: none"> DS - Detected by simulation DI - Detected by implication PT - Possibly detected <ul style="list-style-type: none"> AP - ATPG untestable possibly detected NP - Not analyzed, possibly detected UD - Undetectable <ul style="list-style-type: none"> UU - Undetectable unused UT - Undetectable tied UB - Undetectable blocked UR - Undetectable redundant AU - ATPG untestable <ul style="list-style-type: none"> AN - ATPG untestable not-detected ND - Not detected <ul style="list-style-type: none"> NC - Not controlled NO - Not observed
<code>-test_point_file <file_name></code>	Specifies the name of the output file that will contain the list of test-points. This file uses the same exact file format as the <code>run_observe_analysis</code> command in order to remain compatible with DFT Compiler.
<code>-replace</code>	Causes TetraMAX to replace any existing output file (specified by the <code>-test_point_file</code> option).
<code>-dont_touch <instance></code>	This option prevents the creation of test points in the specified instance. Multiple <code>-dont_touch</code> options can be used for additional instances.

Running the Flow

The flow for running test-point insertion based on undetected fault topology is as follows:

- Run the `run_atpg -auto` command or any other method for generating patterns. If ATPG is not done prior to running the `run_testpoint_analysis` command, all undetected faults will be analyzed, which may result in very long run times.
- Run the following command to generate a list of test points:

```
run_testpoint_analysis <options>
```

- TetraMAX can estimate the test coverage improvement by reading in the test points:

```
run atpg -auto -observe_file <test-point file>
```

Note: the total number of faults shown after ATPG will not include the faults from the additional test points.

- Use DFT Compiler to insert the test points by reading in the file generated by the `run_testpoint_analysis` command, then rerun TetraMAX on the new netlist to generate the final ATPG patterns and coverage.

Limitations

Note the following limitations associated with this feature:

- If you have an LSSD design, you can use the `run_testpoint_analysis` command in TetraMAX; however, DFT Compiler does not support insertion of observe and control test points on this style of scan. In this case, a TESTXG-61 message will be issued in DFT Compiler.
- Only observe scan cell test points are supported at the current time.

5

On-Chip Clocking Support

On-Chip Clocking (OCC) support is common to all scan ATPG and Adaptive Scan environments. This implementation is intended for designs that require ATPG in the presence of PLL and clock controller circuitry.

OCC support includes phase-locked loops, clock shapers, clock dividers and multipliers, etc. In the scan-ATPG environment, scan chain load_unload are controlled through an ATE clock. However, internal clock signals that reach state elements during capture are PLL-related.

This chapter contains the following sections:

- [Background](#)
- [Supported Flows](#)
- [Definitions](#)
- [Pattern Support](#)
- [Limitations](#)
- [Design Flows](#)
- [OCC Support in TetraMAX](#)
- [OCC-Specific DRC Rules](#)

Background

At-speed testing for deep submicron defects requires not only more complex fault models for ATPG and fault simulation, like transition faults and path delay faults, but also requires the accurate application of two high-speed clock pulses to apply the tests for these fault models. The time delay between these two clock pulses, referred to as the launch clock and the capture clock, is the effective cycle time at which the circuit will be tested.

A key benefit of scan-based at-speed testing is that only the launch clock and capture clock need to operate at the full frequency of the device under test. Scan shift clocks and shift data may operate at much slower speed, thus reducing the performance requirements of the test equipment. However, complex designs often have many different high frequency clock domains, and the requirement to deliver a precise launch and capture clock for each of these from the tester can add significant or prohibitive cost on the test equipment. Furthermore, special tuning is often required to properly control the clock skew to the device under test.

One common alternative for at-speed testing is to leverage existing on-chip clock generation circuitry. This approach uses the active controller, rather than off-chip clocks from the tester, to generate the high speed launch and capture clock pulses. This type of approach generally reduces tester requirements and cost, and can also provide high speed clock pulses from the same source as the device in its normal operating mode without additional skews from the test equipment or test fixtures.

To use this approach, additional on-chip controller circuitry is included to control the on-chip clocks in test mode. The on-chip clock control is then verified, and at-speed test patterns are generated which apply clocks through proper control sequences to the on-chip clock circuitry and test mode controls. DFT Compiler and TetraMAX ATPG support a comprehensive set of features to ensure that:

- The test mode control logic for the OCC operates correctly and has been connected properly.
- Test mode clocks from the OCC circuitry can be efficiently used by TetraMAX for at-speed test generation.
- OCC circuitry can operate asynchronously to shift and other clocks from the tester.

Supported Flows

OCC is supported in the following flows:

- DFT Compiler-to-TetraMAX flow (for details, please see Chapter 7, “Using On-Chip Clocking,” in the DFT Compiler User Guide Vol. 1: Scan)

- Non-DFT Compiler to TetraMAX Flows:
 - Basic Scan with On-Chip Clocking
 - Adaptive Scan with On-Chip Clocking
-

Definitions

Note the following definitions as they apply to OCC in this chapter:

- **Reference Clocks** — The frequency reference to the PLL. It must be maintained as a constantly pulsing and free-running oscillator or the circuitry will lose synchronization.
 - **PLL Clocks** — The output of the PLL. A free-running source that also runs at a constant frequency which may or may not be the same as the reference clock.
 - **ATE Clocks** — Shifts the scan chain typically slower than a reference clock. You must manually add this signal (a port) when inserting the OCC. Note that the ATE clock cannot be a reference clock, and it does not capture.
 - **Internal Clocks** — The OCC is responsible for gating and selecting the PLL clocks and ATE clocks, and for creating the internal clocks, which satisfy ATPG requirements.
 - **External Clocks** — The primary inputs of a design which clock flip-flops directly through combinational logic not generated from PLLs.
-

Pattern Support

Note the following Pattern Support available in OCC:

Table 5-1 OCC Pattern Support

Argument Format	Description Synchronous Single Pulse	Description Synchronous Multi-Pulse	Argument Asynchronous
STIL	Yes	Yes	Yes
STIL99	Yes	Yes	No
WGL	Yes	Yes	No
Verilog	Yes	No	No
Others	Yes	No	No

Limitations

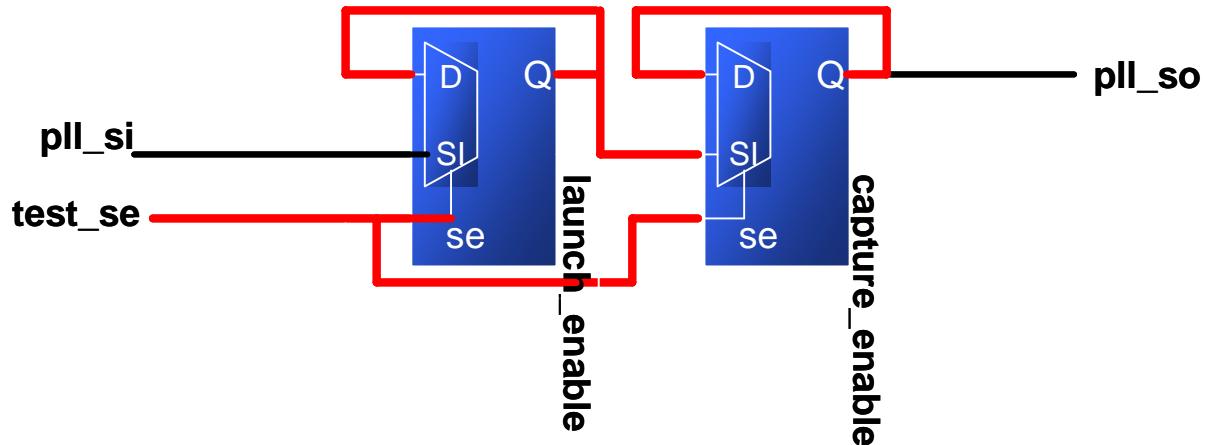
Note the following limitations:

- You must use multiplexed flip-flop scan style.
- You must use generic capture procedures for internal/external clocking. For more information, see “[Creating Generic Capture Procedures](#)” on page 9-18.
- The OCC from DFT Compiler cannot be used for last shift launch.
- Multi-cycle paths can only be tested if the launch and capture events are sufficiently separated in time.
- The clock frequency of the PLL generating internal clocks cannot change dynamically — must be constant (i.e., programmable bits must be non-scan and constant during ATPG).
- Do not use the reference clock as your ATE clock or shift clock.
- End-of-cycle measure is not compatible with PLL reference clocks. With PLL reference clocks defined, ATPG can generate patterns with the following sequence of events:
 - forcePI
 - measurePO
 - pulse reference clocks

When writing such patterns out in STIL (or any other external format), the vector that contains the measurePO must also pulse reference clocks (by definition reference clocks must be pulsed in every vector). But the end-of-cycle measure timing means the order of events is reversed in this vector: pulse reference clocks measurePO. This is incorrect and the pattern will likely fail on silicon. A new message has been added that will flag you to correct the timing:

Warning: Reference Clock <ON_time> < measure_time> in waveformtable. All PO measures were masked. (M664)

- Clock bits must hold state during capture.



- Avoid using reference clock for flip-flops inside the design.
- Mixing Clock (control) bit scan chains is allowed with legacy scan- based design scan chains. Mixing of Clock (control) bit scan chains with internal scan chains is not supported for Adaptive Scan designs as the Clock (control) bit scan chain must not be connected through the Compressor/Decompressor logic.
- Programmable PLLs (test_setup is critical and must not become corrupt during the entire ATPG process).

```

MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        C {
            "all_inputs" = \r26 N;
            "all_outputs" = \r8 X;
        }
        V {
            "ateclk" = P;
            "clk" = P;
            "pll_reset" = 1;
        }
        V {
            "test_mode" = 1;
            "pll_bypass" = 0;
            "pll_reset" = 0;
            "test_se" = 0;
        }
    }
}

```

The `pll_reset` must be constrained to stay in a consistent state while shifting data from the clock chain. The OCC Controller goes through the initialization sequence once and returns to a state to be controlled from the clock chain only. Therefore, the `pll_reset` must be constrained to stay in a consistent state.

- If the reference clock period is an integer divisor of the `test_default_period`, then patterns can be written in the STIL, STIL99 and WGL formats.
- If the reference clock is not an integer divisor to the `test_default_period`, the only format that can be written in a completely correct way is STIL. Other formats (including STIL99) cannot include the reference clock pulses and a warning is printed indicating that these pulses must be added back to the patterns manually.
- Make sure you constrain the scan enable to the off-state in the TetraMAX command file since it is not specified in the OCC protocol file.
- The `tmax2pt.tcl` script supports OCC. However, since there is no timing information for internal clocks in the TMAX database, the timing that is written out is nominal and may not match the design's actual clock timing.

Design Flows

This section discusses the following design flows:

- [DFT Compiler-to-TetraMAX Flow](#)
- [Non-DFT Compiler to TetraMAX Flows](#)

DFT Compiler-to-TetraMAX Flow

This seamless, easy-to-follow flow automatically writes out the SPF file for TetraMAX and the Verilog netlist.

For details on this flow, please refer to Chapter 7, “Using On-Chip Clocking,” in the DFT Compiler User Guide Vol. 1: Scan).

[Figure 5-1](#) illustrates the basic ATPG design flow. This flow consists of the following steps:

1. Read in the Verilog netlist.
2. Read in the library models.
3. Build the ATPG design model.
4. Read in the STIL test protocol file, automatically generated by DFT Compiler. Perform test DRC and make any necessary corrections.

- To run with PLL active, specify the following command:

```
run drc <STIL_file> -patternexec <test_mode>
```
- To run with PLL bypassed, specify the following command:

```
run drc <STIL_file> -patternexec \
<test_mode>_occ_bypass
```

When using default test modes, use one of the following:

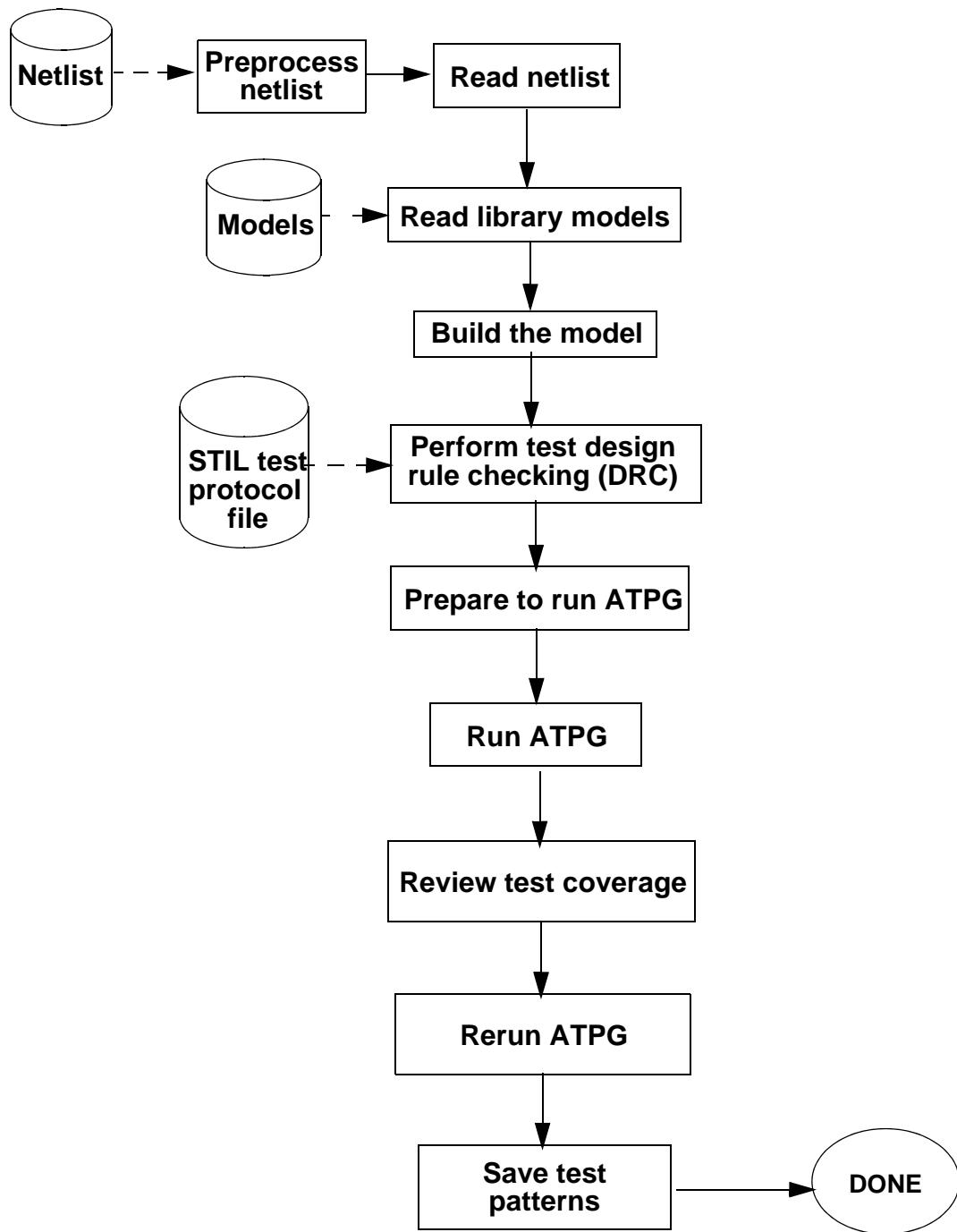
```
run drc <scan_STIL_file> -patternexec Internal_scan
run drc <scan_STIL_file> -patternexec \
Internal_scan_occ_bypass
run drc <compression_STIL_file> -patternexec \
ScanCompression_mode
run drc <compression_STIL_file> -patternexec \
ScanCompression_mode_occ_bypass
```

5. To prepare the design for ATPG, set up the fault list, and set the ATPG options.
6. Run automatic test pattern generation.
7. Review the test coverage and rerun ATPG if necessary.
8. Save the test patterns and fault list.

Note:

The individual steps in this flow are described in detail in [Chapter 4, “ATPG Design Flow.”](#)

Figure 5-1 DFT Compiler to TetraMAX Flow



Non-DFT Compiler to TetraMAX Flows

It is not recommended that you use non-DFT Compiler OCC IP with TetraMAX. If you have this type of IP, you should refer to the “User-Defined Instantiated Clock Controller and Chain Insertion Flow” section in the DFT Compiler User Guide.

OCC Support in TetraMAX

OCC support in TetraMAX provides for automated handling of internal clocks in a generic manner. This automation is enforced by using clock design rules that validate user-specified clock controller settings.

When a design contains both internal clocks (commonly driven by PLL sources), and external (primary input) clocks, the TetraMAX default operation is to use both clock sources for test generation. In some clock-tracing situations, internal clocks will take precedence over external sources, however this may not eliminate all ambiguity, especially when both clock sources are presented to the same internal element.

TetraMAX allows for control of capture clocks that are issued during ATPG on a per-pattern basis. This gives ATPG the flexibility of deciding what internal clocks that should be pulsed in a given capture cycle, instead of incurring the overhead of pulsing all internal clocks every capture cycle. Note that generic capture procedures should be used exclusively. Also, because the pulse placements of different OCC clocks cannot be predicted, you should always use the following command:

```
set_delay -common_launch_capture_clock
```

Scan ATPG Flow

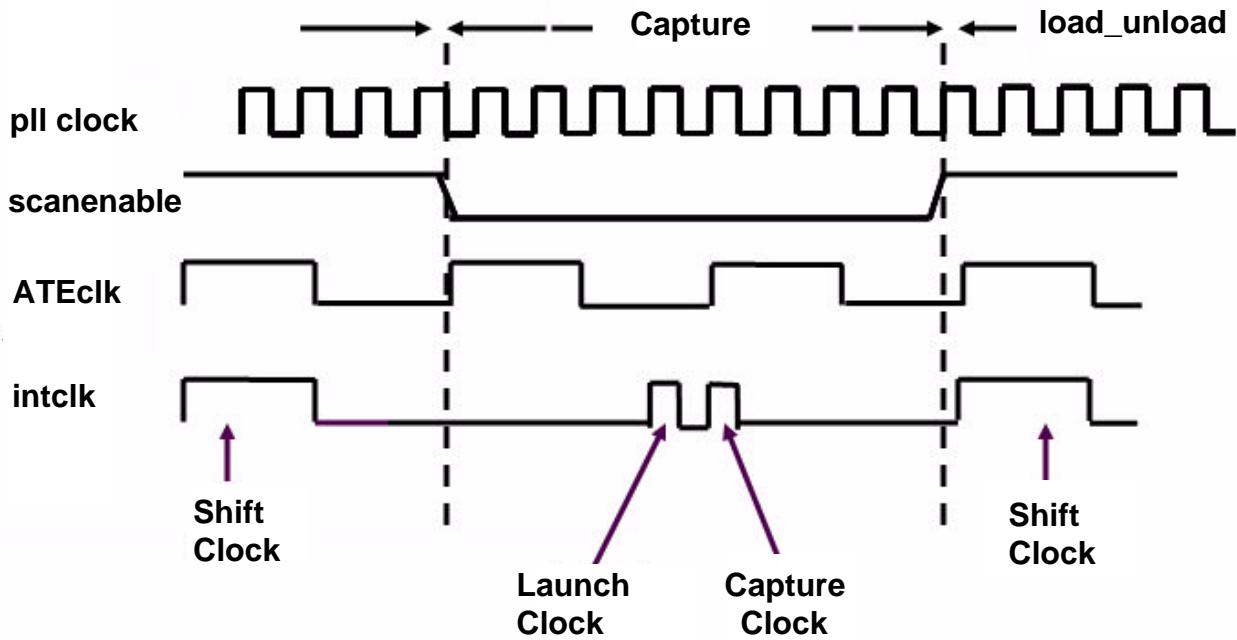
The ATPG flow consists of the following steps:

1. Read the design files.
2. Build the design.
3. Run DRC with the TetraMAX SPF file created by DFT Compiler after scan insertion in presence of PLL circuitry.
4. Run ATPG.

Waveform and Capture Cycle Example

Figure 5-2 shows an example of the relationship between various clocks when the design contains OCC controller.

Figure 5-2 Waveform and Capture Cycle Example



Note in Figure 5-2 that the `refclk` must pulse in every vector.

Please refer to Figure 5-2 for information about `pllclk`, `ateclk`, and `intclk`.

OCC-Specific DRC Rules

Test DRC involves analysis of many aspects of the design. Among other things, DRC checks the following:

- C28 - Invalid PLL source for internal clock
- C28 - Invalid PLL source for internal clock
- C29 - Undefined PLL source for internal clock
- C30 - Scan PLL conditioning affected by non-scancells

- C31 - Scan PLL conditioning not stable during capture
- C31 - Scan PLL conditioning not stable during capture
- C32 - BIST PLL conditioning affected by scan cells
- C33 - BIST PLL conditioning not stable during interval
- C34 - Unsensitized path between PLL source and internal clock
- C35 - Multiple sensitizations between PLL source and internal clock
- C36 - Mistimed sensitizations between PLL source and internal clock
- C37 - Cannot satisfy all internal clocks off for all cycles
- C38 - Bad off-conditioning between PLL source and internal clock
- C39 - Nonlogical clock C connects to scan cell

Reference clocks are used only during design rule checking and are non-logical for pattern generation. PLL clocks are used during scan design rule checking (Category S – Scan Chain Rules) and clock design rule checking (Category C – Clock Rules). Pattern generation does not consider PLL clocks. Internal clocks are used for all capture operations, and normal clock rule checking is applied to these so that TetraMAX can perform these and other DRC checks, you must provide information about clock ports, scan chains, and other controls by means of a STIL test protocol file. The STIL file can be generated from DFT Compiler, or you can create one manually as described in [Chapter 9, “STIL Procedure Files.”](#)

6

Working With Design Netlists and Libraries

This chapter provides information on reading and processing design netlists and library modules.

This chapter contains the following sections:

- [Using Wildcards to Read Netlists](#)
- [Controlling Case-Sensitivity](#)
- [Identifying Missing Modules](#)
- [Using Black Box and Empty Box Models](#)
- [Handling Duplicate Module Definitions](#)
- [Memory Modeling](#)
- [Building the ATPG Design Model](#)
- [Binary Image Files](#)

Using Wildcards to Read Netlists

If your library cells are contained in multiple individual files, you can read them in all at once using wildcards. TetraMAX supports the asterisk (*) to match occurrences of any character, and the question mark (?) to match any single character. For other combinations, see the topic “Limited Regular Expressions” in online Help. Following are some examples.

To read in all files in the directory `mylib` that have the extension `.v`, use the command:

```
BUILD> read netlist mylib/*.v
```

To read in all files in `mylib`, enter:

```
BUILD> read netlist mylib/*
```

To read in all files that begin with `DF` and end with `.udp`, in all subdirectories in `mylib` that end in `_lib`, enter:

```
BUILD> read netlist mylib/*_lib/DF*.udp
```

To read in all files that begin with `DFF`, end in `.v`, and have any two characters in between, enter:

```
BUILD> read netlist DFF???.v
```

You can also use wildcards in the Read Netlist/Image dialog box. Use the Browse button to select any file from the directory of interest and click OK. Then replace the file name with an asterisk.

When you use wildcards, you might find it convenient to use these options:

- Verbose: Produces a message for each file rather than the default message for the sum of all files.
- Abort on error: Determines whether TetraMAX stops reading files when it encounters an error with an individual file.

For more information about the controls in the Read Netlist/Image dialog box, see online Help for the `read netlist` command.

Controlling Case-Sensitivity

Netlist formats differ in whether or not the instance, pin, net, and module names are case-sensitive. When TetraMAX reads a netlist, it chooses case-sensitive or case-insensitive based on the type of netlist by default as follows:

- Verilog Netlists: case-sensitive
- EDIF Netlists: case-insensitive
- VHDL Netlists: case-insensitive

You can override the defaults by using the `-sensitive` or `-insensitive` option of the `read netlist` command. For example, to read in all files ending in `.v` in directory `mylib`, using case-insensitive rules, use the following command:

```
BUILD> read netlist mylib/*.v -insensitive
```

Identifying Missing Modules

If your design references undefined modules, TetraMAX sends you error messages during execution of `run build_model`. To identify all currently referenced undefined modules, you can use the `Netlist > Report Modules` menu command, or you can enter the `report module -undefined` command at the command line. For example:

```
BUILD> report modules -undefined
```

An example of such a report is shown in [Figure 6-1](#).

Figure 6-1 Report Modules Window Listing Undefined Modules

The screenshot shows a Windows application window titled "Report Modules". The menu bar includes "File", "Edit", "Search", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a table with the following columns: module name, pins (tot, i/o, o/, io), inst, refs(def'd), and used. The table lists various undefined modules, each with a total pin count of 0 and all other fields containing 0 or (N).

module name	pins				refs(def'd)	used
	tot	i/	o/	io)		
ICNH	0(0/	0/	0)	0	(N)
ON4	0(0/	0/	0)	0	(N)
AND2	0(0/	0/	0)	0	(N)
DFFRLP	0(0/	0/	0)	0	(N)
DFFRP	0(0/	0/	0)	0	(N)
EXNOR	0(0/	0/	0)	0	(N)
OR2	0(0/	0/	0)	0	(N)
ICN	0(0/	0/	0)	0	(N)
BICN	0(0/	0/	0)	0	(N)
BON4T	0(0/	0/	0)	0	(N)
INC4H	0(0/	0/	0)	0	(N)
MUX2H	0(0/	0/	0)	0	(N)
DFFP	0(0/	0/	0)	0	(N)
OR2H	0(0/	0/	0)	0	(N)
NAN2	0(0/	0/	0)	0	(N)
AND3	0(0/	0/	0)	0	(N)
NOR2	0(0/	0/	0)	0	(N)
DFFLP	0(0/	0/	0)	0	(N)
EXOR	0(0/	0/	0)	0	(N)
NAN3	0(0/	0/	0)	0	(N)
INV	0(0/	0/	0)	0	(N)
MUX2A	0(0/	0/	0)	0	(N)

In the report, the columns for the total number of pins, input pins, output pins, I/O pins, and number of instances all contain 0. Because the corresponding modules are undefined, this information is unknown. In the “refs(def’d)” column, the first number indicates the number of times the module is referenced by the design, and (N) indicates that the module has not yet been defined.

For additional variations of the `report modules` command, refer to the online documentation under “Understanding Reported Data.”

Any undefined module referenced by the design causes a B5 rule violation when you attempt to use the `run build_model` command. The default severity of rule B5 is error, so the build process stops.

If you set the B5 rule severity to warning, TetraMAX automatically inserts a black box model for each missing module when you build the design. In a black box model, the inputs are terminated and the outputs are tied to X. For more information, see “[Using Black Box and Empty Box Models](#).”

To change the B5 rule severity to warning, use the following command:

```
BUILD> set rule B5 warning
```

With this severity setting, when you use the `run build_model` command, missing modules do not cause the build process to stop. Instead, TetraMAX converts each missing module into a black box. After this process, use the `report violations` command to view an explicit list of the missing modules:

```
DRC> report violations B5
```

Leaving the B5 rule severity set to warning might cause you to miss true missing module errors later. To be safe, you should set the rule severity back to error. Before you do this, use the `set build` command to explicitly declare the black box modules in the design, as explained in the next section. Then you can set the B5 rule severity back to error and still build your design successfully.

Using Black Box and Empty Box Models

There might be blocks in a design for which you do not want ATPG to be performed, such as a phase-locked loop block, an analog block, a block that is bypassed during test, or a block that is tested separately, such as a RAM block.

Some ATPG tools require you to build a black box model to represent such a function in your design. TetraMAX, however, lets you declare any block in the design to be a black box or an empty box by using one of the following commands:

```
set build -black_box module_name  
set build -empty_box module_name
```

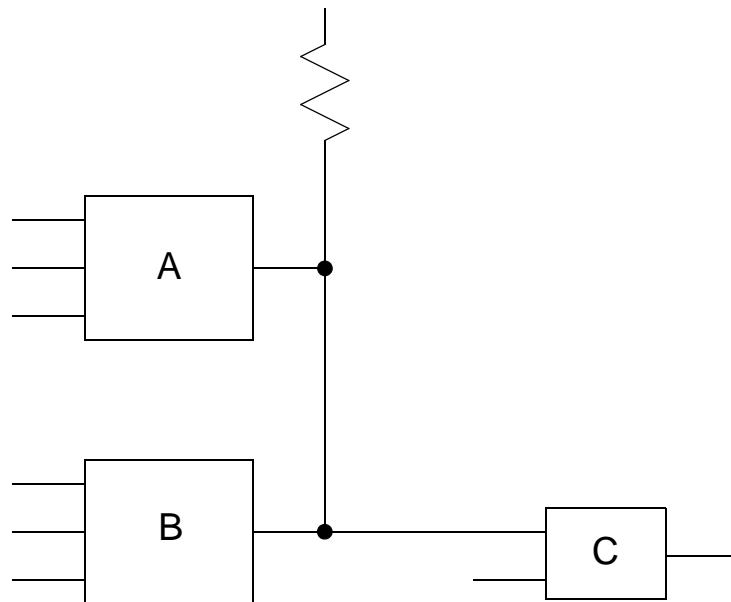
If you declare a block to be a black box, TetraMAX ignores the contents of the block when you build the model with the `run build_model` command. Instead, it terminates the block inputs and connects TIEX primitives to the outputs. Thus, the block outputs are unknown (X) for ATPG.

An empty box is the same as a black box, except that the outputs are connected to TIEZ rather than TIEX primitives. Thus, the block outputs are assumed to be in the high-impedance (Z) state for ATPG.

The black box model is the usual and more conservative model for any block that is to be removed from consideration for ATPG. In certain cases, however, this model can cause contention, thereby preventing patterns from being generated for logic outside of the black box. In these cases, the empty box model is a better choice.

For example, suppose that you have two RAM blocks called A and B, both with three-state outputs. The block outputs are tied together and connected to a pullup resistor, as shown in [Figure 6-2](#). If the enabling logic is working properly, no more than one RAM will be enabled at any given time, thus preventing contention at the outputs.

Figure 6-2 RAM Blocks Modeled As Empty Boxes



If you declare blocks A and B to be black boxes, their outputs will be unknown (X), resulting in a contention condition that could prevent pattern generation for logic downstream from the outputs. However, if you are sure that both block A and block B will be disabled during test, you can declare these two blocks to be empty boxes. In that case, their outputs will be Z, and the pullup will pull the output node to 1 for ATPG.

Be careful when you use an empty box declaration. The pattern generator cannot determine whether the outputs are really in the Z state during test. If they are not really in the Z state, the generated patterns might result in contention at the empty box outputs.

You can build your own black box and empty box models if you prefer to do so. Here is an example of a model that works just like a black box declaration:

```

module BLACK (i1,i2, o1, o2, bidil, bidiz);
    input i1, i2;
    output o1, o2;
    inout bidil, bidiz;
    _TIEX (i1, i2, o1); // terminate inputs & drive
output
    _TIEX (o2);
    _TIEX (bidil);
    _TIEX (bidiz);
endmodule

```

Here is an example of a model that works just like an empty box declaration:

```

module EMPTY (i1,i2, o1, o2, bidil, bidiz);
    input i1, i2;
    output o1, o2;
    inout bidil, bidiz;
    _TIEZ (i1, i2, o1);
    _TIEZ (o2);
    _TIEZ (bidil);
    _TIEZ (bidiz);
endmodule

```

Note that an empty box is not the same as a model without any internal components or connections, such as the following example:

```

module NO_GOOD (i1,i2, o1, o2, bidil, bidiz);
    input i1, i2;
    output o1, o2;
    inout bidil, bidiz;
endmodule

```

If you use such a model, TetraMAX interprets it literally, resulting in multiple design rule violations (unconnected module inputs and undriven module outputs). The unconnected inputs are considered “unused,” so the gates that drive these inputs might be removed by the ATPG optimization algorithm, thus affecting the gate count and fault list. Each unconnected output triggers a design rule violation and is connected to a TIEZ primitive, which becomes an X on most downstream gate inputs.

To avoid these problems, create a model like one of the earlier examples, or use the set build command to declare the block to be a black box or empty box.

Behavior of RAM Black Boxes

When the behavior of your RAM black box is not what you expected, you should consider how the memory itself was modeled. The six cases described below revolve around how the memory module is or is not in the netlist, and how TetraMAX treats that memory device. Additionally, pros and cons are provided for each case.

Case 1

Netlist Contains: No module definition for memory

TetraMAX Session: Defines memory as an EMPTY BOX

In this case, because you do not have a module definition for the RAM, use the `set build -empty_box MYRAM` command to tell TetraMAX to treat the module as an empty box.

Pros: No modeling required.

Cons: If the memory has an output enable that is not held off, then this model is not accurate. TetraMAX will have a false environment where it sees no contention but there could really be contention occurring.

Case 2

Netlist Contains: No module definition for memory

TetraMAX Session: Defines memory as a BLACK BOX

In this case, because you do not have a module definition for the RAM, use the `set build -black_box MYRAM` command to instruct TetraMAX to treat the module as a black box.

Pros: No modeling required.

Cons: If multiple black box or empty box devices are connected together, then TetraMAX may not be able to determine if a pin is an input or an output. An output pin that is mistakenly considered an input means a TIEX that might have exposed a contention problem will go unnoticed.

Case 3

Netlist Contains: Null module definition for memory

TetraMAX Session: Defines memory as an EMPTY BOX

In this case, you take the memory module port definition from your simulation model and delete the behavioral or gate level description, leaving only the input/output definition list. This is known as a "null" module, because it has no gates within it. You then optionally use the `set build -empty_box MYRAM` command to explicitly document that this module is an empty box. The `set build -empty_box` command in this particular case is actually not needed, but it is good practice to record in the log file that the model is intentionally and explicitly to be an empty box. Without this, someone reviewing your work at a later time would have to know what was in the RAM ATPG model definition to know what type of model was chosen.

Pros: Modeling takes just a few minutes if you already have a simulation model.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

Cons: If the memory has an output enable that is not held off, then this model is not accurate. TetraMAX will have a false environment where it sees no contention, but there could really be contention occurring.

Here's an example null module:

```
module MYRAM (read, write, cs, oe,
              data_in, data_out, read_addr, write_addr );
    input  read, write, cs, oe;
    input  [7:0] data_in;
    input  [3:0] read_addr;
    input  [3:0] write_addr;
    output [7:0] data_out;
    // all core gates deleted to form NULL module
endmodule
```

Note:

Null module definitions generate numerous Nxx warnings about unconnected inputs. These can be eliminated by adding a TIEZ gate and connecting all input pins to this gate so that they are terminated and connecting the output to a dummy net.

Case 4

Netlist Contains: Null module definition

TetraMAX Session: Defines memory as a BLACK BOX

In this case, you create a null module as in Case 3, but you use the `set build -black_box MYRAM` command to instruct TetraMAX that the outputs of the module should be connected to TIEX drivers. The `set build` command is not optional for this case, or you would have an empty box instead of a black box.

Pros: Modeling takes just a few minutes if you already have a simulation model.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

There is no danger of creating a false environment where potential contention is masked by the model as in Cases 1, 2, or 3.

Cons: If the RAM has tristate outputs considered constantly TIEX, then an overly pessimistic environment is created. When a design has multiple RAMs whose outputs are tied together, this pessimistic model will produce contention that cannot be avoided. Depending on the contention settings chosen for ATPG pattern generation, TetraMAX may discard all the patterns produced.

Case 5

Output enable modeling

In this case, you start with a null module definition and add only enough gates to properly model the tristate output of the device. This is usually a few AND/OR gates and BUFIF gates enabled by some sort of chip select or output enable.

Pros: Modeling effort is light to medium. Most models can be created in less than half an hour with experience.

There is no ambiguity within TetraMAX as to which pins are inputs or outputs as in Case 2.

There is no danger of creating a false environment where potential contention is masked by the model as in Cases 1, 2, or 3.

There is no danger of an overly pessimistic output that introduces contention problems as in Case 4.

Cons: Although this model solves most problems, it does not let the TetraMAX generate patterns that would use the RAM to control and observe circuitry around the RAM, thereby leaving faults in the "shadow" of the RAM undetected.

The following is an example memory module with OEN modeling.

```
module MYRAM (read, write, cs, oe,
              data_in, data_out, read_addr, write_addr );
  input  read, write, cs, oe;
  input  [7:0] data_in;
  input  [3:0] read_addr;
  input  [3:0] write_addr;
  output [7:0] data_out;

  and u1 (OEN, cs, oe);           // form output enable
  buf u2 (TX, 1'bx);

  bufif1 do_0 (data_out[0], TX, OEN);
  bufif1 do_1 (data_out[1], TX, OEN);
  bufif1 do_2 (data_out[2], TX, OEN);
  bufif1 do_3 (data_out[3], TX, OEN);
  bufif1 do_4 (data_out[4], TX, OEN);
  bufif1 do_5 (data_out[5], TX, OEN);
  bufif1 do_6 (data_out[6], TX, OEN);
  bufif1 do_7 (data_out[7], TX, OEN);

endmodule
```

Case 6

Full functional modeling

In this case, you create a functional RAM model for ATPG using the limited Verilog syntax supported by TetraMAX.

Pros: Eliminates all problems of Cases 1 through 5.

Cons: Most time consuming. Can be as quick as an hour or as long as multiple days to construct, test, and verify an ATPG model for a memory.

Here's an example memory module with full functional modeling (see "[Memory Modeling](#)" on page 6-13 for additional examples):

```
//  
// --- level sensitive RAM with active high chip select, read,  
//      write, and output enable controls.  
  
module MYRAM (read, write, cs, oe,  
              data_in, data_out, read_addr, write_addr );  
    input  read, write, cs, oe;  
    input  [7:0] data_in;  
    input  [3:0] read_addr;  
    input  [3:0] write_addr;  
    output [7:0] data_out;  
    reg    [7:0] memory [0:15];  
    reg    [7:0] DO_reg, data_out;  
    event WRITE_OP;  
    and u1 (REN, cs, read);      // form read enable  
    and u2 (WEN, cs, write);    // form write enable  
    and u3 (OEN, cs, oe);       // form output enable  
    always @ (WEN or write_addr or data_in) if (WEN) begin  
        memory[write_addr] = data_in;  
        #0; ->WRITE_OP;  
    end  
    always @ (REN or read_addr or WRITE_OP)  
        if (REN) DO_reg = memory[read_addr];  
    always @ (OEN or DO_reg)  
        if (OEN) data_out = DO_reg;  
        else     data_out = 8'bZZZZZZZZ;  
endmodule
```

Troubleshooting Unexplained Behavior

Specific items you could double-check when you see unexplained behavior from your RAM block box are described next.

1. Did you follow the guidelines in the above cases in terms of how the memory module was (or was not) defined in the netlist, as well as what command was issued in TetraMAX?
2. Was the `set build -black_box` command used properly? That is, in particular, the target of this command must be the module name of the RAM, and not a particular instance of the RAM; for example,

```
set build -black_box MY_RAM1
    // MY_RAM1 is the module name
```

If you're still not sure, consider the commands

```
// report on as yet undefined modules;
// A black box showing up in the rightmost column of this
report
// indicates that the module is recognized as a black box:
report modules -undefined
```

OR

```
// report on what TetraMAX thinks are memories:
report memory -all -verbose
```

If you have properly performed steps 1 and 2 listed earlier, but are still seeing unexplained behavior, determine if your RAM has bidirectional (inout, tristate) ports. If so, then

- Determine why you've opted for a black box instead of an empty box. The black box model uses TIEX to drive outputs, whereas the empty box model uses TIEZ. When RAM or ROM devices have inout/tristate ports used as outputs, they drive "Z" (not "X") when disabled. Hence, an empty box model would be more appropriate here.
- If you determine that a black box is still desired for a RAM having inout ports used as outputs, then you have some choices to make because there is no way that TetraMAX can determine whether a particular inout should be an "in" or an "out" given only the null module declaration in the netlist:
 - Make a TetraMAX ATPG model for the black box RAM using TIEX ATPG primitives inside the model to force the inout ports to TIEX, and read this in as yet another source file (for example, MY_RAM1_BBmodel.v, which will in essence redefine the module MY_RAM1 to now have these TIEX primitives on its inout ports). The RAM inouts will now act as outputs driving out 'X' values. The TetraMAX graphical schematic viewer (GSV) will show you only the TIEXs representing the RAM at this point, not the RAM itself.
 - Make a TetraMAX ATPG mode similar to the above, but instead of placing TIEX ATPG primitives in the model, use actual tristate driver ATPG models (TSD) to drive the inout ports being used as outputs. Also, tie the TSD enable and input pins to TIEX primitives, and the result will be not only a RAM whose inout ports now drive out "X", but also a RAM that will be visible in the GSV.

For additional information, see the "Memory Modeling," "TIEX Primitive," and "TSD Primitive" topics in online Help.

Handling Duplicate Module Definitions

You can read a module definition more than once. By default, TetraMAX uses the most recently read module definition and issues an N5 rule violation warning for any subsequent module definitions that have the same name.

You can change this default behavior so that the first module defined is always kept, using the `-redefined_module` option of the `set netlist` command. Alternatively, you can choose Netlist > Set Netlist Options and use the Set Netlist dialog box or click the Netlist button on the command toolbar and use the Read Netlist/Image dialog box.

If you are certain that there are no module name conflicts, you can change the severity of rule N5 from warning to error:

```
BUILD> set rules n5 error
```

With a severity setting of error, the process stops when TetraMAX encounters the error, thus preventing redefinition of an existing module by another module with the same name.

When you use the `read netlist` command, you can use the `-master_modules` option to mark all modules defined by the file being read as “master modules.” A master module is not replaced when other modules with the same name are encountered. This mechanism can be useful for reading specific modules that are intended as module replacements, independent of the reading order. Note that a master module can be replaced by a module with the same name if the `-master_modules` switch is again used.

Memory Modeling

You can define RAM and ROM models using a simple Verilog behavioral description. Memory models can have these functions:

- Multiple read and write ports
- Common or separate address bus
- Common or separate data bus
- Edge-sensitive or level-sensitive read and write controls
- One qualifier on the write control
- One qualifier on the read control
- A read off state that can hold or return data to 0/1/X/Z
- Asynchronous set and/or reset capability

- Memory initialization files

You create a ROM by defining a RAM that has an initialization file and no write port.

Note:

Write ports cannot simultaneously be both level-sensitive and edge-sensitive. However, the read ports can be mixed edge-sensitive and level-sensitive, and can be different from the write ports.

TetraMAX uses a limited Verilog behavioral syntax to define RAM and ROM models for ATPG use. In concept, this is equivalent to defining some simple RAM/ROM functional models.

For detailed information on RAM and ROM modeling, see the online Help topic “Memory Modeling.” The topics covered in online Help include defining write ports and read ports, read off behavior, memory address range, multiple read/write ports, contention behavior, memory initialization, and memory model debugging.

A Basic Template

[Example 6-1](#) is a basic template for a 16-word by 8-bit RAM that can be applied to a ROM.

Example 6-1 Basic Memory Modeling Template

```
module MY_ATPG_RAM ( read, write, data_in, data_out,
read_addr,write_addr );
    input read, write;
    input [7:0] data_in;           // 8 bit data width
    input [3:0] read_addr;        // 16 words
    input [3:0] write_addr;       // 16 words
    output [7:0] data_out;        // 8 bit data width
    reg [7:0] data_out;          // output holding register
    reg [7:0] memory [0:15];     // memory storage

    event WRITE_OP;              // declare event for write-thru
    ...memory port definitions...
endmodule
```

It consists of a Verilog module definition in which you define

- The inputs and outputs (in any order and with any legal port name)
- The output holding register, “data_out” in this example
- The memory storage array, “memory” in this example

This basic structure changes very little from RAM to RAM. The port list may vary for more complicated RAMs or ROMs with multiple ports, but the template is essentially the same. Note that the ATPG modeling of RAMs requires that bused ports be used.

Initializing RAM and ROM Contents

Note:

If a RAM is to be initialized, you must provide the vectors that initialize it.

If your design contains ROMs, you must initialize the ROM image by loading data into it from a memory initialization file. You create a default initialization file and reference it in the ROM's module definition.

If you want to use a different memory initialization file for a specific instance, use the `read memory_file` command to refer to the new memory initialization file. In TetraMAX, ROMs and RAMs are identical in all respects except that the ROM does not have write data ports. Thus, the following discussion about ROMs also applies to RAMs.

The Memory Initialization File

ROM memory is initialized by a hexadecimal or binary ASCII file called a memory initialization file. [Example 6-2](#) shows a sample hexadecimal memory initialization file.

Example 6-2 Memory Initialization File

```
// 16x16 memory file in hex
0002
0004
0008
0010
0020
0040
0080
0100
0200
0400
0800
1000
2000
4000
8000
```

For additional examples of Memory Initialization Files, see the online Help topic “ROM Data File.”

Default Initialization

To establish the default memory initialization file, specify its file name in the module definition of the ROM. [Example 6-3](#) defines a Verilog module for a ROM that has 16 words of 16 data bits.

Example 6-3 16x16 ROM Model

```
module rom16x16 (ren, a, dout);
  parameter addrbits = 4;
  parameter addrmax  = 15;
  parameter databits = 16;
  input ren;
  input [addrbits-1:0] a;
  output [databits-1:0] dout;
  reg [databits-1:0] mymem [0:addrmax];
  reg [databits-1:0] dout ;
  initial $readmemh("rom_init.dat", mymem);
  always @ (ren) dout <= mymem[a] ;
endmodule
```

The `initial $readmemh` statement in this example indicates that the data in the `rom_init.dat` file is used to initialize the memory core `mymem`. The `$readmemh()` function is for hexadecimal data; there is a similar function, `$readmemb()`, for binary data.

Verilog defines the order in which data is loaded into the `mymem` core. This order is based on how you define the `mymem` index, as follows:

- The format `mymem [0:15]` indicates that the first data word in the file is to be loaded into address 0 and the last data word into address 15.
- The format `mymem [15:0]` indicates that the first data word in the file is to be loaded into address 15 and the last data word into address 0.

In [Example 6-3](#), the line

```
reg [databits-1:0] mymem [0:addrmax];
```

indicates that the first data word is loaded into address 0 and the last data word is loaded into the address specified by `addrmax`.

Instance-Specific Initialization

If you use more than one ROM instance in your design, you might not want to initialize all the ROMs from the same memory initialization file.

For each specific ROM instance, you can override the memory initialization file specification in the module definition using the Read Memory File dialog box, or you can enter the `read memory_file` command at the command line.

Using the Read Memory File Dialog Box. To use the Read Memory File dialog box to override the memory initialization file specification in the module definition for a specific ROM instance, follow these steps:

1. From the menu bar, choose the Primitives > Read Memory File. The Read Memory File dialog box appears.

2. Enter the instance and then enter or browse to the memory initialization file.

For more information about the controls in this dialog box, see the online Help for the `read memory_file` command.

3. Click OK.

Using the `read memory_file` Command. You can also override the memory initialization file specification in the module definition using the `read memory_file` command. For example:

```
DRC> read memory_file i007/u1/mem/rom1/rom_core i007.d3 -hex
```

For more information about the controls in this dialog box, see the online Help for the `read memory_file` command.

The following example indicates that the instance `/TOP/BLK1/rom1/rom_core` is to be initialized using the hexadecimal file `U1_ROM1.dat`.

```
DRC> read memory /BLK1/rom1/rom_core U1_ROM1.dat -hex
```

Note:

In responding to the `read memory_file` command, TetraMAX always loads the first word in the data file into memory address 0, the second word into address 1, and so on, regardless of how the memory index is defined in the Verilog module.

Building the ATPG Design Model

Building the ATPG design model is the process of using your design and the library modules it references to build an in-memory image of the design in a form ready for DRC analysis and ATPG.

Processes During `run build_model` Execution

During execution of the `run build_model` command, the following processes occur:

- The targeted top module for build, usually the top module, is used to form an in-memory image. Each instance in the top level is replaced by the gate-level representation of that instance; this process is repeated recursively until all hierarchical instantiations have been replaced by references to ATPG simulation primitives.
- Special ATPG simulation primitives are inserted for inputs, outputs, and bidirectional ports.

- Special ATPG simulation primitives are inserted to resolve BUS and WIRE nets. Unused gates are deleted based on the last setting of the `set build -delete_unused_gates` command.
- Each primitive is assigned a unique ID number.
- Some BUF devices are inserted at top-level ports that have direct connections to sequential devices. No fault sites are added by these buffers.
- Various design and module-level rule checks (the “B” series) are performed to determine the following:
 - Missing module definitions
 - Floating nets internal to modules
 - Module ports defined as bidirectional with no internal drivers (These could have been input ports.)
 - Module ports defined as outputs with no internal drivers (These possibly should have been inputs.)
 - Module input ports that are not connected to any gates within the module (These might be extraneous ports.)
 - Instances that have undriven input pins (These might be floating-gate inputs.)
- TIE0, TIE1, TIEZ, and TIEX primitives are inserted into the design where appropriate as a result of determining floating inputs or pins tied to a constant logic level.
- Statistics on the number of ATPG simulation primitives as well as the types of ATPG primitives are collected.

[Example 6-4](#) presents an example transcript of the `run build_model` command.

Example 6-4 Transcript of run build_model Command Output

```
BUILD> run build_model asic_top
-----
Begin build model for topcut = asic_top ...
-----
Warning: Rule B7 (undriven module output pin) failed 178 times.
Warning: Rule B8 (unconnected module input pin) failed 923 times.
Warning: Rule B10 (unconnected module internal net) failed 32
times.
Warning: Rule B13 (undriven instance pin) failed 2 times.
End build model: #primitives=101071, CPU_time=3.00 sec,
Memory=34529279
-----
```

Controlling the Build Process

There are several parameters you can control in the build process. To set these parameters, you can either use the Set Build dialog box or you can enter the `set build` command at the command line.

Using the Set Build Dialog Box

To use the Set Build dialog box to enter parameters to control the build process, follow these steps:

1. Click the Build button in the command toolbar at the top of the TetraMAX main window. Then, in the Run Build Model dialog box, click the Set Build Options button. The Set Build dialog box appears.
2. Enter your build, optimization, and model options. For descriptions of these controls, see the online Help for the `set build` command.
3. Click OK.

Using the `set build` Command

You can also enter parameters to control the build process using the `set build` command. For example:

```
DRC> set build -hierarchy . -nodelete_unused_gates
```

For the complete syntax and option descriptions, see online Help for the `set build` command.

ATPG-Specific Learning Processes

Immediately following the build model processing, some ATPG-specific learning processes begin, so TetraMAX ATPG can determine information useful for performing simulation and test generation.

The learning process performs the following analyses:

- Identifies feedback paths
- Orders gates and feedback networks by rank
- Identifies easiest-to-control input and easiest-to-observe fanout for all gates
- Identifies equivalence relationships between gates
- Identifies the potential functional behavior of circuit fragments

- Identifies tied value gates
- Identifies fault blockages that result from tied gates
- Identifies tied gates and blockages that result from gates whose inputs come from a common or equivalent source
- Identifies equivalent DFF and DLAT devices; that is, those with identical inputs
- Identifies implication relationships between gates

You can control the ATPG learning algorithms by using the Run Build Model dialog box, or you can use the `set learning` command at the command line.

Using the Run Build Model Dialog Box

To use the Run Build Model dialog box to control the learning algorithms, follow these steps:

1. Click the Build button in the command toolbar at the top of the TetraMAX main window. The Run Build Model dialog box appears.
2. Enter your choices in the Set Learning section. For descriptions of these controls, see online Help for the `set learning` command.
3. Click OK.

Using the set learning Command

You can also enter options to control the learning algorithms using the `set learning` command. For example:

```
BUILD> set learning -atpg_equivalence
```

For the complete syntax and option descriptions, see online Help for the `set learning` command.

Binary Image Files

A binary image file (or image file) is a data file that contains design data in an efficient and proprietary format for reading into TetraMAX. It contains a flattened version of the design along with TetraMAX settings.

You can create an image file by using a `write image` command and then read it by using a `read image` command. Image files can be created in two ways, depending on what mode is active when the `write image` command is issued:

- When the DRC mode is active, only the build data is stored in the image file.

- When the Test mode is active, both build and DRC data is stored in the image file.

After reading the image, TetraMAX will be in the mode in which the image was created (DRC or Test). Note that the image file does not create an identical session as when the image file was created. Some settings and data are not in the image file. Settings that are stored can be found by issuing a `report settings -all -command_report` command. Certain design data, such as net names and intermediate levels of hierarchy, are not stored. Thus, TetraMAX can only operate in primitive view and not design view.

Image files provide a number of benefits. They can be used to simplify file management, because all netlist, library, and SPF details are stored in a single file. This is beneficial for archiving and simpler data transfer between users. When TetraMAX is reading the image, there is no need to go through the build and, possibly, DRC phases again. This results in time savings on large designs. Image files can also provide intellectual property protection.

An image file can be made secure by using a combination of `set commands` and `write image` commands as described later in this section. When read, a secure image file allows only a restricted set of commands chosen by the image creator. The allowed commands are stored in the encrypted image file. Moreover, you can control whether schematic viewing is allowed.

When a secure image file is read, the TetraMAX session is said to be in a secure state. Only the allowed commands can be executed. If you specify a disallowed command, the tool will not be able to execute it and issues a warning message.

Note:

When using a secure image file, the following neutral commands are always allowed:
`exit`, `alias`, `unalias`, `help`, `source`, `cd`, `pwd`, `usage`.

This functionality is implemented through these commands:

```
set commands [-secure <command> | -all>] [-nosecure  

<command> | -all>]  
  

report commands [-secure]  
  

write image <file_name> [-password <string>]  

[-schematic_view]  
  

read image <file_name> [-password <string>]
```

See Online Help for descriptions of the commands.

TetraMAX can also obfuscate instance, net and module names when creating a garbled image. This provides an additional level of security by hiding design context. Instance names are changed to be of the format `u###`, net names become `n###` and module names become `m###`, where "###" is an integer number of any length. The original names are not

stored in the image file. The `-garble` option of the `write image` command causes the names to be modified before the image is written. Then, this garbled and secure image can be sent to a third party with controlled data access. The garbled instance and net names can be translated back to the original names by using the `-ungarble` option of the `report nets` command and `report instances` command by the image creator, if the original design database or an ungarbled image file is accessible

Flow Example

The following command sequences illustrate the flow in creating a nonsecure image file:

```
read netlist top.v
read netlist my_lib.v -library
run build my_chip
run drc my_chip.spf
write image my_chip_post_drc.img -violations
-replace
```

To read the image during a subsequent run, enter

```
read image my_chip_post_drc.img
```

The following command sequences illustrate the use of the secure image file feature in a diagnosis context:

```

// Flow to create the image
read netlist mynetlist.v
run build ...
run drc ...
// Commands to allow in the secure image
set commands -sec add equivalentnofaults
set commands -sec addnofaults
set commands -sec sources
set commands -sec help
set commands -sec read faults
set commands -sec readnofaults
set commands -sec removenofaults
set commands -sec report licenses
set commands -sec reportnofaults
set commands -sec report patterns
set commands -sec report version
set commands -sec setsimulation
set commands -sec run diagnosis
set commands -sec runsimulation
set commands -sec set patterns
set commands -sec set diagnosis
write image image_enc.gz -password top_secret \
    -schematic_view -replace -garble
run atpg -auto
write patterns pat.bin -format binary

// Flow to read the image
build -force
read image image_enc.gz -password top_secret
// Read binary pattern format
set pattern external pat.bin
// Create STIL/WGL patterns to be used with garbled
image
write patterns pat_garbled.wgl -format wgl -ext
write patterns pat_garbled.stil -format stil -ext
run diagnosis fail.log

```

To translate a garbled name back to the original name:

```

// read in original design database
read netlist mynetlist.v
run build ...
run drc ...
// supply garbled name as argument to get ungarbled
name in output
report instances u43259 -ungarble

```


7

Using the GSV For Review and Analysis

This chapter describes the Graphical Schematic Viewer (GSV) within the context of its use for interactive analysis and correction of test design rule checking (DRC) violations and test pattern generation problems.

This chapter contains the following sections:

- [Getting Started With the GSV](#)
- [Displaying Pin Data](#)
- [Displaying Other Design Information](#)
- [Analyzing DRC Violations](#)
- [Analyzing Buses](#)
- [Analyzing ATPG Problems](#)

A series of examples demonstrates the methodology for iteratively viewing, correcting, and reevaluating violations with the GSV.

Getting Started With the GSV

The GSV graphically displays design information in schematic form for review and analysis. It selectively displays a portion of the design related to a test design rule violation, a particular fault, or some other design-for-test (DFT) condition. You use the GSV to find out how to correct violations and debug the design.

There are several different ways to invoke the GSV, depending on the type of design information you want to show:

- Click the SHOW button in the GSV toolbar and specify the Block ID, path, trace, cell, or other design object that you want to view.
 - Click the ANALYZE button in the GSV toolbar and specify the rule violation or fault that you want to view.
 - In the transcript, right-click the error or warning message of interest, and from the pop-up menu, select Analyze Violation.
-

Starting the GSV Using the SHOW Button

To start the GSV and display a particular part of the design,

1. Click the SHOW button.

The SHOW menu appears, which lets you choose what to show: a named object, trace, scan path, and so on.

2. To display a named object, select Named.

The Show Block dialog box appears.

3. In the Block ID/PinPath Name text field, enter a primitive ID, instance, or pin path name to the object to display. (If you do not know what instance or pin names are available, enter 0; this is the primitive ID of the first primary input port to the top level.)

Note:

For information on the design's port names and hierarchy, review the list of top-level ports using the `report primitives -ports` command.

4. Click the Add button.

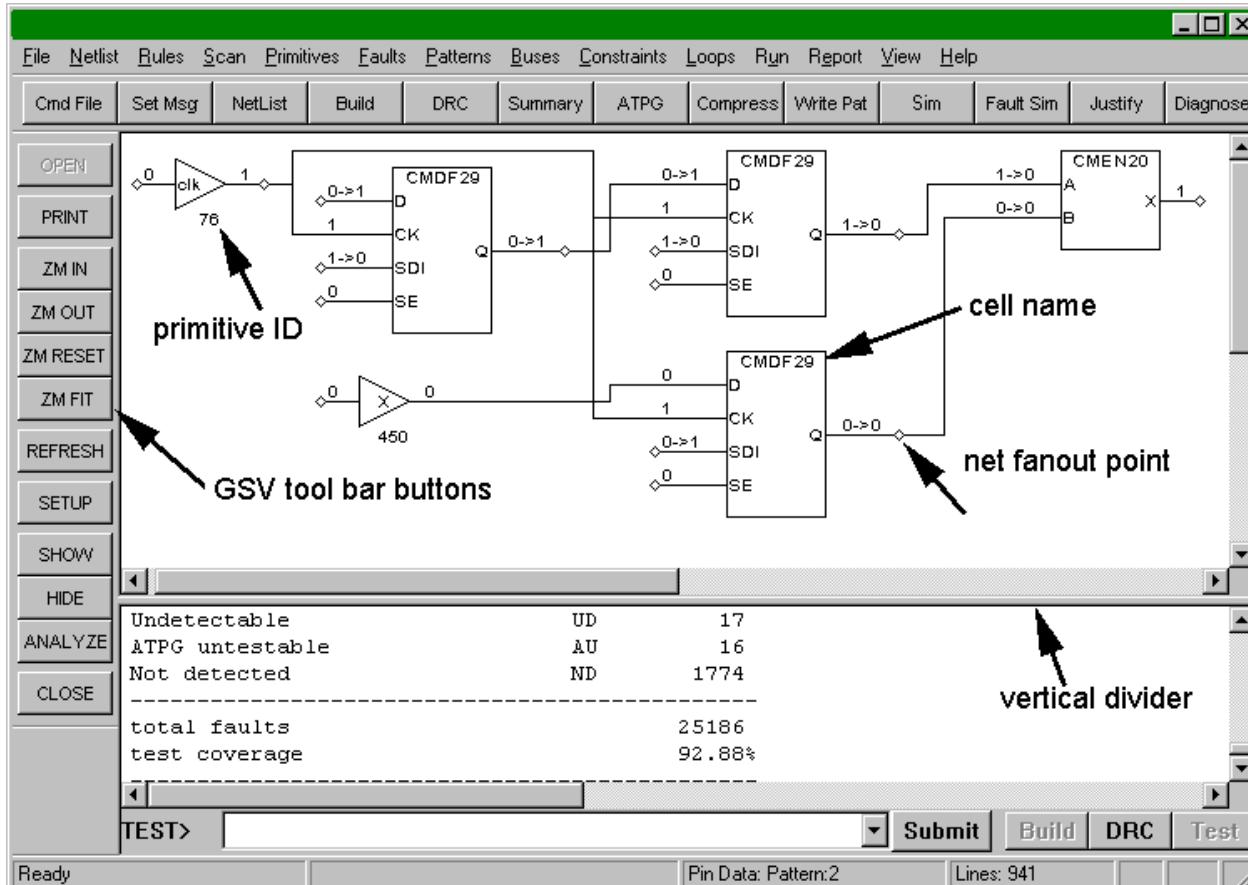
Your entry is added to the list box.

5. Repeat steps 3 and 4 to add all the parts of the design that you want to view.

6. Click OK.

Figure 7-1 shows the main TetraMAX window split by the movable divider. The top window shows a GSV schematic containing the specified objects. The bottom window contains the transcript.

Figure 7-1 GSV in the Main TetraMAX Window

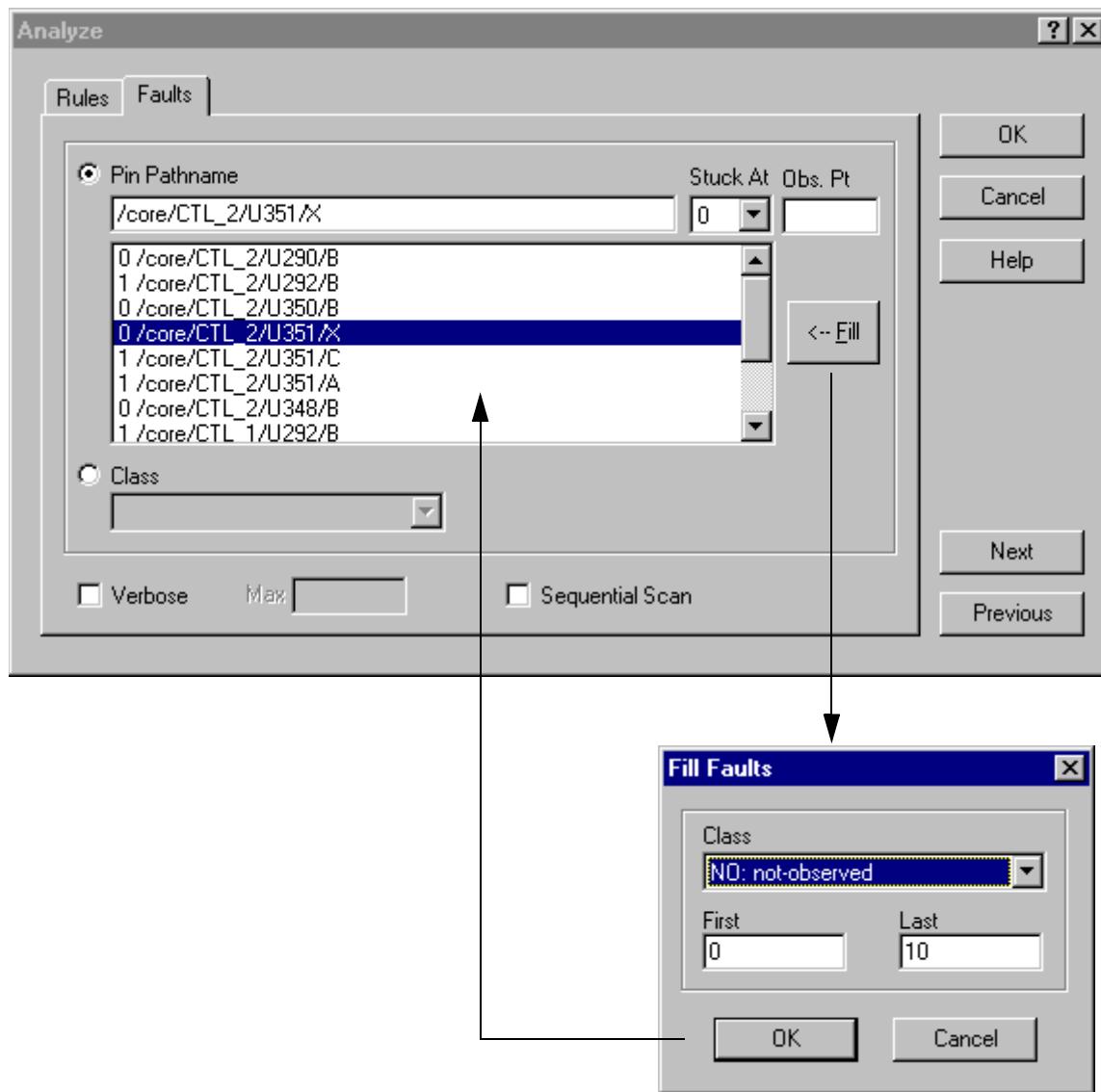


Starting the GSV From a DRC Violation or Specific Fault

You can start the GSV and view a specific DRC violation by using the Analyze dialog box. To do so,

1. Click the ANALYZE button in the GSV toolbar. The Analyze dialog box appears as shown in [Figure 7-2](#).

Figure 7-2 Analyze and Fill Faults Dialog Boxes

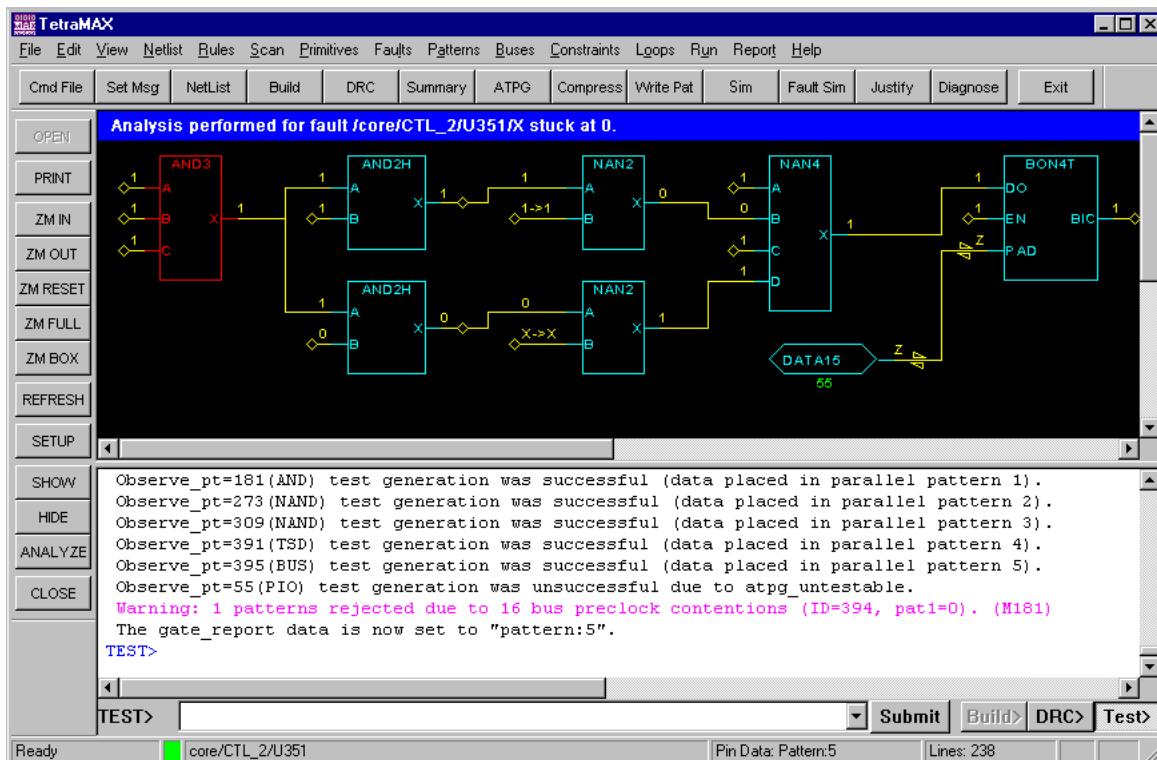


2. Click the Faults tab if it is not already active.
3. Select the Pin Pathname option if it is not already selected.
4. Click the Fill button. This opens the Fill Faults dialog box.
5. Using the Class field, select the class of faults that you would like to see listed, such as "NO: not-observed." You can also specify the range of faults within that class that are to be listed. Click OK to fill in the list box in the Analyze window, as shown in [Figure 7-2](#).

6. From the list, select the specific fault you would like displayed, such as "0 /core/CTL_2/U351/X." The fields at the top of the dialog box are filled in automatically from your selection.
7. Click OK. The Analyze dialog box closes, and the GSV displays the logic associated with the selected fault location.

Figure 7-3 shows the schematic displayed for a selected fault. The title at the top of the GSV window indicates the fault location displayed and appears on any printouts of the GSV.

Figure 7-3 GSV Window With a Fault Displayed



The command-line equivalent to the Analyze dialog box is the `analyze faults` command. This command and the resulting report appear in the transcript window, as shown in [Example 7-1](#).

Example 7-1 analyze faults Transcript of an NO Fault

```
TEST> analyze faults /core/CTL_2/U351/X -stuck 0 -display
-----
Fault analysis performed for /core/CTL_2/U351/X stuck at 0 (output
of AND
    gate 178).
Current fault classification = NO (not-observed).
-----
Connection data: to=CLKPO,MASTER from=CLOCK
Fault site control to 1 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to abort.
Observe_pt=181(AND) test generation was successful (data placed
in
    parallel pattern 1).
Observe_pt=273(NAND) test generation was successful (data placed
in
    parallel pattern 2).
Observe_pt=309(NAND) test generation was successful (data placed
in
    parallel pattern 3).
Observe_pt=391(TSD) test generation was successful (data placed
in
    parallel pattern 4).
Observe_pt=395(BUS) test generation was successful (data placed
in
    parallel pattern 5).
Observe_pt=55(PIO) test generation was unsuccessful due to
atpg_untestable.
Warning: 1 patterns rejected due to 16 bus preclock contentions
        (ID=394, pat1=0). (M181)
The gate_report data is now set to "pattern:5".
```

The details of this type of report are described later in “[Example: Analyzing a NO Fault](#)” on [page 7-45](#).

Navigating Within the GSV

To navigate within the GSV window, use the horizontal or vertical slider; the up, down, left, and right arrow keys on the keyboard; and the ZM IN, ZM OUT, ZM RESET, ZM FULL, and ZM BOX buttons.

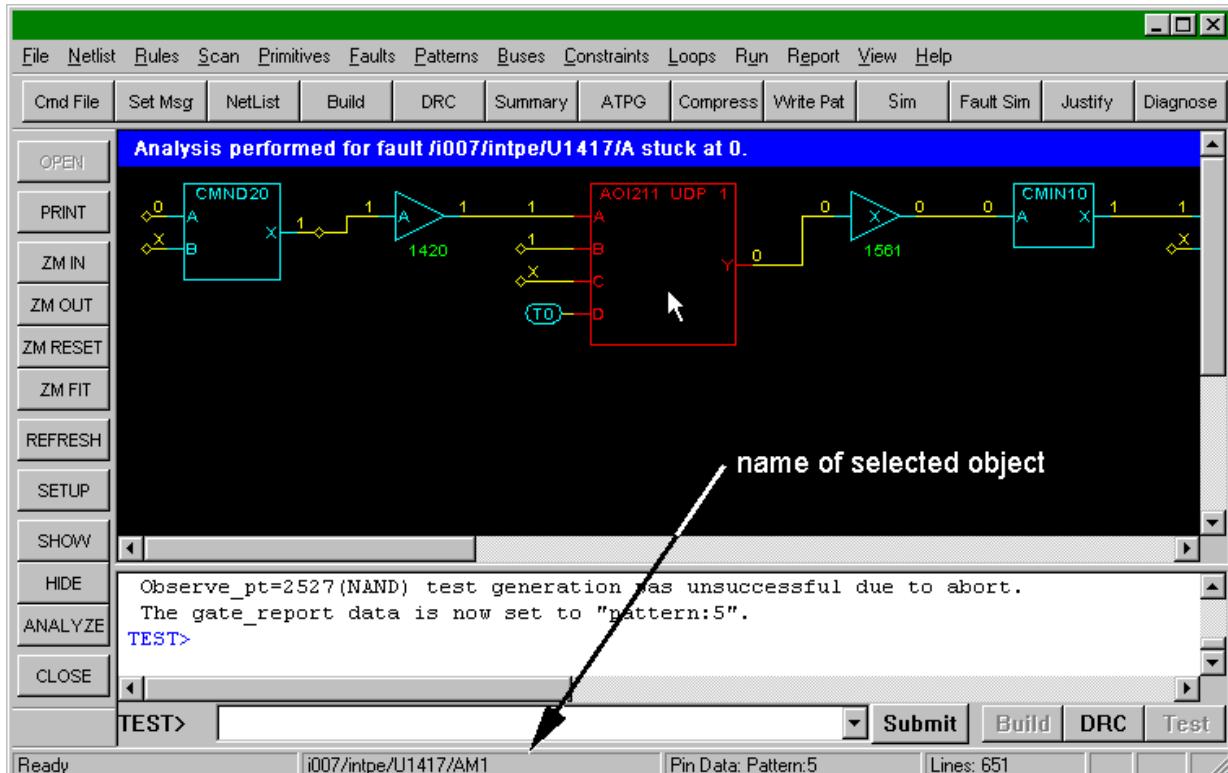
To zoom in to a specific area, click the ZM BOX button and then drag a box around the area to be magnified.

Selecting Objects in the GSV Schematic

To select an object, click it. The selected object color changes to red. The net or instance name of the selected object appears in the lower status bar, as shown in [Figure 7-4](#).

To deselect the object, click it again. To select more than one object, hold down the Shift key and click each object.

Figure 7-4 Selected Object Name



Hiding Objects in the GSV Schematic

To hide an object in the GSV,

1. Select the object by clicking it.
2. Click the HIDE button. The HIDE menu appears.
3. Choose Selected.

The selected object is hidden. Alternatively, you can choose Named to hide a named object, or All to hide all objects. You can also press the Delete key to hide selected objects.

Using the Block ID Window

You can find out the instance name, parent module, and connection data for any displayed object using the Block ID window.

To open the Block ID window,

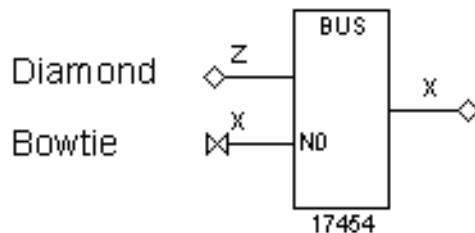
1. Click the object of interest; the object color changes to red.
 2. With the right mouse button, click the object again. A menu appears.
 3. With the left mouse button, click the Display Gate Info option in the menu. The Block ID window appears with information about the selected object.
 4. To display information for other objects, with the Block ID window still open, click each object with the right mouse button while holding down the Control key.
-

Expanding the Display From Net Connections

In the schematic display, net connections to undisplayed nets appear with one of two termination symbols, as shown in [Figure 7-5](#).

- The diamond symbol represents a unidirectional net connection.
- The bow tie symbol represents a bidirectional net connection.

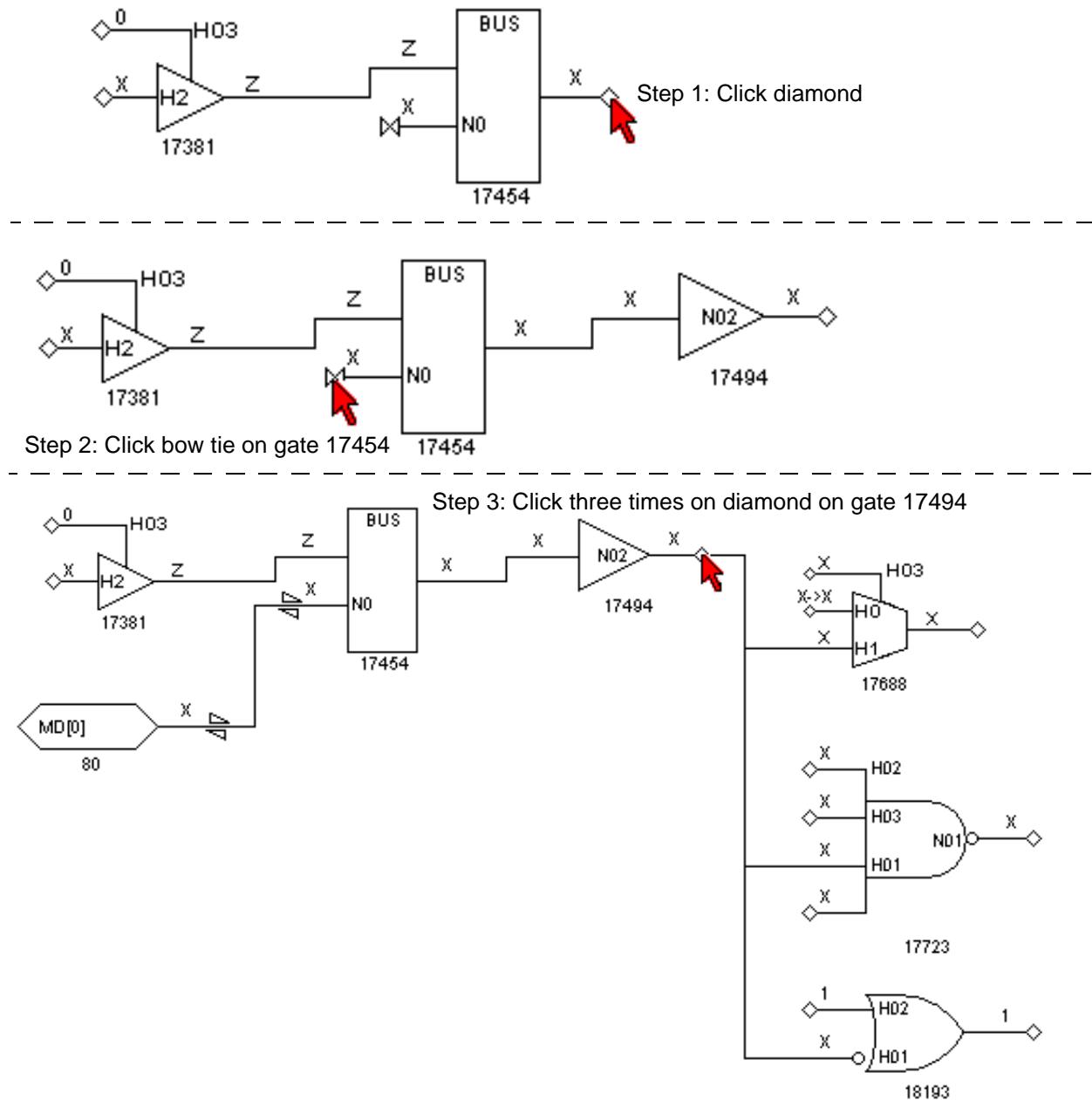
Figure 7-5 Net Expansion Symbols: Diamond and Bow Tie



To expand the display from a specific connection, click the diamond or bow tie that represents the connection of interest. The schematic expands to include the next gate or component forward or backward from the selected connection. Each click adds one component to the display. If a net has multiple additional components, you can click repeatedly and display more components until the diamond or bow tie no longer appears.

[Figure 7-6](#) shows an example of the results obtained by clicking the diamond and bow tie connection points.

Figure 7-6 Expanding Net Fanout Connections



To traverse a specific route from output pin to input pin without displaying all the fanout connections:

1. Right-click the net diamond.

2. From the pop-up menu, select Show Unconnected Fanout.

The Unconnected Fanout dialog box appears, which lists all of the paths from the net that are not currently shown in the schematic.

3. Select from the list the path you want to traverse.
4. Click OK. The GSV adds the selected path to the GSV display.

Hiding Buffers and Inverters in the GSV Schematic

When you display a design at the primitive level, you can save display space by removing the buffer and inverter gates and displaying them as double slashes and bubbles instead.

To hide buffer and inverter gates,

1. Click the SETUP button on the GSV toolbar.

The GSV Setup dialog box appears. The Hierarchy selection lets you specify whether to display primitives or design components. (For a discussion of primitives, see “[ATPG Model Primitives](#)” on page 7-11.)

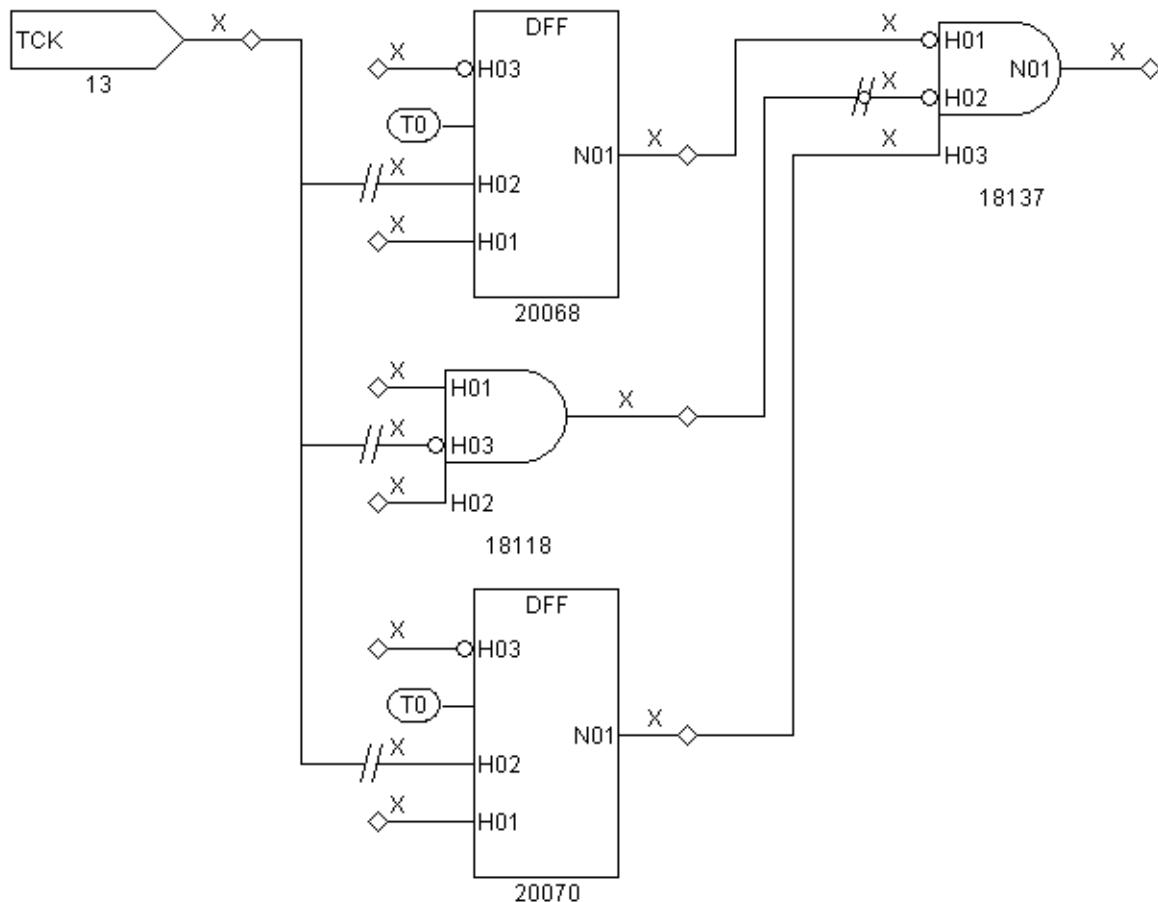
2. Select the BUF/INVs check box in the Hide section.
3. Click OK.

TetraMAX redraws the schematic without the usual buffer and inverter symbols.

As you redraw items in the schematic, the buffers and inverters are displayed as double slashes and bubbles, as shown in [Figure 7-7](#). Double slashes across the net represent a hidden gate with no logic inversion; double slashes around a bubble represent a hidden gate with logic inversion.

When you look at schematics that contain hidden gates, be aware of any hidden gates that invert logic.

Figure 7-7 Schematic With Buffers and Inverters Hidden



ATPG Model Primitives

This section describes the set of TetraMAX primitives that are used in GSV displays when Primitive is selected in the GSV Setup dialog box. (The case where Design is selected is described in “[Displaying Symbols in Primitive or Design View](#)” on page 7-17.)

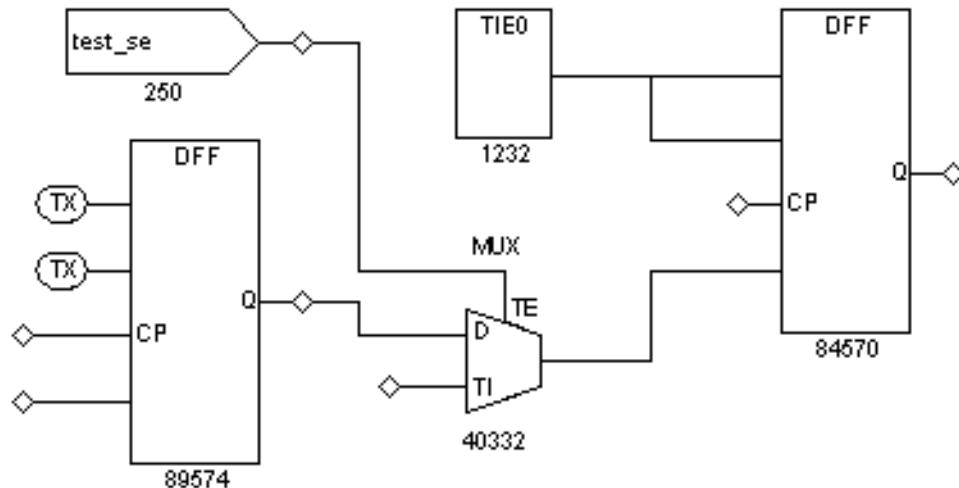
Tied Pins

A pin can be tied to 0, 1, X, or Z and can be represented in one of the following two ways:

- By an oval containing the label T0, T1, TX, or TZ connected to the pin. For example, in [Figure 7-8](#), the DFF on the left has two of its input pins connected to ovals labeled TX, indicating that the two pins are tied to X (unknown).

- By a separate connection to a TIE primitive. For example, in [Figure 7-8](#), the DFF on the right has two of its input pins connected to the TIE0 primitive, indicating that the two pins are tied to 0.

Figure 7-8 GSV Representation of Tied Pins

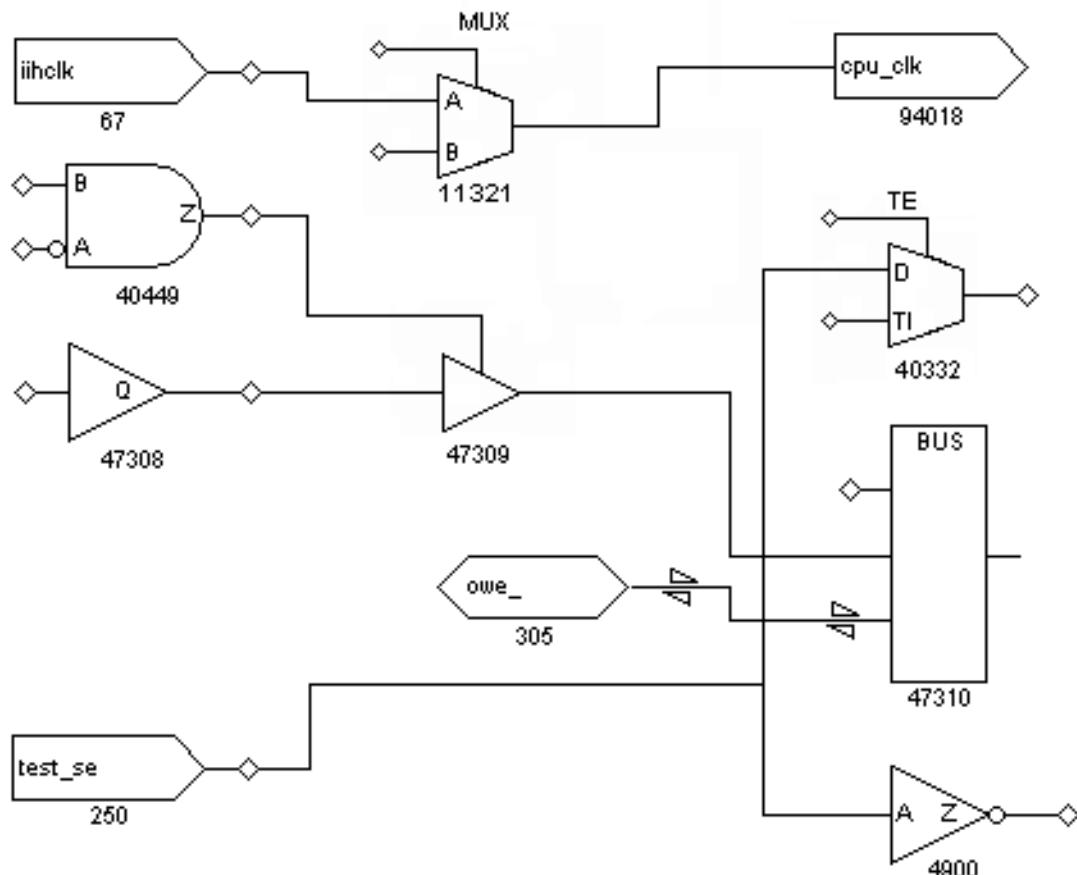


Primary Inputs and Outputs

Primary (top-level) inputs and outputs are identified as follows:

- Primary inputs are identified with the symbol shown for gates 67 and 250 in [Figure 7-9](#). Primary input ports always appear at the left of the schematic, and the symbol contains the port label (for example, `iihclk` and `test_se`).
- Primary outputs are identified with the symbol shown for gate 94018 in [Figure 7-9](#). Primary outputs always appear at the right of the schematic, and the symbol contains the port label (for example, `cpu_clk`).
- Primary bidirectional ports are identified with the symbol shown for gate 305 in [Figure 7-9](#). Primary bidirectional ports can appear anywhere in the schematic, and the symbol contains the port label (for example, `owe_`). The two bidirectional triangular wedges on bidirectional nets distinguish them from unidirectional nets.

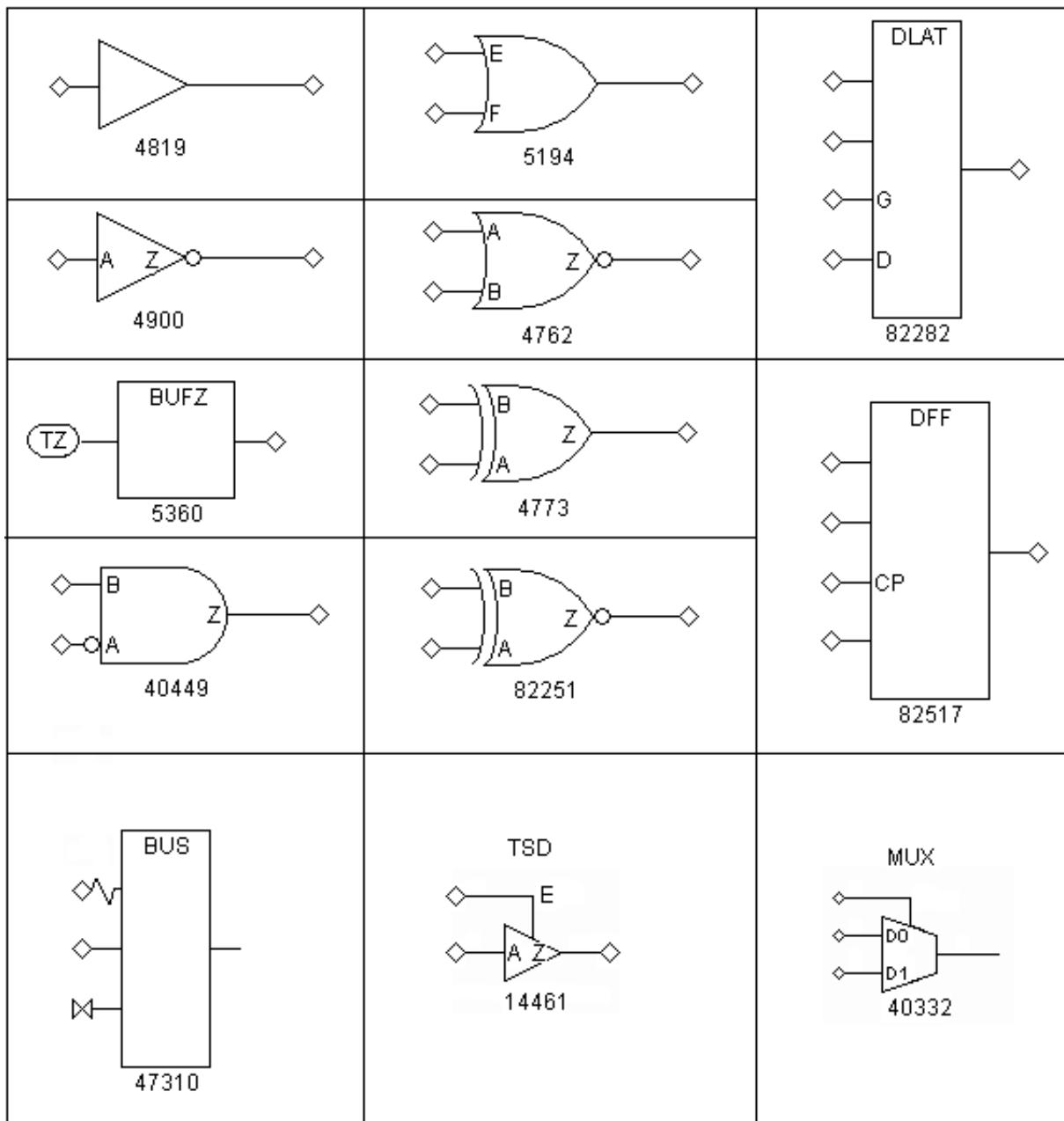
Figure 7-9 Primary I/O and Bidirectional Port Symbols



Basic Gate Primitives

Figure 7-10 shows representative symbols for many of the more commonly used TetraMAX primitives. The combinational gates AND, OR, NOR, XOR, and XNOR are shown with two inputs, but can have any number of inputs. For a complete list, refer to the online reference topic “ATPG Simulation Primitives.”

Figure 7-10 Some Basic Gate Primitives



Additional Visual Characteristics

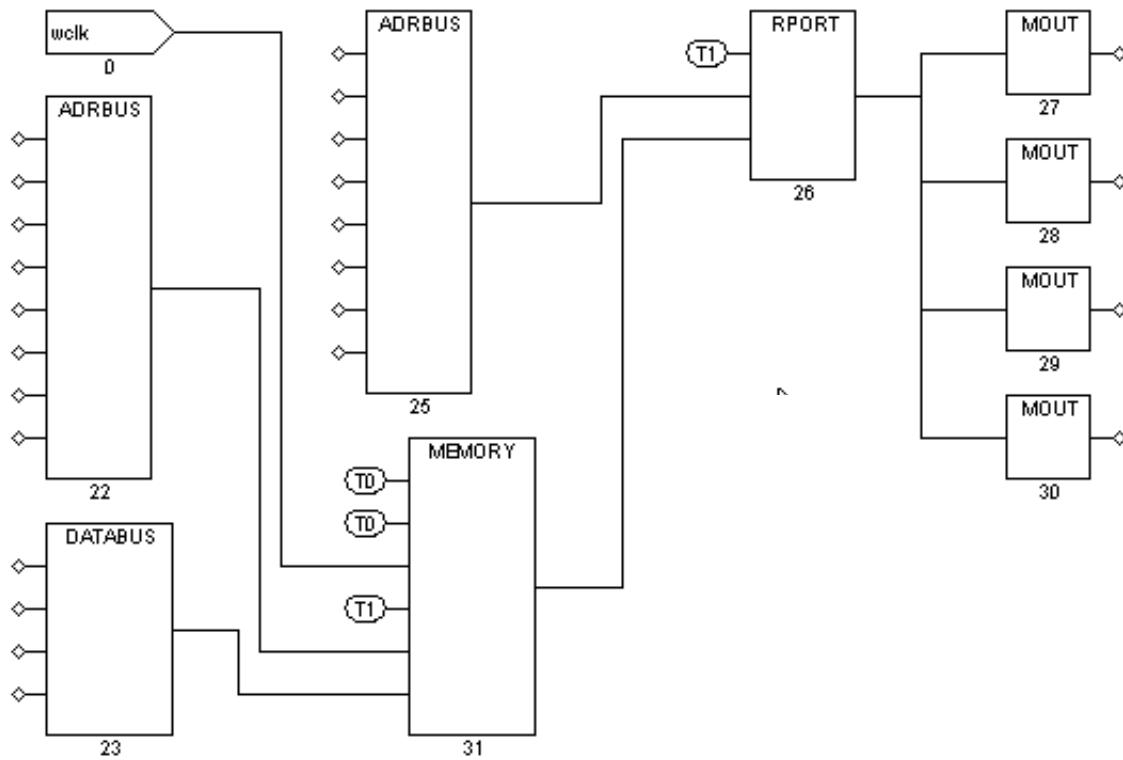
Here are some additional visual characteristics of ATPG primitives.

- Merged inverters: Inverters can be merged into the drawn symbol to make the schematic more compact. For example, in [Figure 7-10](#), the AND gate (ID 40449) shows an inversion bubble on the A input, indicating that an inverter that preceded this pin has been merged into the AND gate.
- Merged resistors: Resistors can be merged into the drawn symbol to show a gate that has a weak output drive strength. For example, in [Figure 7-10](#), the BUS gate (ID 47310) shows a resistor on one of its input pins, indicating a resistive input.
- Pin Name Labels: Some pins are labeled with pin names, and some are not. A pin name on a primitive indicates that the pin maps directly to the identical pin on the defining module in the library cell. For example, in [Figure 7-10](#) the TSD or three-state device (ID 14461) shows pins labeled A, E, and Z, meaning that those pins are all directly mapped to the pins of the defining module. However, the DFF (ID 82517) shows only pin CP labeled, meaning that CP is mapped directly to the defining module's pin called CP, but the unnamed pins are connected to other TetraMAX primitives. A single module can be represented by several TetraMAX primitives; in that case, the labels do not all appear on the same TetraMAX primitive.
- Pin order: The order of pins on the TSD, DLAT, DFF, and MUX primitives is significant. Refer to [Figure 7-10](#); pins are displayed in the following order, starting at the top:
 - For the DLAT (level-sensitive latch) primitive: asynchronous set, asynchronous reset, active-high enable, and data inputs.
 - For the DFF (edge-triggered flip-flop) primitive: asynchronous set, asynchronous reset, positive triggered clock, and data inputs.
 - When the display mode is set to Primitive, you can control the appearance of DFF/ DLAT symbols in the Environment dialog box (Edit > Environment). In the dialog box, click the Viewer tab and set the DFF/DLAT option to Mode 1, Mode 2, or Mode 3. For details, see [“Displaying Symbols in Primitive or Design View” on page 7-17](#) and the online help topic “DFF Primitive.”

RAM and ROM Primitives

For readability, instead of a single rectangle with numerous pins, a RAM or ROM block is represented as a collection of the special primitives shown in [Figure 7-11](#). The example represents a simple 256x4 RAM with a single write port and a single read port, each with its own address and control pins. Other RAMs can have multiple read and write ports. Although the RAM and ROM primitives are shown with specific bit widths (for example, ADRBUS has eight bits and DATABUS has four bits), all bit widths are supported, as required by the design.

Figure 7-11 RAM and ROM Primitives



The RAM and ROM primitives are

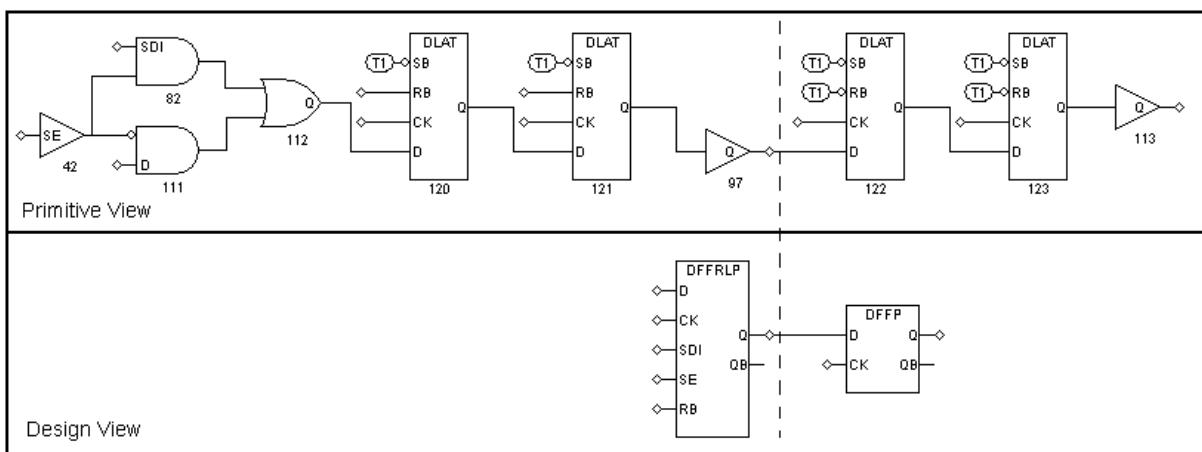
- ADRBUS: Merges the eight individual address lines at the left into the single 8-bit address bus at the right. In this example, the write port uses a separate address from the read port.
- DATABUS: Merges the four individual data write lines at the left into the single 4-bit data bus at the right.
- MEMORY: The core of the RAM or ROM; holds the stored contents. Starting from the top left, pins are as follows: an active-high set; an active-high reset (both tied to 0 in the example); a single data write port consisting of a write clock (wclk); a write enable (tied to 1); the write port address bus (8 bits); and the write port data bus (4 bits). A memory block can have multiple read and write ports; a memory without a write port represents a ROM. The module where the ROM is defined must give a path name to a memory initialization file.
- RPORT: Provides a single read port. It has a read clock or read enable pin (tied to 1 in the example), an 8-bit address bus input, and a 4-bit data bus input from the memory core. Its output is a 4-bit data bus.
- MOUT: Splits a single bit from the 4-bit RPORT data bus.

Displaying Symbols in Primitive or Design View

You can choose to display a schematic using the TetraMAX primitives (Primitive view) or using the higher-level symbols that represent the library cells (Design view).

To specify the type of view, in the GSV Setup dialog box, select either Primitive or Design in the Hierarchy box and click OK. [Figure 7-12](#) shows two different views of a design. The schematic labeled Primitive View uses TetraMAX primitives; the schematic labeled Design View uses cells in the technology library.

Figure 7-12 Comparison of Primitive and Design Views



Displaying Instance Path Names

You can display the instance path name above each instance in the schematic. To do so, select Edit > Environment in the menu bar. In the Environment dialog box, click the Viewer tab. Select the Display Instance Names check box, and then click OK.

Displaying Pin Data

You can display various types of pin data on the schematic to help you analyze DRC problems or view logic states for specific patterns, constrained and blocked values, or simulation results. For example, you might want to see the ripple effects of pins tied to 0 or 1, identify all nets that are part of a clock distribution, or see logic values on nets resulting from a STIL shift procedure.

The data values displayed are generated either by DRC or by ATPG. Data values generated by DRC correspond to the simulation values used by DRC in simulating the STIL protocol to check conformance to the test rules. Data values generated by ATPG are the actual logic values resulting from a specific ATPG pattern.

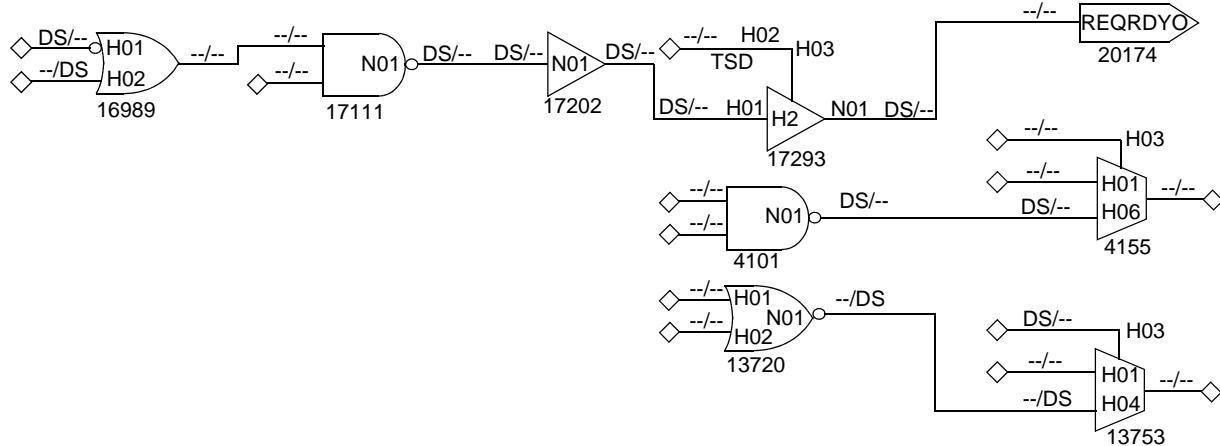
When you analyze a rule violation or a fault, TetraMAX automatically selects and displays the appropriate type of pin data. You can also manually select the type of pin data to be displayed by using the SETUP button in the GSV toolbar, or you can use the `set pindata` command at the command line.

Using the Setup Dialog Box

To display pin data on the schematic using the Setup dialog box,

1. With a schematic displayed in the GSV (for example, as shown in [Figure 7-13](#)), click the SETUP button on the GSV toolbar. The Setup dialog box opens.
2. Using the Pin Data Type pull-down menu, select the type of pin data you want to display.
3. Click OK. TetraMAX redraws the schematic using the new pin data type.

Figure 7-13 GSV Display: Pin Data Type Set to Tie Data



Using the set pindata Command

To set the pin data display mode from the command line, use the `set pindata` command. For example:

```
TEST> set pindata clock_cone CLK
```

For the complete syntax and option descriptions, see the online help for the `set pindata` command.

Pin Data Types

Table 7-1 lists each pin data type, a description of the data displayed in the GSV, and its typical use. Following the table are detailed descriptions of some of the pin data types. For the others, you can find information in the online help for the `set pindata` command.

Table 7-1 Pin Data Types

Pin data type	Data displayed	Typical Use
Clock Cone	Cone of influence and effect cones for the selected clock	Debugging clock (C) violations
Clock Off	Simulated values when all clocks are held in off state	Debugging clock (C) violations
Constrain Value	Simulated values that result from tied circuitry and ATPG constraints	Analysis of the effects of constrained signals
Debug Sim Data	Imported external simulator values	Debugging golden simulation vector mismatches
Error Data	Simulated values associated with the current DRC error	Analysis of DRC violations with severity of error
Fault Data	Current fault codes	Analysis of fault coverage (for advanced users of fault simulation)
Fault Sim Results	Good machine and faulty machine values for a selected fault	Displaying results of Basic-Scan fault simulation (for advanced users of fault simulation)
Full-Seq SCOAP Data	SCOAP controllability and observability measures using Full-Sequential ATPG	Identification of logic that is difficult to test with Full-Sequential ATPG
Full-Seq TG Data	Full-Sequential test generator logic values, showing the sequence of logic values used to achieve justification	Analysis of logic controllability using Full-Sequential ATPG
Good Sim Results	The good machine value for the selected ATPG pattern	Displaying ATPG pattern values
Load	Simulated values for the <code>load_unload</code> procedure	Debugging problems in a STIL <code>load_unload</code> macro

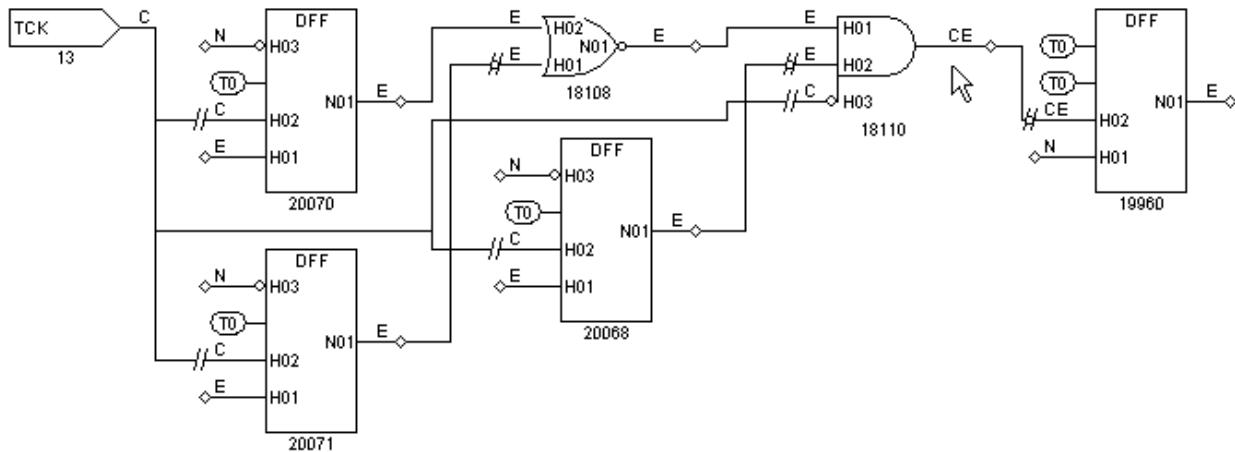
Table 7-1 Pin Data Types (Continued)

Pin data type	Data displayed	Typical Use
Master Observe	Simulated values for the <code>master_observe</code> procedure	Debugging problems in a STIL <code>master_observe</code> procedure
Pattern	Simulated values for a selected pattern	Fault analysis; displays ATPG generated values
SCOAP Data	SCOAP controllability and observability measures	Identification of logic that is difficult to test
Sequential Sim Data	Currently stored sequential simulation data	Displaying results of sequential fault simulation (for advanced users of fault simulation)
Shadow Observe	Simulated values for the <code>shadow_observe</code> procedure	Debugging problems in a STIL <code>shadow_observe</code> procedure
Shift	Simulated values for the <code>shift</code> procedure	Debugging DRC T (scan chain tracing) violations
Stability Patterns	Simulated values for the <code>load_unload</code> , <code>Shift</code> , and <code>capture</code> procedures	Analysis of classification of nonscan cells
Test Setup	Simulated values for the <code>test_setup</code> macro	Debugging problems in a STIL <code>test_setup</code> macro
Tie Data	Simulated values that result from tied circuitry	Analysis of the effects of tied signals

Displaying Clock Cone Data

To display clock cone data, select Clock Cone as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown [Figure 7-14](#). This example shows the clock cones and effect cones of the TCK clock port.

Figure 7-14 GSV Display: Pin Data Type Set to Clock Cone



Nets labeled “C” are in the clock’s clock cone. A clock cone is an area of influence that begins at a single point, spreads outward as it passes through combinational gates, and terminates at a clock input to a sequential gate.

Nets labeled “E” are in the clock’s effect cone. An effect cone begins at the output of the sequential gate affected by the clock, spreads outward as it passes through combinational gates, and also terminates at a sequential gate.

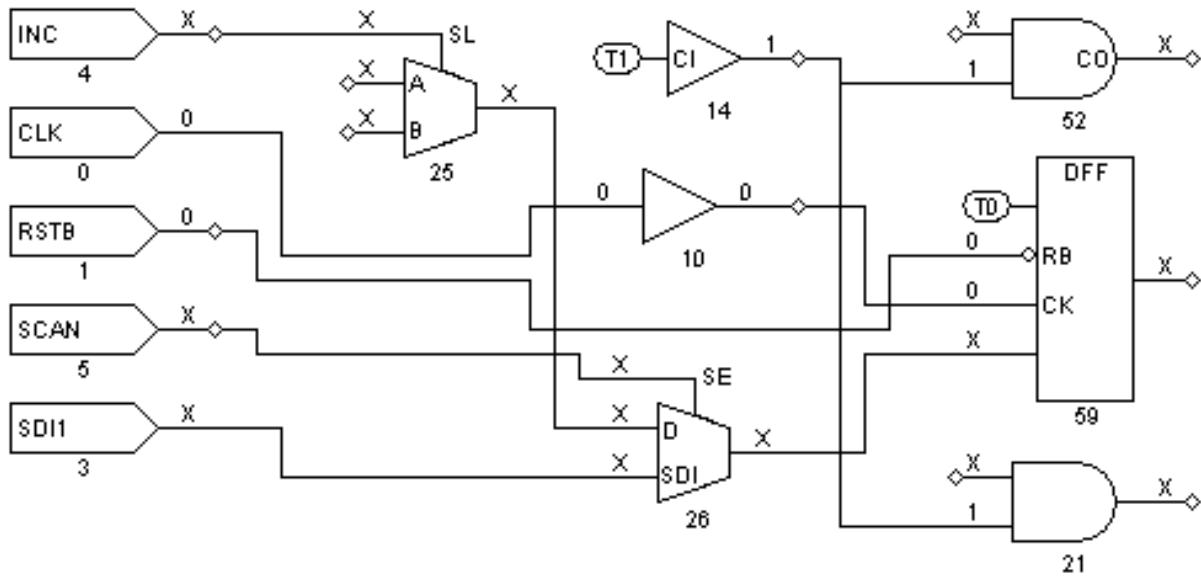
Nets labeled “CE” are in both the clock and effect cones because of a feedback path through a common gate that allows the effect cone to merge with the clock cone.

Nets labeled “N” are in neither the clock nor effect cones.

Displaying Clock Off Data

To display clock off data, select Clock Off as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 7-15](#).

Figure 7-15 GSV Display: Pin Data Type Set to Clock Off



In Figure 7-15, nets that are part of a clock distribution are shown with the logic values they have when the clocks are at their defined off states. Nets not affected by clocks are shown with Xs.

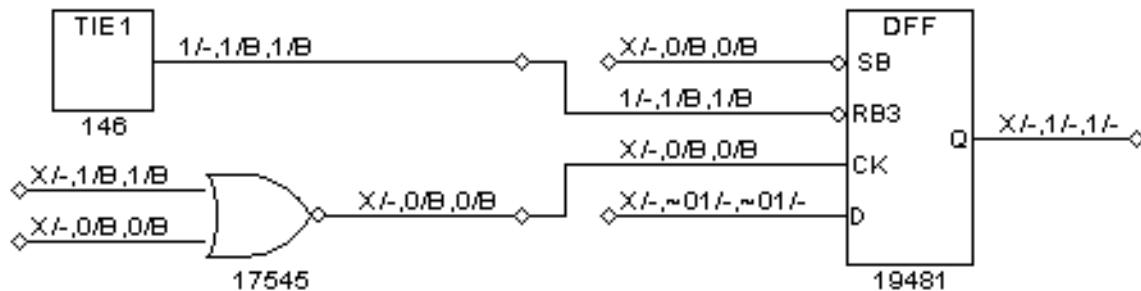
In this design, the clock ports are CLK and RSTB and their nets have values of 0. The 0 value from the CLK net is propagated to the input of gate 10, the output of gate 10, and the CK input of gate 59 (the DFF). The 0 value of RSTB is propagated to the RB input of the same DFF, gate 59. Notice that the RB pin has an inversion bubble; this is an active-low reset. When the clocks are off, there is a logic 0 value on this pin, which results in a C1 violation (unstable scan cells when clocks off).

The solution to the problem detected here is to delete the clock RSTB and redefine it with the opposite polarity. Then, execute `run drc` again and verify that this particular DRC violation is no longer reported.

Displaying Constrain Values

To display constrain values, select Constrain Value as the Pin Data type in the GSV Setup dialog box and click OK. Figure 7-16 shows a schematic displaying the constrain values.

Figure 7-16 GSV Display: Pin Data Type Set to Constrain Value



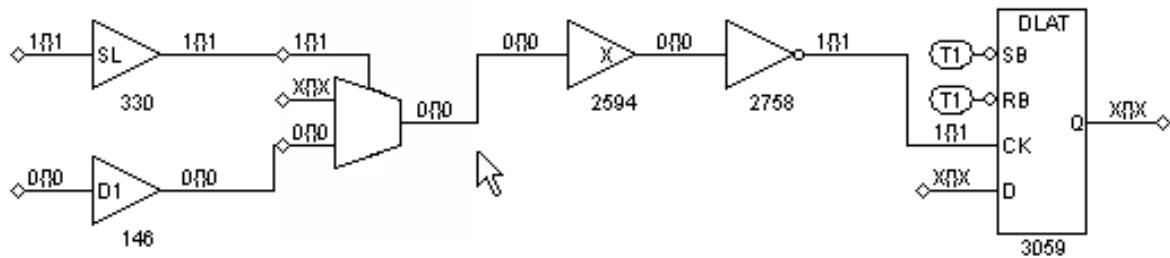
Constrain values are shown as three pairs of characters in the format T/B1, C/B2, S/B3.

- T is the pin's value that is a result of tied circuitry, if any exists. An “X” indicates that there is no value due to tied logic.
- B1 indicates whether faults are blocked on the pin because of the tied value “T.” A value of “B” indicates that the fault is blocked; a dash (-) indicates that the fault is not blocked.
- C is the constant value on the pin that results from constrained circuitry during Basic-Scan ATPG, if any. A tilde (~) preceding the character indicates values that cannot be achieved. For example, ~1 means that a value of 1 cannot be achieved, so the value is either 0 or X. An “X” indicates that there is no constant value due to constraints during Basic-Scan ATPG.
- B2 indicates whether faults are blocked on the pin because of the constrained value “C.” A value of “B” indicates that the fault is blocked; a dash (-) indicates that the fault is not blocked.
- S is similar to C, except that it is the constant value on the pin that results from constrained circuitry during sequential ATPG.
- B3 is similar to B2, except that it indicates whether faults are blocked on the pin because of the constrained value “S.”

Displaying Load Data

To display logic values during the `load_unload` procedure, select Load as the Pin Data type in the GSV Setup dialog box and click OK. [Figure 7-17](#) shows a schematic displaying the load data.

Figure 7-17 GSV Display: Pin Data Type Set to Load



The logic values are shown in the format “AAA{ }SBB.”

- AAA is one or more logic states associated with test cycles defined at the beginning of the `load_unload` procedure.

For each test cycle defined before the `Shift` procedure within the `load_unload` procedure, AAA has only one logic state if there were no events during that cycle.

For example, if three test cycles within the `load_unload` procedure precede the `Shift` procedure and an input port is forced to a 1 in the first cycle, the input port might show logic values 111{ }1. If, however, the port is pulsed and an active-low pulse is applied in the third test cycle, the port would show logic values 11101{ }1. In this case, the third test cycle is expanded into three time events and produces the third, fourth, and fifth characters, --101{ }-.

Curly braces { } represent application of the `Shift` procedure as many times as needed to shift the longest scan chain. For single-bit shift chains, the actual data simulated for the shift pattern is used rather than the { } placeholder.

- S represents the final logic value at the end of the `Shift` procedure.
- BB represents the logic values from cycles in the `load_unload` procedure that occur after the `Shift` procedure. TetraMAX determines the logic values for multibit shift chains as follows:
 - It places all constrained primary inputs at their constrained states.
 - It simulates all test cycles within the `load_unload` procedure before the `Shift` procedure, in the order that they occur.
 - It sets to X all other input ports and scan inputs that are not constrained or explicitly set.
 - It pulses the shift clock repeatedly until the circuit comes to a stable state.
 - It simulates all test cycles that are defined within the `load_unload` procedure that occur after the `Shift` procedure.

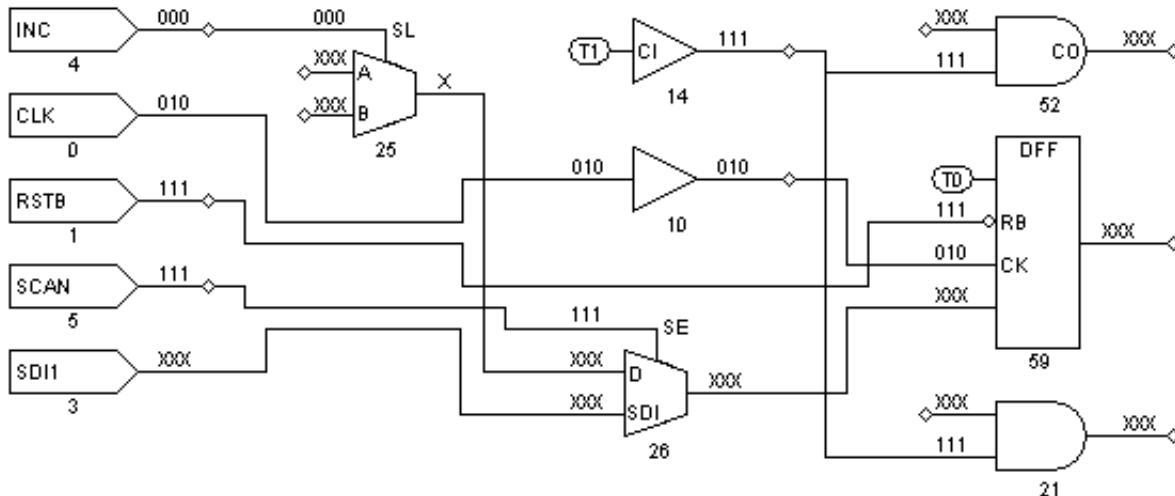
If no test cycles in the `load_unload` procedure occur after the `Shift` procedure, BB is an empty string. Otherwise, the string displayed for BB contains characters: one character for each test cycle that can be represented with a single time event, and multiple characters for any test cycles that require multiple time events. This is similar to how a single cycle in A is expanded into three characters when the port is pulsed; see the preceding discussion of AAA.

Displaying Shift Data

To display logic values during the `Shift` procedure, select Shift as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 7-18](#).

In [Figure 7-18](#), the pins show logic values that result from simulating the `Shift` procedure. The CLK port shows a simulation sequence of 010, and during the same three time periods, the RSTB pin is 111 and the SCAN pin is 111.

Figure 7-18 GSV Display: Pin Data Type Set to Shift



These are all appropriate values for the STIL `Shift` procedure shown in [Example 7-2](#).

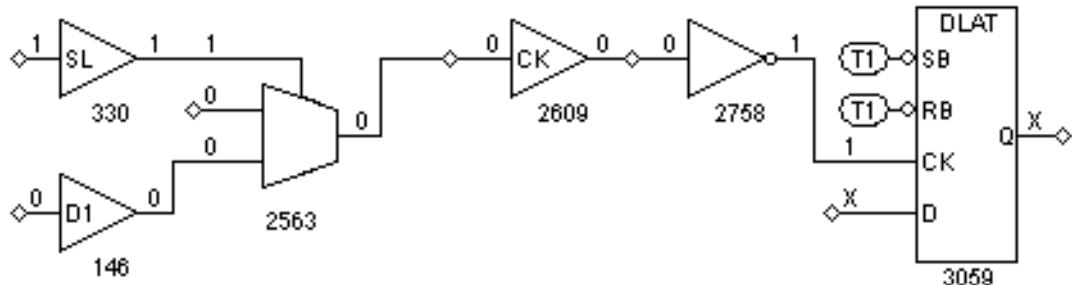
Example 7-2 STIL Shift Procedure

```
Shift
  v { _so = #;    _si = #;  INC = 0; CLK = P; RSTB = 1; SCAN = 1; }
```

Displaying Test Setup Data

To display logic values simulated during the `test_setup` macro, select Test Setup as the Pin Data type in the GSV Setup dialog box and click OK. An example schematic with Test Setup data is shown in [Figure 7-19](#).

Figure 7-19 GSV Display: Pin Data Type Set to Test Setup



By default, only a single logic value is shown, which corresponds to the final logic value at the exit of the `test_setup` macro. To show all logic values of the `test_setup` macro, you must change a DRC setting using the `set drc` command, then rerun the DRC analysis as follows:

```
TEST> drc
DRC> set drc -store_setup
DRC> run drc
```

Displaying Pattern Data

You can use the GSV to display logic values for a specific ATPG pattern within the last 32 patterns processed. The GSV can also show the values for all 32 patterns simultaneously.

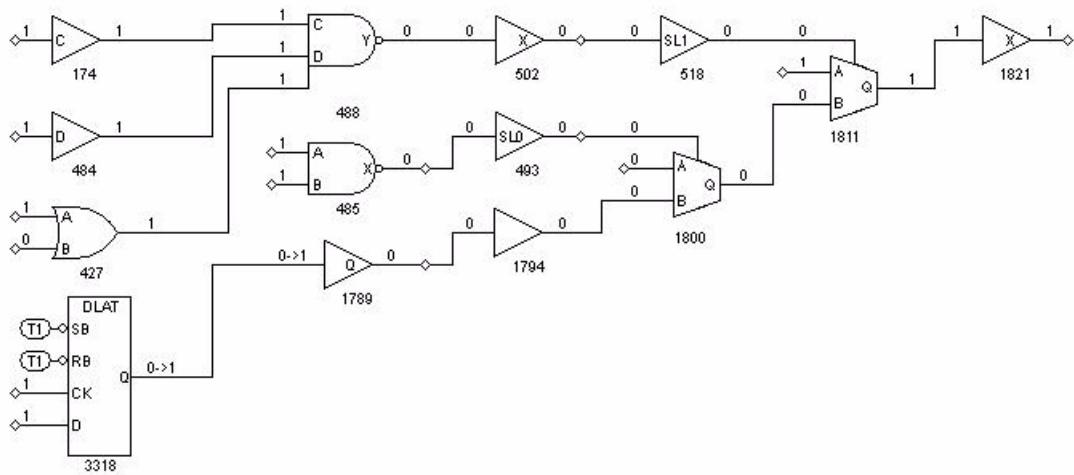
Displaying Logic Values for a Specific Pattern

To display logic values for a particular pattern,

1. Display some design gates in the schematic window.
2. Click the SETUP button in the GSV toolbar.
3. In the Setup dialog box, set the Pin Data type to Pattern.
4. In the Pattern No. text box, choose the specific pattern number to be displayed.
5. Click OK.

The logic values that result from the selected ATPG pattern are displayed on the nets of the schematic, as shown in [Figure 7-20](#).

Figure 7-20 GSV Display: Pin Data Type Set to a Pattern Number

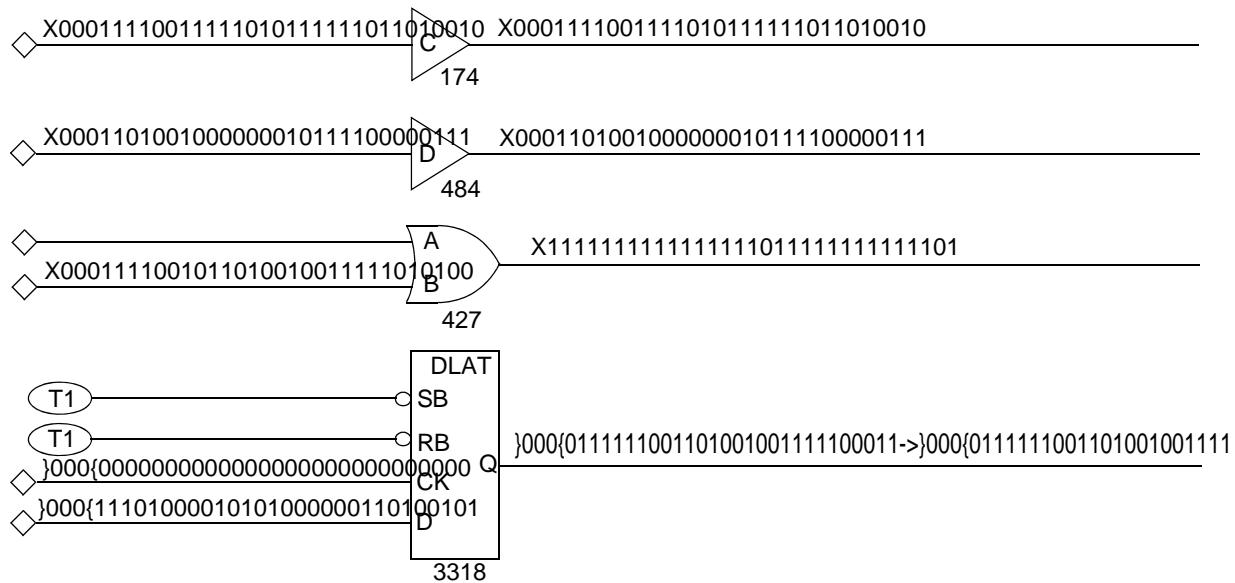


The logic values shown with an arrow, as in 0->1, show the preclock state on the left and the postclock state on the right. A logic state shown as a single character represents the preclock state. For a clock pin, a single character represents the clock-on state.

Displaying Logic Values for Multiple Patterns

To display logic values for all patterns, choose All Patterns in the GSV Setup dialog box and click OK. [Figure 7-21](#) shows all 32 patterns on the pins. You read the values from left to right. The leftmost character is the logic value resulting from pattern 0, and the rightmost character is the logic value resulting from pattern 31.

Figure 7-21 GSV Display: Pin Data Type Set to Pattern All



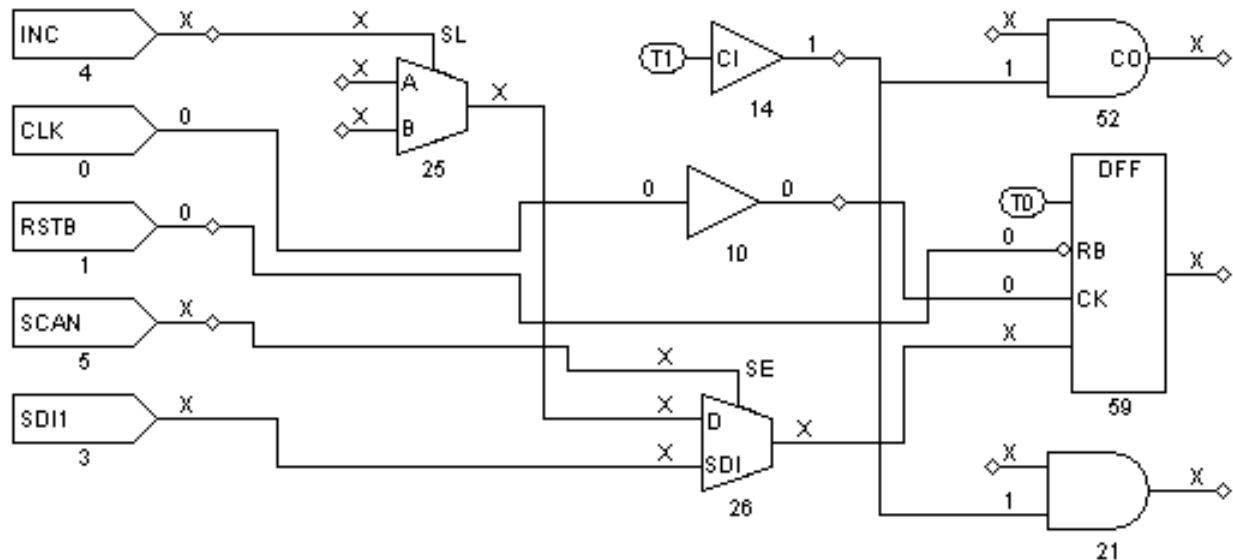
To examine a pattern that is not in the final 32 patterns processed, choose Good Sim Results in the GSV Setup dialog box and click OK.

For an additional method of viewing the logic values from a specific pattern, see “[Running Logic Simulation](#)” on page 12-21. By simulating a fault on an output port, you can display the logic values for any pattern.

Displaying Tie Data

To display tie data, select Tie Data as the Pin Data type in the GSV Setup dialog box and click OK. The schematic is redrawn as shown in [Figure 7-22](#).

Figure 7-22 GSV Display: Pin Data Displayed



In Figure 7-22, logic values are shown on nets affected by pins tied to 0 or 1. Thus, the output of gate 14 is shown with logic value 1, because its input is tied to 1. The tied value of 1 is propagated to the inputs of gates 52 and 21. Nets not affected by tied values are shown with Xs.

Displaying Other Design Information

This section contains examples of other design information you can display using the GSV.

Analyzing a Feedback Path

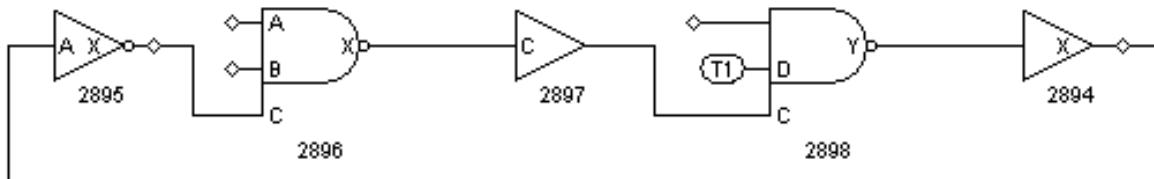
You can use the GSV to review combinational feedback loops in the design. [Example 7-3](#) shows the use of the `report feedback paths` command to obtain a summary of all combinational feedback paths and details about a specified feedback path. The five gates involved in this feedback path example are identified by their instance path names (under “id#”) and gate IDs.

Example 7-3 report feedback paths Transcript

```
TEST> report feedback paths -all
id#  #gates  #sources  sensitization_status
---- -----
0      2        1  pass
1      10       1  pass
2      10       1  pass
3      10       1  pass
4      10       1  pass
5      10       1  pass
6      5        1  pass
7      10       1  pass
8      8        1  pass
TEST> report feedback paths 6 -verbose
id#  #gates  #sources  sensitization_status
---- -----
6      5        1  pass
BUF  /amd2910/register/U70 (2894), cell=CMOA02
INV  /amd2910/register/sub_23/U11 (2895), cell=CMIN20
NAND /amd2910/register/U86 (2896), cell=CMND30
BUF  /amd2910/register/U70 (2897), cell=CMOA02
NAND /amd2910/register/U70/M1 (2898), cell=OAI211_UDP_1
```

To view a particular feedback path in the GSV, click the SHOW button, select Feedback Path, and specify the feedback path in the Show Feedback Path dialog box. [Figure 7-23](#) shows the resulting schematic display for feedback path number 6 in [Example 7-3](#).

Figure 7-23 GSV Display: a Feedback Path



Checking Controllability and Observability

You can use the Run Justification dialog box or the `run justification` command, along with the GSV's ability to display pattern data, to determine the following:

- Whether a single internal point is controllable and observable
- Whether a single internal point is controllable and observable within existing ATPG constraints
- Whether multiple points can be set to desired states simultaneously

You specify one or more internal pin states to achieve. TetraMAX attempts to find a pattern that achieves the specified logic states. If a pattern can be found, it is placed in the internal pattern buffer, and you can write it out or display it in the schematic by specifying the Pattern pin display format in the Setup dialog box.

By default, the `run justification` command uses Basic-Scan ATPG; or if you have enabled Fast-Sequential ATPG with the `set atpg -capture_cycles` command, it uses Fast-Sequential ATPG. If you want justification performed with Full-Sequential ATPG, use the `-full_sequential` option of the `run justification` command, or enable the Full Sequential option in the Run Justification dialog box.

Using the Run Justification Dialog Box

To specify pin states using the Run Justification dialog box,

1. From the menu bar, choose Run > Run Justification. The Run Justification dialog box appears.
2. In the Gate ID/Pin path name text field, type the gate ID number of a gate whose state you want to specify or the pin path name of the pin you want to specify.
3. In the Value field, use the drop-down menu to choose the value you want to specify for that gate or pin (0, 1, or Z).
4. Click Add.

The value and gate ID are added to the list in the dialog box.

5. Repeat steps 2, 3, and 4 for each gate or pin that you want to specify. When you are finished, click OK.

The Run Justification dialog box closes. TetraMAX attempts the justification and reports the results.

Using the `run justification` Command

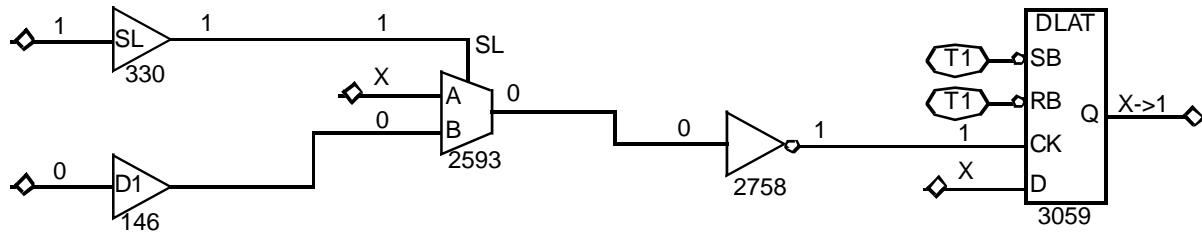
[Example 7-4](#) shows the use of the `run justification` command to request that gate ID 330 be set to 1 while gate ID 146 is simultaneously set to 0. The message indicates that the operation was successful and that the pattern is stored as pattern 0, available for pattern display.

Example 7-4 Using the run justification Command

```
TEST> run just -set 330 1 -set 146 0 -store
Successful justification: pattern values available in pattern 0.
```

[Figure 7-24](#) shows a schematic that displays the data for pattern number 0 in [Example 7-4](#). Gate 146 is at logic 0 state and gate 330 is at logic 1, as requested. Justification was successful, and TetraMAX was able to create a pattern to satisfy the list of set points.

Figure 7-24 GSV Display: Logic Values From Run Justification



Analyzing DRC Violations

This section contains a series of examples that demonstrate the process of troubleshooting a set of DRC violations. This is a summary of the process:

1. Decide on a particular rule violation to investigate.
2. From the fault list for this rule, select a specific violation.
3. Use the ANALYZE button to display the violation in the GSV; the appropriate pin data is automatically displayed.
4. Determine the cause of the violation and correct it.
5. Execute `run drc`.
6. List the violations of the same rule, verify the absence of the violation you just corrected, and examine the remaining violations. (Often, correcting one violation corrects others as well, but might also create new violations.)
7. Return to step 2 and repeat the same process until all violations of the rule have been corrected.

Scan Chain Blockage

An S1 rule violation is referred to as a scan chain blockage and is a common DRC violation. The S1 violation occurs when DRC cannot successfully trace the scan chain because a signal somewhere in the circuit is in an incorrect state and is blocking the scan chain.

Suppose you decide to troubleshoot S1 rule violations (step 1 in the troubleshooting process).

For step 2 in the troubleshooting process, suppose you select violation S1-13 from the fault list obtained through the Analyze dialog box. [Example 7-5](#) shows the transcript message for violation S1-13.

Example 7-5 S1-13 Violation Message

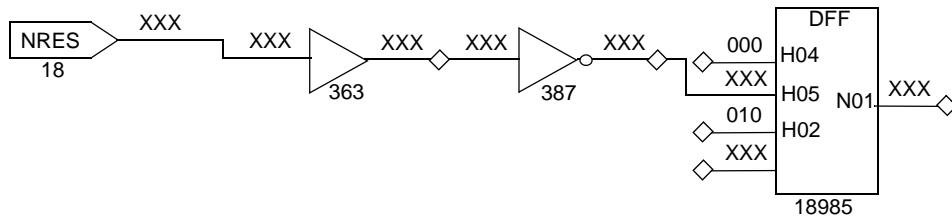
```
Error: Chain c16 blocked at DFF gate /my_asic/alu/bits/AD_DATIN/
ff_reg (18985)
after tracing 3 cells. (S1-13)
```

To view the violation (step 3 of the troubleshooting process),

1. Click the ANALYZE button on the GSV toolbar. The Analyze dialog box opens.
2. Click the Rules tab if it is not already active.
3. Type S1-13 in the Rule Violation box.
4. Click OK.

The schematic in [Figure 7-25](#) displays the violation. The pin data type has been automatically set to Shift, and the shift data is displayed. The schematic shows the gate identified in the S1-13 violation message and the gates feeding its second pin (the reset pin).

Figure 7-25 GSV Display: DRC Violation S1-13



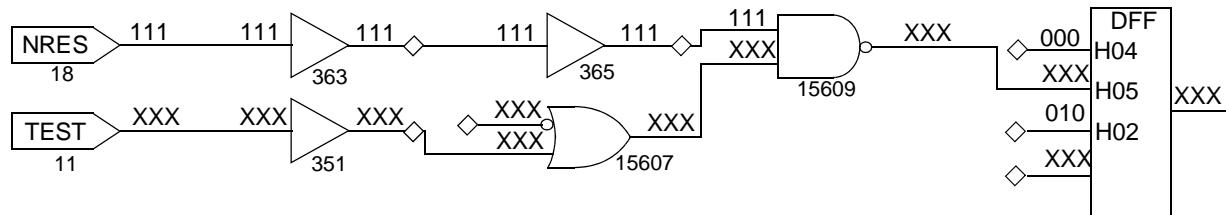
To find the signal blocking the scan chain at gate 18985 (step 4 in the troubleshooting process), proceed as follows:

1. Check the clock and asynchronous pins, starting with the DFF clock pin (H02); it has a 010 simulated state from the shift procedure, which is correct.
2. Check the DFF reset pin (H05); it has an XXX value, which is unacceptable. For a successful shift, H05 must be held inactive.
3. Trace the XXX value back from the H05 pin. The source is the primary input NRES.

NRES has an unknown value, either because it was not declared as a clock (as it should have been because of its asynchronous reset capability) or because the STIL `load_unload` procedure does not force NRES to an off state. You investigate these possibilities and correct the problem, then execute `run drc` and examine the new list of violations (step 5 in the troubleshooting process).

After correcting the NRES input problem and executing `run drc`, you select violation S1-9 from the list of remaining S1 violations. As before, you display the violation using the GSV; [Figure 7-26](#) shows the resulting schematic with Shift pin data displayed.

Figure 7-26 GSV Display: DRC Violation S1-9

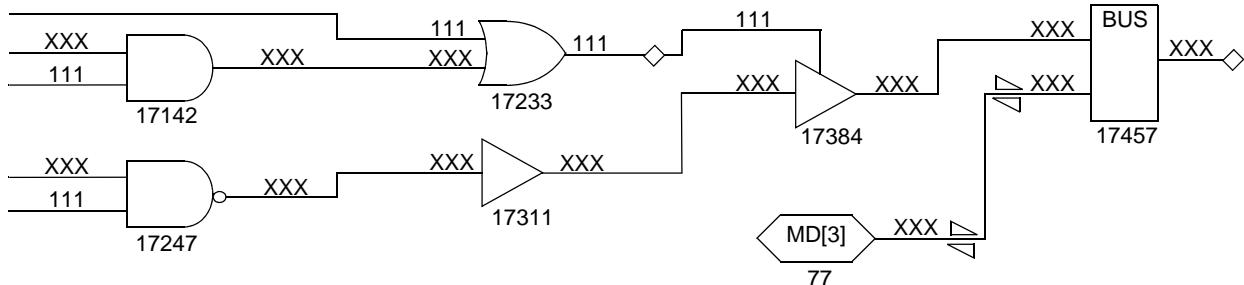


In the figure, although NRES is now correctly defined as a clock with an off state of 1, there is a problem with the reset pin, pin H05, on gate 19766 (DFF). Tracing the XXX values back as in the previous example, you find that the source is the primary input TEST. In this case, TEST was not defined as a constrained port in the STIL file.

To correct the problem, you edit the STIL file to define TEST as a primary input constrained to a logic 1, make entries in the STIL procedures for `load_unload` and `test_setup` to initialize this primary input, and execute `run drc` again.

The number of DRC violations decreases with each iteration, but there are still S1 violations. You select another violation and display it in the GSV as shown in Figure 7-27.

Figure 7-27 GSV Display: Another S1 Violation



This time, the problem is associated with the bus device, which is a gate inserted by TetraMAX during ATPG design building to resolve multidriver nets. Both potential sources for the bus inputs appear to be driving, and both have values of X. One of the sources that has an X value is the MD[3] bidirectional port; you can correct this by driving the port to a Z state. You edit the STIL file to add the declaration `MD [3] = Z` to one of the `V{..}` vectors at the start of the `load_unload` procedure (see Chapter 9, "STIL Procedure Files.")

After you make this correction, you execute `run drc` again and find no further S1 violations.

Example: A Bidirectional Contention Problem

Bidirectional contention on ports and internal pins is checked by the Z rules. In [Example 7-6](#), you use the `report rules` command to get a listing of Z rules that have failed; the report shows 108 Z4 failures and 24 Z9 failures. Suppose you decide to troubleshoot the Z4 failures. You use the `report violations` command and get a list of five violations, as shown in the example. From those, you select the Z4-1 violation to troubleshoot first.

Example 7-6 report rules Listing of Violation Messages

```
TEST> rep rules -fail

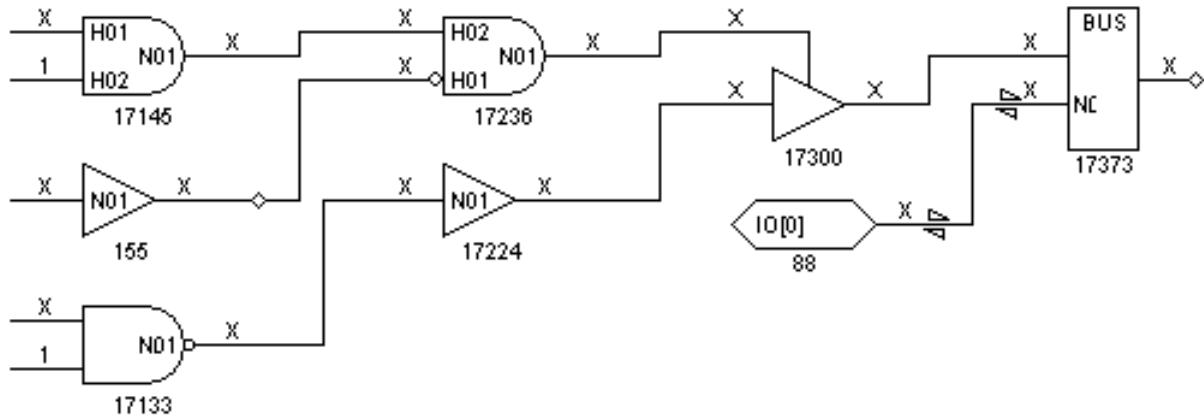
rule  severity  #fails  description
----  -----  -----
S19   warning      201  nonscan cell disturb
C2    warning      201  unstable nonscan DFF when clocks off
C17   warning      17   clock connected to PO
C19   warning      1    clock connected to non-contention-free BUS
Z4    warning     108   bus contention in test procedure
Z9    warning      24   bidi bus driver enable affected by scan cell

TEST> rep violations z4 -max 5

Warning: Bus contention on /my_asic/L030 (17373)
        occurred at time 0 of test_setup procedure. (Z4-1)
Warning: Bus contention on /my_asic/L032 (17374)
        occurred at time 0 of test_setup procedure. (Z4-2)
Warning: Bus contention on /my_asic/L034 (17375)
        occurred at time 0 of test_setup procedure. (Z4-3)
Warning: Bus contention on /my_asic//L036 (17376)
        occurred at time 0 of test_setup procedure. (Z4-4)
Warning: Bus contention on /my_asic/L038 (17377)
        occurred at time 0 of test_setup procedure. (Z4-5)
```

According to the violation error message in [Example 7-6](#), the problem is bus contention at time 0 of the `test_setup` macro. You display the violation using the GSV, as shown in [Figure 7-28](#); the schematic shows the `test_setup` data.

Figure 7-28 GSV Display: DRC Violation Z4-1

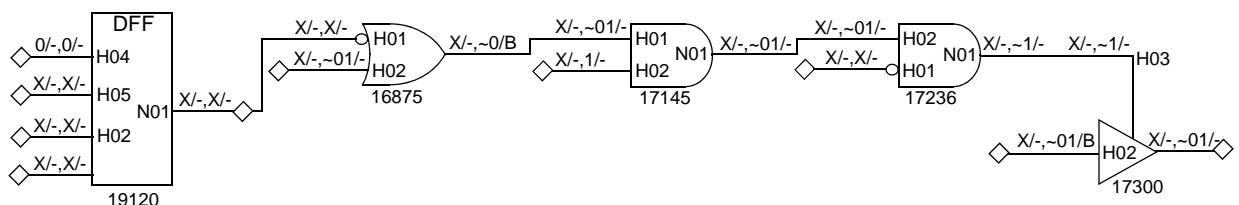


The schematic display shows a bidirectional port, `IO[0]`, which is at an `X` state. In addition, `BUS` has both inputs driven at `X`; at least one should be a `Z` value. Tracing back from `BUS`, you find a three-state driver TSD (gate 17300) whose enable and data values are both `X`. There appear to be numerous potential causes of the contention.

The violation message indicates that the violation occurred at time 0 of the `test_setup` macro. Therefore, you examine the `test_setup` macro in the STIL procedure file and find that the `IO[0]` port has not been explicitly set to the `Z` state. You edit the `test_setup` macro in the STIL file to add lines that set `IO[0]` and all the other bidirectional ports to `Z`.

After eliminating the bus contention, you execute `run drc` and find no `Z4` violations. However, `Z9` violations are still reported. You select `Z9-1` for analysis. The pin data is changed to Constrain Value, and the schematic display of the `Z9` violation appears as shown in Figure 7-29.

Figure 7-29 GSV Display: DRC Violation Z9-1



The `Z9-1` violation indicates that the control line to a three-state enable gate is affected by the contents of a scan chain cell. Thus, if a scan chain is loaded with a known value and then a capture clock or reset strobe is applied, the state of the scan cell probably changes and therefore the three-state driver control changes. Depending on the states of the other drivers on this multidriver net, the result might be a driver contention.

You can deal with this violation in one of the following ways:

- Accept the potential contention, especially if the only other driver of the net is the top-level bidirectional port. In this case, you can set the Z9 rule to ignore for future runs.
- Alter the design to provide additional controls on the three-state enable. In test mode you might block the path from the scan cell or redirect the control to some top-level port by means of a MUX.
- Adjust the contention checking to monitor bus contention both before and after clock events. TetraMAX then discards patterns that result in contention and tries new patterns in an attempt to find a pattern to detect faults without causing contention. To set bus contention checking, you enter the following command:

```
SETUP> set contention bus -capture
```

Analyzing Buses

During the DRC process, TetraMAX analyzes bus and wire gates to determine whether they can be in contention.

All bus and wire gates are analyzed to determine if two or more drivers can drive different states at the same time. Bus gates are also analyzed to determine whether they can be placed at a Z state. Drivers that have weak drive outputs are not considered for contention.

This analysis is performed before a DRC analysis of the defined STIL procedures; the information is used to prevent issuing false contention violations for the STIL procedures.

Bus Contention Status

Based on the results of the contention analysis, a bus or wire gate is assigned one of the following contention status types:

- Pass: Indicates that the bus or wire gate can never be in contention. These gates do not have to be checked further.
- Fail: Indicates that the bus or wire gate can be in contention. These gates must be monitored by TetraMAX during ATPG to avoid patterns with contention.
- Abort: Indicates that the analysis for determining a pass/fail classification was aborted. Because these gates were not identified as "pass," they must be monitored during ATPG.
- Bidi: Indicates a bus gate that has an external bidirectional connection; any internal drivers are not capable of contention. TetraMAX can avoid contention by controlling the bidirectional ports.

In addition to a contention status, bus gates undergo an additional analysis to determine whether the driver can achieve a Z state. This produces a Z-state status for each pass, fail, abort, or bidirectional gate.

The Contention Checking Report

At the conclusion of the contention check, TetraMAX displays a report like the one shown in [Example 7-7](#). The report identifies the number of bus and wire gates and the number of gates that were placed into each contention and Z-state category.

Example 7-7 DRC Report for Contention Checking

```
SETUP> run drc my_stil_file.spf
// -----
// Begin scan design rules checking...
// -----
// Begin reading test protocol file my_stil_file.spf...
// End parsing STIL file my_stil_file.spf with 0 errors.
// Test protocol file reading completed, CPU time=0.05 sec.
//
// Begin Bus/Wire contention ability checking...
// Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
#keepers=0
// Contention status: #pass=257, #bidi=31, #fail=289, #abort=2,
#not_analyzed=0
// Z-state status : #pass=160, #bidi=128, #fail=286, #abort=3,
#not_analyzed=0
// Warning: Rule Z1 (bus contention ability check) failed 289
times.
// Warning: Rule Z2 (Z-state ability check) failed 289 times.
// Bus/Wire contention ability checking completed, CPU time=7.19
sec.
```

The “Bus summary” line in the report provides the following information:

- #bus_gates: the total number of bus gates in the circuit
- #bidi: the number of bus gates with an external bidirectional port
- #weak: the number of bus gates that have only weak inputs
- #pull: the number of bus gates that have both strong and weak inputs
- #keepers: the number of bus gates connected to a bus keeper

Reduction of Aborted Bus and Wire Gates

Bus gates for which contention checking is aborted are still checked for contention during ATPG; the ATPG algorithm expends effort attempting to avoid contention. If contention checking is aborted for some gates, it is a good idea to increase the effort used to classify

as pass, fail, or bidirectional, rather than abort. You can do this by using the Analyze Buses dialog box, or you can use the `set atpg -abort_limit` and `analyze bus` commands on the command line, as shown in [Example 7-8](#).

Example 7-8 Using set atpg -abort_limit and analyze bus

```
TEST> report bus -summary
Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
#keepers=0
    Contention status: #pass=257, #bidi=31, #fail=89, #abort=200,
#not_analyzed=0
    Z-state status   : #pass=160, #bidi=128, #fail=231, #abort=58,
#not_analyzed=0
TEST> set atpg -abort 50
TEST> analyze bus -all -update
Bus Contention results: #pass=257, #bidi=31, #fail=289, #abort=0,
CPU time=0.00
TEST> analyze bus -zstate -all -update
Bus Zstate ability results: #pass=160, #bidi=128, #fail=289,
#abort=0, CPU
time=0.80
TEST> report bus -summary
Bus summary: #bus_gates=577, #bidi=128, #weak=0, #pull=0,
#keepers=0
    Contention status: #pass=257, #bidi=31, #fail=289, #abort=0,
#not_analyzed=0
    Z-state status   : #pass=160, #bidi=128, #fail=289, #abort=0,
#not_analyzed=0
Learned behavior : none
```

Using the Analyze Buses Dialog Box

To reduce the number of aborted bus and wire gates,

1. From the menu bar, choose Buses > Analyze Buses. The Analyze Buses dialog box appears.
2. In the Gate ID text field, choose -All.
3. In the Analysis Type text field, choose Prevention.
4. Enable the Update Status option.
5. Click OK.

Using the `set atpg` and `analyze bus` Commands

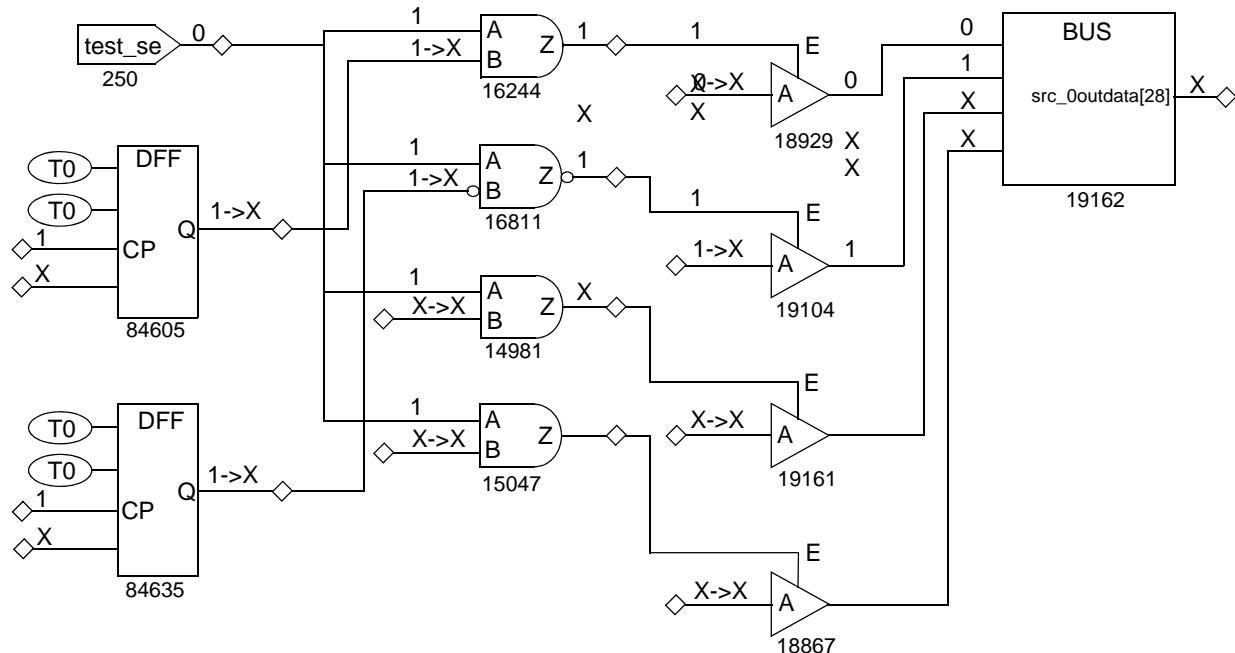
To reduce the number of aborted bus and wire gates from the command, use the `set atpg -abort_limit` and `analyze bus` commands.

Causes of Bus Contention

After attempting to eliminate bus or wire gates originally classified as aborted, you might want to review some of the bus or wire gates that were classified as failing. To review these gates, view a violation ID from the Z1 or Z2 class. The Z1 class deals with buses that can potentially be in contention, and the Z2 class deals with buses that can potentially be floating.

Figure 7-30 shows the GSV display of a Z1 violation and the logic that contributes to the three-state driver control. In this case, a pattern was found to cause contention on the bus device, gate 19162. The first two input pins of the bus device are conflicting non-Z values. The remaining two inputs are X; if a conflict is found, TetraMAX does not fill in the details of the remaining inputs. The source of the potential contention is inherent in the design; with the test_se port at 0, the TSD driver enables are controlled by the contents of the two independent DFF devices on the left. Although there might not be any problem during normal design operation, contention is almost certain to occur under the influence of random patterns.

Figure 7-30 GSV Display: DRC Violation Z1



You can deal with the Z1 and Z2 violations in one of these ways:

- Ignore the warnings, with the following consequences:
 - Contention will probably occur during pattern generation, and TetraMAX will discard those patterns that result in contention, possibly increasing the runtime.

- The resulting test coverage could be reduced. TetraMAX might be forced to discard patterns that would otherwise detect certain faults.
- Floating conditions will probably occur. Although floating conditions might have very little impact on ATPG patterns, internal Z states quickly become X states after passing through a gate, leading to an increased propagation of Xs throughout the design. These Xs eventually propagate to observe points and must be masked off, thus potentially increasing the demands on tester mask resources.
- Modify the design to attempt to eliminate potential contention or, in the case of the Z2 violation, a potential floating internal net. You accomplish this by using DFT logic that ensures that one and only one driver is on at all times, even when the logic is initialized to a random state of 1s and 0s.

Analyzing ATPG Problems

This section contains examples that demonstrate the process of analyzing ATPG problems that appear as fault sites classified as untestable. The general analysis process is as follows:

1. View the fault list by using the `report faults` command, by writing a fault list, or by opening the Analyze dialog box and clicking the Faults tab.
2. From the fault list, select a specific fault class and fault location to view.
3. Display the fault in the GSV. Use the `analyze faults` command or the Analyze dialog box.
4. Study the schematic and transcript to determine the cause of the problem.

Example: Analyzing an AN Fault

In this example, suppose you have viewed the fault list as previously described, and have selected for analysis an AN (ATPG untestable—not detected) fault identified as:

/amd2910/stack/U948/D1

[Example 7-9](#) shows a transcript of an `analyze faults` command for this fault. TetraMAX analyzes the fault, draws its location in the GSV, generates one or more patterns, and places them in the internal pattern buffer. You can examine these patterns to determine the controllability and observability issues encountered in classifying the fault.

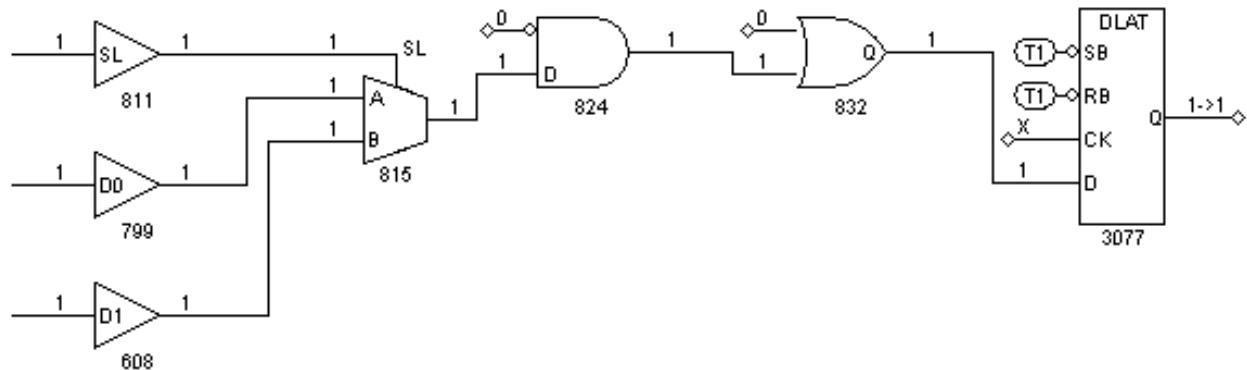
Example 7-9 Transcript of analyze faults Results for an AN Fault

```
TEST> analyze faults /amd2910/stack/U948/D1 -stuck 0 -display
-----
Fault analysis performed for /amd2910/stack/U948/D1 stuck at 0
(input 0 of
    BUF gate 608).
Current fault classification = AN (atpg_untestable-not_det).
-----
Connection data: to=TLA
Fault site control to 1 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to
atpg_untestable.
Observe_pt=815(MUX) test generation was successful (data placed
in parallel
    pattern 1).
Observe_pt=824(AND) test generation was successful (data placed
in parallel
    pattern 2).
Observe_pt=832(OR) test generation was successful (data placed
in parallel
    pattern 3).
Observe_pt=3077(DLAT) test generation was unsuccessful due to
atpg_untestable.
```

[Figure 7-31](#) shows the GSV schematic display of the untestable fault location. From the schematic and the messages in [Example 7-9](#), you conclude the following:

- The fault site was controllable; it could be set to 1. Therefore, controllability is not the reason the fault is untestable.
- Attempts to observe the fault at gates 815, 824, and 832 were successful; therefore, observability at these gates is not the reason the fault is untestable.
- Attempts to observe the fault at gate 3077 (DLAT) were unsuccessful, so observability at this gate could be the reason the fault is untestable. The DLAT is not in a scan chain and is not in transparent mode with this particular pattern (CK pin = X), so the fault cannot be propagated to an observe site.

Figure 7-31 GSV Display: an AN Fault



The source of the problem seems to be an observability blockage at the DLAT device. You could now explore whether you can place the DLAT in a transparent state using the `run justification` command, following the method described under “[Checking Controllability and Observability](#)” on page 7-30.

Example: Analyzing a UB Fault

In this example, suppose you used the Analyze dialog box to view the fault list and selected a UB (undetectable-blocked) fault to analyze.

To view the fault list and select a fault using the Analyze dialog box,

1. Click the ANALYZE button in the GSV toolbar.
The Analyze dialog box opens.
2. Click the Faults tab.
3. Click Pin Pathname if it is not already selected.
4. Click the Fill button.
5. In the Fill Faults dialog box, select “UB: undetectable-blocked” as the Class type.
6. Enter 100 in the Last field.
7. Click OK in the Fill Faults dialog box.

In the Analyze dialog box, the first 100 UB faults appear in the scrolling window under the Faults tab. Scroll through the list and select a fault to analyze.

8. Click OK.

TetraMAX analyzes the fault selected and displays in the transcript window the equivalent analyze faults command and the results of the analysis, as shown in [Example 7-10](#).

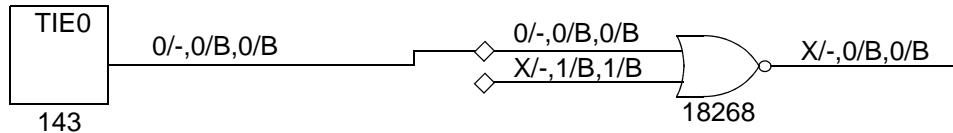
Example 7-10 Transcript of analyze faults Results for a UB Fault

```
TEST> analyze faults /JTAG_IR/U51/H02 -stuck 0 -display
-----
-
Fault analysis performed for /JTAG_IR/U51/H02 stuck at
0 \
    (input 1 of OR gate 18268).
Current fault classification = UB \
    (undetectable-blocked).

Fault is blocked from detection due to tied values.
Blockage point is gate /MAIN/JTAG_IR/U51 (18268).
Source of blockage is gate /MAIN/U354 (143).
```

[Figure 7-32](#) shows the graphical representation of the section of the design associated with the fault.

Figure 7-32 GSV Display: a UB Fault



The fault analysis message provides information similar to that in the schematic display. A stuck-at-0 fault at pin H02 of gate 18268 cannot be detected because the input to pin H01 of this OR gate comes from a tied-to-0 source.

Notice that the schematic contains the fault site as well as the gates involved with the source of the blockage. In addition, the pin data type has been set to Constrain Data and the constraint information is displayed directly on the schematic. For an interpretation of constrain values, see ["Displaying Constrain Values" on page 7-22](#).

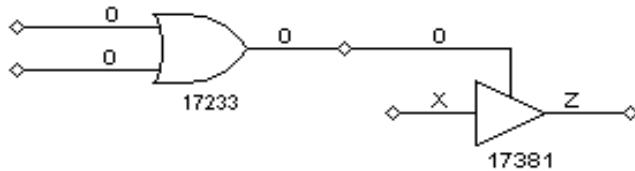
In the example in [Figure 7-32](#), TetraMAX has analyzed a stuck-at-0 fault on the H02 pin in the schematic. The transcript shows that this fault is UB, that the blockage point is gate 18268, and that the source of the blockage is gate 143.

You review the GSV display in [Figure 7-32](#) to gain some additional insight. Gate 143 is a tie-off cell that ties the H01 input of gate 18268 to 0, forcing the output of gate 18268 to a logic 1 and blocking the propagation of faults from pin H02.

Example: Analyzing a NO Fault

In this example, you selected the fault class NO (not-observed), obtained a list, and selected one of the faults for analysis, as previously described. The corresponding schematic is shown in [Figure 7-33](#).

Figure 7-33 GSV Display: a NO Fault



The fault report in [Example 7-11](#) states that the fault site is controllable but not observable. A pattern that controlled the fault site to 0 to detect a stuck-at-1 fault was placed in the internal pattern buffer as pattern 0, but was later rejected because the pattern failed bus contention checks.

Example 7-11 analyze faults Report for a NO Fault

```
TEST> analyze faults /U317/H02 -stuck 1 -display
-----
Fault analysis performed for /U317/H02 stuck at 1 (output of OR
gate 17233).
Current fault classification = NO (not-observed).
-----
Connection data: to=REGPO,MASTER,TS_ENABLE
Fault site control to 0 was successful (data placed in parallel
pattern 0).
Observe_pt=any test generation was unsuccessful due to abort.
Observe_pt=17381(TSD) test generation was unsuccessful due to
atpg_untestable.
Warning: 1 pattern rejected due to 32 bus contentions (ID=17373,
pat1=0). (M181)
```

The fault report mentions two observe points. The first, “any test generation,” was unsuccessful because an abort limit was reached. The second observe point at gate 17381 was unsuccessful because of ATPG-untestable conditions at that gate. The first observe point might succeed if you increase the abort limit and try again.

Printing Schematic to a File

To create a grayscale PostScript file, which captures the schematic displayed in the Graphical Schematic Viewer (GSV), use the following command:

```
GSV_Print -FILE <file> -BANNER <string> Y
```

You can add the GSV_Print command into the TetraMAX scripts and capture schematic output automatically. The computing host must have PostScript drivers installed (usually with lpr/lp). You can enclose the arguments in double quotes ("").

8

Using the Simulation Waveform Viewer

This chapter describes the TetraMAX Simulation Waveform Viewer (SWV). You can use the SWV to debug internal, external, and imported functional pattern mismatches by displaying the failing simulation values and TetraMAX simulated values of the test_setup procedure.

This chapter contains the following sections:

- [Getting Started With the SWV](#)
- [Supported Pin Data Types and Definitions](#)
- [Invoking the SWV](#)
- [Using the SWV Interface](#)
- [Using the SWV With the GSV](#)
- [Using the SWV Without the GSV](#)
- [SWV Inputs and Outputs](#)
- [Analyzing Violations](#)

Getting Started With the SWV

Before you start using the SWV, it is recommended that you familiarize yourself with the Graphical Schematic Viewer (GSV). For more information, see [Chapter 7, “Using the GSV For Review and Analysis.”](#)

The GSV graphically displays design information in schematic form for review and analysis. It selectively displays a portion of the design related to a test design rule violation so that you can debug a test setup, and or debug internal, external pattern mismatches. You use the GSV to find out how to correct violations and/or debug the design.

The SWV is intended to add a third level of dimension to DRC debugging. The following methods are currently used with DRC:

1. Create or parse the SPF, edit the SPF, and rerun DRC
2. Use the GSV to identify and resolve shift errors, and also to view test_setup
3. Use the SWV when you want to view large amounts of net instance data in the GSV, such as large test_setup (item 2 above) or run simulation data

It is important to note that the SWV is only a viewer; its primary purpose is to enhance what you see in the GSV.

In the GSV, you can view simulation values (or pin data values) on the nets of the design. By default, these values are 10 data bits, although this is user-configurable. Simulation values displayed in the GSV are a subset of values, followed by an ellipsis. You can change this display by changing the default setting, or by moving the data display within the GSV cone of logic. This data synchronizes with the SWV.

When the simulation string becomes more than 20 characters, the space required to display such a long string makes the GSV display impractical. In the SWV, the simulation strings do not need to be displayed in full, because you can look up the transition in the waveforms. When tracing between the GSV cone of logic, the SWV is dynamically updated with the data from the GSV. When you select and move your pointer, the SWV highlights the corresponding bit in the GSV. You can change the default display of simulation values using the following command:

```
set environment viewer -max_pindata_length <d>
```

Supported Pin Data Types and Definitions

[Figure 8-1](#) shows the TetraMAX pin data type setup menu. The SWV does not support all pin data types. It supports only test_setup, debug_sim_data, and sequential_sim_data.

Two of the pin_data types require data to be stored internally in TetraMAX.

By default, only a single logic value is shown, which corresponds to the final logic value at the exit of the test_setup macro. To show all logic values of the test_setup macro, you must change a DRC setting using the set drc command, then re-run the DRC analysis as follows:

```
TEST> drc  
DRC> set drc -store_setup  
DRC> run drc
```

The pin_data type test_setup requires set drc -store.

Sequential simulation data is typically from functional patterns. This type of data is stored in the external pattern buffer. When the simulation type in the Run Simulation dialog box is set to Full Sequential, you can select a range of patterns to be stored.

After the simulation is completed, you can display selected data from this range of patterns using the pin data type sequential_sim_data. For example:

```
TEST> set simulation -data 85 89  
TEST> run simulation -sequential
```

The pin_data type seq_sim_data requires the output of the set simulation -data command.

Figure 8-1 Setting Pin Data Type

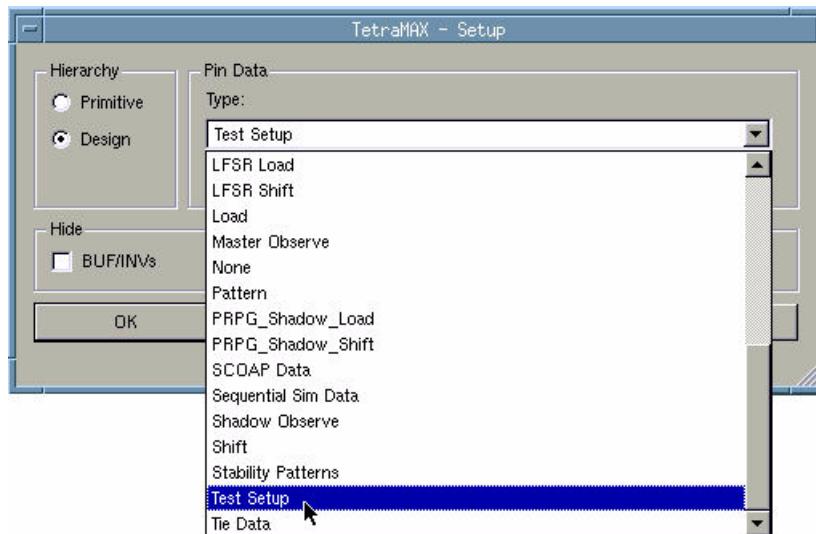


Table 8-1 lists and describes the pin data types supported by the SWV.

Table 8-1 Supported Pin Data Types

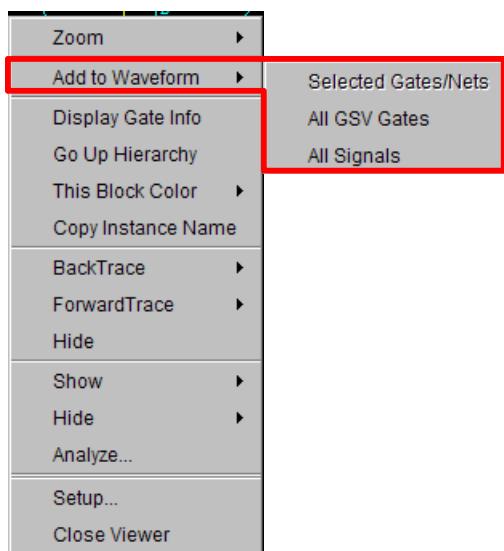
Pin Data Types	Definition
Test Setup	Displays simulated values for the test_setup macro displaying debugging problems in a STIL test_setup macro
Debug Sim Data	Displays imported external simulator values used for debugging golden simulation vector mismatches
Sequential Sim Data	Displays currently stored sequential simulation data used for displaying results of sequential fault simulation (for advanced users of fault simulation)

Invoking the SWV

You can specify commands, select buttons, or use your right mouse button to open menus that cause TetraMAX to launch the SWV either directly from the GSV or without the GSV.

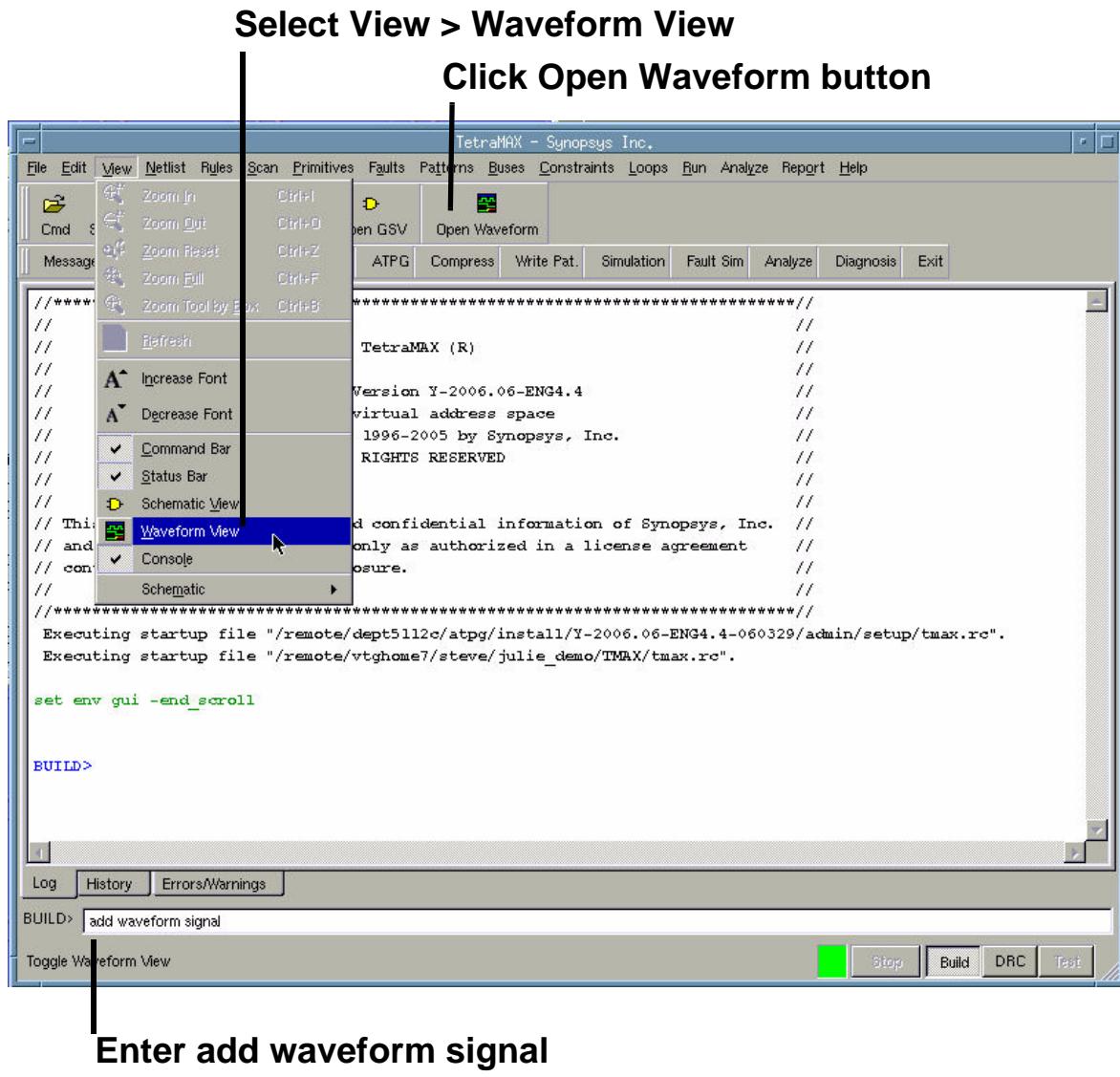
[Figure 8-2](#) shows the SWV menu that appears when you right-click after selecting the nets and/or gates. You can add signals, gates, and nets to the waveform using this menu.

Figure 8-2 Opening the SWV using your right mouse button



[Figure 8-3](#) shows the three ways to invoke the SWV from the GSV.

Figure 8-3 Three Ways to Open the SWV



Using the SWV Interface

This section describes the basic features of the SWV interface, including the following:

- Understanding the SWV Layout
- Using the Signal List Pane

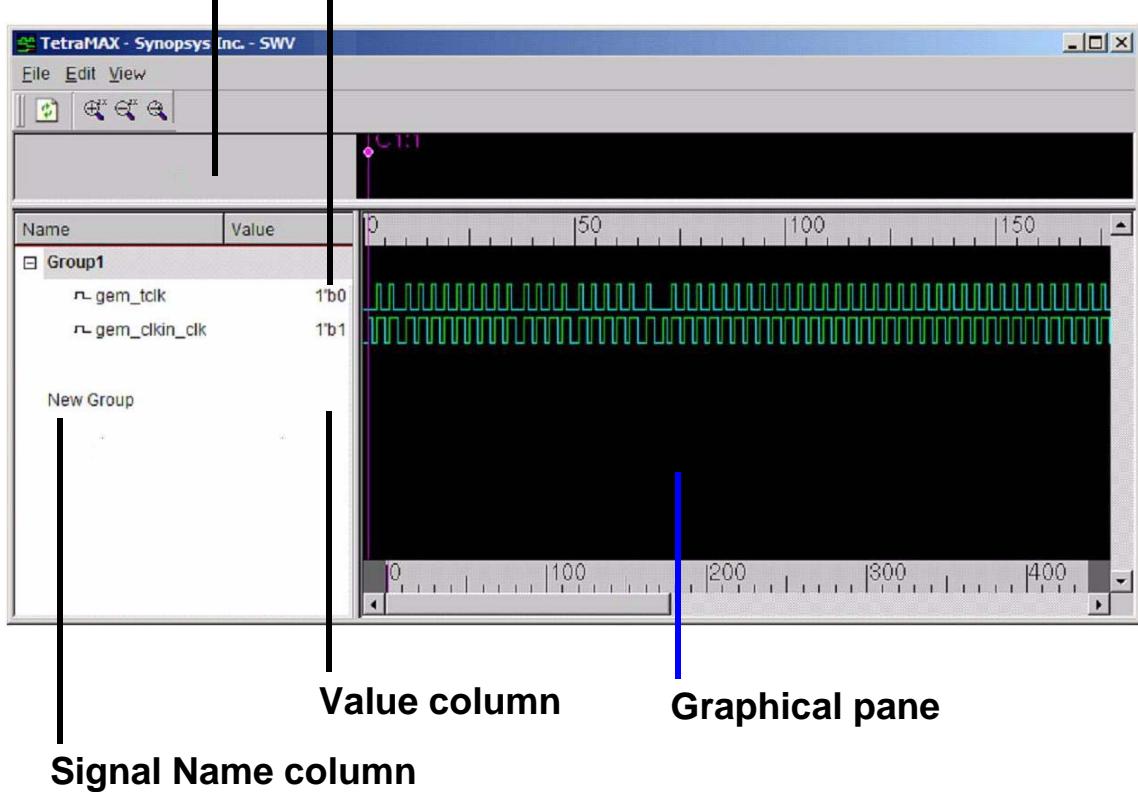
- Refreshing the View
- Identifying Signal Types in the Graphical Pane
- Using the Time Scales
- Using the Marker Header Area

Understanding the SWV Layout

The layout of the SWV is shown below in [Figure 8-4](#).

Figure 8-4 SWV layout

Signal List pane 1-bit signal with binary value 0 and 1



Note that the SWV contains a scrollable list view (the Signal List pane) and a corresponding graphical pane (the Graphical pane). The Signal List pane contains two columns: the first column is the Signal Group tree view with the signal/bus names, and the second column is the value according to the reference cursor.

The Graphical pane consists of equivalent rows of signal in graphical drawing. Also, the reference cursor and marker can be manipulated in the Graphic pane to perform measurement between events. There are two timescales (upper and lower). The upper timescale denote the current view port time range and the lower timescale represent the global (full) time range with data.

Using the Signal List Pane

The Signal List pane, located on the left side of the SWV, is organized into the following three-level tree view:

- The root node is the group name.
- The second level is the signal or bused signal name.
- The third level is the individual bit of the bused signal (if applicable).

Signals are grouped together according to the target to which it is added to. New groups can be created with a signal dropped to the (default) new group tree node.

Signal groups provide a logical way to organize your signals. For example, you can keep all input signals in one group and output signals in another group. You can expand or collapse the signal list by clicking the + sign to the left of the group name. The sign changes to - when you expand it.

You can edit group names, but you cannot edit signal names. The SWV enables you only to view the design. You can not edit or make any changes to it.

Manipulating Signals

This section describes how to add, delete, insert, and copy signals in the SWV.

Adding Signals

You can add any number of signals to the SWV base at the current insertion point. By default, the insertion point is to create a new signal group. After a signal is added, the insertion point is advanced to the most recently visited group.

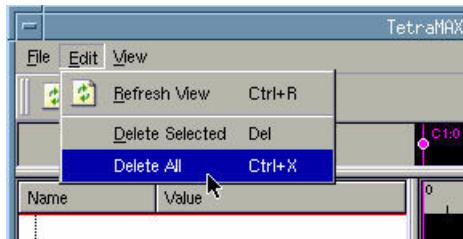
To add a signal, middle-click the signal from the waveform to select it. The red insertion line appears around the signal. Then drag it to the required group.

To add a range of signals, press Shift and click the signals to select the range. A red rubber band box appears around the range. Then drag the box into a group.

Deleting Signals

To delete a signal, or multiple signals, and groups, select the signal (s) to be deleted, and press the Delete button or choose Edit > Delete Selected. To delete multiple signals or groups, choose Edit > Delete All. [Figure 8-5](#) shows the Edit menu.

Figure 8-5 Selecting Delete All in the Edit Menu



Inserting Signals

An insertion point is denoted by a red line. There might be times when you need to copy or duplicate a signal (shift + left-click) and move it to other groups. To do this, you can drag the insertion point into the required group.

The target of the insertion point can be specified to be a “New Group” or any group that already exists.

When an insertion point is applied to a group, the signal is added to the bottom of the list. When the insertion point is at a particular position, the signal is added below the position of the insertion point in the signal list view. If the insertion point is in the new group item view, it creates a new group. If the insertion point is in the list view, it just adds the signal to the group.

[Figure 8-6](#) shows an empty waveform table, and [Figure 8-7](#) illustrates signal insertion.

Figure 8-6 Empty Waveform Table

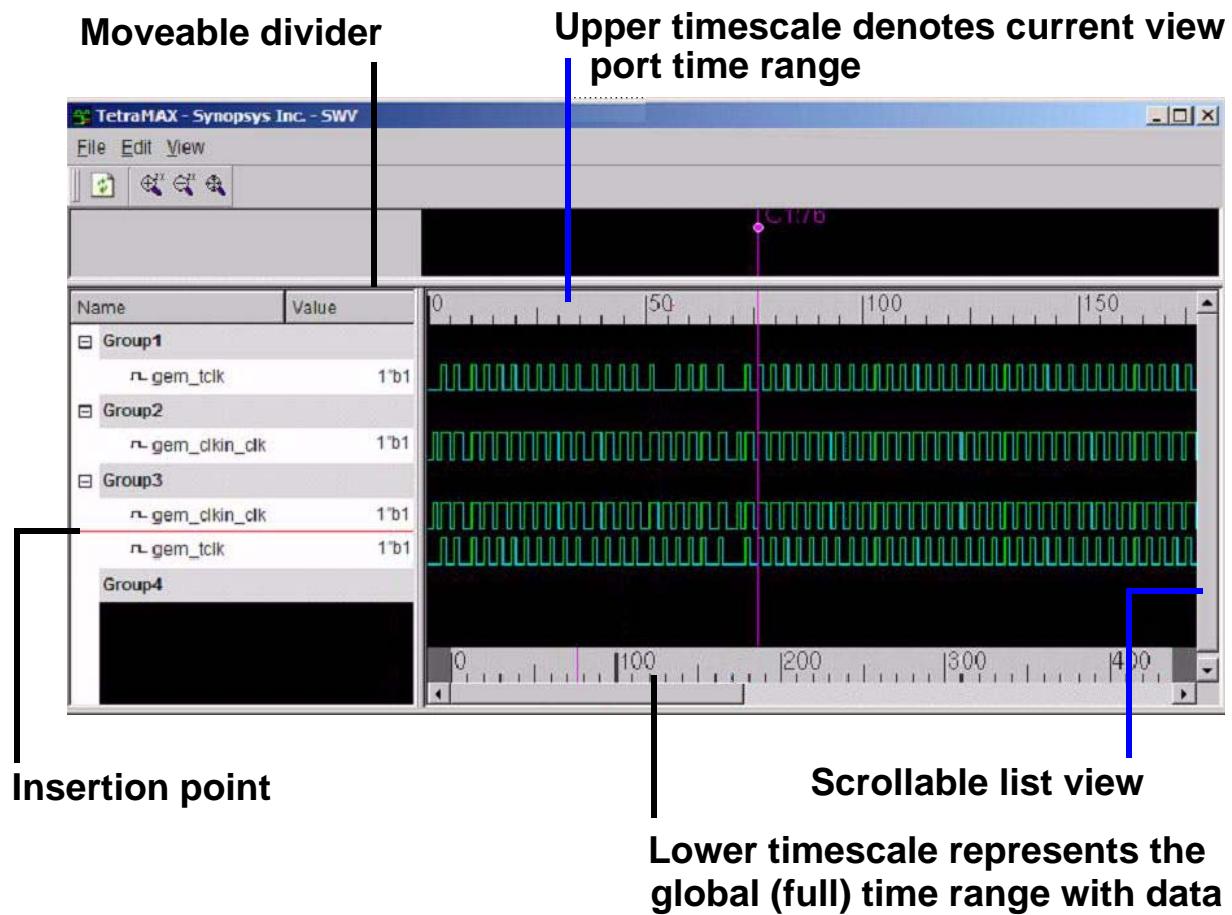
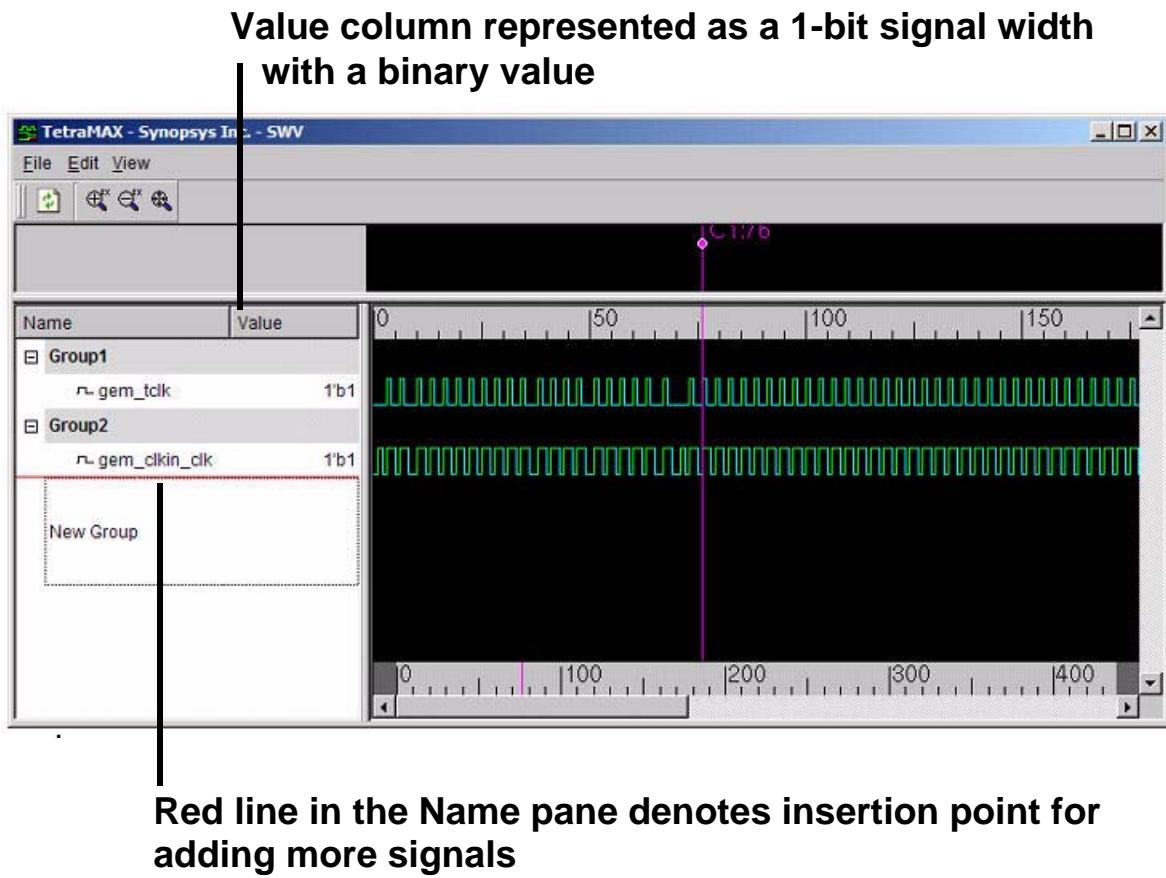


Figure 8-7 Inserting Signals



Refreshing the View

To refresh the view (similar to the GSV), click the Refresh button or select Edit > Refresh View.

Understanding the Simulation Waveform Viewer Color Codes

The simulation waveform viewer uses the color codes shown in the table below. [Figure 8-8](#) shows the colors in the waveform viewer.

Argument	Description
Red	Insertion line
Pink	Cursor measurement
White	Pattern marker
Green	Load signal
Yellow	Capture signal

Figure 8-8 Simulation Waveform Viewer Colors



Identifying Signal Types in the Graphical Pane

Most signals contain events, and each event change is represented by a transition in the drawing. The viewer signals are classified into scalar type. A scalar signal carries a single bit transition between the values 0, 1, Z, X. A signal band is divided into vertical subsections to draw the values. A line drawn at the bottom of a signal band refers to event 0, while a line drawn on the top of the band refers to event 1. A Z value is drawn in the middle of the band and a filled band denotes an unknown X value.

When a vector contains an X value, it is drawn in the red event (default) color. When the vector contains some Z value, it is drawn in yellow. When all values of the transition vector are unknown, a filled red rectangle is used, and if all are Z, a horizontal yellow line is drawn in the middle of the signal band.

Using the Time Scales

As shown in [Figure 8-6 on page 8-9](#), there are two time scales: upper and lower. The upper time scale denotes the current view port time range and the lower time scale represents the global (full) time range with data. Both of these time scales are explained in the following subsections.

Upper Time Scale

The upper-time-scale area displays the current viewing time range in x10ps. You can drag markers or cursors visible in the upper time scale to other locations in the view. In addition, you can perform zoom operations in the upper time scale area by clicking your left mouse button and horizontally dragging to specify a horizontal zoom area. When you release the left mouse button, the current view refreshes with the zoomed in view in the current wave list. You won't need to further adjust the vertical alignment. When in full zoom view, the upper time scale will display the same value and range as the lower time scale.

Lower Time Scale

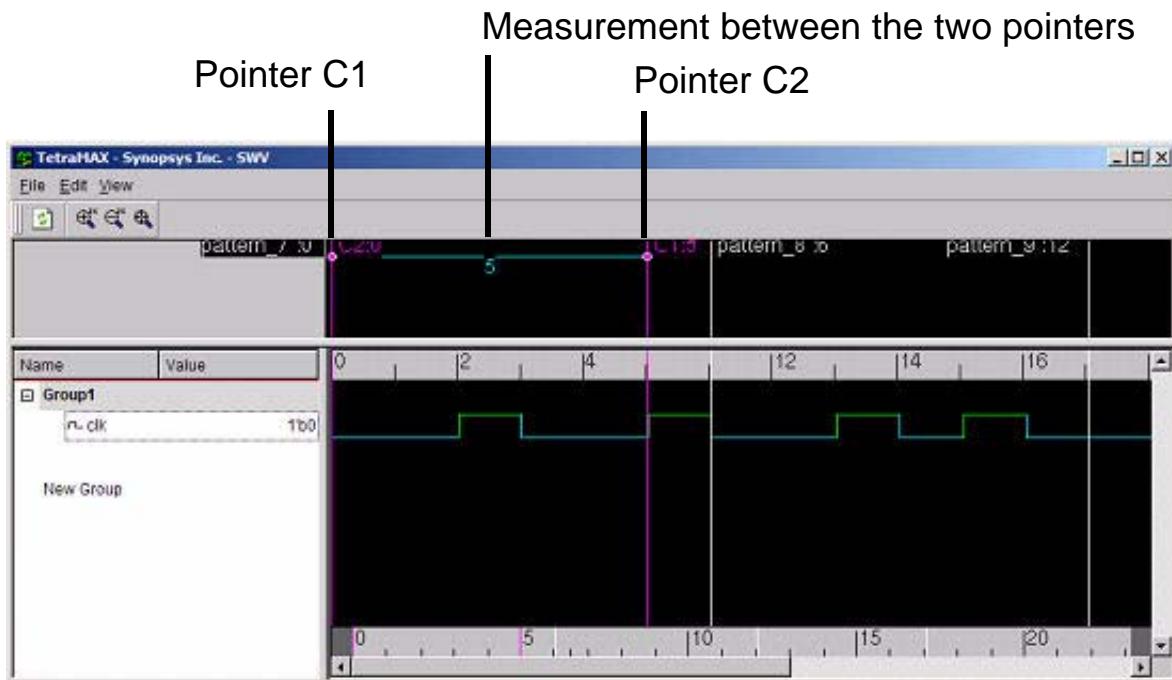
The lower time scale area shows the full time range the data occupies. You can control zoom operation using your left mouse button, which causes an adjustment in the current view time range. The width of the scroll thumb in the horizontal scrollbar shows the approximate view area in proportion to the full data time range. Reference and marker cursors are shown in the lower timescale for easy identification of marked location and to maintain the context for navigation.

Using the Marker Header Area

The SWV provides two reference pointer: C1 and C2. These pointers are drawn in magenta, whereas other marker cursors are in white. A marker identifier (a circle) in the marker header area is used for marker selection by the pointer.

The graphical pane shows a graphical representation of equivalent rows of signals. You can manipulate the reference cursors and markers in the graphical pane to perform measurement between events (as shown in [Figure 8-9](#).)

Figure 8-9 Reference Pointers



Adding and Deleting Pointers

To add the default reference pointer C1 or C2, you can drag the C1 pointer to the clicked location, or you can use the middle mouse button to drag the C2 reference pointer.

You can delete all markers by first selecting the markers, and then choosing the Delete Selected command (or the Delete This Marker command if you selected only one marker).

Moving a Marker Pointer

There are two methods you can use to move a marker pointer:

- Drag the marker identifier (a circle) in the marker header area to the new location. This method is limited to relocating the marker identifier to a region in the current viewable time range. See [Figure 8-10](#) below.
- Drag the left marker, then click to release it.

Figure 8-10 Moving a marker cursor



Measuring Between Two Pointers

As shown in [Figure 8-9 on page 8-13](#), you can use any pointer as a reference point for measurement. The other pointer value will change according to the currently selected reference cursor.

Using the SWV With the GSV

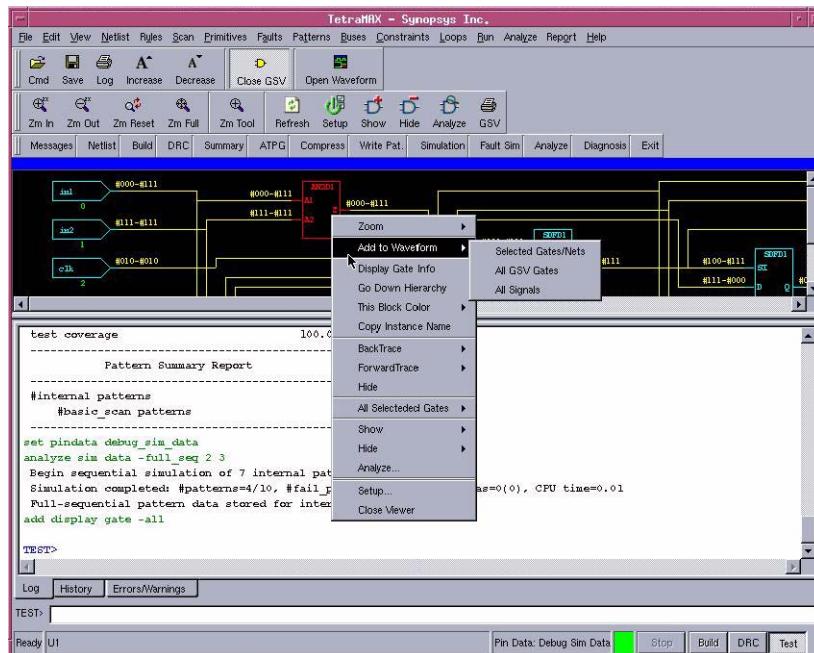
The main aspect of the TetraMAX GUI is a schematic viewer containing annotated simulation values during DRC. You can expand the viewer to ease DRC debugging. The rest of the TetraMAX GUI consists of dialog boxes and a console. The GUI displays these patterns in a logic cone view, which contains a cone of logic from each design, derived by tracing back from a pair of matched points. When there are DRC warnings and error messages, the logic cones are produced.

To launch the simulation waveform window from a selected logic cone view: Choose View->Waveform View (as shown in [Figure 8-3](#)), then choose “Setup...” (as shown in [Figure 8-11](#)) and select “Pin Data Type” as “Test Setup,” then use the RMB “add to waveform” “all GSV gates.”

The simulation waveform is initialized with pattern data associated with the cone view from which it was created during DRC. There is a one-to-one correspondence between GSV and SWV when a DRC violation is used. If the GSV is closed, its corresponding waveform view is not closed. If the SWV is closed, its corresponding GSV is not closed, and the pattern annotations on it are not cleared.

[Figure 8-11](#) shows how to use the SWV with the GSV.

Figure 8-11 Using the SWV With the GSV

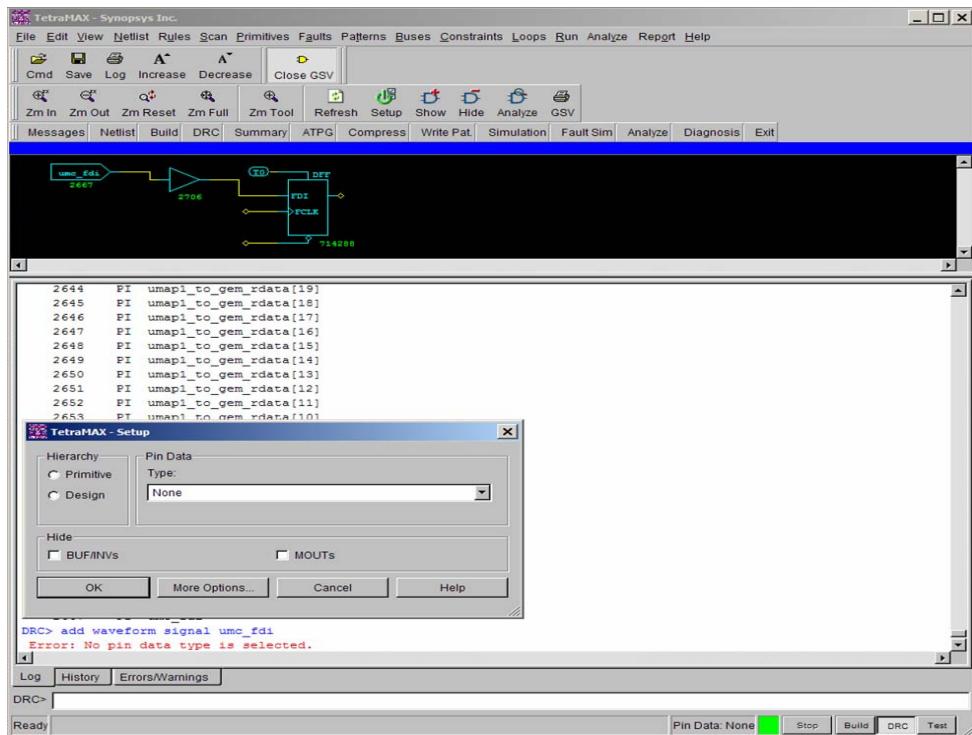


The data values displayed are generated either by DRC or by ATPG. Data values generated by DRC correspond to the simulation values used by DRC in simulating the STIL test_setup protocol to check conformance to the test protocol. Data values generated by ATPG are the actual logic values resulting from a specific ATPG pattern.

When you analyze a rule violation or a fault, TetraMAX automatically selects and displays the appropriate type of pin data in the GSV. You can also manually select the type of pin data to be displayed by using the SETUP button in the GSV toolbar, or you can use the set pindata command on the command line.

The SWV can use only the pin data types listed in [Table 8-1](#). [Figure 8-12](#) shows an error caused when you do not select a valid pin data type that is supported by the SWV.

Figure 8-12 Pin Data Type Error



Using the SWV Without the GSV

You might need to launch the SWV without the GSV when you have failing external patterns (read externally into TetraMAX) and you want to see the patterns for an overall evaluation of how TetraMAX interprets them. You can view the values of gates and nodes of a design for a particular pattern, or you can just view a waveform if you are already familiar with the circuit nets and nodes and you are running iterative loops in TetraMAX.

The following are sample flows using the SWV. Enter these commands in a command file.

Sample Flow

```
set pldata test_setup // test_setup is one of the many
//pin_data_types
add disp gate -all // this invokes the GSV containing the gates
// of interest.
set pldata test_setup // test_setup is one of the many
// pin_data_types required for SWV
add WAveform Signals < > // this invokes the SWV containing the
// waveforms for the gates of interest. The user may know the
// gates from a previous run in the GSV.
```

Example 2

```
Set simulation -data 85 89 // specify the values to store by
//patterns start/end run simulation
Run simulation -sequential // execute a sequential simulation
Set pldata seq_sim_data // required for SWV
Add waveform signals <> // this invokes the SWV containing the
// waveforms for the IOs of the patterns 85 through 89
```

Example 3

```
set pattern external patterns.stil
Analyze sim data pats1.vcd -fast 1
add display gate < >
Set pldata debug_sim_data // should be the default setting
```

SWV Inputs and Outputs

There are two input flows:

- Either the streaming pin_pathname | gate_id from the GSV to the SWV
- Streaming the externally read pattern data to the SWV displaying all I/Os

The output is an SWV. The output includes messages, warnings, and errors.

Analyzing Violations

This section describes the TetraMAX error messages and others messages related to the SWV.

Error: No pin data type is selected

You cannot select any nets or gates because the pin_data types require data to be stored internally to TetraMAX using the set drc -store_setup or the set simulation -data command. See “[Supported Pin Data Types and Definitions](#)” on page 8-2.

```
Error: Invalid argument "TOP_template_DW_tap_inst/  
U34/QN". <M1>
```

This means that a gate was selected and added to the SWV but the QN pin is not valid or not used due to no net attached.

```
TOP_template_DW_tap_inst/U10_1/CP (Gate 41) is  
already in waveform list as  
TOP_template_DW_tap_inst/U34/CP.
```

This message is informative when you select two gates that have the same clock, and add them to the SWV. The GSV picks one name and displays a message that the other pin has the same name.

```
/U1_out (Gate 5) is already in waveform list as U1/Z
```

This message is informative when you select two gates that have the same clock and add them to the SWV. The GSV picks one name and displays a message that the other pin has the same name.

9

STIL Procedure Files

This chapter contains instructions and guidelines for creating and editing a Standard Test Interface Language (STIL) procedure file, the vehicle through which TetraMAX obtains information needed for test design rule checking (DRC).

This chapter contains the following sections:

- [STIL Usage in TetraMAX](#)
- [Creating a New STIL Procedure File](#)
- [Editing the STIL Procedure File](#)
- [Testing the STIL Procedure File](#)
- [STIL Procedure File Guidelines](#)
- [JTAG/TAP Controller Variations for the load_unload Procedure](#)
- [Multiple Scan Groups](#)
- [Limiting Clock Usage](#)

STIL Usage in TetraMAX

STIL is an emerging standard for simplifying the number of test vector formats that automated test equipment (ATE) vendors and computer-aided engineering (CAE) tool vendors must support. TetraMAX supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data.

In constructing a STIL procedure file (SPF), you can define the minimum information needed by TetraMAX. However, any STIL files written as TetraMAX output contain an expanded form of the minimum information and might also contain pattern/response data that the ATPG algorithm produces. TetraMAX reads and writes in STIL, so once a STIL file is generated for a design, TetraMAX can read it again at a later time to recover the clock/constraint/chain data, the pattern/response data, or both.

For general information on the STIL standard (Standard Test Interface Language (STIL) for Digital Test Vectors, IEEE Std. 1450.0-1999), see the STIL home page at the following Web address:

<http://grouper.ieee.org/groups/1450/index.html>

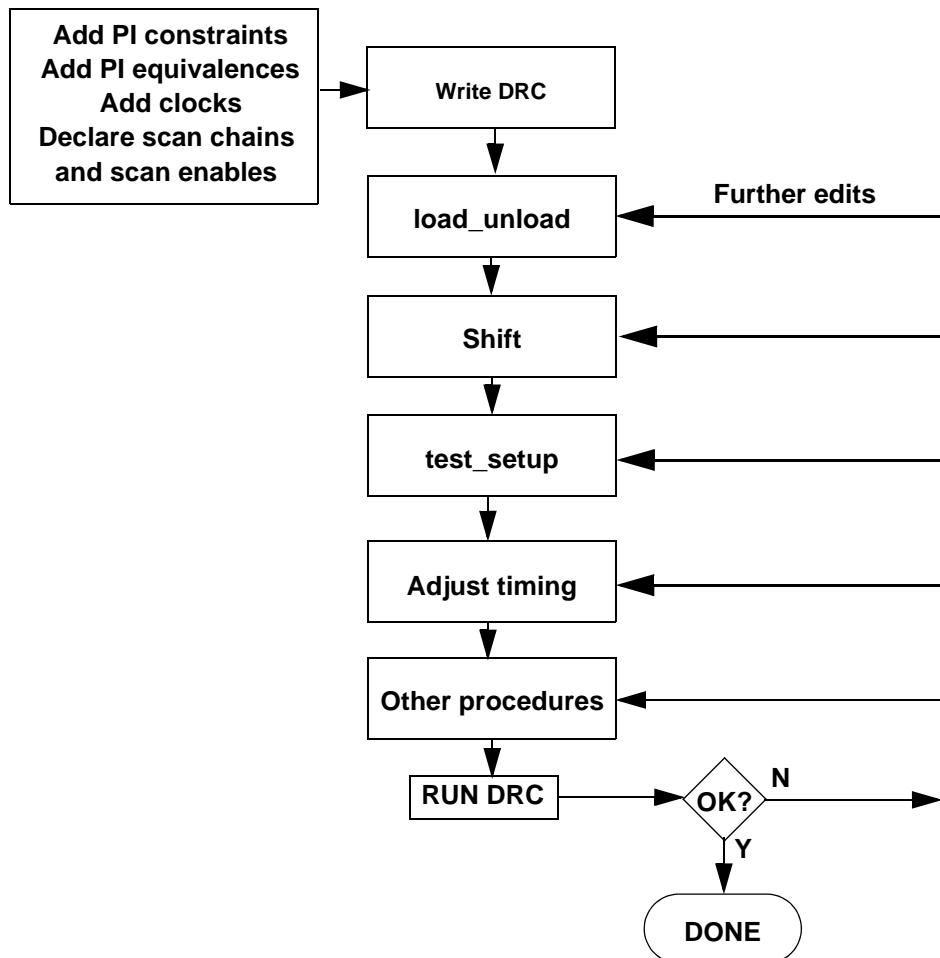
TetraMAX makes use of the Extensions to STIL for Semiconductor Design Environments, IEEE Std. 1450.1. For details, see See “[Ltran Translation Utility](#)” on page D-2.”

Creating a New STIL Procedure File

This section describes the steps involved in creating an SPF with no prior input. First, you define primary input (PI) constraints, clocks, and scan chain information. Next, you use TetraMAX to write out the initial STIL template. You edit the STIL template to define the required STIL procedures and port timing.

[Figure 9-1](#) shows a process flow for creating the initial SPF. This flow starts with no prior input.

Figure 9-1 Flow for Creating an SPF



Declaring Primary Input Constraints

If one or more top-level ports put the design into the ATPG or another test mode, as is common in design-for-test (DFT) practices, usually those ports must be held constant so that the ATPG algorithm can succeed.

In the STIL procedures, you can force a constrained port to a state other than the requested constrained value for a few tester cycles, and then the port must be returned to its constrained value. For example, you might want to hold a global reset port to an off state for general ATPG patterns but allow it to be asserted to initialize the design. For more information, see [“Defining the test_setup Macro” on page 9-9](#).

You declare such a port using the Add PI Constraints dialog box, using the `add pi constraints` command, or using a definition in the SPF.

Note:

A port that enables a test mode for a design is different from the scan_enable and other ports that change state during the shift and capture operations.

Using the Add PI Constraints Dialog Box

To use the Add PI Constraints dialog box to declare a PI constraint,

1. From the menu bar, choose Constraints > PI Constraints > Add PI Constraints. The Add PI Constraints dialog box appears.
2. In the Port Name field, enter the name of the port you want to constrain. To select from a list of ports, click the down-arrow button at the end of the Port Name field.

In this case, a port named TEST_MODE must be held to a constant state of logic 1 for all patterns generated by the ATPG algorithm.

3. From the Value list, choose the value to which you want to constrain the port.
4. Click Add.

The dialog box remains open so that you can add more constraints if needed.

5. Click OK.

Using the add pi constraints Command

You can also declare a PI constraint by using the `add pi constraints` command. For example:

```
DRC> add pi constraint 1 TEST_MODE
```

For the complete syntax and option descriptions, see online Help for the `add pi constraints` command.

Declaring Clocks

Declare ports as clocks only if those ports affect the state of flip-flops and latches or control RAM/ROM read or write ports. You declare clocks in terms of their natural off states. An active-high clock has an off state of 0, and an active-low clock has an off state of 1.

You can declare a clock by using the Add Clocks dialog box, the Edit Clocks dialog box, the DRC dialog box, the `add clocks` command, or by editing the timing block in the STIL procedure file.

Using the Edit Clocks Dialog Box

To use the Edit Clocks dialog box,

1. From the menu bar, choose Scan > Clocks > Edit Clocks. The Edit Clocks dialog box appears.
2. To declare a clock, select the port name from the Port Name list, specify the off state (0 or 1), and specify whether the clock is used for scan shifting. Clock signals used for asynchronous set/reset or RAM/ROM control are not used for scan shifting and are not pulsed during shift procedures.
3. If you want to specify the test cycle period, leading edge time, trailing edge time, and measure time of the clock, fill in the corresponding fields (Period, T1, T2, and Measure) and set the time units (ns or ps). In the absence of explicit timing specifications, the default values are: Period=100, T1=50, T2=70, Measure=40, and Unit=ns.

The same period, measure time, and units apply to all clocks in the system, but each clock can have its own leading and trailing edge times (T1 and T2). A measure time less than T1 implies a preclock measure protocol, whereas a measure time greater than T2 implies an end-of-cycle measure protocol.

4. Click Add. The clock declaration is added to the list box.
5. Repeat steps 2 to 4 for each clock input in the design. You can also remove, copy, or modify an existing clock definition.
6. Click OK to implement the changes you have made in the dialog box.

Using the add_clocks Command

You can also declare clocks, and define the test cycle period and timing parameters by using the `add_clocks` command. For example:

```
DRC> add_clocks 0 CLK1 -timing 200 50 80 40 -unit ns -shift
```

For the complete syntax and option descriptions, see the online help for the `add_clocks` command.

Asynchronous Set and Reset Ports

By default, latches and flip-flops whose set and reset lines are not off when all clocks are at their off state are treated as unstable cells. Because they are unstable, their output values are unknown and they cannot be used during test pattern generation.

One way to make these elements stable is to declare their asynchronous set/reset input signals to be clocks. During ATPG, TetraMAX holds these inputs inactive while other clocks are being used. However, test coverage surrounding the elements might still be limited.

To have these latches and flip-flops treated as stable cells without declaring their set/reset inputs to be clocks, use the `set drc -allow_unstable_set_resets` command. For details, see “[Cells With Asynchronous Set/Reset Inputs](#)” on page 4-18.

Declaring Scan Chains and Scan Enables

To declare the scan chains and scan enable inputs, you can use either the DRC dialog box or enter a command at the command line (see “[Using a Command](#)”).

Using the DRC Dialog Box

To use the DRC dialog box,

1. Click the DRC button in the command toolbar at the top of the TetraMAX main window. The DRC dialog box appears.
2. Click the Quick STIL tab if it is not already selected. Under the tab, select the Scan Chains/Scan Enables view if it is not already selected.

Note:

Selecting the Clocks view displays the Edit Clocks dialog box, as described in “[Declaring Clocks](#)” on page 9-4.

3. To specify a scan chain, enter a name for the scan chain in the Name field. Specify the Scan In and Scan Out ports by selecting the port names from the pull-down lists.
4. Click Add. The scan chain definition is added to the list.
5. To define a scan enable input, select the port name from the Port Name pull-down list. In the Value field, specify the port value during scan shifting.
6. Click Add. The scan enable port definition is added to the list.
7. When you finish specifying the scan chain and scan enable information, click OK.

Using a Command

You can also use the following commands to declare, report, and remove scan chains and scan enables:

- `add scan chains`
- `add scan enables`
- `report scan chains`
- `report scan enables`
- `remove scan chains`

- remove scan enables

For the complete syntax and option descriptions, see online Help for these commands.

Writing the Initial STIL Template

You can write out a template STIL file at any point after executing a `run build_model` command. To do so, use the `write drc_file` command or the Write tab in the DRC dialog box. If you have declared any clocks, PI equivalences, PI constraints, or scan chain information, then those items will appear in the STIL file written by TetraMAX.

Editing the STIL Procedure File

After you create an SPF template file, you need to edit the template to accomplish the following tasks:

- Defining the `load_unload` Procedure
- Defining the Shift Procedure
- Defining the `test_setup` Macro
- Controlling Bidirectional Ports
- Defining Pulsed Ports
- Defining Basic Signal Timing
- Defining System Capture Procedures
- Creating Generic Capture Procedures
- Defining a Sequential Capture Procedure
- Defining the End-of-Cycle Measure
- Defining Constrained Primary Inputs
- Defining Equivalent Primary Inputs
- Defining Reflective I/O Capture Procedures
- Defining Other STIL Procedures
- Using Quick STIL Commands with an On-Chip Clock Controller

Note that STIL keywords are case-sensitive. When you enter a keyword in an SPF, ensure that you use uppercase and lowercase letters correctly (for example, `ScanStructures`, `ScanChain`, `ScanIn`, `ScanOut`). Incorrect case is a common cause of syntax errors.

Throughout the STIL examples in this chapter, text strings are sometimes enclosed in quotation marks and sometimes not. The general rule in STIL procedure files is that quotation marks are optional unless the text string contains parentheses "()", braces "[]", or spaces.

Defining the load_unload Procedure

The `load_unload` procedure, which TetraMAX requires, contains information about placing the scan chains into a shiftable state and shifting one bit through them. TetraMAX creates this procedure if you define the scan enable information before you write out the STIL file.

Although specifying the scan chain length is required in standard STIL syntax, it is optional for STIL input files to TetraMAX. When writing a STIL pattern file, TetraMAX determines the scan chain lengths and defines the correct length of each scan chain while writing STIL output.

[Example 9-1](#) shows the syntax used to define scan chains. This example consists of the STIL header followed by the `ScanStructures` keyword and four scan chains. In this example, the scan chains are named `c1` through `c4`. The `Procedures` section defines a procedure called `load_unload`, which consists of one test cycle (a "`V { . . . }`" vector statement). In the test cycle, the clocks `CLOCK` and `RESETB` are set to their off states and the `SCAN_ENABLE` port is driven high to enable the scan chain shift paths.

Example 9-1 SPF: Defining Scan Chain Loading and Unloading

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE=1; }
    }
}
```

Defining the Shift Procedure

The `Shift` procedure specifies how to shift the scan chains within the definition of the `load_unload` procedure. The bold text shown in [Example 9-2](#) defines the `Shift` procedure.

Example 9-2 SPF: Defining the Scan Chain Shift Procedure

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
```

The `Shift` procedure consists of a test cycle (`V`) in which the scan inputs `_si` are set from the next available stimulus data (`#`); the scan outputs `_so` are measured from the next available expected data (`#`); and the port `CLOCK` is pulsed (`P`). There are four `#` symbols, one for each scan chain defined.

When the `load_unload` procedure is applied, the `Shift` procedure is applied repeatedly as necessary to shift as many bits as are in the longest scan chain.

A test cycle has been added after the `Shift` procedure to ensure that the clocks and asynchronous reset/set ports are at their off states. This is an optional cycle if all procedures start out by ensuring that the clocks and asynchronous set/reset ports are at the off state.

The `_si` and `_so` grouping names are expected by TetraMAX. They refer to the scan inputs and scan outputs. The STIL file output generated by TetraMAX completely describes the port names and ordering contained in the groupings `_si` and `_so`; you do not have to enter this information.

Defining the `test_setup` Macro

The `test_setup` macro is optional. It defines any initialization sequences that the design might need for test mode, or to ensure that the device is in a known state.

In [Example 9-3](#), the bold text is a macro that has been added with the name `test_setup`, which consists of three test cycles. The first cycle sets the inputs `TEST_MODE`, `PLL_TEST_MODE`, and `PLL_RESET` all to 1. The second cycle changes `PLL_RESET` to 0, and the third cycle returns `PLL_RESET` to 1.

Example 9-3 SPF: Defining the `test_setup` Macro

```
STIL;
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
```

```

    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si #####; _so #####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
MacroDefs {
    test_setup {
        V { TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1; }
        V { PLL_RESET = 0; }
        V { PLL_RESET = 1; }
    }
}

```

If you need to initialize a port to X in the `test_setup` macro, the STIL assignment character for this is “N”. Use of “X” indicates an output measure is to be performed and the result is to be masked.

Using Loop Statements

You can use loops in `test_setup` procedures, however you must limit the usage of `Loop` statements in this context, otherwise:

- the size of the `test_setup` macro will grow unreasonably
- the time to simulate the clock pulses will be unreasonable

It is recommended that you represent only the necessary events required to initialize the device for ATPG efforts in the `test_setup` procedure. Loops that represent device test, for example, one million vectors to lock a PLL clock at test, will not be appropriate or necessary in the ATPG environment when a PLL clock is black-boxed.

Vectors may be extracted before the Loop and after the Loop, and the Loop count decremented as appropriate for each extracted vector.

Each extracted vector must contain the exact same sequence of clocks as specified in the vector inside the `Loop` statement. Empty vectors (no events) may appear between the vectors that contain clock pulses — but it is critical that any vector that contains a signal assignment, match in order with the signal assignments for the vector inside the `Loop`. Otherwise, this extracted vector will not be recognized as consistent with the internal vector, and the extracted vector will not be “re-rolled” into the Loop count, causing DRC analysis errors.

The only supported contents of a Loop in a *setup procedure are C {} condition statements, V {} vectors, or WaveformTable "W" statements.

Example 1:

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        C { "all_inputs" = NNN; "all_outputs" = \r6 X; }
            Loop 10 { V { "s_in"=0; "clk"=P; } }
    }
}
```

Example 2:

```
MacroDefs {
    "test_setup" {
        W "_default_WFT_";
        V { "CK"=0; }
            Loop 4 { V { "s_in"=0; "clk"=P; } }
    }
}
```

Loops in STIL may contain other references, for example calls to other macros and procedures. These constructs are not supported within the setup environment today.

Controlling Bidirectional Ports

During scan chain shifting defined by the `load_unload` procedure, the control logic for bidirectional ports could possibly be at a random state, causing DRC violations of the Z class. You can prevent Z class violations resulting from random states of bidirectional ports by:

- Placing a Z value on the bidirectional ports, which turns off the ATE tester drive
- Providing for a top-level control port, enabled only for test mode, that globally disables all bidirectional drivers

[Example 9-4](#) illustrates a design that has a top-level bidirectional control port called `BIDI_DISABLE` (shown in bold). This example uses the `SignalGroups` section to define an ordered grouping of ports referenced by the label `bidi_ports`, thus facilitating assignment to multiple ports.

Example 9-4 SPF: Controlling Bidirectional Ports

```
STIL;
SignalGroups {
    bidi_ports = '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]" +
"D[5]" + "D[6]" +
        + "D[7]" + "D[8]" + "D[9]" + "D[10]" + "D[11]" + "D[12]" +
        + "D[13]" + "D[14]" + "D[15]";
```

```

}
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        V {
            CLOCK=0; RESETB=1; SCAN_ENABLE = 1;
            BIDI_DISABLE = 1;
            bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;
        }
        V {}
        V { bidi_ports = \r4 1010 ; }
        Shift {
            V { _si=####; _so=####; CLOCK=P; }
        }
        V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
    }
}
MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
           BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZ; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}

```

Additions for bidirectional control have been made within both the `load_unload` procedure and the `test_setup` macro. The additions to the `load_unload` procedure are as follows:

- These lines have been added to the first test cycle:

```

        BIDI_DISABLE = 1;
        bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;

```

Setting the `BIDI_DISABLE` port to 1 disables all bidirectional drivers in the design. Assigning Z states to the `bidi_ports` ensures that the ATE tester does not try to drive the bidirectional ports.

- A second, empty test cycle has been added, as follows:

```

        V {}

```

The empty braces indicate that no signals are changing. This provides a cycle of delay between turning off bidirectional drivers with `BIDI_DISABLE=1` and forcing the bidirectional ports as inputs in the third cycle. This is not usually necessary, but illustrates one technique for adding delay using an empty test cycle.

- A third test cycle has been added, as follows:

```
v { bidi_ports = \r4 1010 ; }
```

Here, the `bidi_ports` are driven to a non-Z state so that they do not float while the drivers are disabled. The `\r4` syntax indicates that the following string is to be repeated four times. In other words, the pattern applied to the `bidi_ports` group is 10101010101010.

In the `test_setup` macro, the following line has been added to the first test cycle:

```
BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZ;
```

Here, the `BIDI_DISABLE` port is forced high and the `bidi_ports` are set to a Z state.

Defining Pulsed Ports

Pulsed ports for clocks and asynchronous sets and resets can be defined within TetraMAX using the `add clocks` command or the Add Clocks dialog box; or they can be described in an optional section in the SPF.

The bold text in [Example 9-5](#) defines two pulsed ports, `CLOCK` and `RESETB`, by adding a `Timing{ .. }` section and a `WaveformTable` definition with the special-purpose name recognized by TetraMAX, `_default_WFT_`.

Example 9-5 SPF: Defining Pulsed Ports

STIL;

```
Timing {
  WaveformTable "_default_WFT_" {
    Period '100ns';
    Waveforms {
      CLOCK { P { '0ns' D; '50ns' U; '80ns' D; } }
      CLOCK { 01Z { '0ns' D/U/Z; } }
      RESETB { P { '0ns' U; '10ns' D; '90ns' U; } }
      RESETB { 01Z { '0ns' D/U/Z; } }
    }
  }
}

ScanStructures {
  ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
  ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
  ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
  ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}

Procedures {
  "load_unload" {
    V {
      CLOCK=0; RESETB=1; SCAN_ENABLE = 1;
    }
  }
}
```

```

        BIDI_DISABLE = 1;
        bidi_ports = ZZZZ ZZZZ ZZZZ ZZZZ;
    }
V {}
V { bidi_ports = \r4 1010 ; }
Shift {
    V { _si=####; _so=####; CLOCK=P; }
}
V { CLOCK=0; RESETB=1; SCAN_ENABLE = 0; }
}
}

MacroDefs {
    test_setup {
        V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
            BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZ; }
        V {PLL_RESET = 0; }
        V {PLL_RESET = 1; }
    }
}
}

```

This timing definition has the following features:

- The period of the test cycle is 100 ns.
- The following line defines the port `CLOCK` as a positive-going pulse that starts each cycle at a low value (`D` = force down), transitions up (`U` = force up) at an offset of 50 ns into the cycle, then transitions down at an offset of 80 ns:

```
CLOCK { P { '0ns' D; '50ns' U; '80ns' D; } }
```

- The next line indicates that for test cycles in which the `CLOCK` port has a constant value, the change to that value occurs at an offset of 0 ns into the test cycle:

```
CLOCK { 01Z { '0ns' D/U/Z; } }
```

- The following lines define the port `RESETB` as a negative-going pulse:

```
RESETB { P { '0ns' U; '10ns' D; '90ns' U; } }
RESETB { 01Z { '0ns' D/U/Z; } }
```

`RESETB` is defined nearly identically to `CLOCK`, with two exceptions. First, it starts each pulse cycle in the `U` (force up) position, transitions to `D` (force down), and then to `U` again. Second, the timing is slightly different, with the first transition at an offset of 10 ns into the cycle and the last transition at an offset of 90 ns.

Defining Basic Signal Timing

Defining signal timing is not required for performing DRC or generating patterns, but it is necessary for writing out patterns with meaningful timing. If you do not define the signal timing explicitly, TetraMAX uses its own default values.

Caution!

Do not edit signal timing values in ATPG-generated pattern files; doing so can result in simulation mismatches or ATE mismatches for your patterns. Instead, define signal timing in your SPF and run DRC with that procedure file before generating handoff patterns with ATPG.

[Example 9-6](#) contains many additions to [Example 9-5](#) to define signal timing. Line numbers have been added for reference.

Example 9-6 SPF: Defining Timing

```
1. STIL;
2. UserKeywords PinConstraints;
3. PinConstraints { "TEST_MODE" 1; "PLL_TEST_MODE" 1; }
4. SignalGroups {
5.     bidi_ports '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]" +
6.                 "D[5]" + "D[6]" + "D[7]" + "D[8]" + "D[9]" + "D[10]" +
7.                 "D[11]" + "D[12]" + "D[13]" + "D[14]" + "D[15]" ';
8.     input_grp1 'SCAN_ENABLE + BIDI_DISABLE + TEST_MODE +
9. PLL_TEST_MODE' ;
10.    input_grp2 'SDI1 + SDI2 + DIN + "IRQ[4]"' ;
11.    in_ports 'input_grp1 + input_grp2';
12.    out_ports 'SDO2 + D1 + YABX + XYZ';
13. }
14. Timing {
15.     WaveformTable "BROADSIDE_TIMING" {
16.         Period '1000ns';
17.         Waveforms {
18.             CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } } // clock
19.             CLOCK { 01Z { '0ns' D/U/Z; } }
20.             RESETB { P { '0ns' U; '400ns' D; '800ns' U; } } /
21.             RESETB { 01Z { '0ns' D/U/Z; } }
22.             input_grp1 { 01Z { '0ns' D/U/Z; } }
23.             input_grp2 { 01Z { '10ns' D/U/Z; } }
24.             // outputs are to be measured at t=350
25.             out_ports { HLTX { '0ns' X; '350ns' H/L/T/X; } }
26.             // bidirectional ports as inputs are forced at t=20
27.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
28.             // bidirectional ports as outputs are measured at t=350
29.             bidi_ports { X { '0ns' X; } }
30.             bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
31.         }
32.     } // end BROADSIDE_TIMING
33.     WaveformTable "SHIFT_TIMING" {
34.         Period '200ns';
35.         Waveforms {
36.             CLOCK { P { '0ns' D; '100ns' U; '150ns' D; } }
37.             CLOCK { 01Z { '0ns' D/U/Z; } }
38.             RESETB { P { '0ns' U; '20ns' D; '180ns' U; } }
39.             RESETB { 01Z { '0ns' D/U/Z; } }
40.             in_ports { 01Z { '0ns' D/U/Z; } }
41.             out_ports { X { '0ns' X; } }
42.             out_ports { HLT { '0ns' X; '150ns' H/L/T; } }
43.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
44.             bidi_ports { X { '0ns' X; } }
```

```

40.         bidi_ports { HLT { '0ns' X; '100ns' H/L/T; } }
41.     }
42.   } // end SHIFT_TIMING
43. }
44. ScanStructures {
45.   ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
46.   ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
47.   ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
48.   ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
49. } // end scan structures
50. Procedures {
51. "load_unload" {
52.   W "BROADSIDE_TIMING" ;
53.   V {CLOCK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1;
bidi_ports = \r16 Z;}
54.   V {}
55.   V { bidi_ports = \r4 1010 ; }
56.   Shift {
57.     W "SHIFT_TIMING" ;
58.     V { _si=#####; _so=#####; CLOCK=P; }
59.   }
59.   W "BROADSIDE_TIMING" ;
60.   V { CLOCK=0; RESETB=1; SCAN_ENABLE=0; }
61. } // end load_unload
62. } //end procedures
63. MacroDef {
64. "test_setup" {
65.   W "BROADSIDE_TIMING" ;
66.   V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
67.     BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
68.   V {PLL_RESET = 0; }
69.   V {PLL_RESET = 1; }
70. } // end test_setup
71. } //end procedures

```

The lines were added for the following purposes:

- Lines 6–9. Defines some additional signal groups so that timing for all inputs or outputs can be defined in just a few lines, instead of explicitly naming each port and its timing.
- Lines 12–27. This is a waveform table with a period of 1000 ns that defines the timing to be used during nonshift cycles.
- Lines 28–42. This is another waveform table, with a period of 200 ns, that defines the timing to be used during shift cycles.
- Line 52. Addition of the `w` statement ensures that the `BROADSIDE_TIMING` is used for `v` cycles during the `load_unload` procedure.
- Line 57. Addition of the `w` statement ensures that the `SHIFT_TIMING` is used during application of scan chain shifting.
- Line 65. Causes the `test_setup` macro to use `BROADSIDE_TIMING`.

Defining System Capture Procedures

For each declared clock port, TetraMAX uses a default procedure that defines how that port is pulsed for a system (nonscan) test cycle. The name of each procedure is `capture_clockname`, where `clockname` is the clock port name. In addition, there is a procedure named `capture` where no clocks are active. The default procedure usually contains three test cycles for forcing inputs, measuring outputs, and (optionally) pulsing the clock/set/reset port.

If you have defined ports named `CLOCK` and `RESETB` to be clocks, when you issue the `write drc` command, the output file contains default capture procedures similar to those shown in [Example 9-7](#).

Example 9-7 Default Capture Procedures

```
"capture_CLOCK" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
    "pulse": V { "CLOCK"=P; }
}
"capture_RESETB" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
    "pulse": V { "RESETB"=P; }
}
"capture" {
    W "_default_WFT_";
    "forcePI": V { "_pi"=\r10 # ; }
    "measurePO": V { "_po"=#####; }
}
```

If you want to use nondefault timing or sequencing, copy the definitions for the capture procedures from the default output template into the `Procedures` section of your SPF and edit the procedures.

Note:

TetraMAX defaults to the first WaveformTable encountered in the file if a WaveformTable is not specified in the `sequential_capture` procedure when present or defined in a capture procedure in the DRC file. This WaveformTable can be, but does not need to be named `"_default_WFT_"`. In other words, if your SPF had two waveform tables, say `"_first_WFT_"` followed later by `"_default_WFT_"`, and you did not list your capture clocks in the SPF, then TetraMAX would use `"_first_WFT"` for waveform timing information.

The bold text in [Example 9-8](#) shows some typical modifications to the capture procedure files, in which the three cycles are merged into a single cycle and nondefault timing is assigned in the form of the `BROADSIDE_TIMING` that was illustrated in [Example 9-7](#).

Example 9-8 Modified Capture Procedure Examples

```
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING" ;
        V { CLOCK=0; RESETB=1; SCAN_ENABLE=1;
            BIDI_DISABLE=1; bidi_ports = \r16 z; }
        V {}
        V { bidi_ports = \r4 1010 ; }
        Shift {
            W "SHIFT_TIMING" ;
            V { _si=####; _so=####; CLOCK=P; }
        }
        W "BROADSIDE_TIMING" ;
    }

    "capture_CLOCK" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; "CLOCK"=P; }
    }
    "capture_RESETB" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; "RESETB"=P; }
    }
    "capture" {
        W "BROADSIDE_TIMING";
        V { "_pi"=\r10 # ; "_po"=#####; }
    }
}

MacroDefs {
    "test_setup" {
        W "BROADSIDE_TIMING" ;
        V { TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
            BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
        V { PLL_RESET = 0; }
        V { PLL_RESET = 1; }
    }
}
```

Creating Generic Capture Procedures

This section describes the process for writing out a set of single-cycle generic capture procedures. These procedures include: `multiclock_capture()`, `allclock_capture()`, `allclock_launch()`, `allclock_launch_capture()`.

This section covers the following topics:

- [Advantages of Generic Capture Procedures](#)
- [Writing Out Generic Capture Procedures](#)
- [Controlling Multiple Clock Capture](#)
- [Using Allclock Procedures](#)

- Using load_unload for Last-Shift-Launch Transition
- Example Post-Scan Protocol
- Limitations

Advantages of Generic Capture Procedures

Generic capture procedures offer the following advantages:

- The single cycle capture procedure is efficient.
- It matches the event ordering (force PI, measure PO, pulse clock) in TetraMAX without any manual modifications.
- Stuck-at and at-speed ATPG can now use a single common protocol file
- The stuck-at _default_WFT_ WaveformTable is used as a template for modifying the timing of the at-speed WaveformTables.

Writing Out Generic Capture Procedures

To write out a set of generic capture procedures, you will need to specify the `write drc -generic_captures` command. This command suppresses the default-generated procedures (`capture_<clockname>` will not be produced if they were not defined), however, any explicitly defined clocked capture procedure from a prior run `drc` command will still be written out. The unclocked capture procedure will not be written. Also the default timing produced will be written to be compatible with single-cycle capture procedures (a Z event will be produced by default for the measure events H, L, T, and X, at time zero) when this option is used.

WaveformTables. If the default timing is defined, only one WaveformTable will be generated in the output file, and all procedures will reference that same timing. If you desire multiple WaveformTables to be created ("`_launch_WFT_`", "`_capture_WFT_`", and "`_launch_capture_WFT_`"), then use the command `set faults -model transition` or `set faults -model path_delay` before the `write drc` command to identify that the data should be generated to cover this mode of operation.

Note the following:

- It is a requirement to use the generic capture procedures for Internal/External Clocking.
- Capture procedures using the internal clocks must use `_multiclock_capture_WFT_` procedures, which is appropriate because the PLL pulse trains are internally generated independently of the external timing. The timings that should be changed to get at-speed transition fault testing on the external clocks are in the `_allclock_` Waveform Tables (`launch_WFT`, `capture_WFT`, `launch_capture_WFT`). Be careful not to change the

Period or the timings of the Reference Clocks or else the PLLs may lose lock. Only change the rise and fall times of the external clocks. (For more information, see the "Creating Generic Capture Procedures" section in the *DFT Compiler User Guide*.)

- Each 2-clock transition fault test consists of a launch cycle using `_allclock_launch_WFT_` followed by a capture cycle using `_allclock_capture_WFT_`. The active clock edges of these two cycles should be moved close to each other. Make sure that the clock leading edge comes after the `all_outputs` strobe times, and adjust those times (for all values: L, H, T and X) in the `_allclock_capture_WFT_` if necessary. The remaining Waveform Table, `_allclock_launch_capture_WFT`, is only used when launch and capture are caused by opposite edges of the same clock. Here, the only important timing is from the clock leading edge to the same clock's trailing edge. In practice, this only happens in Full-Sequential ATPG. and in most cases it can be ignored.

Controlling Multiple Clock Capture

You can control multiple clock capture by specifying a single general capture procedure, called `multiclock_capture`, in an SPF file. This procedure enables you to map all capture behaviors irrespective of the number of clocks present, to use this procedure. In addition to supporting capture operations that contain multiple clocks, this procedure also eliminates the need to manually define a full set of clock-specific capture procedures or allow them to be defined by default.

There are several different methods associated with specifying multiple clock capture:

- [Specifying for a Single Vector](#)
- [Specifying for Multiple Vectors](#)
- [Specifying Macros for BIST Operation](#)
- [Using Multiple Capture Procedures](#)

Specifying for a Single Vector. The following example shows how to specify `multiclock_capture` for a single vector, which is the simplest form of this procedure:

```
Procedures {
    "multiclock_capture" {
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_po"=\r9 # ; "_pi"=\r11 # ; }
    }
}
```

Note that the single vector form does not require an explicit parameter to support the clock pulses because the clocks are always listed in the `_pi` arguments, and also in the `_po` arguments for any clocks that are bidirectional. It is strongly recommended that you specify

an initial Condition statement to set the _po states to an X in this procedure. A default value should be present in this procedure because not all calls from the Pattern data provide explicit output states.

As is the case with all capture procedures, the single-vector form of multiclock_capture requires the timing in the WaveformTable to follow the TetraMAX event order for captures. This means that all input transitions must occur first, all output measures must occur next, and all clock pulses must be defined as the last event.

Specifying for Multiple Vectors. As with standard capture procedures, the multiclock_capture procedure can consist of multiple vectors. In this case, you need to specify an additional argument to hold the variable clock-pulse information, as shown in the following example:

```
Procedures {
    "multiclock_capture" { // 2-cycle
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_pi"=\r11 # ; "_po"=\r9 # ; }
        C { "_po"=\r9 X ; }
        V {"_clks"= ###; }
    }
    "multiclock_capture" { // 3-cycle
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_pi"=\r11 # ; }
        V { "_po"=\r9 # ; }
        C { "_po"=\r9 X ; }
        V {"_clks"= ###; }
    }
}
```

Specifying Macros for BIST Operation. In BIST patterns, most functions are defined as macros instead of procedures. Accordingly, in TetraMAX you can define generic clocked capture procedures as macros instead of procedures.

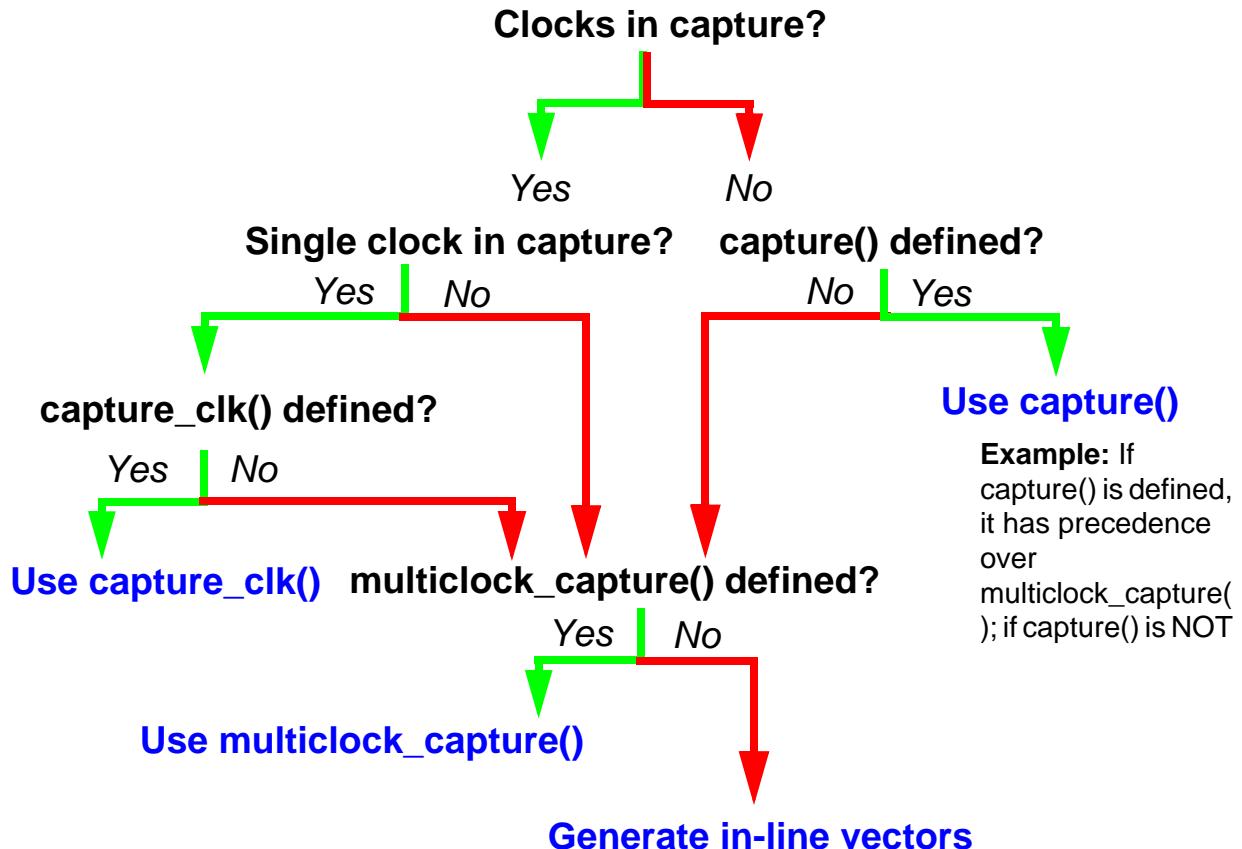
BIST patterns also typically assert only clock pulses for capture operations and rely on the previous context for all other signal states (from the point of view of STIL incremental vector constructs). Therefore, in BIST contexts the only parameter you can specify in the function is a group containing only the clocks of the design. In this case, it is strongly recommend that you define the capture operation containing the clock group, as shown in the following example:

```
MacroDefs {
    "multiclock_capture" {
        W "TS1";
        V { "_clks"= ### ; }
    }
}
```

}

Using Multiple Capture Procedures. Figure 9-2 shows how the `multiclock_capture` procedure is used when other `capture()` or `capture_clk()` procedures are defined.

Figure 9-2 Using Multiple Capture Procedures



Using Allclock Procedures

Allclock procedures directly replace specifically named WaveformTables (WFTs) by designating launch, capture, and launch_capture-specific timing parameters. This approach replaces an inline vector and WFT switch with a procedure call.

You can specify a set of allclock procedures that are used in specific contexts in which a sequence of capture events supports a launch and capture operation. These sequences are generated in system-clock-launched transition tests. Full-sequential patterns, including path delay, will use in-line vectors and not procedure calls. This is because the full-sequential operation has dependencies on the sequential_capture definition, which affects how capture

operations will occur. Because of the use of in-line vectors, transition and path delay timing is controlled with the use of fixed WaveformTable names (and not the allclock capture procedures) for full-sequential patterns.

Note that last-shift-launch contexts do not identify the launch or the capture operation. This means a last-shift-launch uses a standard capture procedure designation and does not reference allclock procedures even if they are present.

Standard capture procedure designation will apply `multiclock_capture` in this situation if it is present (based on the presence of other capture procedures as diagrammed in [Figure 9-2](#)), and you may define the timing of the transition capture operation from this procedure. The timing of the launch operation will be defined by the last vector of the `load_unload` procedure for a last-shift-launch context.

TetraMAX supports the following allclock procedures:

- `allclock_capture()` — Applies to tagged capture operations in launch/capture contexts only.
- `allclock_launch()` — Applies to tagged launch operations in launch/capture contexts only
- `allclock_launch_capture()`— Applies to tagged launch-capture operations only.

Specifying a Typical Allclock Procedure. By default, an allclock procedure applies to a single vector, although it doesn't have to carry the redundant clock parameter. An allclock procedure may reference any WFT for each operation. See the example `allclock_capture()` procedure below:

```
Procedures {
    "allclock_capture" {
        W "TS1";
        C { "_po"=\r9 X ; }
        V { "_po"=\r9 # ; "_pi"=\r11 # ; }
    }
}
```

Specifying Allclock Procedures and BIST. in BIST contexts, the only parameter you can specify in a function is a group containing the clocks of the design. Also, as noted earlier, a function is defined as a macro instead of a procedure.

In last-shift-launch contexts, you can assert a `_pi` group during the launch, and measure a `_po` group during the capture. In these situations, the appropriate parameters must be present in the allclock routines, as shown in the following examples:

```
MacroDefs {
    "allclock_launch" {
        W "launch";
```

```

    C { "_po"=\r9 X ; }
    V { "_pi"=\r11 # ; } // clocks are part of _pi
}
"allclock_capture" {
    W "supper";
    V { "_po"=\r9 #; "_ck"=\r3 #; }
}
}

```

Interaction of the Allclock and Multiple Clock Procedures. A defined `multiclock_capture()` procedure is always used for any capture operation that is not controlled by another defined procedure. This means that if an allclock procedure is not defined, the `multiclock_capture` procedure is applied in its place.

Interaction of Allclock Procedures and Named Waveform Tables. If an allclock procedure is defined, a named WFT is not applied on inline vectors even if it is defined. This is because allclock procedures always replace the generation of inline vectors in pattern data, and WFT names are supported only when inline vectors are generated.

It is strongly recommended that you define a sufficient set of allclock procedures for a particular context, even if the procedures are identical. This preserves pattern operation information that might otherwise be difficult to identify.

Using `load_unload` for Last-Shift-Launch Transition

The `load_unload` procedure supports passing the `pi` data into the first vector of the `load_unload` operation. This means `load_unload` supports last-shift-launch transition tests, presents the leading PI states at the time of the last shift operation (the launch), and supports transitioning those states.

Because of this implementation, it is important to provide sufficient information as part of the `load_unload` definition to permit stand-alone operation of the `load_unload` procedure. It is important to consider that the `load_unload` procedure is also used to validate scan chain tracing. Required states on inputs necessary to support scan chain tracing must be provided to this routine even if these signals are subsequently presented as parameterized values to the procedure.

For example,

```

Procedures {
    "load_unload" {
        W "_default_WFT_";
        C { "test_se"=1; } // required for scan chain tracing
        V { "_pi"=\r34 #; }
        Shift {
            V { "_ck"=\r3 P; "_si"=\r8 #; "_so"=\r8 #; }
        }
    }
}

```

```
}
```

Example Post-Scan Protocol

```
Procedures {
    "multiclock_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = ####;
            "_pi" = \r9 #;
        }
    }
    "allclock_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = ####;
            "_pi" = \r9 #;
        }
    }
    "allclock_launch" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = ####;
            "_pi" = \r9 #;
        }
    }
    "allclock_launch_capture" {
        W "_default_WFT_";
        C {
            "_po" = XXXX;
        }
        V {
            "_po" = ####;
            "_pi" = \r9 #;
        }
    }
    "load_unload" {
        W "_default_WFT_";
        C {
            "all_inputs" = NN0011NN1; // moved scan enable here
            "all_outputs" = XXXX;
        }
    }
    "Internal_scan_pre_shift" : V {   "_pi" = \r9 #; }
    Shift {
}
```

```

    V {
        "_clk" = PP11;
        "_si" = ##;
        "_so" = ##;
    }
}
}
}

```

Limitations

Note the following limitations related to generic capture procedures:

- WGL patterns are not supported if the multiclock_capture is multiple cycle and/or the clock (_clk) parameter is used; in this case, the WGL will not contain the clock pulses. WGL pattern format is only supported with single-cycle multiclock_captures that do not use a "clock" parameter (_clk).
- WGL and legacy Verilog formats do not support 3-cycle generic capture procedures.
- Using the DFT Compiler flow, the timing from the _default_WFT_ waveform table is copied to the allclock waveform tables (launch_WFT, capture_WFT, launch_capture_WFT). You will need to modify these multiple identical copies of this information with the correct timing before running at-speed ATPG.
- TetraMAX transition delay ATPG using the command set delay -launch_cycle last_shift is not supported with the allclock capture procedures, only system_clock launch is supported.
- Muxclock is not supported (D, E, P waveforms).
- Using the TetraMAX only flow, these generic capture procedures are all non-default procedures; that is, they must be manually defined in the SPF in order to be used in the patterns.

Defining a Sequential Capture Procedure

A sequential capture procedure lets you customize the capture clock sequence applied to the device during Full-Sequential ATPG. For example, you can define the clocking sequence for a two-phase latch design, where CLKP1 is followed by CLKP2. Using a sequential capture procedure is optional, and it only affects Full-Sequential ATPG, not Basic-Scan or Fast-Sequential ATPG. (For more information on ATPG modes, see “[ATPG Modes](#)” on page 12-2.)

With Full-Sequential ATPG and a sequential capture procedure, the relationships between clocks, tester cycles, and capture procedures can be more flexible and more complex. Using Basic-Scan ATPG results in one clock per cycle, one clock per capture procedure, and one capture procedure per TetraMAX ATPG pattern. Using Full-Sequential ATPG and a

sequential capture procedure, a cycle can be defined with one or more clocks, a capture procedure can be defined with any number of cycles, and an ATPG pattern can contain multiple capture procedures.

A sequential capture procedure can pulse multiple clocks, define clocks that overlap, and specify both optional and required clock pulses. A very long or complex sequential capture procedure is more computationally intensive than a simple one, which can affect the Full-Sequential ATPG runtime.

Using the Default Capture Procedures

By default, all ATPG modes use the same `capture_clockname` procedures described in ["Defining System Capture Procedures" on page 9-17](#). The Full-Sequential algorithm assumes the same order of events for each vector as the other algorithms. Under these default conditions, the Full-Sequential algorithm uses a fixed capture cycle consisting of three time frames, in which the tester does the following:

1. (Optional) Loads scan cells, changes inputs, and measures outputs
2. Applies a leading clock edge
3. Applies a trailing clock edge, and optionally unloads scan cells

The Full-Sequential ATPG algorithm can choose any one of the available capture procedures for each vector, including the one that does not pulse any clocks. The algorithm can produce patterns using any sequence of these capture procedures in order to detect faults.

Using a Sequential Capture Procedure

To use a sequential capture procedure, add a procedure called `sequential_capture` to the STIL file, and use that procedure by setting the `-clock -seq_capture` option of the `set drc` command, as in the following example:

```
DRC> set drc -clock -seq_capture
```

Using this command option causes the Full-Sequential ATPG algorithm to use only the sequential capture procedure and to ignore the `capture_clockname` procedures defined by the STIL file or the `add_clocks` command. This option has no effect on the Basic-Scan and Fast-Sequential algorithms. See the online help for additional information about sequential capture procedures in STIL.

Sequential Capture Procedure Syntax

A sequential capture procedure can be composed of one or more vectors. You can specify each vector using any of the following events:

- Force PI (must occur before clock pulses; required for the first vector)
- Measure PO (may occur before, during, or after clock pulses)
- Clock pulse (no more than one per clock input)

Each vector corresponds to a tester cycle. Be sure to consider any hardware limitations of the test equipment when you write the sequential capture procedure.

You can specify an optional clock pulse, which means that the clock is not required to be pulsed in every sequence. The Full-Sequential ATPG algorithm determines when to use or not use the clock. To define such a clock pulse, use the following statement:

```
V {"clock_name"=#;}
```

You can specify a required (mandatory) clock pulse, which means that the clock must be pulsed in every capture sequence. To define such a clock pulse, use the following statement:

```
V {"clock_name"=P;}
```

Here is an example of a sequential capture procedure:

```
"sequential_capture"
    W "_default_WFT_";
    F {"test_mode"= 1; }
    V {"_pi"= \r48 #; "_po"= \r12 X ; }
    V {"CLK1"= P; CLK2= #; }
    V {"CLK3"= P; }
    V {"_po"= \r12 #; }
}
```

A sequential capture procedure can contain multiple tester cycles by supporting one or more vectors (multiple `V` statements), but there can be only one `WaveformTable` reference (`W` statement).

The procedure can have one force PI event per input per vector. Each force PI event must occur before any clock pulse events in that cycle. All inputs must be forced in the first vector of the sequential capture procedure; each input holds its state in subsequent vectors unless there is an optional change caused by another force PI event.

The procedure can have one required (`=P`) or optional (`=#`) clock pulse event per clock input per vector. Nonequivalent clocks can be pulsed at different times, and these clock pulses can overlap or not overlap.

The procedure can have one measure PO event per output per vector, which can occur anywhere in the cycle. However, no input or clock pulse events can be specified between the earliest and latest output measurements. The procedure also supports equivalence relationships and input constraints (`E` and `F` statements).

Sequential ATPG and simulation can model input changes only in the first time frame of each cycle. TetraMAX adds more time frames only as necessary to model discrete clock pulse events. It strobes outputs in no more than one of the existing time frames for each cycle.

Defining the End-of-Cycle Measure

The preferred ATPG cycle has the measure point coming before any clock events in the cycle. However, an end-of-cycle measure is possible with a few minor adjustments to the SPF.

The SPF of [Example 9-9](#) illustrates the two changes that allow TetraMAX to accommodate an end-of-cycle measure:

- The timing of the measure points defined in the `Waveforms` section is adjusted to occur after any clock pulses.
- A measure scan out ("`_so"="###` ") is placed within the `load_unload` procedure and before the `Shift` procedure.

In addition, the capture procedures must be either the default of three cycles or a two-cycle procedure where the force/measure events occur in the first cycle and the clock pulse occurs in the second.

Example 9-9 End-of-Cycle Measure

```
Timing {
    WaveformTable "BROADSIDE_TIMING" {
        Period '1000ns';
        Waveforms {
            measures { X { '0ns' X; } }
            CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
            CLOCK { 01Z { '0ns' D/U/Z; } }
            RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
            RESETB { 01Z { '0ns' D/U/Z; } }
            input_grp1 { 01Z { '0ns' D/U/Z; } }
            input_grp2 { 01Z { '10ns' D/U/Z; } }
            bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
            measures { HLT { '0ns' X; '950ns' H/L/T; } }
        }
    }
    WaveformTable "SHIFT_TIMING" {
        Period '200ns';
        Waveforms {
            measures { X { '0ns' X; } }
            CLOCK { P { '0ns' D; '100ns' U; '150ns' D; } }
            CLOCK { 01Z { '0ns' D/U/Z; } }
            RESETB { P { '0ns' U; '20ns' D; '180ns' U; } }
            RESETB { 01Z { '0ns' D/U/Z; } }
            in_ports { 01Z { '0ns' D/U/Z; } }
            bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
            measures { HLT { '0ns' X; '190ns' H/L/T; } }
        }
    }
}
ScanStructures {
    ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
    ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
    ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
    ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING";
        V { CLOCK=0; RESETB=1; SCAN_ENABLE=1;
            BIDI_DISABLE=1; bidi_ports = \r16 Z; }
        V { "_so" = #####; }
        V { bidi_ports = \r4 1010; }
        Shift {
            W "SHIFT_TIMING";
            V { _si=####; _so=####; CLOCK=P; }
        }
    }
}
```

Defining Constrained Primary Inputs

Primary inputs that need to be held to a constant value can be defined in the SPF as an alternative to using the add pi constraints command or the Constraints menu.

In [Example 9-10](#), the following line has been added to each clock capture procedure to define the “fixed” ports:

```
F { TEST_MODE = 1; PLL_TEST_MODE = 1; }
```

These F statements indicate that the ports listed should be fixed (held at constant values) throughout the current capture procedure. All F statements must be identical in every capture procedure.

Example 9-10 SPF: Defining Constrained Ports

```
Procedures {
    "capture_CLOCK" {
        W "BROADSIDE_TIMING";
        F { TEST_MODE = 1; PLL_TEST_MODE = 1; }
        V { "_pi"=\r10 # ; "_po"=#####; "CLOCK"=P; }
    }
    "capture_RESETB" {
        W "BROADSIDE_TIMING";
        F { TEST_MODE = 1; PLL_TEST_MODE = 1; }
        V { "_pi"=\r10 # ; "_po"=#####; "RESETB"=P; }
    }
    "capture" {
        W "BROADSIDE_TIMING";
        F { TEST_MODE = 1; PLL_TEST_MODE = 1; }
        V { "_pi"=\r10 # ; "_po"=#####; }
    }
}
```

Defining Equivalent Primary Inputs

Primary inputs that need to be held at the same values or at complementary values can be defined in the SPF as an alternative to using the add pi equivalences command.

[Example 9-11](#) shows how to define equivalent primary inputs in the SPF.

Example 9-11 SPF: Defining Equivalent Ports

```
Procedures {
    "capture" {
        W "_default_WFT_";
        E "ck1" "ck2";
        C { "all_inputs"=\r30 N; "all_outputs"=\r30 X ; {}}
        V { "_pi"=\r35 # ; "_po"=\r30 # ; {}}
    }
    "capture_ck1" {
        W "_default_WFT_";
        E "ck1" "ck2";
        C { "all_inputs"=\r30 N; "all_outputs"=\r30 X ; {}}
        "measurePO": V { "_pi"=\r35 # ; "_po"=\r30 # ; {}}
        C { "InOut1"=X; "PA1"=X; "DOA"=X; "NA1"=X; "NA2"=X; {}}
        "pulse": V { "ck1"=P; {}}
    }
    "load_unload" {
```

Defining Reflective I/O Capture Procedures

A few ASIC vendors have special requirements for the application of the tester patterns when the design contains bidirectional pins. These vendors require the design to contain a global disable control, available in ATPG test mode, which is used to turn off all potential bidirectional drivers. Further, the following sequence is required during the application of nonshift clocking and nonclocking capture procedures:

1. Force primary inputs with bidirectional ports enabled
2. Measure values on outputs as well as bidirectional ports
3. Disable bidirectional drivers
4. Use tester to force bidirectional ports with values measured in step 2
5. (Optional) Apply clock pulse

You tell TetraMAX which port acts as the global bidirectional control by using the `-bidi_control_pin` option of the `set drc` command. For example, to indicate that the value 0 on the port `BIDI_EN` disables all bidirectional drives, enter:

```
DRC> set drc -bidi_control_pin 0 BIDI_EN
```

To define the corresponding reflective I/O capture procedures, you use `%` characters instead of `#` as data placeholders. In [Example 9-11](#), each capture procedure measures primary outputs with the string `%%%%%` instead of the string `#####`. A few cycles later, the string `%%%` appears in an assignment of the symbolic group `"_io"`, which is shorthand for the bidirectional ports.

The number of ports in the `"_po"` symbolic list is usually larger than the set of bidirectional ports referenced by `"_io"`, so it is common for the `%%%%%` string for `"_po"` to be longer than the string for the `"_io"` reference where the reflected data is reapplied. TetraMAX understands the correspondence needed for proper pattern data.

Example 9-12 Capture Procedures With Reflective I/O Syntax

```
"capture_CLOCK" {
    W "_default_WFT_"; // force PI, measure PO, BIDI_EN=1
    V {"_pi"=\r10 # ; "_po"=%%%%%% ; } // disable bidis, mask
PO measures
    V { BIDI_EN=0; "_po"=XXXXXX; } // reflect bidis, pulse CLOCK
    V { "_io"=%%% ; CLOCK=P; }

}

capture_RESETB {
    W "_default_WFT_"; // force PI, measure PO, BIDI_EN=1
    V {"_pi"=\r10 # ; "_po"=%%%%%% ; } // disable bidis, mask
PO measures
    V { BIDI_EN=0; "_po"=XXXXXX; } // reflect bidis, pulse RESETB
    V { "_io"=%%% ; RESETB=P; }

}

capture {
    W "_default_WFT_";
    V {"_pi"=\r10 # ; "_po"=##### ; } // force PI, measure PO
    V { "_po"=XXXXX; } // mask measures
    V { } // pad procedure to 3 cycles
}
```

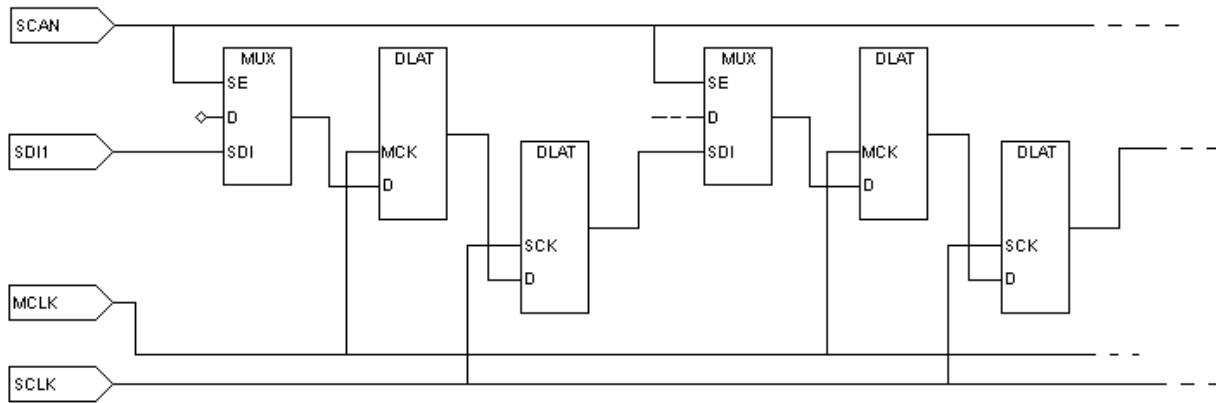
Defining Other STIL Procedures

Some design styles require additional STIL procedures. Two of the more common are the `master_observe` and `shadow_observe` procedures.

The `master_observe` Procedure

Use the `master_observe` procedure if the design has separate master-slave clocks to capture data into scan cells, as shown in [Figure 9-3](#). In system (nonscan) mode, after applying the `capture_clockname` procedure corresponding to the master clock, you must apply the slave clock to propagate the data value captured from the master latches to the slave latches. In the `master_observe` procedure, you describe how to pulse the slave clock and thereby observe the master.

Figure 9-3 Master-Slave Scan Chain



[Example 9-13](#) shows a `master_observe` procedure that uses two tester cycles. In the first cycle, all clocks are off except for the slave clock, which is pulsed. In the second cycle, the slave clock is returned to its off state.

Example 9-13 Sample master_observe Procedure

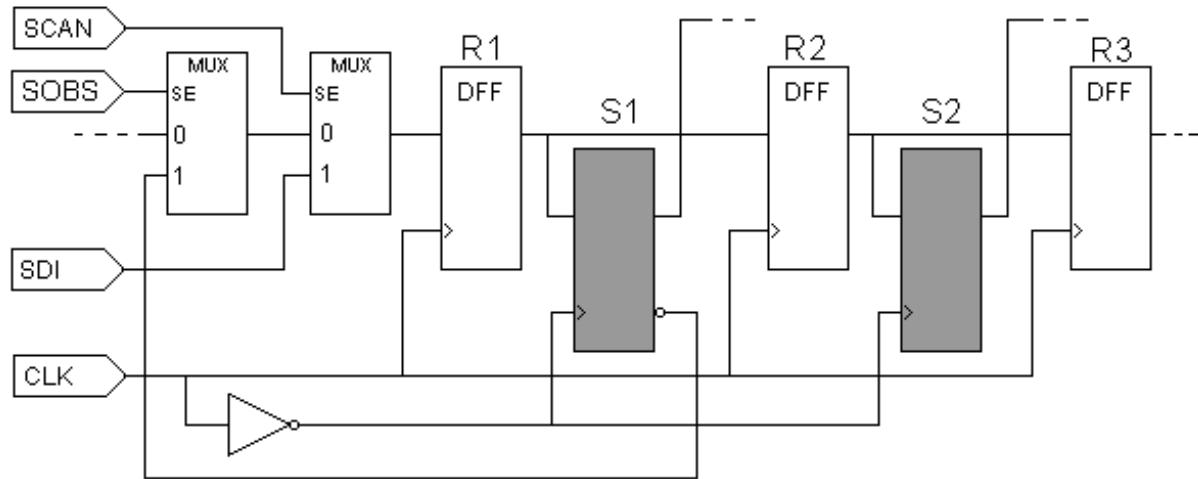
```
Procedures {
    "load_unload" {
        W "BROADSIDE_TIMING"
        V { MCLK=0; SCLK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1; }
        V { bidi_ports = \r16 Z ; }
        Shift {
            W "SHIFT_TIMING";

            V { _si=##; _so=##; MCLK=P; SCLK=0; }
            V { MCLK=0; SCLK=P; }
        }
        V { SCLK=0; }
    }
    master_observe {
        W "BROADSIDE_TIMING";
        V { MCLK=0; SCLK=P; RESETB=1; }
        V { SCLK=0; }
    }
}
```

The shadow_observe Procedure

A `shadow_observe` procedure is used when a design has shadow registers and each shadow register output is observable at the scan cell to which it is a shadow. [Figure 9-4](#) shows two shadow registers, S1 and S2, which are shadows of R1 and R2, respectively. Shadow S1 has a combinational path back to its scan cell and would benefit from the definition of a `shadow_observe` procedure. Shadow S2 does not have a path back to R2 and would not benefit from a `shadow_observe` procedure.

Figure 9-4 A Shadow Register



[Example 9-14](#) shows a `shadow_observe` procedure that corresponds to [Figure 9-4](#). The first cycle places all clocks at off states and sets up the path from `S1` back to `R1` by setting `SCAN=0` and `SOBS=1`. The second cycle pulses the `CLK` port, and the third cycle turns off `CLK` and returns `SOBS` to zero.

Example 9-14 Sample shadow_observe Procedure

```
Procedures {
    load_unload {
        V { CLK=0; RSTB=1; SCAN=1; }
        Shift { V { _si=##; _so=##; CLK=P; } }
        V { CLK=0; }
    }

    shadow_observe {
        V { CLK=0; RSTB=1; SCAN=0; SOBS=1; }
        V { CLK=P; }
        V { CLK=0; SOBS=0; }
    }
}
```

Using Quick STIL Commands with an On-Chip Clock Controller

The following Quick STIL commands are used for a default DFT Compiler-inserted On-Chip Clock (OCC) controller:

```
add scan chains ...
add scan enables 1 test_se
add pi constraints 1 test_mode
add pi constraints 0 test_se pll_reset pll_bypass
set drc -num_pll_cycles ...
add clocks 0 <EXTERNAL_CLOCKS> -shift -timing <4_VALUES>
```

```

add clocks0 <ATE_AND_REF_CLOCKS> -shift -refclock -timing <4_VALUES>
add clocks 0 <ASYNC_REF_CLOCKS> -refclock -ref_timing <3_VALUES>
add clocks 0 <PLL_CLOCKS> -pllclock
add clocks 0 <INT_CLOCK> -intclock -pll_source <PLL_CLOCK> -cycle ...
write drc temp.spf -generic_captures

```

Make the following changes from the temp.spf file to the final protocol file:

1. Copy and paste the entire WaveformTable (WFT) "_default_WFT_" { ... } block four times.
2. Rename new WFT blocks as follows:

```

"_multiclock_capture_WFT_"
"_allclock_capture_WFT_"
"_allclock_launch_WFT_"
"_allclock_launch_capture_WFT_"

```

3. Change the WFT for each procedure (except load_unload) as follows:

```

"multiclock_capture" { W "_multiclock_capture_WFT_" ;
"allclock_capture" { W "_allclock_capture_WFT_" ;
"allclock_launch" { W "_allclock_launch_WFT_" ;
"allclock_launch_capture" { W "_allclock_launch_capture_WFT_" ;

```

4. In load_unload, add the following just before Shift loop, with only ATE_CLOCKS and SYNC_REF_CLOCKS specified:

```
v { "clkate"=P; "clkref0"=P; }
```

5. In test_setup, copy the v statement and do the following:

- Change the polarity of pll_reset in the first v statement
- Change 0 to P for all ATE_CLOCKS and SYNC_REF_CLOCKS in both v statements

6. Change the timing of the WFTs as desired.

Note the following:

- To avoid step 5, create a legal SPF with a correct test_setup, and run the command set command noabort and the command run drc -append before specifying the write drc command. The DRC run fails because step 4 must be done first. The result appears to be correct, but you will need to determine if this is easier than hand-editing temp.spf.
- Step 6 can be done in an editor, or another TMAX run can be made to use the update wft and update clock commands.

Testing the STIL Procedure File

After completing edits to the SPF, reread the file and perform syntax checking of your procedures by executing the following command:

```
DRC> run drc filename.spf
```

During DRC, TetraMAX simulates each procedure and checks it against a number of rules. Should you encounter a DRC violation that requires editing the SPF file, you can edit and save the file and use just the `run drc` command (without the file name) to test your changes, as follows:

```
DRC> run drc
```

TetraMAX rereads the SPF file each time the `run drc` command is executed.

Not all DRC violations are severe enough to stop TetraMAX from entering TEST command mode. To rerun DRC when in TEST command mode, first return to SETUP command mode, as follows:

```
TEST> drc  
DRC> run drc
```

STIL Procedure File Guidelines

This section contains some general guidelines, tips, and shortcuts for working with SPFs.

- To save time and avoid typing errors, use the `write drc` command to create the STIL template. The more information that you provide to TetraMAX before the `write drc` command, the more TetraMAX will provide in the template. If possible, build your design model and define all clock and constrained inputs before you create the STIL template.
- Recall that STIL keywords are case-sensitive. All keywords start with an uppercase character, and many contain more than one uppercase character.
- Use `SignalGroups` to define groups of ports so that you can easily assign values and timing.
- At the beginning of the `load_unload` procedure, always place the ports declared as clocks in their off states.
- Except for the `test_setup` and `Shift` procedures, every procedure should include initializing all clocks to their off state and all PI constraints and PI equivalences to their proper values at the beginning of the procedure.

- If you have constrained ports or bidirectional ports, define a `test_setup` macro and initialize the ports.
- A `test_setup` procedure must initialize all clocks to their off states, and all PI constraints and PI equivalences to their proper values, by the end of the procedure. Note that it is not necessary to stop Reference clocks (including what DFT Compiler refers to as ATE clocks). All other clocks still must be stopped.
- Bidirectional ports should be forced to Z within a `test_setup` macro and forced to Z at the beginning of the `load_unload` procedure.
- For non-JTAG designs, it is usually not necessary to apply a reset to the design within a `test_setup` macro.
- When defining pulsed ports, define the 0/1/Z mapping for cycles when the clock is inactive, as in the following example:

```
CLOCK { 01Z { '0ns' D/U/Z; } }
```

JTAG/TAP Controller Variations for the `load_unload` Procedure

The `load_unload` procedure defines how to place the design into a state in which the scan chains can be loaded and unloaded. This typically involves asserting a scan-enable input or other control line and possibly placing bidirectional ports into the Z state. Standard DRC rules also require that ports defined as clocks be placed in their off states at the start of the scan chain load/unload process.

Final Shift With TMS=1

In designs that use the test access port (TAP) controller to set up internal scan chain access or boundary scan access, it is very common to need to perform the very last scan shift with the test mode select (TMS) port asserted. This is accomplished by placing as many scan chain force and measure events outside of the `Shift` procedure as necessary. Usually only one final force/measure event is needed.

The bold text in [Example 9-15](#) shows one additional scan chain force and measure placed outside of the `Shift` procedure. For a scan chain length of N , TetraMAX performs $N-1$ shifts using the vector inside the `Shift` procedure, and the final shift using the vector which follows, where TMS=1.

Example 9-15 JTAG/TAP Controller Adjustments to load_unload

```
Procedures {
    "load_unload" {
        V { TMS=0; TCK=0; CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
        Shift {
            V { _si=####; _so=####; TCK=P; }
        }
        V { TMS=1; _si=####; _so=####; TCK=P; }
    }
}
```

For more examples of `load_unload` procedures, see the online help on the topic “JTAG.”

Multiple Scan Groups

You should set up TetraMAX for multiple scan-group support if your design has multiple scan chains that cannot be accessed simultaneously (for example, they share the same I/O pins). TetraMAX supports designs that have multiple scan groups by using IEEE Std. 1450.1 extensions to STIL.

If you have a design with multiple scan groups that must be accessed in serial, not in parallel, during the `load_unload` process, perform the following steps.

1. Define multiple `ScanStructure` blocks.

Each `ScanStructure` block defines one scan chain group. Use a unique label for each scan chain group. In [Example 9-17 on page 9-42](#), the `ScanStructure` labels are `g1`, `g2`, `g3`, and `g4`. TetraMAX also requires each `ScanChain` label to be unique across all scan chain definitions.

2. Add `ScanStructure` statements to the `load_unload` procedure.

Within the `load_unload` procedure, add the `ScanStructure` statement ahead of any scan input or scan output references. The `ScanStructure` statement identifies the scan group label that is active for any lines that follow.

3. Reference scan inputs and outputs with symbolic labels.

Within the `load_unload` and `Shift` procedures, reference the appropriate set of scan inputs and scan outputs with symbolic labels: `_si1`, `_so1`, `_si2`, `_so2`, and so on.

TetraMAX associates these symbolic labels with the scan inputs and scan outputs of the appropriate scan group. You are not required to use the `_so` prefix on scan output symbolic labels, but if you use the `_so` prefix, you must also use the `_si` prefix on symbolic labels for the scan input.

If STIL patterns will be written out from this data, then each scan signal in each Shift Vector needs a unique symbolic label. A V14 warning will be generated when this constraint is not followed, identifying the signal that needs a unique symbolic label. In most other situations, all scan signals can be referenced with a single symbolic label, such as `Shift { V { _si=##; _so=##; ... } }`. However, if STIL patterns will be written out, then each scan signal used across more than one scan group will require a separate symbolic label in order to associate scan data with this specific scan block. [Example 9-15](#) does not follow this constraint (and would generate V14 warnings which may be ignored if STIL patterns are not generated), however [Example 9-16](#) (with only one scan chain per Shift) does. [Example 9-16](#) demonstrates a multiple scan chain per Shift implemented with this restriction.

When symbolic labels must be associated with individual scan signals, it is necessary to define a `SignalGroups` block to establish these associations and define the symbolic labels. [Example 9-16](#) identifies the necessary `SignalGroup` definitions that must be part of this context.

Example 9-16 Four Scan Groups Structured for STIL Pattern Generation

```
STIL 1.0;
SignalGroups {
    _s11="SDI[1]" {ScanIn;} _s12="SDI[2]" {ScanIn;} _s13="SDI[3]" 
    {ScanIn;}
    _s01="SDO[1]" {ScanOut;} _s012="SDO[2]" {ScanOut;} _s013="SDO[3]" 
    {ScanOut;}
    _s21="SDI[1]" {ScanIn;} _s22="SDI[2]" {ScanIn;} _s23="SDI[3]" 
    {ScanIn;}
    _s021="SDO[1]" {ScanOut;} _s022="SDO[2]" {ScanOut;} _s023="SDO[3]" 
    {ScanOut;}
    _s31="SDI[1]" {ScanIn;} _s32="SDI[2]" {ScanIn;} _s33="SDI[3]" 
    {ScanIn;}
    _s031="SDO[1]" {ScanOut;} _s032="SDO[2]" {ScanOut;} _s033="SDO[3]" 
    {ScanOut;}
    _s41="SDI[1]" {ScanIn;} _s42="SDI[2]" {ScanIn;} _s43="SDI[3]" 
    {ScanIn;}
    _s041="SDO[1]" {ScanOut;} _s042="SDO[2]" {ScanOut;} _s043="SDO[3]" 
    {ScanOut;}
}

ScanStructures g1 {
    ScanChain g1_0 { ScanIn "SDI[1]"; ScanOut "SDO[1]"; }
    ScanChain g1_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain g1_2 { ScanIn "SDI[3]"; ScanOut "SDO[3]"; }
}
ScanStructures g2 {
    ScanChain g2_0 { ScanIn "SDI[2]"; ScanOut "SDO[1]"; }
    ScanChain g2_1 { ScanIn "SDI[3]"; ScanOut "SDO[2]"; }
    ScanChain g2_2 { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
}
ScanStructures g4 {
```

```

ScanChain g4_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }

ScanChain g4_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
ScanChain g4_2 { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
}

ScanStructures g3 {

ScanChain g3_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }

ScanChain g3_1 { ScanIn "SDI[1]"; ScanOut "SDO[2]"; }

ScanChain g3_2 { ScanIn "SDI[2]"; ScanOut "SDO[3]"; }
}

Procedures {
load_unload {

V { mclk=0; clk=0; rst=1; scan_en=1; inc=0; }
V { mode=0; }
V { chain_sel = 0; mclk=P; }
V { }

ScanStructures g1;
"single_shift0:" V { _s11=#; _s12=#; _s13=#; _s011=#;
_s012=#; _s013=#; clk=P; mclk=0; }

Shift { ScanStructures g1; V { _s11=#; _s12=#; _s13=#; _s011=#;
_s012=#; _s013=#; clk=P; } }

"single_shift1:" V { _s11=#; _s12=#; _s13=#; _s011=#; _s012=#;
_s013=#; clk=P; }

V { chain_sel = 0; mclk=P; clk=0; }

V { chain_sel = 1; }

V { mclk=0; }

ScanStructures g2;

Shift { V { _s121=#; _s122=#; _s123=#; _s021=#; _s022=#; _s023=#; clk=P;
} }

"single_shift2:" V { _s121=#; _s122=#; _s123=#; _s021=#; _s022=#; _s023=#;
clk=P; }

V { chain_sel = 1; mclk=P; clk=0; }

ScanStructures g3;

V { chain_sel = 0; mclk=P; }

Shift { V { _s131=#; _s132=#; _s133=#; _s031=#; _s032=#; _s033=#; clk=P;
mclk=0; } }
V { clk=0; }

V { chain_sel = 1; mclk=P; }
}

```

```

V { }

ScanStructures g4;

Shift { V { _si41=#; _si42=#; _si43=#; _so41=#; _so42=#; _so43=#; clk=P;
mclk=0; } }

V { clk=0; mclk=0; mode=1; }
}

MacroDefs {
    "test_setup" {

V { "mclk"=0; "clk"=0; "rst"=1; scan_en=0; inc=0; mode=1; }
}
}

```

[Example 9-16](#) identifies the necessary expansion to the symbolic references, to support proper STIL pattern generation of a design containing scan groups that are sequentially shifted. This example shows:

- The SignalGroups definitions necessary to support association of the individual signals in the load_unload procedure.
- The use of the symbolic references in the load_unload procedure to reference individual scan signals.
- The presence of pre-shift and post-shift vectors that also consume scan data. Look for the labels single_shift0, single_shift1, and single_shift2 in the [Example 9-16](#).

Note:

It is a DFT requirement that the scan cells of one scan group not be disturbed during the scan shifting of other scan groups. You must consider this restriction when you plan to use multiple scan groups.

[Example 9-17](#) illustrates syntax for a design with four different groups of scan chains that must be accessed serially during the load_unload process.

Example 9-17 Four Scan-Chain Groups Loaded Serially

```

ScanStructures g1 {
    ScanChain g1_0 { ScanIn "SDI[1]"; ScanOut "SDO[1]"; }
    ScanChain g1_1 { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }
    ScanChain g1_2 { ScanIn "SDI[3]"; ScanOut "yama"; }
}

ScanStructures g2 {
    // STIL allows same chain name in another group,
    // but TMAX does not
    ScanChain GROUP2_0 { ScanIn "SDI[2]"; ScanOut "data23"; }
    ScanChain GROUP2_1 { ScanIn "SDI[3]"; ScanOut "SDO[2]"; }
}

ScanChain "g4_0" { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
ScanChain "g4_1" { ScanIn "SDI[2]"; ScanOut "SDO[2]"; }

```

```

        ScanChain "g4_2" { ScanIn "SDI[1]"; ScanOut "SDO[3]"; }
    }
ScanStructures g3 {
    ScanChain g3_0 { ScanIn "SDI[3]"; ScanOut "SDO[1]"; }
    ScanChain g3_1 { ScanIn "SDI[1]"; ScanOut "SDO[2]"; }
    ScanChain g3_2 { ScanIn "SDI[2]"; ScanOut "SDO[3]"; }
}
ScanStructures g4 {
Procedures {
    load_unload {
        V { mclk=0; clk=0; rst=1; scan_en=1; inc=0; }
        V { mode=0; }
        ScanStructures g1;
        V { chain_sel = 0; mclk=P; }
        V { chain_sel = 0; mclk=P; }
        Shift {
            V { _si1=###; _so1=###; clk=P; mclk=0; }
        }
        ScanStructures g2;
        V { chain_sel = 0; mclk=P; clk=0; }
        V { chain_sel = 1; mclk=P; }
        V { mclk=0; }
        Shift {
            V { _si2=##; _so2=##; clk=P; mclk=0; }
        }
        ScanStructures g3;
        V { chain_sel = 1; mclk=P; clk=0; }
        V { chain_sel = 0; mclk=P; }
        Shift {
            V { _si3=###; _so3=###; clk=P; mclk=0; }
        }
        ScanStructures g4;
        V { clk=0; }
        V { chain_sel = 1; mclk=P; }
        V { chain_sel = 1; mclk=P; }
        Shift {
            V { _si4=###; _so4=###; clk=P; mclk=0; }
        }
        V { clk=0; mclk=0; mode=1; }
    }
}

```

The design for the multiple scan group protocol in [Example 9-17](#) has the following elements:

- Four scan chain groups. Three groups have three scan chains and the fourth has two. A simple MUX control selects the active scan group by marching a 2-bit code into the `chain_sel` port using the `mclk` clock.
- The `load_unload` procedure begins with two `v{ ... }` statements to place the design into a shift mode.
- The first `ScanStructures` statement makes group `g1` active for the lines that follow.
- A `Shift{ ... }` procedure uses the symbolic label `_si1`. This symbolic label is associated with the scan input pins defined in the `ScanStructures g1` block.

- Following the first scan group are three additional sequences of `ScanStructures`, followed by `v{...}` statements that select the appropriate chain group, and a `Shift{...}` procedure.

As you saw in [Example 9-17](#), a simple MUX control accomplished the sharing of similar I/O pins across four scan groups. But some boundary-scan designs that need to support multiple scan chains can have more complicated control sequences. For example, it is not uncommon to require the final shift of the TAP-controlled scan chain to be done outside of the `Shift` procedure.

The concepts and rules for supporting multiple scan groups are the same for a design with boundary scan as for a design without boundary scan.

[Example 9-18](#) shows a more complicated sequence for a design with three scan groups of one scan chain each. In this design, to load an instruction that accesses each internal scan chain through its test data in (TDI) and test data out (TDO) pins, the TAP controller must be stepped through each of its various states.

Example 9-18 Design With Three Scan Groups

```

STIL 1.0;
ScanStructures A { ScanChain "A1" { ScanIn "tdi"; ScanOut "tdo"; } }
ScanStructures B { ScanChain "B1" { ScanIn "tdi"; ScanOut "tdo"; } }
ScanStructures C { ScanChain "C1" { ScanIn "tdi"; ScanOut "tdo"; } }
//
// Instructions to enable scanning of each of the above 3 groups:
//
// Group           Tap instruction
// -----
// 1   SCAN_MODULE_A 7'b00011
// 2   SCAN_MODULE_B 7'b00101
// 3   SCAN_MODULE_C 7'b00111
//
Procedures {
    load_unload {

        V { clock=0; test_enab=1; scan_enab=1; _io=Z ;
            tms=0; tck=0; resetN=1; TBC=0; }
        ScanStructures A;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
        V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
        V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=1xxxx
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=11xxx
        V { tms=0; tdi=0; tck=P; } // shift IR, inst=011xx
        V { tms=0; tdi=0; tck=P; } // shift IR, inst=0011x
        V { tms=1; tdi=0; tck=P; } // shift IR, inst=00011, mv to
EXIT1-IR
        V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
        V { tms=0; tdi=0; tck=P; } // move to IDLE
        V { tms=0; tdi=0; tck=0; } // clocks off
        Shift { V { _si1=# ; _so1=# ; clock=P; } }
        ScanStructures B;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
        V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
        V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=00101
        V { tms=0; tdi=0; tck=P; } // shift IR
        V { tms=0; tdi=1; tck=P; } // shift IR
        V { tms=0; tdi=0; tck=P; } // shift IR
        V { tms=1; tdi=0; tck=P; } // shift IR, move to EXIT1-IR
        V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
        V { tms=0; tdi=0; tck=P; } // move to IDLE
        V { tms=0; tdi=0; tck=0; } // clocks off
        Shift { V { _si2=# ; _so2=# ; clock=P; } }
        ScanStructures C;
        V { tms=1; tdi=0; tck=P; clock=0; } // move to SELECT-DR
        V { tms=1; tdi=0; tck=P; } // move to SELECT-IR
        V { tms=0; tdi=0; tck=P; } // move to CAPTURE-IR
        V { tms=0; tdi=0; tck=P; } // move to SHIFT-IR
        V { tms=0; tdi=1; tck=P; } // shift IR, inst=00111
        V { tms=0; tdi=1; tck=P; } // shift IR
        V { tms=0; tdi=1; tck=P; } // shift IR
        V { tms=0; tdi=0; tck=P; } // shift IR
        V { tms=1; tdi=0; tck=P; } // shift IR, move to EXIT1-IR
    }
}

```

```

V { tms=1; tdi=0; tck=P; } // move to UPDATE-IR
V { tms=0; tdi=0; tck=P; } // move to IDLE
V { tms=0; tdi=0; tck=0; } // clocks off
Shift {
    V { _si3=# ; _so3=# ; clock=P; }
}
V { tms=1; tdi=#; tck=#; } // move to EXIT1-DR
V { tms=1; tdi=0; tck=0; } // move to UPDATE-DR
V { tms=1; tdi=0; tck=0; } // move to SELECT-DR
V { tms=0; tdi=0; tck=0; } // move to CAPTURE-DR
} // end load_unload
capture_tck {
    V { _pi=# ; _po=# ; tck=P; }
}
capture_clock {
    V { _pi=# ; _po=#; clock=P; }
}
capture_resetN {
    V { _pi=# ; _po=# ; resetN=P; }
}
capture {
    V { _pi=# ; _po=# ; }
}
}
MacroDefs {
    test_setup {
        V { _io=Z ; tms=1; tdi=0; tck=0; resetN=1; test_enab=1;
            scan_enab=0; clock=0; TBC=0; }
        V { tms=1; tdi=0; tck=0; resetN=P; clock=0; } // move to RESET
        V { tms=1; tdi=0; tck=P; resetN=1; clock=P; } // stay in RESET
        V { tms=0; tdi=0; tck=P; resetN=1; clock=P; } // move to IDLE
        V { tms=0; tdi=0; tck=0; } // clocks off
    }
}

```

Limiting Clock Usage

It is sometimes necessary to limit the clocks used by the ATPG algorithm during the capture procedures. For example, sometimes only the TCK clock should be used or the TAP controller state machine will get out of step. If you need to restrict usage of defined clocks to a single clock, use the `-clock` option of the `set drc` command:

```
DRC> set drc -clock TCK
```

Use of this option restricts the ATPG algorithm to using this clock, and only this clock, for capture.

10

Fault Lists and Faults

TetraMAX ATPG puts faults into different fault classes, which are organized into categories. This chapter describes the fault classes and explains how TetraMAX calculates test coverage statistics.

This chapter contains the following sections:

- [Fault Lists](#)
- [Fault Categories and Classes](#)
- [Fault Summary Reports](#)
- [Reporting Clock Domain-Based Faults](#)

Fault Lists

TetraMAX maintains a list of potential faults for a design, together with the categorization of each fault. You can read and write fault list files with the `read faults` and `write faults` commands.

A fault list file is an ASCII file with one fault entry per line, as shown in [Example 10-1](#). Each entry consists of three items separated by one or more spaces. The first item indicates the stuck-at value (sa0 or sa1), the second item is the two-character fault class code, and the third item is the pin path name to the fault site. Any additional text on the line is treated as a comment.

If the fault list contains equivalent faults, then the equivalent faults must immediately follow the primary fault on subsequent lines. Instead of a class code, an equivalent fault is indicated by a fault class code of “--”, as shown in [Example 10-1](#).

Example 10-1 Typical Fault List File Showing Equivalent Faults

```
// entire lines can be commented
sa0  DI   /CLK           ; comments here
sa1  DI   /CLK
sa1  DI   /RSTB
sa0  DS   /RSTB
sa1  AN   /i22/mux2/A
sa1  UT   /i22/reg2/lat1/SB
sa0  UR   /i22/mux0/MUX2_UDP_1/A
sa0  --   /i22/mux0/A    # equivalent to UR fault above it
sa0  DS   /i22/reg1/MX1/D
sa0  --   /i22/mux1/X
sa0  --   /i22/mux1/MUX2_UDP_1/Q
sa1  DI   /i22/reg2/r/CK
sa0  DI   /i22/reg2/r/CK
sa1  DI   /i22/reg2/r/RB
sa0  AP   /i22/out0/EN
sa1  AP   /i22/out0/EN
```

TetraMAX ignores blank lines and lines that start with a double slash and a space (//).

Note:

You can control whether the fault list contains equivalent faults or primary faults by using the `-report` option of the `set faults` command or the `-collapsed` or `-uncollapsed` option of the `write faults` command.

Using Fault List Files

You can use fault list files to manipulate your fault list, in the following ways:

- Add faults from a file, ignoring any fault classes specified

- Add faults from a file, retaining any fault classes specified
- Delete faults specified by a fault list file
- Add nofaults (sites where no faults are to be placed) specified by a fault list file

To access fault list files, you use the `read faults` and `read nofaults` commands, which have the following syntax:

```
READ Faults file_name [-Retain_code] [-Add | -Delete]
READ NOFaults file_name
```

The `-retain_code` option retains the fault class code but behaves differently depending on whether the faults in the file are new or replacements for existing faults. These differences are discussed next.

New Faults With `-retain_code`

For any new fault locations encountered in the input file, if the fault code is DS or DI, the new fault is added to the fault list as DS or DI, respectively. For all other fault codes, TetraMAX determines whether the fault location can be classified as UU, UT, UB, DI, or AN. If the fault location is determined to be one of these fault classes, the new fault is added to the fault list and the fault code is changed to the determined fault class. If the fault location was not found to be one of these special classes, the new fault is added with the fault code as specified in the input file.

Existing Faults With `-retain_code`

For any fault locations provided in the input file that are already in the internal fault list, the fault code from the input file replaces the fault code in the internal fault list. TetraMAX does not perform any additional analysis.

Collapsed and Uncollapsed Fault Lists

To improve performance, most ATPG tools collapse all equivalent faults and process only the collapsed set. For example, the stuck-at faults on the input pin of a BUF device are considered equivalent to the stuck-at faults on the output pin of the same device. The collapsed fault list contains only the faults at one of these pins, called the primary fault site. The other pin is then considered the equivalent fault site. For a given list of equivalent fault sites, the one chosen to be the primary fault site is purely random and not predictable.

You can generate a fault summary report using either the collapsed or uncollapsed list using the `-report` option of the `set faults` command.

[Example 10-2](#) shows both a collapsed and an uncollapsed fault summary.

Example 10-2 Collapsed and Uncollapsed Fault Summary Reports

```
TEST> set faults -report collapsed
TEST> report faults -summary

      Collapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   120665
Possibly detected    PT   3749
Undetectable         UD   1374
ATPG untestable     AU   6957
Not detected         ND   6452
-----
total faults          139197
test coverage        88.91%
-----

TEST> set faults -report uncollapsed
TEST> report faults -summary

      Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   144415
Possibly detected    PT   4003
Undetectable         UD   1516
ATPG untestable     AU   8961
Not detected         ND   7607
-----
total faults          166502
test coverage        88.74%
-----
```

Random Fault Sampling

Using a sample of faults rather than all possible faults can reduce the total runtime for large designs. You can create a random sample of faults using the `-retain_sample percentage` option of the `remove faults` command.

The `percentage` argument of the `-retain_sample` option indicates a probability of retaining each individual fault and does not indicate an exact percentage of all faults to be retained. For example, if `percentage = 40`, for a fault population of 10,000, TetraMAX does not retain exactly 4,000 faults. Instead, it processes each fault in the fault list and retains or discards each fault according to the specified probability. For large fault populations, the exact percentage of faults kept will be close to 40 percent, but for smaller fault populations, the actual percentage might be a little bit more or less than what is requested, because of the granularity of the sample.

For example, the following sequence requests retaining a 25 percent sample of faults in `block_A` and `block_B` and a 50 percent sample of faults in `block_C`.

```
TEST> add faults /my_asic/block_A
TEST> add faults /my_asic/block_B
TEST> remove faults -retain_sample 50
TEST> add faults /my_asic/block_C
TEST> remove faults -retain_sample 50
```

You can combine the `-retain_sample` option with the capabilities of defining faults and nofaults from a fault list file for flexibility in selecting fault placement.

As an alternative to the `remove faults` command, you can choose **Faults > Remove Faults** to access the Remove Faults dialog box.

Fault Dictionary

In some products, a “fault dictionary” is used to translate a fault location into a pattern that tests that location, and to translate a pattern number into a list of faults detected by that pattern.

TetraMAX ATPG does not produce a traditional fault dictionary. Instead, it supports a diagnostics mode that translates tester failure data into the design-specific fault location identified by the failure data. For more information, see [Chapter 19, “Diagnosing Manufacturing Test Failures.”](#)

Fault Categories and Classes

TetraMAX maintains a list of potential faults in the design and assigns each such fault to a fault class according to its detectability status. Faults classes are organized into categories. A two-character symbol is used as an abbreviated name for both classes and categories.

There are five higher-level fault categories containing a total of 11 lower-level fault classes, organized as follows:

```

DT: detected
DR: detected robustly
DS: detected by simulation
DI: detected by implication
PT: possibly detected
AP: ATPG untestable-possibly detected
NP: not analyzed-possibly detected
UD: undetectable
UU: undetectable unused
UT: undetectable tied
UB: undetectable blocked
UR: undetectable redundant
AU: ATPG untestable
AN: ATPG untestable-not detected
ND: not detected
NC: not controlled
NO: not observed

```

“Undetectable” means that the fault cannot be detected by any means. A straightforward example of this is an unused Q-bar output of a flip-flop, which is classified as UU (undetectable unused). It is impossible to detect faults at these types of locations because they cannot be observed, either directly or indirectly.

“Untestable” means that TetraMAX could not find a successful detection pattern while maintaining all design constraints (for example, constant ports, contention avoidance). This type of fault might become testable if restrictions are relaxed.

Detected Faults

The DT (detected) category of faults includes three classes:

- DR (detected robustly) faults are determined during Path Delay ATPG or fault simulation. During ATPG, at least one pattern that caused the fault to be placed in this class is retained. For additional information, see [Chapter 15, “Path Delay Fault Testing.”](#)
- DS (detected by simulation) faults are determined by generating patterns and simulating to verify that the patterns result in the faults being detected.
- DI (detected by implication) faults do not have to be detected by specific patterns, because these faults result from shifting scan chains. The faults in the DI class usually occur along the scan chain paths and include clock pins and scan-data inputs and outputs of the scan cells.

If design rule checking (DRC) determines that the scan chains are controlled and connected, and if the shift data can be passed from a scan chain data input to a scan chain data output, then those faults along the scan chain associated with the data and clock paths are classified as detected by implication; no specific patterns need to be generated to test for them. By default, a special chain test is written into the ATPG patterns to ensure testing of DI faults.

Possibly Detected Faults

The PT (possibly detected) category of faults contains these two classes:

- The AP (ATPG possibly detected) class contains faults for which the difference between the good machine and the faulty machine results in a simulated output of X rather than 1 or 0. Analysis proved that the fault cannot be definitely detected under current ATPG conditions, only possibly detected. For example, with faults on the enable line of an internal three-state driver, the off state of the enable can only be possibly detected because the resulting Z state on the data bus quickly becomes an X state as it is captured into a scan cell or passes through other internal logic.
- The NP (not analyzed-possibly detected) class also contains faults for which the difference between the good machine and the faulty machine results in a simulated output of X rather than 1 or 0. However, the analysis to prove that the fault cannot be definitely detected using current ATPG conditions was not conclusive. Like the AP class, the simulation cannot tell the expected output of the faulty machine.

For both the AP and NP classes of faults, the good machine value is known but the faulty machine value simulates as X and is unknown. On the actual device, an X is never generated; the output in the presence of a fault is either a 1 or a 0. If the faulty device output is 1 and the expected data is 0, the fault is detected. If the faulty device output is 0 and the expected data is 0, the fault is not detected, and so these types of faults are identified as possibly detected.

Partial credit is given for possibly detected faults in the test coverage calculation. This partial credit is by default 50 percent, but you can adjust the credit using the `set faults -pt_credit` command. For details, see [“Test Coverage” on page 10-10](#).

Undetectable Faults

The UD (undetectable) category of faults contains faults that cannot be tested by any means: ATPG, functional, parametric, or otherwise. Usually, when TetraMAX calculates test coverage, these faults are subtracted from the total faults of the design.

The UD category includes four classes:

- The UU (undetectable unused) class contains faults located on unused outputs or, in general, outputs that have no electrical connection to any other logic. A fault located on one of these fault sites has no logic simulation effect on any other logic in the design.
- The UT (undetectable tied) class contains faults located on pins that are tied to a logic 0 or 1, which are usually unused inputs that have been tied off. A stuck-at-1 fault on a pin tied to a logic 1 cannot be detected and has no fault effect on the circuit. Similarly, a stuck-at-0 fault on a pin tied to a logic 0 has no effect.

- The UB (undetectable blocked) class contains faults at locations for which controllability and observability are hindered by redundant faults. UB faults are companion faults to UR faults.
 - The UR (undetectable redundant) class contains faults for which there are redundant logic paths. A fault in one path cannot be detected because the other redundant logic path masks the fault effect. Because of the self-protecting nature of redundant logic design, a single fault cannot be detected and is indistinguishable from the good machine behavior as seen from the design outputs and scan cells.
-

ATPG Untestable Faults

The AU (ATPG untestable) category of faults contains faults that are not necessarily intrinsically untestable, but are untestable using ATPG methods. These faults cannot be proven to be undetectable and might be testable using other methods (for example, functional tests).

The AU category has one class, AN (ATPG untestable, not detected). The primary reasons for this classification are:

- A fault cannot be controlled or observed because setting up the required patterns would violate a PI or ATPG constraint.
 - Faults associated with nonscan sequential devices (latches and flip-flops) are not testable with simple ATPG methods.
-

Not Detected Faults

TetraMAX classifies a fault as ND (not detected) when the analysis for that fault was not completed or was aborted. Incomplete or aborted analyses could be caused by the default ATPG iteration limits or by designs that are too complex for the ATPG algorithm to solve.

The ND category has two classes:

- The NC (not controlled) class contains faults that the ATPG algorithm could not control to achieve both a logic 0 and a logic 1 state. Nodes that are always at an X state are classified as NC because ATPG cannot achieve either a logic 0 or a logic 1.
- The NO (not observed) class contains faults that could be controlled, but could not be propagated into a scan chain cell or to a design output for observation.

Usually a design begins with most faults in the NC class. As the ATPG algorithm progresses, the faults are moved to the NO class and then to a DS or other class. If too many NC or NO faults remain after test pattern generation, you can try to increase the test coverage by increasing the ATPG effort level. For example, you can use a larger value for the `-abort_limit` option of the `set atpg` command.

Fault Summary Reports

By default, TetraMAX displays fault summary reports using the five categories of fault classes, as shown in [Example 10-3](#).

Example 10-3 Fault Summary Report: Test Coverage

```
-----  
Uncollapsed Fault Summary Report  
-----  
fault class           code   #faults  
-----  
Detected              DT     144361  
Possibly detected     PT     4102  
Undetectable          UD     1516  
ATPG untestable       AU     8828  
Not detected          ND     7695  
-----  
total faults          166502  
test coverage         88.74%  
-----
```

For a detailed breakdown of fault classes, use the `-summary verbose` option of the `set faults` command:

```
TEST> set faults -summary verbose
```

[Example 10-4](#) shows a verbose fault summary report, which includes the fault classes in addition to the fault categories.

Example 10-4 Verbose Fault Summary Report

Uncollapsed Fault Summary Report

fault class	code	#faults
Detected	DT	144415
detected_by_simulation	DS	(117083)
detected_by_implication	DI	(27332)
Possibly detected	PT	4003
atpg_untestable-pos_detected	AP	(403)
not_analyzed-pos_detected	NP	(3600)
Undetectable	UD	1516
undetectable-unused	UU	(4)
undetectable-tied	UT	(565)
undetectable-blocked	UB	(469)
undetectable-redundant	UR	(478)
ATPG untestable	AU	8961
atpg_untestable-not_detected	AN	(8961)
Not detected	ND	7607
not-controlled	NC	(503)
not-observed	NO	(7104)
total faults		166502
test coverage		88.74%

The test coverage figure at the bottom of the report provides a quantitative measure of the test pattern quality. You can optionally choose to see a report of the fault coverage or ATPG effectiveness instead.

The three possible quality measures are defined as follows:

- Test coverage = detected faults / detectable faults
- Fault coverage = detected faults / all faults
- ATPG effectiveness = ATPG-resolvable faults / all faults

Test Coverage

Test coverage gives the most meaningful measure of test pattern quality and is the default coverage reported in the fault summary report. Test coverage is defined as the percentage of detected faults out of detectable faults, as follows:

$$\text{Test Coverage} = \frac{\text{DT} + (\text{PT} \times \text{PT_credit})}{\text{All_Faults} - \text{UD} - (\text{AN} \times \text{AU_credit})} \times 100$$

PT_credit is initially 50 percent and AU_credit is initially 0. You can change the settings for PT_credit or AU_credit using the set_faults command.

By default, the fault summary report shows the test coverage, as in [Example 10-3](#) and [Example 10-4](#).

Fault Coverage

Fault coverage is defined as the percentage of detected faults out of all faults, as follows:

$$\text{Fault Coverage} = \frac{\text{DT} + (\text{PT} \times \text{PT_credit})}{\text{All_Faults}} \times 100$$

Fault coverage gives no credit for undetectable faults; PT_credit is initially 50 percent.

To display fault coverage in addition to test coverage with the fault summary report, use the `-fault_coverage` option of the `set faults` command. [Example 10-5](#) shows a fault summary report that includes the fault coverage.

Example 10-5 Fault Summary Report: Fault Coverage

```
TEST> set faults -fault_coverage
TEST> report faults -summary
-----
Uncollapsed Fault Summary Report
-----
fault class           code   #faults
-----
Detected              DT     144361
Possibly detected     PT     4102
Undetectable          UD     1516
ATPG untestable       AU     8828
Not detected          ND     7695
-----
total faults          166502
test coverage         88.74%
fault coverage        87.93%
```

ATPG Effectiveness

ATPG effectiveness is defined as the percentage of ATPG-resolvable faults out of the total faults, as follows:

$$\text{ATPG_eff} = \frac{\text{DT} + \text{UD} + \text{AN} + (\text{NP} \times \text{PT_credit})}{\text{All_Faults}} \times 100$$

In addition to faults that are detected, full credit is given for faults that are proven to be untestable by ATPG. PT_credit is initially 50 percent.

To display AGPG effectiveness with the fault summary report, use the -atpg_effectiveness option of the set_faults command. [Example 10-6](#) shows a fault summary report that includes the ATPG effectiveness.

Example 10-6 Fault Summary Report: ATPG Effectiveness

```
TEST> set faults -atpg_effectiveness
TEST> report faults -summary
-----
Uncollapsed Fault Summary Report
-----
fault class          code  #faults
-----
Detected             DT    144361
Possibly detected   PT    4102
Undetectable        UD    1516
ATPG untestable     AU    8828
Not detected        ND    7695
-----
total faults         166502
test coverage        88.74%
fault coverage       87.93%
ATPG effectiveness  94.30%
```

Reporting Clock Domain-Based Faults

TetraMAX includes a set of command options that enable you to report fault coverage for transition or stuck-at faults on a per-clock domain basis. You can also add or remove faults for particular clock domains so that ATPG or fault simulation targets only those clock domains that are of interest.

Note the following when using this feature:

- It is important to understand how TetraMAX distinguishes faults captured by a clock and launched by a clock:
 - Faults are considered to be captured by a clock when they feed a logic cone that enters the data input of a flip-flop clocked by that clock.
 - Faults are considered to be launched by a clock when they are fed by a logic cone starting from the output of a flip-flop clocked by that clock.
 - The clock, set, and reset inputs of flip-flops are not considered when determining capture; faults leading to them are captured by the NO_CLOCK domain.

- Faults within the logic core of more than one clock are not considered to belong to either domain. Instead, they are put into a separate category called MULTIPLE. Thus, the clock domain faulting is called exclusive because each clock domain excludes the effects of other clocks.
- Faults given the status Detected by Implication (DI) are detected by the scan chain load/unload sequence. This sequence uses shift constraints which can differ dramatically from the capture constraints that are used to calculate launch and capture clocks for reporting faults by clock domain. This often results in DI faults being reported as captured by the NO_CLOCK domain if the shift path is blocked by the capture constraints. If shift-only clocks are used, this can result in DI faults being both launched and captured by the NO_CLOCK domain.

The table below explains all the commands and command options associated with reporting clock domain-based faults.

Command	Description
<code>add faults -launch <launch_clock></code>	Specifies the launch clock of the faults to be added. You can use this switch independently, or in conjunction with the <code>-capture</code> switch (described below). The <code>-launch</code> switch adds all faults to the fault list that are launched exclusively by the specified clock. This switch also accepts the keywords PI, NO_CLOCK, or MULTIPLE, in either all upper or all lowercase, to add faults that are driven by PI/PIO, no clock domains, or multiple clock domains, respectively.
<code>add faults -capture <clock_name></code>	Specifies the capture clock of the faults to be added. You can use this switch independently, or in conjunction with the <code>-launch</code> switch described previously. The <code>-capture</code> switch adds all faults to the fault list that are captured exclusively by the specified clock. This switch also accepts the keywords PO, NO_CLOCK, or MULTIPLE, in either all upper or all lowercase, to add faults that are observed by PO/PIO, no clock domains, or multiple clock domains, respectively.
<code>add faults -exclusive</code>	Specifies that only the faults that are driven and captured exclusively (using a single launch and a single capture) are to be added. Faults exclusively driven by PI or observed by PO are also added.

Command	Description
add faults -shared	Specifies that only the faults that are launched or captured by multiple clocks should be added. This excludes all PI and PO faults described in the add faults options described previously.
Note: This command is equivalent to: add faults -launch MULTIPLE add faults -capture MULTIPLE	
add faults -inter_clock_domain	Adds only exclusive faults that are driven and captured by different clock domains.
add_faults -intra_clock_domain	Adds only exclusive faults that are driven and captured by the same clock domains.
Note: All of the add faults options, other than -launch and -capture, are exclusive and cannot be issued at the same time, or with any of the old options. If you issue these options together, a UI error message will appear.	
remove faults -launch <clock_name>	Specifies the launch clock of the faults to be removed. You can use this switch independently, or in conjunction with the -capture switch (described later).
	The -launch switch removes all faults to the fault list that are launched exclusively by the specified clock. This switch also accepts the keywords PI, NO_CLOCK, or MULTIPLE, in either all upper or all lowercase, to remove faults that are driven by PI/PIO, no clock domains, or multiple clock domains, respectively.
remove faults -capture <clock_name>	Specifies the capture clock of the faults to be removed. You can use this switch independently, or in conjunction with the -launch switch (described previously).
	The -capture switch removes all faults to the fault list that are captured exclusively by the specified clock. This switch also accepts the keywords PO, NO_CLOCK, or MULTIPLE, in either all upper or all lowercase, to remove faults that are observed by PO/PIO, no clock domains, or multiple clock domains, respectively.

Command	Description
remove faults -exclusive	Specifies that only the faults that are driven and captured exclusively (using a single launch and a single capture) are to be removed. Faults exclusively driven by PI or observed by PO are also removed.
remove faults -shared	Specifies that only the faults that are launched or captured by multiple clocks should be removed. This excludes all PI and PO faults (described in the remove faults options previously).
remove faults -inter_clock_domain	Removes only exclusive faults that are driven and captured by different clock domains.
remove faults -intra_clock_domain	Removes only exclusive faults that are driven and captured by the same clock domains.
Note: All of the remove faults options, other than -launch and -capture, are exclusive and cannot be issued at the same time, or with any of the old options. If you issue these options together, a UI error message will appear	
report faults -per_clock_domain	All specified faults are reported with extra information for their launch and capture clocks. Note that all clocks are reported, even for "shared" or "multiple" categories. The format is as follows: <pre><type> <code> <fault_name> (Launch clocks: <clock1> <clock2> ...) (Capture clocks: <clock1> <clock2> ...) str NC u_ext1/y_reg_3/Q (Launch clocks: clkext1) (Capture clocks: TOTO/ U8 clkext3)</pre>

Command	Description
report summaries fault -per_clock_domain	<p>Specifies that the clock report should be divided on a per clock domain basis as shown in the following example. All shared faults are reported as one category.</p> <pre>// Uncollapsed Stuck Fault Summ Report // From: Clock Domain A // To: Clock Domain A // ----- // fault class code #faults // ----- // Detected DT 603 // Possibly detected PT 32 // Undetectable UD 10 // ATPG untestable AU 2 // Not detected ND 3 // ----- // total faults 650 // test coverage 96.72% // -----</pre>

(Report continues for each clock domain.

Command	Description
<pre>report summaries fault -launch <clock_name></pre>	<p>Specifies the launch clock of the faults to be reported on. This switch can be used independently, or in conjunction with the <code>-capture</code> switch (described below). This switch also accepts the keywords PI, NO_CLOCK, and MULTIPLE in either all upper or all lowercase, to report on faults that are driven by PI/PIO, no clock domains, or multiple clock domains, respectively.</p>

The `-launch` option creates individual reports for each capture clock as follows:

```
BUILD> report summaries f -launch A

// Uncollapsed Stuck Fault Summ Report
//      From: Clock Domain A
//      To: Clock Domain A
// -----
//      fault class          code #faults
// -----
//      Detected            DT    603
//      Possibly detected   PT     32
//      Undetectable        UD     10
//      ATPG untestable    AU      2
//      Not detected       ND      3
// -----
//      total faults           650
//      test coverage          96.72%
// -----
```

(Report continues for each clock domain.)

Command	Description
report summaries fault -capture <clock_name>	<p>Specifies the capture clock of the faults to be reported on. This switch can be used independently or in conjunction with the -launch switch (described previously).</p> <p>This switch also accepts the keywords PO, NO_CLOCK, and MULTIPLE in either all upper or all lowercase, to report on faults that are observed by PO/PIO, no clock domains, or multiple clock domains, respectively.</p> <p>The -capture switch creates individual reports for each launch clock as follows:</p> <pre>BUILD> report summaries f -capture B // From: Clock Domain A // To: Clock Domain B // ----- // fault class code #faults // ----- ---- ----- // Detected DT 8 // Possibly detected PT 2 // Undetectable UD 1 // ATPG untestable AU 0 // Not detected ND 1 // ----- // total faults 12 // test coverage 81.82% (Report continues for each clock domain.)</pre>
report summaries fault -exclusive	Excludes the multiple launch and capture section from the report.
report summaries fault -shared	Reports only the section relating to multiple launch and capture clocks.
report summaries fault -inter_clock_domain	Reports on only the exclusive faults that are driven and captured by different clock domains.

Command	Description
<code>report summaries fault -intra_clock_domain</code>	Reports on only the exclusive faults that are driven and captured by the same clock domains.

Note: All of the `report summaries` options, other than `-launch` and `-capture`, are exclusive and cannot be issued at the same time, or with any of the old options. If you issue these options together, a UI error message will appear.

Using Signals That Conflict With Reserved Keywords

The names MULTIPLE, NO_CLOCK, PI, and PO are reserved keywords when you use the `-launch` and `-capture` options. If a clock signal uses one of these names, the clock signal always takes priority when these options are used.

For example, if a clock is named MULTIPLE, then the command `add fault -launch MULTIPLE` adds faults launched exclusively by the clock named MULTIPLE. In this case, if you want to add faults launched by multiple clocks, you can use the command `add faults -launch multiple`. This command works as expected because the reserved names can be all uppercase or all lowercase; however, the actual clock names are case-sensitive.

Finding Particular Untested Faults Per Clock Domain

If you specify the command `report summaries fault -per_clock`, TetraMAX provides only aggregate results. To find individual faults, you need to do the following:

1. Save the patterns.
2. Delete all the faults.
3. Add only the faults of interest.
4. Resimulate the patterns.
5. Run the `report fault` command.

11

Fault Simulation

This chapter describes the information needed for fault simulation (also known as fault grading of functional patterns), outlines the basic fault simulation flow, and provides a detailed description of the fault simulation steps.

This chapter contains the following sections:

- [Fault Simulation Design Flow](#)
- [Preparing the Functional Test Patterns for Fault Simulation](#)
- [Preparing the Design for Fault Simulation](#)
- [Reading the Functional Test Patterns](#)
- [Initializing the Fault List](#)
- [Performing Good Machine Simulation](#)
- [Performing Fault Simulation](#)
- [Combining ATPG and Functional Test Patterns](#)

Procedures that are the same as for the ATPG design flow are treated in less detail. This chapter provides cross-references to corresponding sections in [Chapter 4, “ATPG Design Flow.”](#)

Fault Simulation Design Flow

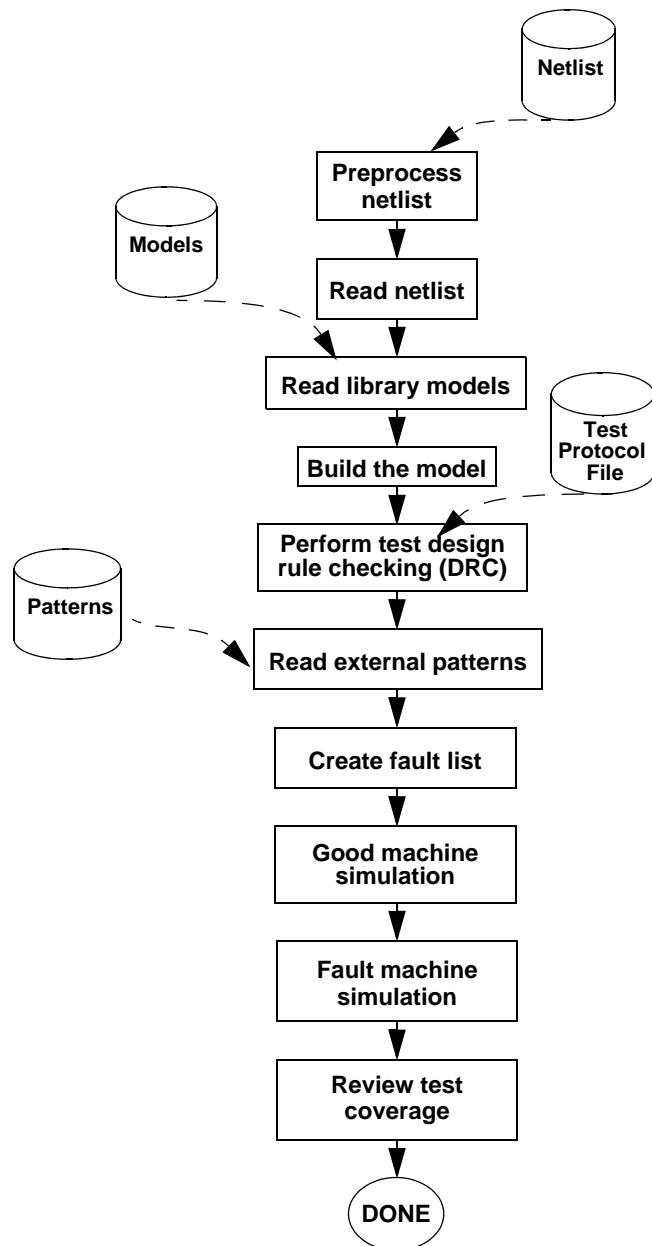
Performing fault simulation lets you determine the test coverage obtained by an externally generated test pattern. For fault simulation, you need functional test patterns that have been developed to test the design and have been previously simulated in a logic simulator to verify correctness. The functional test patterns should contain the expected values unless you are using the Extended Value Change Dump (VCD) format. The expected values tell TetraMAX when and what to measure.

[Figure 11-1](#) shows the basic fault simulation design flow. The steps performed in this flow are as follows:

1. Prepare the functional test patterns for fault simulation.
2. Prepare the design for fault simulation.
3. Read in the external functional patterns.
4. Perform a good machine simulation.
5. Perform fault simulation to fault-grade the functional test patterns.
6. Write the fault list.
7. Save the fault list if you intend to perform ATPG later.

Step 2 includes tasks common to those for the ATPG design flow: preprocessing the netlist, reading in the netlist, reading in the library models, building the design model, and performing design rule checking (DRC).

Figure 11-1 Fault Simulation Design Flow



Preparing the Functional Test Patterns for Fault Simulation

Before beginning fault simulation, you must prepare your functional test patterns for the TetraMAX ATPG fault simulator.

The ATPG and fault simulation algorithms emphasize speed and efficiency over the ability to use or simulate gate timing delays. There are corresponding limitations on functional test patterns. The test patterns must meet the following requirements:

- Acceptability to automated test equipment (ATE)
- Timing insensitivity
- Recognizable format

Acceptability to ATE

Because the functional test patterns are ultimately used by ATE, you must verify that the patterns comply with requirements of ATE. Each brand and model of ATE has its own list of restrictions. The following general list of characteristics is usually acceptable:

- The input stimuli, clocks, and expected response outputs can be divided into a sequence of identical tester cycles.
- Each tester cycle is associated with a timing set. There are a fixed number of timing sets.
- The test cycle defines the state values to be applied as inputs and measured as outputs, and the associated timing set defines the cycle period and the timing offsets within the cycle when inputs are applied, clocks are pulsed, and outputs are sampled.
- The functional patterns are regular, with the timing of input changes and clock pulse locations constant from one cycle to the next.
- The functional pattern set maps into four or fewer timing sets.

In addition, every ATE has its own set of rules for timing restrictions, such as the following:

- Minimum and maximum test cycle period
- Minimum and maximum pulse width
- Proximity of a pulsed signal to the beginning or end of a cycle
- Proximity of two signal changes to one another
- Accuracy and placement of measure strobes
- Placement accuracy of input transitions

Checking for Timing Insensitivity

Functional test patterns must be timing insensitive within each test cycle. The design can have no race conditions that depend on gate delays that must be resolved on nets that reach a sequential device, a RAM or ROM, or a primary output port.

To check for timing insensitivity,

1. Simulate the design in a logic simulator without timing and use a unit-delay or zero-delay timing mode.
2. If the simulation passes, simulate it again with all timing events expanded in time by five or ten times.

If the functional test patterns pass under these conditions, they can be considered timing insensitive.

Examples That Cause Timing Sensitivity

- A pulse generator: An edge transition of an input port results in a pulse on an output port or at the data capture input of an internal register. This pulsed value occurs at a specific delay from the input event and, unless the output is measured at the correct time or the internal register is clocked at the correct time, the pulsed value is lost. This type of design fails simulation in the absence of actual timing.

You can correct this situation in one of two ways:

- Hold the triggering port fixed to a constant value in the functional patterns.
- Add some shunting circuitry (enabled in some sort of test mode) that blocks the internal propagation of the pulsed value.
- Timing-critical measurements: An input port event at offset 0 ns turns on an output driver in 100 nanoseconds (ns), but the patterns are set to measure a Z value at 90 ns before the driver is turned on. Although this measurement is correct in the real device, TetraMAX ATPG uses only unit delays and reports a simulation mismatch.

You can correct this situation by measuring at 110 ns and changing the expected data to the appropriate non-Z value.

- Multiple active clocks or asynchronous set/reset ports in the same cycle: With careful attention to timing, correct use of clock trees, and good analysis tools, you can design blocks of logic with intermixed clock zones that operate correctly with functional patterns when more than one clock is active. However, because TetraMAX ATPG uses zero delay and not gate timing, simulating designs that contain more than one active clock can result in the erroneous identification of internal race conditions and subsequent elimination of functional test patterns.

If your design has more than one active clock, you can prevent TetraMAX ATPG from reporting internal race conditions if you do one of the following:

- Use master-slave clocking in your design.
 - Use resynchronization latches between clock domains.
 - Arrange your test patterns so that in any one cycle you have only one active clock.
-

Recognizable Format

TetraMAX can read patterns in Verilog, VHDL, STIL, WGL, VCDE (Extended VCD), and TetraMAX binary formats. A pattern file that has been compressed with GZIP (a common compression utility for UNIX and PC platforms) can be read directly into TetraMAX.

There are two main types of functional pattern files that can be used: scan and nonscan patterns.

A scan functional pattern contains scan chain load/unload sequences and defines structures and procedures that can be recognized as scan-chain related.

A nonscan functional pattern contains no recognizable structures or procedures to indicate that it is a scan pattern. The pattern might exercise scan chains, might be completely functional, or might perform a combination of scan chain and functional testing.

Scan functional patterns rarely need to be read in, because they are generated by TetraMAX ATPG or another ATPG tool, and should have had fault simulation performed and a test coverage calculated by the ATPG tool. Thus, nonscan functional patterns are the more common patterns used with fault simulation, and so this chapter focuses primarily on nonscan functional patterns.

Requirements for Scan Functional Patterns

Your scan functional pattern set should meet the following requirements:

- Provides scan functional patterns in the same style and format that TetraMAX ATPG uses to write ATPG patterns.
- Defines scan chains, clocks, and primary input constraints that match the usage in the patterns.
- Defines the `load_unload`, `Shift`, and other test procedures so that they are consistent with the patterns.

To become more familiar with the requirements of a specific input format and to see how to define clocks, primary inputs constraints, and scan chains for your current design, write a few sample ATPG patterns to a file. This file will provide an example of the syntax needed for input of scan functional patterns.

Requirements for Nonscan Functional Patterns

Your nonscan functional pattern set should meet the following requirements:

- Provides the patterns as a simple, sequential application of input stimulus and output measures.
- Does not define any scan chains or any ATPG-related procedures (for example, `load_unload` or `Shift`).

If the functional patterns do not contain timing information, you can use a STIL procedure file (SPF) to define pin timing, then reference the SPF using the `run drc` command or the Run DRC dialog box.

To become more familiar with the requirements of pattern input, do the following:

1. For your current design, use the `add clocks` command or the Add Clock dialog box to define as clocks all ports in the input data that function as clocks or pulsed ports.
Note that defining the clocks is optional. Some clock violations found during the `run drc` process can affect the simulator and it may be necessary to remove `add clock` commands.
2. Try to switch to TEST mode without the use of an SPF. Typically, you must change the severity of many of the rules from their defaults to either warning or ignore.
3. Once you achieve TEST mode, execute `run atpg` to generate at least one pattern.
4. Write out a few patterns. Because no scan chains have been defined, this pattern file represents a template for nonscan functional pattern input.

[Example 11-1](#) and [Example 11-2](#) show examples of functional patterns in WGL and STIL format. Additional examples of pattern data in STIL, WGL, and Verilog formats are provided in [“Pattern Input” on page 12-13](#).

Example 11-1 Functional Pattern in WGL Format

```
waveform MY ASIC

signal
  clk : input ;  en : input ;  tck : input ;  tms : input ;
  tdi : input ;  trst_n : input ;  tdo : output ;
  in00 : input ;  out00 : output ;
end

timeplate TP1 period 100NS
  clk    := input[0PS:D, 45NS:S, 55NS:D];
  en     := input[0PS:P, 5NS:S];
  tck    := input[0PS:D, 45NS:S, 55NS:D];
  tms    := input[0PS:P, 5NS:S];
  tdi    := input[0PS:P, 5NS:S];
  trst_n := input[0PS:P, 5NS:S];
  in00   := input[0PS:P, 5NS:S];
  tdo    := output[0PS:X, 95NS:Q'edge];
  out00  := output[0PS:X, 95NS:Q'edge];
end

pattern group_ALL (clk, en, tck, tms, tdi, trst_n,
  tdo, in00, out00)

  vector(TP1) := [ 0 X 0 1 X 0 Z X X ];
  vector(TP1) := [ 0 X 1 0 X 1 Z X X ];
  vector(TP1) := [ 0 X 1 0 X 1 0 X X ];
  vector(TP1) := [ 0 X 1 0 1 1 1 X X ];
  vector(TP1) := [ 0 X 1 0 1 1 1 X X ];
  vector(TP1) := [ 0 X 1 0 0 1 0 X X ];
  vector(TP1) := [ 0 X 1 0 0 1 0 X X ];

end  #pattern

end  #waveform
```

Example 11-2 Functional Pattern in STIL Format

```
STIL 1.0 {
    Extension Design P2000.5;
}
Signals {
    clk In; en In; tck In; tms In; tdi In; trst_n In;
    in00 In; tdo Out; out00 Out;
}
SignalGroups {
    _pi = 'clk + en + tck + tms + tdi + trst_n + in00';
    _po = 'tdo + out00';
}
Timing {
    WaveformTable TP1 {      Period '100ns';
        Waveforms {
            _pi { 01ZN { '0ns' D/U/Z/N; } }
            _po { X { '0ns' X; } }
            _po { HLT { '0ns' X; '95ns' H/L/T; } }
            tck { P { '0ns' D; '45ns' U; '35ns' D; } }
            trst_n { P { '0ns' U; '45ns' D; '35ns' U; } }
        }
    }
}
PatternBurst "_burst_" { PatList {
    "func" {
    }
}}
PatternExec {
    PatternBurst "_burst_";
}
Pattern func {
    W TP1;
    V { _pi = ON01NPN; _po = ZX; }
    V { _pi = ONP0N1N; _po = ZX; }
    V { _pi = ONP0N1N; _po = LX; }
    V { _pi = ONP011N; _po = HX; }
    V { _pi = ONP011N; _po = HX; }
    V { _pi = ONP001N; _po = LX; }
    V { _pi = ONP001N; _po = LX; }
}
```

VCDE Input Patterns

TetraMAX supports reading patterns in Extended VCD (VCDE) format. This format is not the same as traditional VCD. To create a VCDE data file, you need a Verilog-compatible simulator that supports the IEEE draft definition of VCDE. (For details, refer to IEEE P1364.1-1999, *Draft Standard for Verilog Register Transfer Level Synthesis*.) The Synopsys VCS simulator, version 5.1 or later, supports this standard.

For a Verilog simulator that supports the creation of VCDE, creating a VCDE data file is fairly simple. Add a single `$dumpports()` system task to the `initial` block of the top-level module. The syntax is similar to the following:

```

initial begin
  //
  // --- other variable inits here
  //
  $dumpports( testbench.DUT, "vcde_output_file");
  ...
end

```

In this example, the simulator captures all of the I/O events for the simulation instance `testbench.DUT` into a file called `vcde_output_file`. If your simulation is performed directly on your design, the path to this file might be `DUT`. If your design is instantiated in a `testbench`, then this path is more likely to be `testbench.DUT`, where `testbench` is the top-level module name and `DUT` is the instance name of the design found within the module `testbench`.

Note:

If you want to generate a VCDE file from a TetraMAX Verilog testbench, you can use the `+define+tmax_vcde` variable to help generate that file. Do this by adding the `+define+tmax_vcde` variable to your VCS command line when you simulate the TetraMAX-generated Verilog testbench. An VCDE file called `sim_vcde.out` is automatically created.

Be careful not to create a VCDE file with complex timing events. The most efficient functional patterns are those most closely resembling what would be applied on a tester. Within a cycle, use as few separate events as possible as in this sequence:

1. Force all inputs at the same time.
2. Pulse the clock.
3. Measure all outputs at the same time.

Functional patterns in VCDE format do not need to have any measures defined. TetraMAX decides what values to expect on output and bidirectional pins by keeping a running tally of the most recently reported values in the VCDE event stream. For an output port, all values other than X are measurable. For a bidirectional port, the values L, H, T, I, and h are measurable; the value X is not measured; and the values 0, 1, and Z indicate input mode (which is not measurable).

In TetraMAX, when you read in VCDE patterns, you specify the cycle period and measure points within each cycle. TetraMAX uses this information to construct internal measure points and expected data. For more information, see “[Specifying Strobes for VCDE Pattern Input](#)” on page 11-15.

Preparing the Design for Fault Simulation

Most of the steps for preparing the design for fault simulation are the same as preparing for the ATPG design flow:

1. Preprocessing the netlist
 2. Reading the design and libraries
 3. Building the ATPG design model
 4. Declaring clocks (optional)
 5. Running DRC
-

Preprocessing the Netlist

If necessary, preprocess the netlist for compatibility with TetraMAX ATPG. For more information, see “[Netlist Requirements](#)” on page 4-4.

Reading the Design and Libraries

As with ATPG, for TetraMAX ATPG fault simulation you first invoke TetraMAX ATPG, read in the design netlist, and read in the library models. For details, see “[Reading the Netlist](#)” on page 4-5 and “[Reading Library Models](#)” on page 4-7.

An example command sequence is as follows:

```
% tmax  
BUILD> read netlist my_asic.v  
BUILD> read netlist my_lib/*.v -noabort
```

Building the ATPG Design Model

To build the ATPG design model for fault simulation, you use the same `run build_model` command as for ATPG. For fault simulation, enter the following:

```
BUILD> run build_model top_module_name
```

[Example 11-3](#) shows a typical transcript.

Example 11-3 run build_model Transcript

```
BUILD> run build_model my_asic
-----
Begin build model for topcut = my_asic ...
-----
End build model: #primitives=101004, CPU_time=13.90 sec,
Memory=34702381
-----
Begin learning analyses...
End learning analyses, total learning CPU time=33.02
```

Declaring Clocks

Although the nonscan functional stimuli provide all inputs, you may wish to declare clocks so that TetraMAX ATPG can perform its clock-related DRC checks. Declaring clocks is optional. Some clock violations found during run drc can affect the simulator and it may be necessary to remove add clock commands.

If certain ports in the functional stimuli are operated in pulsed fashion within a cycle, you may wish to provide this information to TetraMAX ATPG by declaring these ports to be clocks. For detailed information about declaring clocks, see “[Declaring Clocks](#)” on page 9-4.

A typical command sequence for declaring a clock is:

```
DRC> add clock 0 CLK
DRC> add clock 1 RESETB
```

Running DRC

Running DRC with nonscan functional test patterns tends to be simpler than running DRC for ATPG, because the additional check for scan chains and other ATPG-only checks do not need to be performed.

DRC for Nonscan Functional Test Patterns

For nonscan functional test patterns, if you have defined a clock, you do not need to specify an SPF unless it is necessary for defining port timing. To run DRC without a file, enter the following:

```
DRC> set drc -nofile
DRC> run drc
```

To run DRC with a file, enter the following:

```
DRC> run drc filename
```

Note the following:

- If you encounter DRC violations that apply to ATPG but are not relevant to the fault grading of nonscan functional patterns, adjust the DRC rule severity by using the `set rules rule_id warning` command, and then execute the `run drc` command again.
- In some cases, external functional VCDe patterns are not always compliant with TetraMAX behaviors -- particularly when the clocks are active at the same time that PIs change state. The basic rule is to define clocks in DRC if there are no C-rule violations in the design. If there are C violations, consider passing all signals as inputs and not defining any signals as clocks.

[Example 11-4](#) shows an example of `run drc` without a DRC file, together with the TetraMAX ATPG output.

Example 11-4 Running DRC for Nonscan Functional Patterns

```
DRC> set drc -nofile
DRC> run drc
-----
Begin scan design rules checking...
-----
Begin Bus/Wire contention ability checking...
Bus summary: #bus_gates=4, #bidi=4, #weak=0, #pull=0, #keepers=0
Contention status: #pass=0, #bidi=4, #fail=0, #abort=0,
#not_analyzed=0
Z-state status : #pass=0, #bidi=4, #fail=0, #abort=0,
#not_analyzed=0
Bus/Wire contention ability checking completed, CPU time=0.02
sec.
-----
Begin simulating test protocol procedures...
Test protocol simulation completed, CPU time=0.00 sec.
-----
Begin scan chain operation checking...
Scan chain operation checking completed, CPU time=0.00 sec.
-----
Begin clock rules checking...
Warning: Rule C3 (no latch transparency when clocks off) failed
5 times.
Clock rules checking completed, CPU time=0.02 sec.
-----
Begin nonscan rules checking...
Warning: Rule S23 (unobservable potential TLA) failed 5 times.
Nonscan cell summary: #DFF=0 #DLAT=10 tla_usage_type=none
Nonscan behavior: #CX=5 #LS=5
Nonscan rules checking completed, CPU time=0.03 sec.
-----
Begin contention prevention rules checking...
Contention prevention checking completed, CPU time=0.00 sec.

Begin DRC dependent learning...
DRC dependent learning completed, CPU time=0.00 sec.
```

```
-----  
DRC Summary Report  
-----  
Warning: Rule S23 (unobservable potential TLA) failed 5 times.  
Warning: Rule C3 (no latch transparency when clocks off) failed  
5 times.  
There were 10 violations that occurred during DRC process.  
Design rules checking was successful, total CPU time=0.21 sec.  
-----
```

DRC for Scan Functional Test Patterns

For scan functional test patterns, the SPF you specify should contain, at a minimum, the scan chain definitions, the waveform timing definitions, and the `load_unload` and `Shift` procedure definitions. You can define clocks, primary inputs constraints, and primary input equivalences on the command line or within the SPF, or you can use a combination of both.

To run DRC with a STIL procedure file, enter the following:

```
DRC> run drc filename
```

Reading the Functional Test Patterns

After you run DRC to enter TEST mode, the next step is to read in the functional test patterns. You can read in the patterns using the Set Patterns dialog box, or you can enter the `set patterns` command in the command line.

If you are reading external patterns in VCDE format, you need to specify the trigger conditions for measurement. In the Set Patterns dialog box, use the Strobe Position option and related options; or in the `set patterns` command, use the `-strobe` options. For more information, see “[Specifying Strobes for VCDE Pattern Input](#)” on page 11-15.”

Using the Set Patterns Dialog Box

To read in the functional test patterns using the Set Patterns dialog box,

1. From the menu bar, choose Patterns > Set Pattern Options. The Set Patterns dialog box appears.
For descriptions of these controls, see the online help for the `set patterns` command.
2. Click External.
3. In the Pattern File Name text field, enter the name of the pattern file, or locate it using the Browse button.

4. Click OK.
-

Using the set patterns Command

You can also read in the patterns using the command line. For example:

```
TEST> set patterns ext data.vcde -strobe rising CLK -strobe  
offset 50 ns
```

TetraMAX ATPG automatically determines the type of patterns being read and whether they are in standard or GZIP format, and handles all variations automatically.

For the complete syntax and option descriptions, see online Help for the `set patterns` command.

[Example 11-5](#) shows an example transcript.

Example 11-5 Example set patterns external Command

```
TEST> set pattern ext patterns.v  
End parsing Verilog file patterns.v with 0 errors;  
End reading 41 patterns, CPU_time = 0.02 sec, Memory = 2952
```

For examples of functional patterns, see “[Pattern Input](#)” on page 12-13.

Specifying Strobes for VCDE Pattern Input

Functional patterns in VCDE format do not contain any measure information. Therefore, when you read in VCDE patterns with the Set Patterns dialog box or the `set patterns` command, you need to specify the trigger conditions for measuring expected values. You can specify strobes that occur at a fixed periodic interval, or you can specify strobe trigger conditions based upon events occurring at a specified primary input port, output port, or bidirectional port.

In the Set Patterns dialog box, when you select External as the pattern source, the Strobe Position option and related options are displayed. These options apply to reading VCDE patterns only. The set of options changes according to the Strobe Position setting.

The Strobe Position can be set to any one of the following:

- None: This option is not supported for VCDE input.
- Period: Strobes occur at a fixed periodic interval, starting in each cycle at the offset value specified in the Offset field.

- Event: A strobe is triggered by any event occurring on the port specified in the Port Name field. Any event at that port causes a strobe, including a transition with no level change such as 1 to 1 or 0 to 0.
- Rising: A strobe is triggered by each transition to 1 on the port specified in the Port Name field. Any transition to 1 causes a strobe, including 0 to 1, 1 to 1, X to 1, or Z to 1.
- Falling: A strobe is triggered by each transition to 0 on the port specified in the Port Name field. Any transition to 0 causes a strobe, including 1 to 0, 0 to 0, X to 0, or Z to 0.

For the Event, Rising, and Falling strobe modes, you can specify an offset value in the Offset field. By default, the offset is 0, which causes the strobe to occur just before the trigger event. In other words, the measure occurs just before processing of the VCDE data change that is the trigger event.

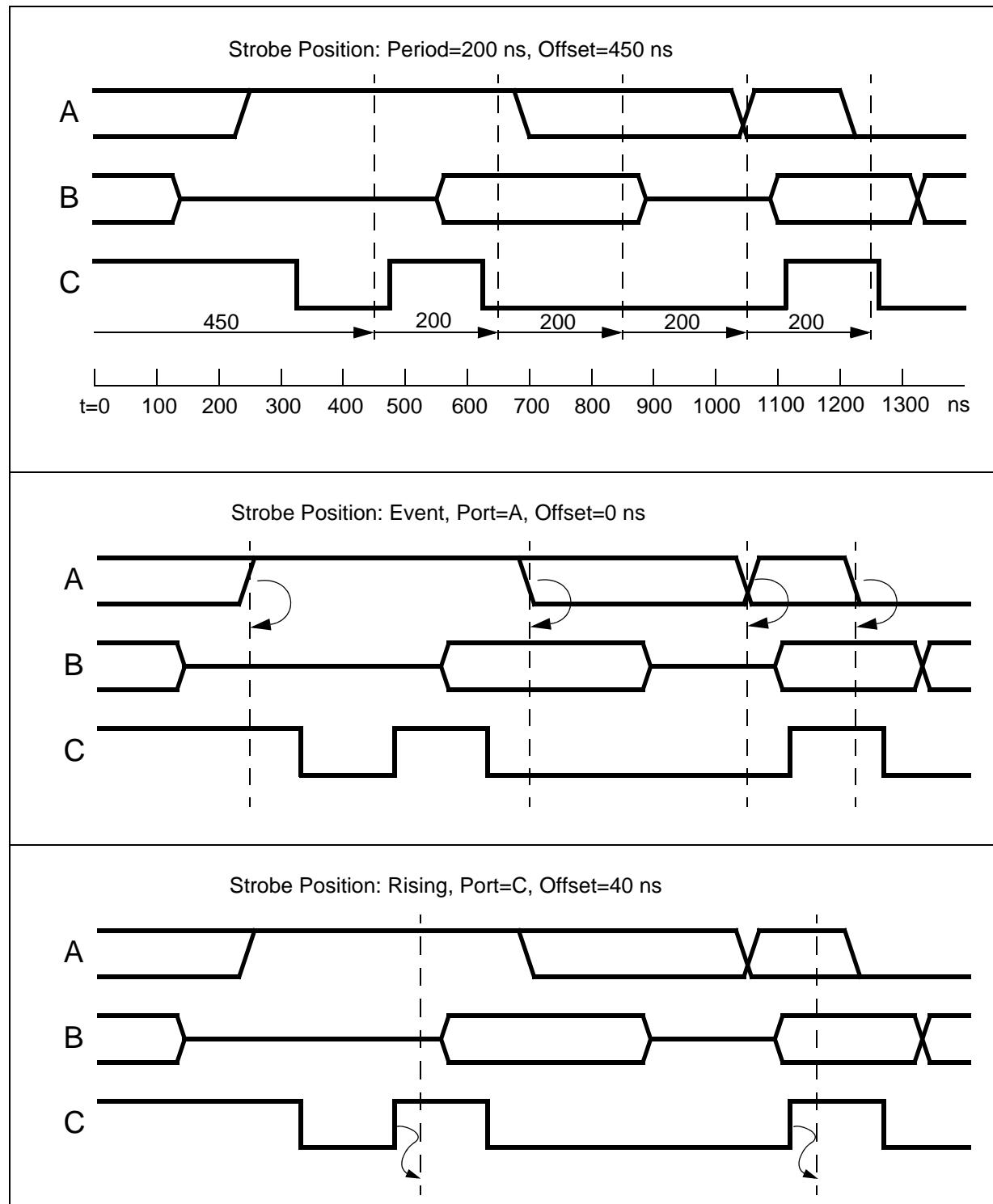
To make the strobe occur at a specific time after the trigger event, specify a positive offset value. Negative offsets for strobes are not supported.

Each period and offset setting must be a positive integer or zero. You specify the time units in the Unit fields: seconds, milliseconds, microseconds, nanoseconds, picoseconds, or femtoseconds.

To specify the strobes using the command-line interface, use the `-strobe` option in the `set patterns` command. For details on the command syntax, see the online help for the `set patterns` command.

[Figure 11-2](#) shows some timing diagrams with the strobe points resulting from various strobe specification settings.

Figure 11-2 VCDE Strobe Specification Examples



Each timing diagram shows the primary I/O signals A, B, and C. The vertical dashed lines represent the strobe times. In the first example, the strobes are periodic and independent of the data stream. In the second and third example, the strobes are based on port A and port C, respectively.

Initializing the Fault List

You can initialize the fault list by using the Add Faults dialog box, or you can enter the `add faults` command at the command line.

Using the Add Faults Dialog Box

To initialize the fault list using the Add Faults dialog box,

1. From the Faults menu, choose Add Faults. The Add Faults dialog box appears.
For descriptions of these controls, see the online help for the `add faults` command.
 2. To add all potential faults (the most common usage), click All.
 3. Click OK.
-

Using the `add faults` Command

You can also initialize the fault list for all faults using the `add faults` command. For example:

```
TEST> add faults -all
```

For the complete syntax and option descriptions, see the online help for the `add faults` command.

You can also define fault lists by reading faults ornofaults from a file or by defining specific hierarchical blocks for adding or removing faults. For example:

```
TEST> read faults saved_faults_file  
TEST> read faults saved_faults_file -retain
```

The double-read sequence shown in this example is necessary to restore the exact fault codes saved to the file.

In addition, you can read in a fault list generated by the TetraMAX ATPG patterns and thereby determine the cumulative fault grade for a combination of ATPG and functional test patterns. For details, see “[Setting Up the Fault List](#)” on page 4-20 and “[Combining ATPG and Functional Test Patterns](#)” on page 11-21.

Performing Good Machine Simulation

You should perform a good machine simulation using the functional patterns before running a fault simulation, to compare the TetraMAX simulation responses to the expected responses found in the patterns. If the good machine simulation reports errors, there is little value in proceeding to run fault simulation.

Setting Up the Good Machine Simulation

As part of setting up the good machine simulation, refer to contention checking as described in “[Choosing Settings for Contention Checking](#)” on page 4-22.

To set up other good machine simulation parameters, you can use the Run Simulation dialog box, or you can enter the `set simulation` and `run simulation` commands on the command line.

Using the Run Simulation Dialog Box

To set up the good machine simulation parameters using the Run Simulation dialog box,

1. Click the Simulation button in the command toolbar at the top of the TetraMAX main window. The Run Simulation dialog box appears.
For descriptions of these controls, see the online help for the `run simulation` command.
2. Select desired options.
3. Click Set to set the simulation options, or click Run to set the options and begin the good machine simulation.

Using the `set/run simulation` Commands

To set up the fault simulator from the command line, use a combination of the `set simulation` command and appropriate options of the `run simulation` command. For example:

```
DRC> set simulation -measure pat -oscillation 20 2 -verbose  
TEST> run simulation -sequential
```

For the complete syntax and option descriptions, see online Help for each command.

[Example 11-6](#) shows a transcript of a simulation run that has no mismatches between the simulated and expected data. For an example with simulation mismatches, see “[Comparing Simulated and Expected Values](#)” on page 12-21.

Example 11-6 Good Machine Simulation Transcript

```
TEST> run simulation -sequential
Begin sequential simulation of 36 external patterns.
Simulation completed: #patterns=36/102, #fail_pats=0(0),
#failing_meas=0(0)
```

Performing Fault Simulation

After performing a good machine simulation to verify that the functional patterns and expected data agree, you can perform a fault grading or fault simulation of those patterns. Performing fault simulation includes setting up the fault simulator, running the fault simulator, and reviewing the results.

Setting Up for Fault Simulation

The `set simulation` command described in section “[Performing Good Machine Simulation](#)” on page 11-19 sets the environment for fault simulation as well as for good machine simulation. Many of the options in the Run Simulation dialog box are also included in the Run Fault Simulation dialog box.

Using the Run Fault Simulation Dialog Box

To set up fault simulation parameters using the Run Fault Simulation dialog box,

1. Click the Fault Sim button in the command toolbar at the top of the TetraMAX main window. The Run Fault Simulation dialog box appears.
For descriptions of these controls, see the online help for the `run fault_sim` command.
2. Select desired options.
3. Click Set to close the dialog box and set the simulation options, or click Run to set the options and begin the faulty machine simulation.

Using the `run fault_sim` Command

You can also set up fault simulation parameters using the `run fault_sim` command. For example:

```
TEST> run fault_sim -store -sequential
```

For the complete syntax and option descriptions, see online Help for the `run fault_sim` command.

A typical transcript of a fault simulation run is shown in [Example 11-7](#).

Example 11-7 Fault Simulation Transcript

```
TEST> run fault_sim -sequential
-----
Begin sequential fault simulation of 4540 faults on 36 external
patterns.
-----
#faults    pass #faults    cum. #faults    test      process
simulated  detect/total  detect/active  coverage   CPU time
-----
1675        550    1675        550    3990    13.57%    3.72
3326        669    1651       1219    3321    29.36%    7.41
4540        390    1214       1609    2931    40.22%   11.13
Fault simulation completed: #faults_simulated=4540,
test_coverage=40.22%
```

You review test coverage in the same way as for ATPG. For details, see [“Reviewing Test Coverage” on page 4-29](#).

The following command generates a summary of fault simulation.

```
TEST> report summaries
```

Writing the Fault List

You write fault lists for fault simulation in the same way as you do for the ATPG flow. The following command writes (saves) a fault list.

```
TEST> write faults file.dat -all -uncollapsed -rep
```

Combining ATPG and Functional Test Patterns

If your design supports scan-based ATPG, you can create ATPG test patterns in addition to functional test patterns. Combining ATPG patterns with functional test patterns can often produce a more thorough and more complete set of test patterns than using either method alone.

If your design allows both ATPG and functional testing, you can combine the resulting test patterns by using one of the following methods:

- Create patterns independently for each test method, with no regard for testing overlap.
- Create ATPG patterns after you create functional patterns, with emphasis on testing only those faults not covered by the functional patterns.
- Create functional patterns after you create ATPG patterns, with emphasis on testing only those faults not covered by ATPG test patterns.

Creating Independent Functional and ATPG Patterns

If you do not want to combine the effects of functional test patterns and ATPG patterns, you can create them independently. The functional test patterns are fault-graded in an appropriate tool, and you obtain a test coverage value for the ATPG patterns that you create using TetraMAX ATPG.

To determine the test coverage overlap, you must perform a detailed comparison of the fault lists from both methods. You should expect overlap, which is not necessarily undesirable. In fact, the resulting redundancy in testing might be your desired goal.

Creating ATPG Patterns After Functional Patterns

If complete functional patterns are to be part of the test flow, use a combined approach with ATPG patterns following functional patterns. The goal of ATPG is to create patterns to test faults not tested by the functional patterns.

This is the typical flow:

1. Use TetraMAX ATPG to fault-grade the functional patterns.
2. Review the resulting test coverage.
3. Write the uncollapsed fault list resulting from the fault simulation.
4. Use TetraMAX ATPG to create ATPG patterns for the fault list you created in step 3.

[Example 11-8](#) shows a command file that implements this flow.

Example 11-8 Creating ATPG Patterns After Functional Patterns

```
//  
// --- ATPG follows Fault Grade flow  
  
read netlist my_design.v -del      // read netlist  
read netlist my_lib.v             // read library modules  
run build_model                  // form in-memory design image  
add clock 0 CLK                 // define clock  
add clock 1 RESETB              // define async reset  
run drc                         // DRC without a procedure file  
set pattern external b010.vin    // read in external patterns  
set simulation -measure pat     // set up for fault sim  
run simulation -sequential      // perform good machine simulation  
add faults -all                 // add all faults  
run fault_sim -sequential       // perform fault grade  
report summaries                // report results  
write fault pass1.flt -all -uncol -rep   // save fault list  
  
//  
// --- switch to SCAN-based ATPG for more patterns  
  
drc -force                      // return to DRC mode  
set patterns -delete             // clear out external patterns  
set patterns internal           // switch to int pattern generation  
run drc my_design.spf           // define scan chains and procedures  
read faults pass1.flt -retain   // start with fault list from pass1  
set atpg -abort 20 -merge high  // setup for ATPG  
run atpg                         // create ATPG patterns  
report summaries                // report coverage results  
write patterns pat.v -form verilog -replace // save patterns  
write fault pass2.flt -all -uncollapsed -rep // save cumulative  
fault list
```

Creating Functional Patterns After ATPG Patterns

Use a combined approach with functional patterns following ATPG patterns if you want to minimize the effort of creating functional test patterns. On a full-scan design, the ATPG patterns achieve a very high coverage and the functional patterns can be created to test for faults that are untestable with ATPG methods.

This is the typical flow:

1. Use TetraMAX ATPG to create ATPG patterns.
2. Review the resulting test coverage.
3. Save the uncollapsed fault list resulting from ATPG.
4. Save the collapsed fault list of the nondetected faults, which are the faults in the ND, AU, and PT categories. For an explanation of these categories, see “[Fault Categories and Classes](#)” on page 10-5.

5. Use the nondetected fault list to guide your construction of functional patterns to test for the remaining faults.
6. When the functional patterns are ready, fault-grade them using the uncollapsed fault list from the ATPG (generated in step 3 above) as the initial fault list.

[Example 11-9](#) shows a command file sequence that illustrates this flow.

Example 11-9 Creating Functional Patterns After ATPG Patterns

```

// --- ATPG before Fault Grade
//
read netlist my_design.v -del      // read netlist
read netlist my_lib.v              // read library modules
run build_model                    // form in-memory design image
add clock 0 CLK                   // define clock
add clock 1 RESETB                // define async reset
add pi constraint 1 TEST          // define constraints
run drc my_design.spf             // define scan chains and procedures
add faults -all                   // seed faults everywhere
run atpg -auto_compression       // create ATPG patterns
write patterns pat.v -form verilog -replace // save patterns
write fault pass1.flt -all -uncollapsed -rep // save cumulative
fault list
//
// --- switch to Fault Grade mode
//
drc -force                         // clocks will still be defined
remove pi constraints -all          // don't constrain when using
ext patterns
set drc -nofile
run drc                             // switch to test mode
set pattern external b010.vin        // read in external patterns
set simulation -measure pat         // set up for fault sim
run simulation -sequential          // perform good machine simulation
read faults pass1.flt               // seed the fault list
read faults pass1.flt -retain       // start with fault list from ATPG
run fault_sim -sequential          // perform fault grade
report summaries                     // report results
write fault pass2.flt -all -uncol -rep // save fault list

```

12

Test Pattern Data

A test pattern is a sequence of input values and expected output values that tests a device for the presence of faults or defects. TetraMAX generates test patterns and reports the test coverage of those patterns. It can also read in externally generated patterns.

This chapter contains the following sections:

- [Controlling ATPG](#)
- [Using Multiple-Session Test Generation](#)
- [Compressing Patterns](#)
- [Pattern Output](#)
- [Pattern Input](#)
- [Running Logic Simulation](#)
- [Per-Cycle Pattern Masking](#)

Controlling ATPG

TetraMAX lets you control the pattern generation effort, CPU limits, ATPG pattern sources, and ATPG modes using the Run ATPG dialog box (see ["Using the Run ATPG Dialog Box" on page 4-23](#)), or a combination of the `set atpg`, `set patterns`, and `set random_patterns` commands. For example:

```
...
TEST> set atpg -patterns 400 -abort_limit 5
...
TEST> set patterns random
...
DRC> set random_patterns -clock -none -length 3000
-observe master
...
```

For the complete syntax and option descriptions, see online Help for each command.

ATPG Modes

TetraMAX supports the following ATPG modes:

- Basic-Scan
- Fast-Sequential
- Full-Sequential

By default, only the Basic-Scan mode is used. For partial-scan designs, the optional Fast-Sequential and Full-Sequential modes can improve test coverage over the Basic-Scan mode by using multiple capture procedures between scan load and unload. These modes are more computationally intensive than the Basic-Scan mode.

Basic-Scan ATPG

The default ATPG mode is Basic-Scan. In this mode, TetraMAX operates as a full-scan, combinational-only ATPG tool. In other words, to get high test coverage, the sequential elements need to be scan elements.

Fast-Sequential ATPG

Fast-Sequential ATPG provides limited support for partial-scan designs (designs with some nonscan sequential elements). In this mode, multiple capture procedures are allowed between scan load and scan unload, allowing data to be propagated through nonscan sequential elements in the design. However, all clock and reset signals to these nonscan elements must be directly controllable at the primary inputs of the device.

You enable the Fast-Sequential mode and specify its effort level by using the `-capture_cycles` option of the `set atpg` command, as in the following example:

```
TEST> set atpg -capture_cycles n
```

The value *n* is a number from 0 to 10 that specifies the number of capture procedures allowed between scan load and unload. A value of 0 disables Fast-Sequential ATPG, resulting in Basic-Scan ATPG operation, which is the default behavior. A nonzero value results in an increase in test coverage where nonscan sequential elements are used.

Full-Sequential ATPG

Full-Sequential ATPG, like Fast-Sequential ATPG, supports multiple capture cycles between scan load and unload, and supports RAM and ROM models, thus increasing test coverage in partial-scan designs. However, clock and reset signals to the nonscan elements do not need to be controllable at the primary inputs and there is no specific limit on the number of capture cycles used between scan load and unload.

You enable the Full-Sequential mode by using the `-full_seq_atpg` option of the `set atpg` command, as in the following example:

```
TEST> set atpg -full_seq_atpg
```

The Full-Sequential mode supports a feature called *sequential capture*. Defining a sequential capture procedure in the STIL procedure file (SPF) lets you customize the capture clock sequence applied to the device during Full-Sequential ATPG. For example, you can define the clocking sequence for a two-phase latch design, where CLKP1 is followed by CLKP2. For more information, see “[Defining a Sequential Capture Procedure](#)” on page 9-26.

ATPG Algorithm Sequence

By default, only the Basic-Scan mode is enabled in TetraMAX. This mode is suitable for full-scan designs, where all sequential elements are scan elements.

For partial-scan designs, you might want to enable the Fast-Sequential mode, or both the Fast-Sequential and Full-Sequential modes, to improve fault coverage around nonscan elements. In that case, TetraMAX performs ATPG using a combination of modes, in the following sequence:

1. Basic-Scan ATPG
2. Fast-Sequential ATPG, if that mode is enabled
3. Full-Sequential ATPG, if that mode is enabled

This is the default ATPG sequence used with the `run atpg` command without any options, or when you click the Run button in the Run ATPG dialog box. This sequence starts with the Basic-Scan algorithm first to detect most of the faults, followed by the more computationally intensive algorithms to detect the remaining faults. This is usually the desired behavior.

You might want to run these algorithms separately in some cases. For example, you might want to try Basic-Scan first and then check the results. If test coverage is not high enough, you can then try Fast-Sequential to detect the remaining faults and check the results again. If the results are still not good enough, you can then try Full-Sequential.

To perform ATPG with just one mode, use the desired “run-only” option with the `run atpg` command:

```
TEST> run atpg basic_scan_only  
TEST> run atpg fast_sequential_only  
TEST> run atpg full_sequential_only
```

If you use the “run-only” options, be sure to use them in order: Basic-Scan, Fast-Sequential, Full-Sequential. Otherwise, the ATPG run might take a very long time because the Fast-Sequential and Full-Sequential algorithms are more computationally intensive than Basic-Scan.

If you run ATPG using just `run atpg` without any options, TetraMAX always uses the Basic-Scan algorithm first, followed by the enabled optional algorithms.

Each run-only option overrides the current `set atpg` configuration. If Fast-Sequential ATPG is not already enabled when you use the `run atpg fast_sequential_only` command, TetraMAX automatically runs the `set atpg -capture_cycles 4` command to enable Fast-Sequential ATPG with the effort level set to medium. This changes the ATPG settings and therefore affects the behavior of subsequent `run atpg` commands in the current session.

Similarly, if Full-Sequential ATPG is not already enabled when you use the `run atpg full_sequential_only` command, TetraMAX automatically runs the `set atpg -full_seq_atpg` command to enable Full-Sequential ATPG. Again, this affects the operation of subsequent `run atpg` commands.

Full-Sequential ATPG Limitations

The following limitations apply to Full-Sequential ATPG:

- It supports stuck-at faults, transition faults, and path delay faults, but not IDQ or bridging faults.
- It does not support the `-fault_contention` option of the `set buses` command.

- It does not support the `-nocapture`, `-nopreclock`, and `-retain_bidi` options of the `set contention` command.
- Patterns generated by Full-Sequential ATPG are not compatible with failure diagnosis using the `run diagnosis` command.
- The following options of the `set simulation` command have not been implemented for Full-Sequential simulation:

```
-bidi_fill | -strong_bidi_fill  
-measure <sim|pat>  
-oscillation
```

Random Decision Option

You can use the Random Decision check box in the Run ATPG dialog box to specify how TetraMAX makes the initial choice for any algorithm decision concerning ATPG pattern generation. By default, Random Decision is off and the initial choice is made based on controllability criteria. Checking Random Decision for ATPG pattern compression can result in a smaller number of patterns.

The following command is equivalent to checking the Random Decision check box:

```
TEST> set atpg -decision random
```

Specifying a Test Coverage Target Value

By default, TetraMAX processes faults and generates patterns in an attempt to achieve 100 percent test coverage. You can specify a lower test coverage target value by entering a decimal number between 0 and 100.0 in the Coverage % field of the Run ATPG dialog box or by issuing a command similar to the following:

```
TEST> set atpg -coverage 88.5
```

You might want to specify a test coverage lower than 100 percent if you want to produce fewer patterns, your design requirements are satisfied with a lower coverage, or you want an alternative to using a pattern limit for decreasing CPU time.

Limiting the Number of Patterns

By default, the number of ATPG patterns TetraMAX produces is limited only by the RAM and disk space of your computer or workstation. You can specify a limit on the number of patterns by entering an integer value in the Max Patterns field of the Run ATPG dialog box, or by issuing a command similar to the following:

```
TEST> set atpg -patterns 1234
```

If there is a pattern limit in effect, you can turn it off by specifying the value 0 as the pattern limit.

Limiting the Number of Aborted Decisions

The search for a pattern by the ATPG algorithm involves making a decision and certain assumptions, setting inputs and scan chain values, and determining whether controllability and observability can be attained. When an assumption is proved false or some restriction or blockage is encountered, the algorithm backs up, remakes the decision, and proceeds until the abort limit is reached or a pattern is found to detect the fault.

To control the level of effort used in searching for a pattern to detect a specific fault, use the `-abort_limit` option of the `set atpg` command or enter a number in the Abort Limit field of the Run ATPG dialog box. The default limit is 10. Higher numbers indicate higher levels of effort.

The default value of 10 has been found to return reasonable results for most designs. Some possible reasons for adjusting the abort limit are:

- You want a quick estimate of total coverage (see “[Quickly Estimating Test Coverage](#)” on [page 4-24](#)).
- You find that after performing pattern generation, you have ND (not detected) faults remaining. See “[Analyzing the Cause of Low Test Coverage](#)” on [page 20-8](#).
- You have aborted buses reported during design rule checking (DRC). See “[Analyzing Buses](#)” on [page 7-37](#).
- You are using a high compression effort and you want to generate enough patterns to ensure that the CPU time spent merging patterns is worthwhile.

Using Multiple-Session Test Generation

You can create patterns using multiple sessions as well as using multiple passes. For an example of using multiple passes, see “[Increasing Effort Over Multiple Passes](#)” on [page 4-27](#).

Here are some examples of situations where you might use multiple sessions:

- Your pattern set is too large for the tester, so you try an additional compression effort. If that is unsuccessful, you truncate the pattern set to a size that fits the tester.

- Your pattern set is too large for the tester, so you split the pattern set into two or more smaller sets.
- You have 2,000 patterns and a simulation failure occurs around pattern 1,800. You want to look at the problem in more detail but do not want to take the time to resimulate 1,799 patterns, so you read in the original patterns and write out the pattern with the error, plus one pattern before and after for good measure.
- You have three separate pattern files from previous attempts, and you want to merge them all into a single pattern file that eliminates duplications.
- Your design has asymmetrical scan chains or other irregularities, and you want to create separate pattern files with different environments of scan chains, clocks, and PI constraints.
- You have changed the conditions under which your existing patterns were generated (for example, by using a different fault list). You want to see how the existing patterns perform with the new fault list.

Some of these examples are explained in more detail in the following sections.

Splitting Patterns

To split patterns, reestablish the exact environment under which the patterns were generated. You do not need to restore a fault list. After achieving test mode, you can split the patterns at the 500-pattern mark by using a command sequence similar to the following:

```
TEST> set patterns external session_1_patterns
TEST> write patterns pat_file1 -last 499 -external
TEST> write patterns pat_file2 -first 500 -external
```

Extracting a Pattern Subrange

To extract part of the pattern, you use the same environment setup rules as for splitting patterns, except that you use the `-first` and `-last` options of the `write patterns` command when writing patterns. After achieving test mode, you can extract a subrange of three patterns using a command sequence similar to the following:

```
TEST> set patterns external session_1_patterns
TEST> write pat subset_file -first 198 -last 200 -ext
```

Merging Multiple Pattern Files

You can merge multiple pattern files only if all the files were generated under the same conditions of clocks and constraints and have identical scan chains. The fault lists do not have to match. To accomplish the merge, reestablish the environment and choose the final fault list to be used. Patterns in the external files are eliminated during the merge effort if they do not detect any new faults based on the current fault list.

After you achieve test mode and initialize a starting fault list, execute commands similar to the following:

```
TEST> set patterns external patterns_1
TEST> run atpg
TEST> set patterns external patterns_2
TEST> run atpg
TEST> set patterns external patterns_3
TEST> run atpg
TEST> report summaries
```

Alternatively, if you want to avoid running ATPG repeatedly or want to avoid potentially dropping patterns, then you can replace the `run atpg` commands with `run simulation -store` commands. For example,

```
TEST> set patterns -delete
TEST> set patterns external patterns_1
TEST> run simulation -store
TEST> set patterns external patterns_2
TEST> run simulation -store
TEST> set patterns external patterns_3
TEST> run simulation -store
TEST> report summaries
```

This alternative approach copies and appends the patterns from an external buffer into an internal one without performing ATPG and without any potential dropping of patterns.

Using Pattern Files Generated Separately

Using multiple sessions to generate patterns, you can use different definitions for clocks, PI constraints, or even scan chains to obtain two or more separate sets of ATPG patterns that achieve a cumulative test coverage effect. The key to determining cumulative test coverage is sharing and reusing the fault list from one session to another.

For example, suppose that you want to create separate pattern files for a design that has the following characteristics:

- 20 scan chains, evenly distributed so that they all are between 240 and 250 bits in length

- 1 boundary scan chain that is 400 bits in length
- 1,500 patterns that have been run through ATPG and produced 98 percent test coverage
- A tester cycle budget of 500,000 cycles

Some rough calculations indicate that the 1,500 patterns require approximately 600,000 tester cycles ($400 \times 1,500$), which exceeds the tester cycle budget. One possible solution is to set up two different environments, one that uses all scan chains and another that eliminates the definition of the 400-bit long scan chain.

Your two ATPG sessions are organized like the following:

- Session 1: You create an SPF that defines all scan chains except the 400-bit chain. You proceed to generate maximum coverage using minimum patterns. After saving the patterns and before exiting, you save the final fault list, as in the following command:

```
TEST> write faults sess1_faults.gz -all- uncollapsed \
      -compress gzip
```

- Session 2: You create an SPF that defines all scan chains. You read in the fault list saved in Session 1, as in the following command:

```
TEST> read faults sess1_faults.gz -retain_code
```

The first session probably achieves less than the original 98 percent coverage, but still consumes approximately 1,500 patterns. More important, the combination of the two sessions matches the original 98 percent test coverage but generates fewer than 20 percent of the original patterns for the second session (about 300 patterns). The total test cycles for both sets of patterns are now as follows:

$$(1,500 \times 250) + (300 \times 400) = 495,000 \text{ tester cycles}$$

The number of patterns has increased from 1,500 to 1,800, but the number of tester cycles has decreased by more than 100,000 and the original test coverage has been maintained.

Note:

When you pass a fault list from one session to another and perform pattern compression, you will see different test coverage results before and after pattern compression. Pattern compression performs a fault grade on the patterns that exist only at that point in time. After pattern compression, the test coverage statistics reflect the coverage of the current set of patterns. The correct cumulative test coverage for both sessions is the output from the last report summaries command executed before any pattern compression.

Compressing Patterns

Test patterns produced by ATPG techniques usually have some amount of redundancy. You can usually reduce the number of patterns significantly by compressing them, which means eliminating some patterns that provide no additional test coverage beyond what has been achieved by other patterns.

Dynamic pattern compression is performed while patterns are being created. With this technique, each time a new pattern is created, an attempt is made to merge the pattern with one of the existing patterns within the current cluster of 32 patterns in the pattern simulation buffer.

To enable dynamic pattern compression, use the `-merge` option of the `set atpg` command or the equivalent options in the Run ATPG dialog box.

Balancing Pattern Compaction and CPU Runtime

Normally, a reasonable number of passes of static compression produces a smaller number of patterns. However, this reduced pattern count results in a CPU runtime penalty.

For a compromise between pattern compactness and CPU runtime, you can use the `-auto_compression` option in the `run atpg` command. This option selects an automatic algorithm designed to produce reasonably compact patterns and high test coverage, using a reasonable amount of CPU time. For more information, look in the online help under the index topic “Automatic ATPG.”

To obtain the maximum test coverage while achieving a reasonable balance of CPU time and patterns,

1. Obtain an estimate of test coverage using the Quick Test Coverage technique (see [“Quickly Estimating Test Coverage” on page 4-24](#)). If you are not satisfied with the estimate, determine the cause of the problem and obtain satisfactory test coverage before you attempt to achieve minimum patterns.
2. Set Abort Limit to 100–300.
3. Set Merge Effort to High.
4. Execute `run atpg -auto_compression`.
5. Examine the results. If there are still some NC or NO faults remaining, increase the Abort Limit by a factor of 2 and execute `run atpg` again.

Compression Reports

[Example 12-1](#) shows a dynamic compression report generated using the `-verbose` option of the `set atpg` command. The `-verbose` option produces this additional information:

- The pattern number within the current group of 32 patterns
- The number of fault detections successfully merged into the pattern (#merges)
- The number of faults that were attempted but could not be merged into the current pattern, which matches the merge iteration limit unless the number of faults remaining is less than this limit (#failed_merges)
- The number of faults remaining in the active fault list (#faults)
- The CPU time used in the merge process

If you monitor the verbose information, you will eventually see a point at which the number of merges approaches zero. At this point, stop the process and reduce the merge effort or disable it because the effect is not producing sufficient benefit to justify the CPU effort expended.

Example 12-1 Verbose Dynamic Compression Report

```
TEST> set atpg -patterns 150 -merge medium -verbose
TEST> run atpg
ATPG performed for 72440 faults using internal pattern source.
-----
#patterns      #faults      #ATPG faults    test      process
stored        detect/active   red/au/abort  coverage   CPU time
-----  -----  -----  -----  -----
Begin deterministic ATPG: abort_limit = 5...
Patn  0: #merges=452  #failed_merges=100  #faults=40083  CPU=1.51
sec
Patn  1: #merges=637  #failed_merges=100  #faults=33938  CPU=2.82
sec
Patn  2: #merges=380  #failed_merges=100  #faults=30325  CPU=3.67
sec
Patn  3: #merges=211  #failed_merges=100  #faults=27403  CPU=4.52
sec
Patn  4: #merges=115  #failed_merges=100  #faults=25827  CPU=5.16
sec
Patn  5: #merges=798  #failed_merges=100  #faults=24633  CPU=6.66
sec
Patn  6: #merges=97   #failed_merges=100  #faults=23436  CPU=7.19
sec
Patn  7: #merges=82   #failed_merges=100  #faults=22431  CPU=7.69
sec
Patn  8: #merges=73   #failed_merges=100  #faults=21348  CPU=8.27
sec
Patn  9: #merges=77   #failed_merges=100  #faults=20340  CPU=8.83
sec
Patn 10: #merges=58   #failed_merges=100  #faults=19906  CPU=9.34
sec
Patn 11: #merges=65   #failed_merges=100  #faults=18231  CPU=9.97
sec
Patn 12: #merges=39   #failed_merges=100  #faults=17414  CPU=10.44
sec
Patn 13: #merges=50   #failed_merges=100  #faults=16759  CPU=10.96
sec
Patn 14: #merges=35   #failed_merges=100  #faults=16383  CPU=11.28
sec
Patn 15: #merges=36   #failed_merges=100  #faults=15994  CPU=11.62
sec
Patn 16: #merges=29   #failed_merges=100  #faults=15588  CPU=11.99
sec
Patn 17: #merges=28   #failed_merges=100  #faults=15112  CPU=12.36
sec
Patn 18: #merges=36   #failed_merges=100  #faults=14763  CPU=12.69
sec
Patn 19: #merges=34   #failed_merges=100  #faults=14510  CPU=13.02
sec
Patn 20: #merges=21   #failed_merges=100  #faults=14289  CPU=13.35
sec
Patn 21: #merges=342  #failed_merges=100  #faults=13933  CPU=14.18
sec
Patn 22: #merges=37   #failed_merges=100  #faults=13711  CPU=14.50
sec
Patn 23: #merges=24   #failed_merges=100  #faults=13570  CPU=14.79
sec
Patn 24: #merges=24   #failed_merges=100  #faults=13438  CPU=15.05
```

```

sec
Patn 25: #merges=20    #failed_merges=100  #faults=13294  CPU=15.32
sec
Patn 26: #merges=23    #failed_merges=100  #faults=13145  CPU=15.59
sec
Patn 27: #merges=134   #failed_merges=57   #faults=12687  CPU=16.93
sec
Patn 28: #merges=27    #failed_merges=100  #faults=12552  CPU=17.28
sec
Patn 29: #merges=23    #failed_merges=100  #faults=12410  CPU=17.54
sec
Patn 30: #merges=29    #failed_merges=100  #faults=12296  CPU=17.82
sec
Patn 31: #merges=22    #failed_merges=100  #faults=12202  CPU=18.09
sec
32      51756 20684      0/0/1    72.80%    19.37
Patn  0: #merges=19    #failed_merges=100  #faults=11909  CPU=19.65
sec
Patn  1: #merges=34    #failed_merges=100  #faults=11755  CPU=19.93
sec
Patn  2: #merges=17    #failed_merges=100  #faults=11666  CPU=20.22
sec

```

Pattern Output

After you generate and compress a pattern set, you will probably want to write the patterns to a file. To do this, use the Write Patterns dialog box or the `write patterns` command. For more information on the command syntax, file format options, and other pattern-writing options, see “[Writing ATPG Patterns](#)” on page 4-31.

The timing of the ATPG patterns written is controlled entirely from the STIL procedure file specified with the last `run drc` command. For more information, see “[Defining Basic Signal Timing](#)” on page 9-14.

Note:

For information on translating adaptive scan patterns into normal scan-mode patterns, see the section “[Pattern File](#)” on page 5 in Chapter 19, “[Diagnosing Manufacturing Test Failures](#).”

Pattern Input

TetraMAX can read in both ATPG (scan) patterns and functional nonscan patterns. To read in a pattern file, use the Set Patterns dialog box or the `set patterns external` command. For details, see “[Selecting the Pattern Source](#)” on page 4-21.

TetraMAX can read ATPG patterns in the same formats it uses to write ATPG patterns: STIL, Verilog, WGL, and VHDL. To see each format and syntax, generate some patterns on a small circuit and write them out in each of the output formats. For more information, see “[Recognizable Format](#)” on page 11-6.

Pattern input examples in STIL, Verilog, and WGL formats are shown and explained next.

STIL Functional Pattern Input

TetraMAX accepts pattern input in STIL format using some limited variations of the example shown in [Example 12-2](#).

The supported format has the following characteristics:

- The `Header` block is optional.
- The `Signals` block is required.
- The `SignalGroups` block is optional.
- The `Timing` block, with at least one `WaveformTable`, is required to define the point in the cycle where the clocks pulse and the outputs are measured.
- The `PatternBurst`, `PatternExec`, and `Pattern` blocks are used to set up a single block of functional patterns.
- The `Pattern` block consists only of `w` and `v` statements.

Example 12-2 Functional Pattern Input in STIL

```
STIL 0.23;
Header { Title "Functional Patterns for Design-X"; }
Signals {
    d11 In;    d10 In;    d9 In;    d8 In;    d7 In;
    d6 In;    d5 In;    d4 In;    d3 In;    d2 In;
    d1 In;    d0 In;    i3 In;    i2 In;    i1 In;
    i0 In;    oe In;    rld In;    ccen In;    ci In;
    cp In;    cc In;    sdi1 In;    sdi2 In;    se In;
    tsel In;   y11 Out;   y10 Out;   y9 Out;   y8 Out;
    y7 Out;   y6 Out;   y5 Out;   y4 Out;   y3 Out;
    y2 Out;   y1 Out;   y0 Out;   full Out;  pl Out;
    map Out;  vect Out;  sdo1 Out;  sdo2 Out;  tout Out;
    vcoctl Out;
}
SignalGroups {
    input_ports = 'd11 + d10 + d9 + d8 + d7 + d6 + d5 + d4 + d3 + d2
                  + d1 + d0 + i3 + i2 + i1 + i0 + oe + rld + ccen + ci
                  + cp + cc + sdi1 + sdi2 + se + tsel';
    output_ports = 'y11 + y10 + y9 + y8 + y7 + y6 + y5 + y4 + y3 + y2
                  + y1 + y0 + full + pl + map + vect + sdo1 + sdo2 + tout
                  + vcoctl';
}
Timing {
```

```

WaveformTable TSET1 {
    Period '250ns';
    Waveforms {
        input_ports { 01Z { '0ns' D/U/Z; } }
        cp { P { '0ns' D; '62ns' U; '187ns' D; } }
        output_ports { X { '0ns' X; } }
        output_ports { LHT { '0ns' X; '240ns' L/H/T; } }
    }
}
PatternBurst functional_burst { FUNC_BLOCK_1; }
PatternExec { Timing; PatternBurst functional_burst; }
Pattern FUNC_BLOCK_1 {
    W TSET1;
    V {
        d1=0; d9=0; sdo2=X; sdi2=0; y9=X; y1=X; d6=0; cp=0; i3=0; cc=0;
        vcoctl=X; y6=X; ci=1; d3=0; i0=0; d11=0; y3=X; y11=X; oe=0; d0=0;
        d8=0; vect=H; map=H; y8=X; y0=X; i2=0; d5=0; sdo1=X; y5=X; sdi1=0;
        tout=X; d2=0; y2=X; d7=0; d10=0; full=X; y7=X; tsel=0; ccen=0;
        se=0;
        y10=X; rld=0; i1=0; d4=0; y4=X; }
        V {tsel=1; tout=T; }
        V {d3=1; y3=H; map=L; i1=1; }
        V {sdo2=L; d3=0; i0=0; y0=H; i2=1; }
        V {sdo2=H; y1=L; i0=1; y3=L; i2=0; d4=1; y4=H; }
        V {y1=H; i3=0; cc=1; y0=L; d4=0; }
        V {y0=H; d10=1; }
        V {sdo2=L; y1=H; i0=0; i2=1; }
        V {y0=H; }
        V {y0=H; }
        V {y1=H; y0=L; }
        V {y0=H; }
        V {y1=L; y3=H; y0=L; sdo1=L; y2=L; }
        V {y0=H; full=L; }
        V {y1=L; y0=L; y2=H; }
        V {y1=H; y0=L; }
        V {y1=L; y3=L; y0=L; y2=L; y4=H; }
        V {y0=H; }
    }
}

```

Verilog Functional Pattern Input

TetraMAX accepts pattern input in Verilog format using some limited variations of the example shown in [Example 12-3](#).

The supported format has the following characteristics:

- `timescale is optional.
- A vector is used for primary outputs, expected data, and mask.
- Each clock capture cycle that can perform a measure is defined in an event procedure.
- Cycles with a measure and no clocks are defined in event procedures.

- Cycles with a clock and no measures are defined in event procedures.
- The data stream occurs within an `initial/end` block.
- Assignment to the variable `pattern` allows TetraMAX to track the pattern boundaries.

Example 12-3 Functional Pattern Input in Verilog

```
`timescale 1 ns / 100 ps

module amd2910_test;
    reg [0:8*9] POnames [19:0];
    integernofails, bit, pattern;
    wire [11:0] d;
    wire [3:0] i;
    wire oe, tsel, ci, rld, ccen, cc, sdi1, sdi2, se, cp;
    wire [11:0] y;
    wire full, pl, map, vect, tout, vcoctl, sdo1, sdo2;
    wire [19:0] PO; // primary output vector
    reg [19:0] XPCT; // expected data vector
    reg [19:0] MASK; // compare mask vector
    assign PO[0] = y[0];
    assign PO[1] = y[1];
    assign PO[2] = y[2];
    assign PO[3] = y[3];
    assign PO[4] = y[4];
    assign PO[5] = y[5];
    assign PO[6] = y[6];
    assign PO[7] = y[7];
    assign PO[8] = y[8];
    assign PO[9] = y[9];
    assign PO[10] = y[10];
    assign PO[11] = y[11];
    assign PO[12] = full;
    assign PO[13] = pl;
    assign PO[14] = map;
    assign PO[15] = vect;
    assign PO[16] = tout;
    assign PO[17] = vcoctl;
    assign PO[18] = sdo1;
    assign PO[19] = sdo2;

    // instantiate the device under test

    amd2910 dut (.o_y11(y[11]), .o_y10(y[10]), .o_y9(y[9]),
        .o_y8(y[8]), .o_y7(y[7]), .o_y6(y[6]), .o_y5(y[5]),
        .o_y4(y[4]), .o_y3(y[3]), .o_y2(y[2]), .o_y1(y[1]),
        .o_y0(y[0]), .o_full(full), .o_pl(pl), .o_map(map),
        .o_vect(vect), .o_sdo1(sdo1), .o_sdo2(sdo2), .tout(tout),
        .vcoctl(vcoctl), .i_d11(d[11]), .i_d10(d[10]), .i_d9(
d[9]),
        .i_d8(d[8]), .i_d7(d[7]), .i_d6(d[6]), .i_d5(d[5]),
        .i_d4(d[4]), .i_d3(d[3]), .i_d2(d[2]), .i_d1(d[1]),
        .i_d0(d[0]), .i_i3(i[3]), .i_i2(i[2]), .i_i1(i[1]),
        .i_i0(i[0]), .i_oe(oe), .i_rld(rld), .i_ccen(ccen),
        .i_ci(ci), .i_cp(cp), .i_cc(cc), .i_sdi1(sdi1),
        .i_sdi2(sdi2),
        .i_se(se), .tsel(tsel));
    end

    // define pulse on "i_cp"
    event pulse_i_cp;
    always @ pulse_i_cp begin
        #500 cp = 1;
        #100 cp = 0;
    end
endmodule
```

```

// define capture event without a clock
event capture;
always @ capture begin
    #0;
    #950; ->measurePO;
end

// define how to measure outputs
event measurePO;
always @ measurePO begin
    if ((XPCT&MASK) !== (PO&MASK)) begin
        $display($time, " ----- ERROR(S) during pattern %0d ---");
    end
    for (bit = 0; bit < 20; bit=bit + 1) begin
        if((XPCT[bit]&MASK[bit]) !== (PO[bit]&MASK[bit])) begin
            $display($time, " : %0s (output %0d), expected %b,
got %b",
                    POnames[bit],           bit, XPCT[bit],   PO[bit]);
            nofails = nofails + 1;
        end
    end
end
end

event capture_i_cp;
always @ capture_i_cp begin
    #0;
    #500 cp = 1; // i_cp
    #100 cp = 0;
    #350; ->measurePO;
end

initial begin
    nofails = 0;
// --- initialize port name table
    POnames[0] = "Y0"; POnames[1] = "Y1"; POnames[2] = "Y2";
    POnames[3] = "Y3"; POnames[4] = "Y4"; POnames[5] = "Y5";
    POnames[6] = "Y6"; POnames[7] = "Y7"; POnames[8] = "Y8";
    POnames[9] = "Y9"; POnames[10] = "Y10"; POnames[11] = "Y11";
    POnames[12] = "full"; POnames[13] = "pl"; POnames[14] = "map";
    POnames[15] = "vect"; POnames[16] = "tout"; POnames[17] =
"vcoctl";
    POnames[18] = "sdo1"; POnames[19] = "sdo2";

    #0; pattern= 0;
    se=0; sdil2=0; sdil1=0; cc=0; ccen=0; ci=0; tsel=0; oe=0;
    cp = 0; i=4'b0010; rld=1; d=12'b0000000000111;
    XPCT=20'bXXXX1011000000000001; MASK=20'b000000000000000000000000;
    ->pulse_i_cp;

    #1000; pattern= 1; i=4'b1110; d=12'b000000000000;
    ->pulse_i_cp;

    #1000; pattern= 2; i=4'b0000; oe=0;
    ->capture;

    #1000; pattern= 3; i=4'b0010; oe=1;

```

```

d=12'b000000000001; XPCT=20'bXXXX1011000000000001;
MASK=20'b000011111111111111;
->capture_i_cp;

#1000; pattern= 4;
d=12'b000000000010; XPCT=20'bXXXX1011000000000010;
MASK=20'b000011111111111111;
->capture_i_cp;

#1000; pattern= 5;
d=12'b000000000100; XPCT=20'bXXXX1011000000000100;
MASK=20'b000011111111111111;
->capture_i_cp;

#1000;
$display("Simulation of %0d cycles completed with %0d errors",
         pattern, nofails );
$finish;
end
endmodule

```

WGL Functional Pattern Input

TetraMAX accepts pattern input in WGL format using some limited variations of the example shown in [Example 12-3](#).

The supported format has the following characteristics:

- The `waveform` function is required.
- The `pmode` function is optional.
- The `signal` block is required.
- The `timeplate` block is required.
- The `pattern` block consists of simple vectors applied sequentially.

Example 12-4 Functional Pattern Input in WGL

```

waveform funct_1
pmode[last_drive];

signal
    TEST : input; RESET_B : input; EXTS1 : input; EXTS0 : input;
    LOBAT : input; SS_B : input; SCK : input; MOSI : input;
    EXTAL : input; TOUTEN : input; TOUTSEL : input;
    XTAL : output; MISO : output; READY_B : output;
    CLKOUT : output; SYMCLK : output; S7 : output;
    S6 : output; S5 : output; S4 : output;
    S3 : output; S2 : output; S1 : output;
    S0 : output; TOUT3 : output; TOUT2 : output;
    TOUT1 : output; TOUT0 : output;
end

timeplate tts0 period 500ns
    TEST := input[0ps:P, 200ns:S];
    RESET_B := input[0ps:P, 200ns:S];
    EXTS1 := input[0ps:P, 200ns:S];
    EXTS0 := input[0ps:P, 200ns:S];
    LOBAT := input[0ps:P, 200ns:S];
    SS_B := input[0ps:P, 200ns:S];
    SCK := input[0ps:P, 200ns:S];
    MOSI := input[0ps:P, 200ns:S];
    EXTAL := input[0ps:P, 100ns:S];
    TOUTEN := input[0ps:P, 200ns:S];
    TOUTSEL := input[0ps:P, 200ns:S];
    XTAL := output[0ps:X, 450ns:Q, 451ns:X];
    MISO := output[0ps:X, 450ns:Q, 451ns:X];
    READY_B := output[0ps:X, 450ns:Q, 451ns:X];
    CLKOUT := output[0ps:X, 450ns:Q, 451ns:X];
    SYMCLK := output[0ps:X, 450ns:Q, 451ns:X];
    S7 := output[0ps:X, 450ns:Q, 451ns:X];
    S6 := output[0ps:X, 450ns:Q, 451ns:X];
    S5 := output[0ps:X, 450ns:Q, 451ns:X];
    S4 := output[0ps:X, 450ns:Q, 451ns:X];
    S3 := output[0ps:X, 450ns:Q, 451ns:X];
    S2 := output[0ps:X, 450ns:Q, 451ns:X];
    S1 := output[0ps:X, 450ns:Q, 451ns:X];
    S0 := output[0ps:X, 450ns:Q, 451ns:X];
    TOUT3 := output[0ps:X, 450ns:Q, 451ns:X];
    TOUT2 := output[0ps:X, 450ns:Q, 451ns:X];
    TOUT1 := output[0ps:X, 450ns:Q, 451ns:X];
    TOUT0 := output[0ps:X, 450ns:Q, 451ns:X];
end

pattern group_ALL (TEST,RESET_B,EXTS1,EXTS0,LOBAT,SS_B,
                    SCK,MOSI,EXTAL,TOUTEN,TOUTSEL,XTAL,
                    MISO,READY_B,CLKOUT,SYMCLK,S7,S6,
                    S5,S4,S3,S2,S1,S0,TOUT3,TOUT2,
                    TOUT1,TOUT0)
    vector(0, 0ps, tts0)   := [0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X
X X X X X X X X X ] (0ps);
    vector(1, 500ns, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X X X
X X X X ] (500ns);
    vector(2, 1us, tts0)  := [0 0 0 0 0 0 0 0 0 0 0 1 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z ] (1us);

```

```

vector(3, 1.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 X 0 Z Z Z Z Z Z Z Z
    Z Z Z Z ] (1.5uS);
vector(4, 2uS, tts0)   := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z ] (2uS);
vector(5, 2.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z Z Z Z
    Z Z Z Z ] (2.5uS);
vector(6, 3uS, tts0)   := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z ] (3uS);
vector(7, 3.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z Z Z Z
    Z Z Z Z ] (3.5uS);
vector(8, 4uS, tts0)   := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z ] (4uS);
vector(9, 4.5uS, tts0) := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z Z Z Z Z
    Z Z Z Z ] (4.5uS);
vector(10, 5uS, tts0)  := [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z ] (5uS);
end

end

```

Running Logic Simulation

Using TetraMAX, you can run logic simulation and use the Graphical Schematic Viewer (GSV) to view the logic simulation results.

For combinational and sequential patterns, you can

- Perform logic simulation using either the internal or external pattern set.
- Check simulated against expected values from the patterns.
- Perform simulation in the presence of a single failure point to determine the patterns that would show differences.
- View the effect of any single point of failure for any single pattern.

For combinational patterns, you can also view the logic simulation value from any single pattern in the most recent 32 patterns in the simulation buffer.

For sequential patterns, you can also save the logic simulation value from any range of patterns and view this data.

Comparing Simulated and Expected Values

You can compare the simulation results against the expected values contained in the patterns during logic simulation. To do this, use the Compare option in the Run Simulation dialog box, or the `-nocompare` option of the `run simulation` command. For more information, see [“Performing Good Machine Simulation” on page 11-19](#).

[Example 12-5](#) shows a transcript of a simulation run that had no comparison errors; 139 patterns were simulated with zero failures.

Example 12-5 Simulation With No Comparison Errors

```
TEST> run simulation
Begin good simulation of 139 internal patterns.
Simulation completed:
#patterns=139, #fail_pats=0(0),
#failing_meas=0(0), CPU time=4.61
```

[Example 12-6](#) shows a transcript of a simulation run with comparison errors. In this report, the first column is the pattern number, and the second column is the output port or scan chain output. The third column is present if the port is a scan chain output and contains the number of scan chain shifts that occurred to the point where the error was detected. The last column, shown in parentheses, is the simulated/expected data.

Example 12-6 Simulation With Comparison Errors

```
TEST> run simulation
Begin simulation of 139 internal patterns.
 1 /o_sdo2 23 (exp=0, got=1)
 4 /o_sdo2 23 (exp=1, got=0)
 6 /o_sdo2 23 (exp=1, got=0)
 7 /o_sdo2 23 (exp=1, got=0)
 8 /o_sdo2 23 (exp=0, got=1)
   :   :   :   :
123 /o_sdo2 23 (exp=0, got=1)
124 /o_sdo2 23 (exp=1, got=0)
129 /o_sdo2 23 (exp=1, got=0)
132 /o_sdo2 23 (exp=1, got=0)
Simulation completed: #patterns=139, #fail_pats=41(0),
#failing_meas=41(0), CPU time=4.97
```

Patterns in the Simulation Buffer

During ATPG, TetraMAX processes potential patterns in groups of 32 using an internal buffer called the Simulation Buffer. Immediately after completion of ATPG, you can select any of the last 32 patterns processed and display the resulting logic values on the pins of objects in the GSV window. You can use the Setup dialog box to select pattern data and provide an integer between 0 and 31 for the pattern number.

Alternatively, you can execute these commands:

```
TEST> set pindata pattern NN
TEST> refresh schematic
```

For an example, see “[Displaying Pattern Data](#)” on page 7-26.

Sequential Simulation Data

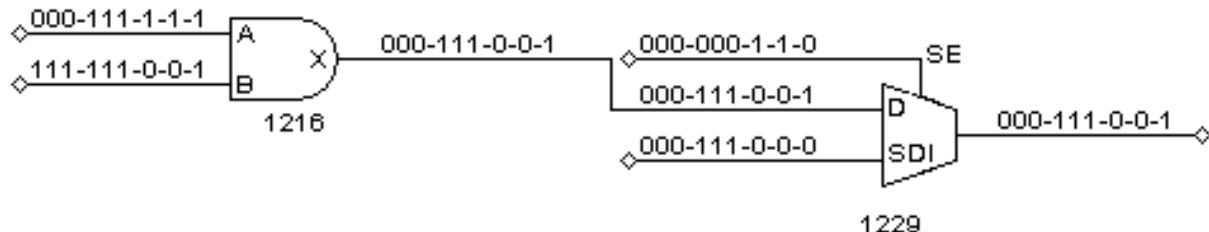
Sequential simulation data is typically from functional patterns. This type of data is stored in the external pattern buffer. When the simulation type in the Run Simulation dialog box is set to Full Sequential, you can select a range of patterns to be stored. After the simulation is completed, you can display selected data from this range of patterns using the pin data type "sequential sim data."

For example, with gates drawn in the schematic window, execution of the following commands generates the display shown in [Figure 12-1](#).

```
TEST> set simulation -data 85 89  
TEST> run simulation -sequential
```

The pin data in the display shows the sequential simulation data values from the five patterns; each pattern has a dash (-) as a separator. Some patterns result in a single simulation event value and other patterns result in three values.

Figure 12-1 Sequential Simulation Data for Five Patterns



Single-Point Failure Simulation

You can simulate any single point of failure for any single pattern by checking the Insert Fault box in the Run Simulation dialog box and specifying the error site and stuck-at value, or by using a command such as the following:

```
TEST> run simulation -maxfails 0 amd2910/ incr/U42/A 1
```

[Example 12-7](#) shows the result of executing this command. TetraMAX reports the signature of the failing data to the transcript as a sequence of pattern numbers and output ports with differences between the expected data and the simulated failure.

Example 12-7 Signature of a Simulated Failure

```
TEST> run simulation -maxfails 0 /amd2910/ incr/U42/A 1
Begin simulation of 139 internal patterns with pin /amd2910/ incr/
U42/A stuck at 1.
  85  /o_sdo2  23  (0/1)
  94  /o_sdo2  23  (0/1)
Simulation completed: #patterns=139, #fail_pats=2(0),
#failing_meas=2(0), CPU time=2.02
```

GSV Display of a Single-Point Failure

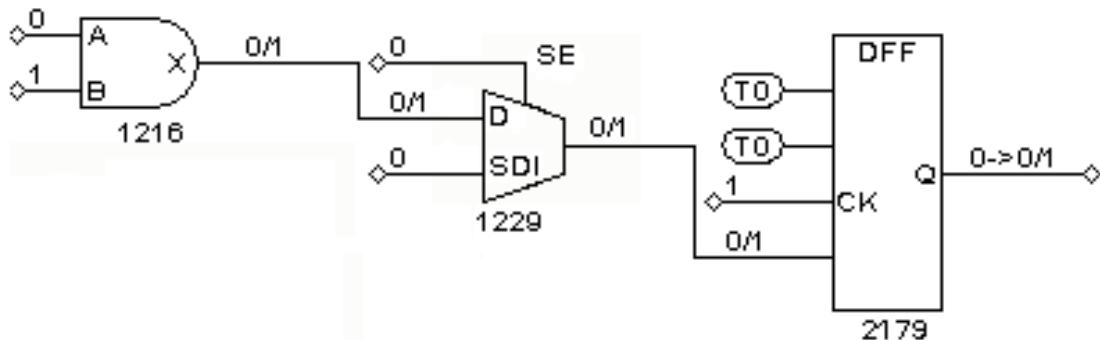
You can display simulation results for a single-point failure in the GSV. To do so, click the SETUP button on the GSV toolbar to display the Setup dialog box.

To view the difference between the good machine and faulty machine simulation for a specific pattern,

1. In the Setup dialog box, under Pin Data, choose Fault Sim Results.
2. Click the Set Parameters button. The Fault Sim Results Parameters dialog box appears.
3. Enter the pin path name or gate ID of the fault site, the stuck-at-0 or stuck-at-1 fault type, and the pattern number that is to be simulated in the presence of the fault.
4. Click OK to close the Fault Sim Results Parameters dialog box.
5. Click OK again to close the Setup dialog box.

The GSV displays the fault simulation results, as shown in [Figure 12-2](#).

Figure 12-2 Fault Simulation Results Displayed Graphically



In [Figure 12-2](#), pin A of gate 1216 is the site of the simulated stuck-at-1 fault. The output pin X shows 0/1, where the 0 is the good machine response and the 1 is the faulty machine response.

You can trace the effect of the faulty machine throughout the design by locating logic values separated by a forward slash (/), representing the good/bad machine response at that pin.

Per-Cycle Pattern Masking

A common practice for test engineers is to replace 0s and 1s with Xs in scan patterns on the tester. The goal, in this case, is to mask specific measures that mismatch on the tester.

The per-cycle pattern masking feature in TetraMAX enables you to use a masks file to identify the measures to mask out. Then, masked patterns can be written out, and, optionally, test coverage can be re-calculated. The steps for this flow are as follows:

1. The original patterns are written out from TetraMAX.
2. A few mismatches occur on the tester.
3. The patterns and mismatches are read into TetraMAX.
4. Mismatches are masked in the pattern.
5. Masked patterns are optionally re-fault simulated.
6. Masked patterns are written out from TetraMAX.
7. All patterns pass on the tester.

This section covers the following topics:

- [Masks File](#)
- [Running the Flow](#)
- [Limitations](#)

Masks File

A masks file contains the measures used to mask in the patterns. It uses the same format as the failure log file used for diagnostics, and can be pattern-based or cycle-based. But note that the pattern-based format with chain name is not supported. See “[Providing Tester Failure Log Files](#)” on page 19-2 for details of the file format.

You can create a masks file as a result of running patterns on a tester. Note that only STIL or WGL patterns files can be used with a cycle-based format masks file. A binary pattern file cannot be masked with the cycle-based format masks file.

Running the Flow

The flow consists of first reading the patterns in the external buffer, along with the masks file. This read step will perform the masking of the patterns. You can then write the updated patterns so you can use them. Finally, you can optionally calculate the new test coverage with the masked cycles.

To read the patterns in the external buffer and read in the masks file, use the following command:

```
set_pattern -external <patterns_file> -resolve_differences <masks_file>
```

For example, the following command reads in the `pat.stil` patterns file and the `mask.txt` masks file, and creates a report that indicates the total number of X measures added in the external patterns:

```
set_pat -ext pat.stil -resol mask.txt
End parsing STIL file pat.stil with 0 errors.
End reading 200 patterns, CPU_time = 33.40 sec, Memory = 5MB
6 X measures were added in the external patterns.
```

Next, use the `write_patterns -external` command to write out the new vectors stored in the external patterns buffer. Then, if you want to calculate the new test coverage, it is recommended that you fault simulate the new patterns with `run_fault_sim`.

The flow is as follows:

```
TEST-T> set_pat -ext pat.stil.gz -resolve mask.txt
TEST-T> write_pat pat.masked.stil.gz -format STIL -compress gzip -external
TEST-T> run_fault_sim
```

An alternate method for fault simulating the patterns and saving them so they can run on the tester is to use first `run_atpg -resolve` and then `write_patterns`. In this case, the difference with previous method is that the `run_atpg - resolve` command fault grades the external patterns with the added masks, and, patterns that don't contribute to the test coverage are removed.

The advantage of using the alternate method is that if a large number of failures are used during per-cycle pattern masking, it is likely that many patterns run on the tester will be useless and thus removing them will increase the speed of the test time. The drawback is that new failures could appear because of the patterns suppression. This is why it is recommended that you perform a check with the `run_sim` command after `run_atpg - resolve`. If new failures occur, it will be necessary to mask the patterns another time using `set_pattern - resolve`.

The alternate flow is as follows:

```
TEST-T> set_pat -ext pat.stil.gz -resol mask.txt
TEST-T> add_faults -all
TEST-T> run_atpg -resolve_differences
TEST-T> run_sim
TEST-T> write_pat pat.masked.stil.gz -format STIL -compress gzip -external
```

Limitations

The limitations to this flow are as follows:

- Reading in mismatches from a Verilog testbench simulation and/or simulating masked patterns with a Verilog testbench are not supported.
- The pattern-based format with chain name is not supported.
- The binary pattern file cannot be masked with a cycle-based format masks file.
- Failures in cycle-based format with cycle offset could not be automatically read.
- For patterns with multiple load-unloads with measures on the scanout in each unload, only the failures for the first unload could be masked.

13

PatternMap

The TetraMAX PatternMap feature allows you to reuse a set of functional test vectors (defined at a module level) for testing the module in the context of the top-level design.

A typical application for this feature is memory testing. Normally, testing embedded memories involves the use of memory BIST (built-in self test). However, if the memory is small enough, inserting BIST logic in your design can increase area. To avoid this impact, you can test the memory with a few functional vectors. Note, however, that you cannot directly control the ports of the embedded module using this approach. TetraMAX PatternMap provides the ability to translate functional test vectors from a memory module to its top-level design.

This chapter contains the following sections:

- [Overview of PatternMap](#)
- [PatternMap Requirements](#)
- [About Pattern Mapping](#)
- [Preparing the Module](#)
- [Running PatternMap](#)
- [Tips for Using PatternMap](#)

Overview of PatternMap

“Module pattern mapping” is a technique used for testing an embedded submodule when functional test patterns are available for testing that module as an independent unit. The TetraMAX implementation of this feature is called PatternMap.

A system-level chip might use a predesigned submodule as part of its overall design. For example, a controller device might use a CPU core or a memory module obtained from an outside vendor, combined with your own proprietary peripheral and I/O logic.

When a predesigned module is embedded within a chip, it presents the problem of how to test the module. You might have a set of functional test vectors for the module operating as an independent unit, but no way to apply those test vectors because the module inputs and outputs are not directly accessible from the chip inputs and outputs.

PatternMap helps you access embedded modules by translating functional vectors, which are defined at the module level and refer to individual module ports, into a set of vectors that refer to top-level primary inputs and scan cells. The resulting new set of patterns, when applied to the chip inputs, results in the application of the functional test patterns to the module, internal to the chip. The output response of the module can be read, with proper conditioning, from the internal scan chains or at the chip outputs.

You invoke PatternMap by using the `run mapping` command. The command syntax is described in the section “[Run Mapping Command](#)” on page 13-15; or see the online help for this command.

PatternMap Requirements

To use PatternMap, the embedded module and its functional test patterns must meet certain requirements.

Embedded Module Requirements

The embedded module must be defined to be a library cell or black box. To establish the cell type, use the `set build` command, and use either the `-design_box` option if there is a simulation view of the model, or the `-black_box` option if it is a true black box module. If it is a true black box module, you need at least a “dummy” module description that defines the port names and port directions.

PatternMap can handle one or more instances of the same module. However, it cannot do pattern mapping with two or more *different* embedded modules in the design.

PatternMap does not look inside the module. It generates only the specified sequence of logic values at the module inputs and reads the values at the module outputs. In other words, pattern mapping is performed only at the boundaries of the embedded instance.

Functional Test Pattern Requirements

The pattern mapping utility takes the functional test patterns for the embedded module and maps those patterns into a new set of test patterns for the whole chip.

The functional test pattern file for the module must meet the following requirements:

- It must be in Extended VCD (VCDE), STIL, or WGL.
- It must contain an ordered pin list consistent with the module pin list.
- It can only contain primary input signals and measurable primary output signals (no module-internal signals).
- It cannot contain any load or unload (scan) operations.
- It can only contain vectors in a simple format acceptable to TetraMAX. See [Example 13-1](#), [Example 13-2](#), and [Example 13-3](#) for suitable examples of module test patterns in VCDE, STIL, and WGL format, respectively.

Example 13-1 Extended VCD 256X4 Synchronous RAM Example

```
$date
    Thu Jul 11 17:57:56 2002
$end
$version
    Chronologic Simulation VCS Release 6.1R18
    $dumpports(ram256x4_test.dut, "my.vcde")
$end
$timescale
    10ps
$end
$comment Csum: 1 91ef8738459d8adb $end
$scope module ram256x4_test.dut $end
$var port      1 <0          wclk $end
$var port      1 <1          rclk $end
$var port      [7:0] <2        wa $end
$var port      [3:0] <3        din $end
$var port      [7:0] <4        ra $end
$var port      [3:0] <5        dout $end
$upscope $end
$enddefinitions $end
#0
$dumpports
pD 6 0 <0
pD 6 0 <1
pDDDDDDDD 66666666 00000000 <2
pUUUU 0000 6666 <3
pUUDUDUDU 00606060 66060606 <4
```

```

pXXXX 6666 6666 <5
$end
#25000
pU 0 6 <0
#28000
pD 6 0 <0
#30000
pDDDDUUUDU 66660060 00006606 <2
pUUDU 0060 6606 <3
pDDDDDDDD 66666666 00000000 <4
#55000
pU 0 6 <1
pHHHH 0000 6666 <5
#58000
pD 6 0 <1
#60000
pDDDUDUUD 66606006 00060660 <2
pDDUD 6606 0060 <3
#80000
pDDDDDDDD 66666666 00000000 <2
pDDDD 6666 0000 <3
pDDDDUUUU 66660000 00006666 <4
#105000
pU 0 6 <0
#108000
pD 6 0 <0
#110000
pDUUUUDDU 60000660 06666006 <2
pUDUU 0600 6066 <3
pDDDDDDDD 66666666 00000000 <4
#135000
pU 0 6 <1
pLLLL 6666 0000 <5
#138000
pD 6 0 <1
#140000
pUDDDDDDDD 06606666 60060000 <2
pUUUU 0000 6666 <3
pDDUDUDDU 66060660 00606006 <4
$vcdclose #160000 $end

```

Example 13-2 STIL Module Test Pattern Example

```

STIL;
SignalGroups {
    _clk= 'tst_clk"      ;
    _pi = 'tst_clk"+ "clock"+ "tst_mode"+ "writerequest" +
"writeaddress[1]" +
        + "writeaddress[0]" + "readaddress[1]" + "readaddress[0]" +
        + "writedata[3]" + "writedata[2]" + "writedata[1]" +
"writedata[0]"      ;
    _po = ' + "readdata[3]" + "readdata[2]" + "readdata[1]" +
"readdata[0]"      ;
}
Timing my_timing {
    WaveformTable TS1 {
        Period '1000ns';
        Waveforms {

```

```

"clock"           { 01NZ { '0ns      'D/U/N/Z; } }
"tst_mode"       { 01NZ { '0ns      'D/U/N/Z; } }
"tst_clk"        { P   { '0ns' D; '500ns' U; '700ns' D; }    }
"tst_clk"        { 01NZ { '0ns' D/U/N/Z; } }
"writerequest"  { 01NZ { '0ns      'D/U/N/Z; } }
"writeaddress[1]" { 01NZ { '0ns      'D/U/N/Z; } }
"writeaddress[0]" { 01NZ { '0ns      'D/U/N/Z; } }
"readaddress[1]" { 01NZ { '0ns      'D/U/N/Z; } }
"readaddress[0]" { 01NZ { '0ns      'D/U/N/Z; } }
"writedata[3]"   { 01NZ { '0ns      'D/U/N/Z; } }
"writedata[2]"   { 01NZ { '0ns      'D/U/N/Z; } }
"writedata[1]"   { 01NZ { '0ns      'D/U/N/Z; } }
"writedata[0]"   { 01NZ { '0ns      'D/U/N/Z; } }
"readdata[3]"    { LHTX { '900ns     'L/H/T/X; } }
"readdata[2]"    { LHTX { '900ns     'L/H/T/X; } }
"readdata[1]"    { LHTX { '900ns     'L/H/T/X; } }
"readdata[0]"    { LHTX { '900ns     'L/H/T/X; } }

}
}

PatternBurst burst1 { block1; }

PatternExec { Timing my_timing ; PatternBurst burst1; }

Procedures {
    sequential_capture {
        W TS1;
        V { _pi = \r12 # ; _po = \r4 # ; tst_clk = P; }
    }
}

Pattern block1 {
    W TS1 ;
    C { _pi = \r12 N ; _po = \r4 X ; } // init all values
"pattern 0": Call sequential_capture {_pi = 001100NN0000 ; _po = XXXX ;}
"pattern 1": Call sequential_capture {_pi = 001101NN00000 ; _po = XXXX ;}
"pattern 2": V { _pi = 001NNN00NNNN ; _po = LLLL ;}
"pattern 3": Call sequential_capture {_pi = 001100NN1111 ; _po = XXXX ;}
"pattern 18": V { _pi = 001NNN01NNNN ; _po = LLLL ;}
}

```

Example 13-3 WGL Module Test Pattern Example

```
waveform module
signal
    "clk1" : input;
    "I0" : input;
    "I1" : input;
    "I2" : input;
    "Q0" : output;
    "Q1" : output;
    "Q2" : output;
    "Q3" : output;
    "reset" : input;
    "se" : input;
end
timeplate "_default_WFT_" period 100ns
    "clk1" := input [0ps:D, 50ns:S, 80ns:D];
    "I0" := input [0ps:S];
    "I1" := input [0ps:S];
    "I2" := input [0ps:S];
    "Q0" := output [0ps:X, 40ns:Q'edge];
    "Q1" := output [0ps:X, 40ns:Q'edge];
    "Q2" := output [0ps:X, 40ns:Q'edge];
    "Q3" := output [0ps:X, 40ns:Q'edge];
    "reset" := input [0ps:S];
    "se" := input [0ps:S];
end
pattern group_ALL ("clk1", "I0", "I1", "I2", "reset", "se", "Q0",
"Q1", "Q2", "Q3")
{ pattern 0 }
{ capture_clk1 }
vector("_default_WFT_") := [ 0 1 1 1 1 0 X X X X ];
vector("_default_WFT_") := [ 1 1 1 1 1 0 X X X X ];
{ pattern 1 }
{ capture_clk1 }
vector("_default_WFT_") := [ 0 0 0 0 0 0 X X X X ];
vector("_default_WFT_") := [ 1 0 0 0 0 0 X X X X ];
{ pattern 2 }
vector("_default_WFT_") := [ 0 1 1 1 0 0 1 0 1 1 ];
end
end
```

About Pattern Mapping

During pattern mapping, TetraMAX ensures that the vectors you are trying to map do not violate any constraints you have on the design, such as PI equivalencies. TetraMAX also protects the data types it manipulates. For example, it won't allow a clock line to be mapped with a data line. Any attempt at this kind of mapping results in an error message and halts the pattern-mapping process.

TetraMAX lets you choose between independent and dependent pattern mapping to map your vectors at the top-level module.

Independent Pattern Mapping

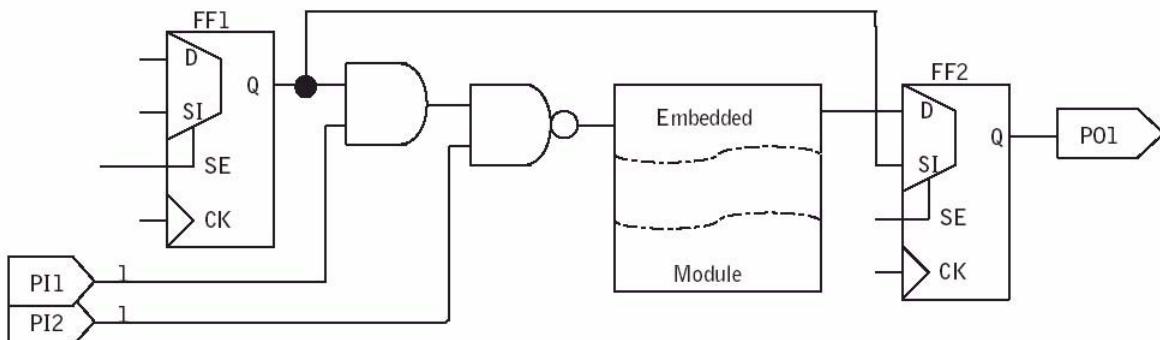
In independent pattern mapping, a unique control point related to a module input or an observe point related to a module output is identified for every port of the embedded component under test. By mapping the value you want to control or observe on a given embedded module port, you enable TetraMAX to observe or control the same element, or correspondence point, in the design.

This form of mapping is considered independent because TetraMAX can observe or control any value on the module port that has been mapped through a unique correspondence point. For example, a valid correspondence point can be a primary input of the top-level design or a scan cell.

A correspondence point might not always connect directly to the module line it controls, in which case TetraMAX may need to set other lines in the design to specific values, called conditioning vectors, to propagate the desired value from the correspondence point to the embedded block.

In [Figure 13-1](#), the two flip-flops FF1 and FF2 are the correspondence points for testing the embedded module. To propagate the test vector from the correspondence point FF1 to the port of the embedded module, some specific values need to be set on PI1 and PI2. A conditioning vector is created by applying a logical “1” on PI1 and a logical “1” on PI2.

Figure 13-1 Example



After identifying the correspondence points, the independent algorithm brings the values corresponding to your functional vectors to your embedded module through those points. Each correspondence point is able to drive all the necessary values for testing your module. Thus, if you want to change a vector, you can edit it directly in the pattern file without having to run TetraMAX PatternMap again because you know the correspondence points.

The independent mapping technique is most suitable when you plan to change the functional vectors. Because each module port has a unique correspondence point, it's clear which scan cell or primary input at the top level controls which module line. As a result, if you need to

modify some functional vectors, you can simply edit the final top-level vector file. However, if the module you plan to test is deeply embedded in the design, TetraMAX may not be able to find a unique correspondence point for each module point. In this case, you should use the dependent mapping technique as described in [“Dependent Pattern Mapping” on page 13-8](#).

For optimal results, insert a scan wrapper around your embedded module. This will provide an excellent correspondence-point candidate. Be aware that scan wrapping logic impacts the die size and propagation time.

Dependent Pattern Mapping

In dependent mapping, ATPG propagation and justification techniques are used to bring vectors to the embedded module. The manufacturing test vector generated at the top-level design depends on the ports. TetraMAX can use different propagation paths to control a given module input to either a logical “1” or a logical “0.” Usually, an ATPG engine is designed to select the path that requires the least effort.

Sometimes TetraMAX cannot find a correspondence point for each port of your embedded module, causing independent pattern mapping to fail. This is likely to happen when your module is deeply embedded in the hierarchy. If you cannot find a correspondence point for a line (all logical values driven or observed through the same point), you might find a way to control a logic 0 through one path and a logic 1 through another path. In this case, the mapping is still possible, but not with correspondence points.

To map without correspondence points, use the dependent mapping algorithm. It behaves just like a standard ATPG algorithm, putting the required values on your module ports and trying to justify them until the values reach either a primary input, a primary output, or a scan cell. The set of vectors generated for the top-level module is dependent on the vectors defined for the embedded module. If you use the dependent mapping algorithm and you want to change a vector for the embedded module level, you must rerun TetraMAX PatternMap.

Dependent mapping is more powerful than independent mapping. However, since there are no correspondence points, you can't modify your functional vectors after the mapping. Therefore, to make any modifications, you must run PatternMap again with a new module vector file.

Note:

You must use either dependent or independent mapping; they cannot be mixed.

Preparing the Module

To perform pattern mapping of a given module, TetraMAX needs to know each of its port directions. If you have a valid TetraMAX design description, simply read in your Verilog file as you would for any other module using the `read netlist` command.

If you do not have a module definition, TetraMAX treats the module as an empty box. As a result, use the `set build -black_box` command to instruct TetraMAX to treat the module as a black box.

A *black box* is a module or block whose function and internal contents are unknown. Only the port names and perhaps port directions are known. When a module is treated as a black box, its input ports are unattached, and its output and bidirectional ports are attached to TIEX primitives.

When you declare a module that does not have a module definition as a black box, TetraMAX must guess the pin directions based on connectivity in the design. This is not always possible with 100% accuracy. To eliminate the possibility of an error, consider defining a NULL MODULE. This null module would list the module header, the pins, and their defined directions, but would have an empty gate list for the internal definition.

An *empty box* is the same as a black box, except that its outputs are floating (connected to TIEZ primitives) rather than tied to an X value. This allows the outputs of multiple empty boxes to be connected together without triggering a contention condition. This should be done only if the empty box outputs are actually in the Z-state during test.

When you declare a module that does not have a module definition as an empty box, TetraMAX must guess the pin directions based on connectivity in the design. This is not always possible with 100% accuracy. To eliminate the possibility of an error, consider defining a NULL MODULE. This null module would list the module header, the pins and their defined directions, but would have an empty gate list for the internal definition

Running PatternMap

The following procedure explains how to translate test vectors generated at a module level into a set of vectors generated at the top level.

Note: The examples given in these steps contain illustrative file names. Change them to match your file names.

1. Write the functional patterns for the embedded module, or reuse existing functional patterns.

2. Ensure that all functional vectors are correct. If you don't have a valid TetraMAX description of your module, you should first verify the accuracy of your functional vectors by running these in a Verilog simulation environment. If you do have a valid TetraMAX design description, then simulate the vectors using the `run simulation` command.

To correct vectors, use the `set pindata` command along with the schematic viewer to analyze the problem.

3. Build the complete design in TetraMAX.
4. If you do not have a Verilog model for your embedded module, mark the module as a black box before building the top-level simulation model.
5. Perform pattern mapping using either the dependent or independent algorithm. For example:

Dependent --

```
run mapping -module module_name -dependent_pattern_mapping  
-pattern_file file_name
```

Independent --

```
run mapping -module module_name -pattern_file file_name
```

6. Use the default independent algorithm if you want to be able to modify vectors without rerunning TetraMAX. Specify the dependent algorithm if your module is deeply embedded in the design. In addition, in the case of unsuccessful mapping, you can turn on dependent mapping.
7. The `run mapping` command is available when you are in the TEST mode only. You can perform mapping on a given module instance, or on all instances of a module. Functional patterns are provided to TetraMAX through the specified pattern file.
8. Run simulation to check your vectors. Since the patterns are functional vectors, use the `-sequential` option of the `run simulation` command to maintain state between vectors.

Note:

If you did not provide a simulation model for your embedded module, TetraMAX issues error messages in the patterns and you will propagate Xs on the module's outputs.

9. Save the ATPG vectors.

```
write patterns my_pat.stil -format stil -replace
```

10. Fault grade the set of ATPG vectors generated by TetraMAX PatternMap.

A fault grade is a simulation. Starting from a fault list and a set of patterns, it shows you what faults are deleted by your patterns. You won't need to target those faults at a later ATPG step.

```
add faults -all  
run fault_sim -store
```

11. Run ATPG to detect the remaining faults in the top-level design.

```
run atpg -auto_compression
```

[Figure 13-2](#) and [Figure 13-3](#) illustrate this process with and without fault simulation. [Figure 13-2](#) follows steps 1 through 6.

Figure 13-2 Generate Vectors From Functional Vectors

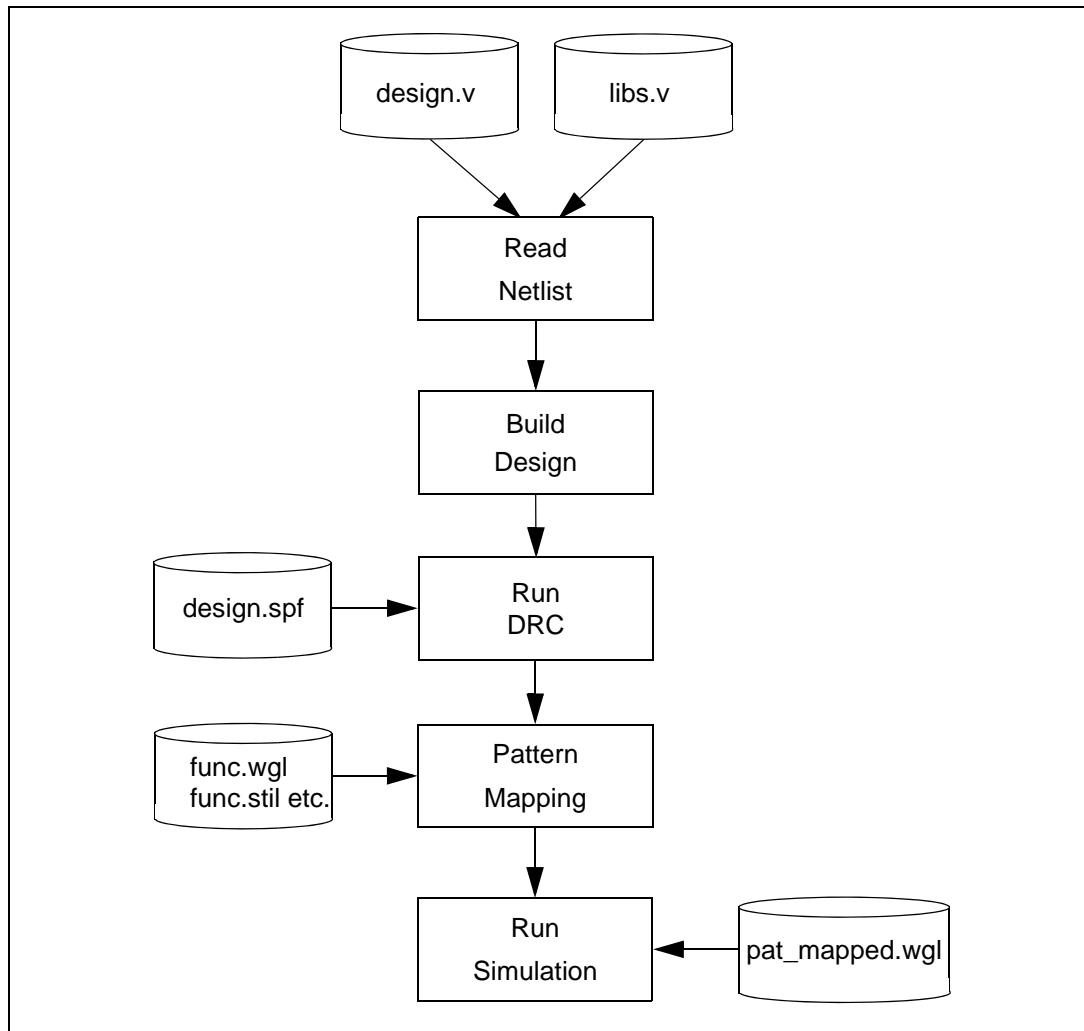
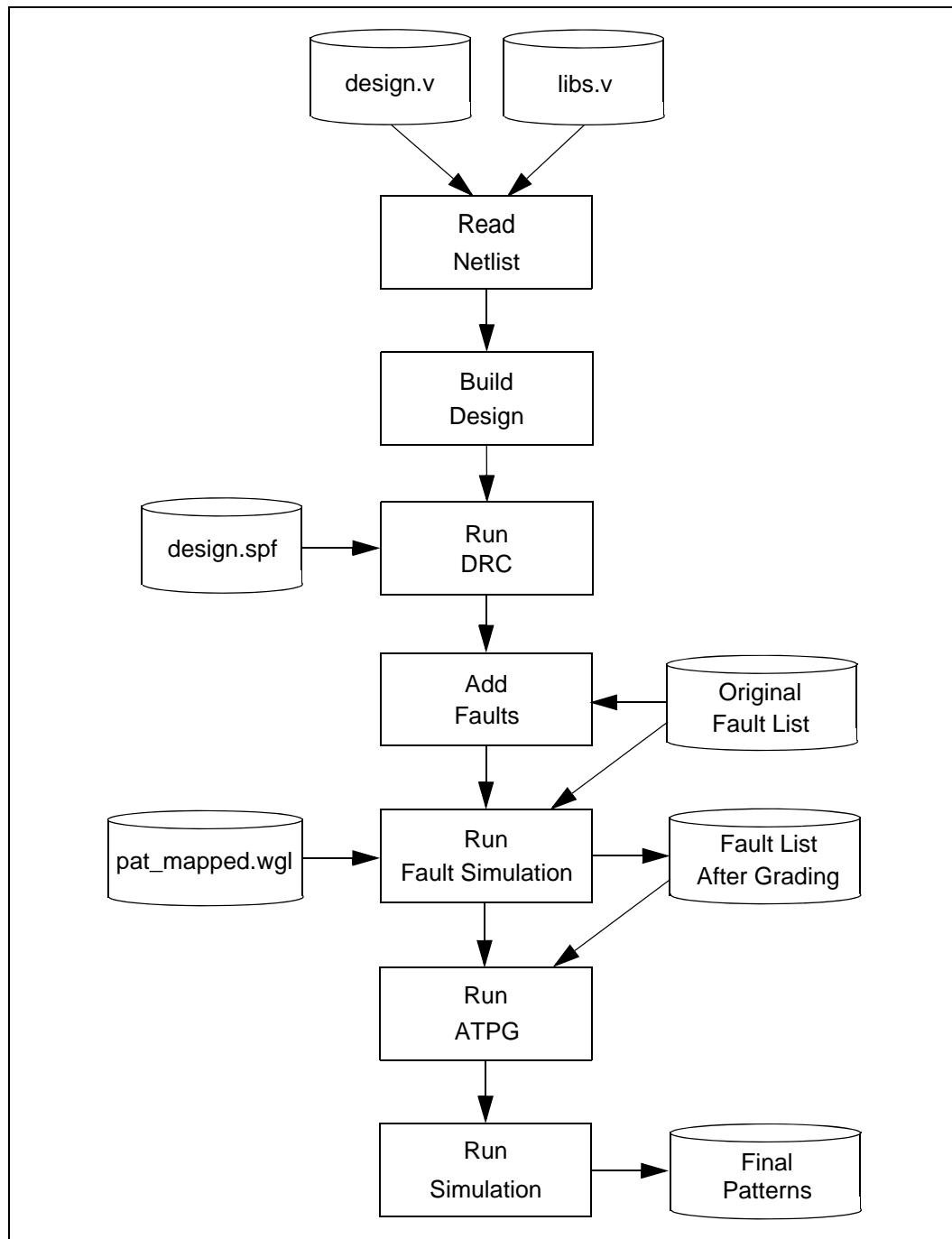


Figure 13-3 illustrates steps 7 and 8. This part of the process consists of running a fault simulation starting from the previously generated vectors (obtained with TetraMAX PatternMap), and running the ATPG flow (`run atpg`) to detect any remaining faults in the design.

Figure 13-3 Running the Fault Simulation

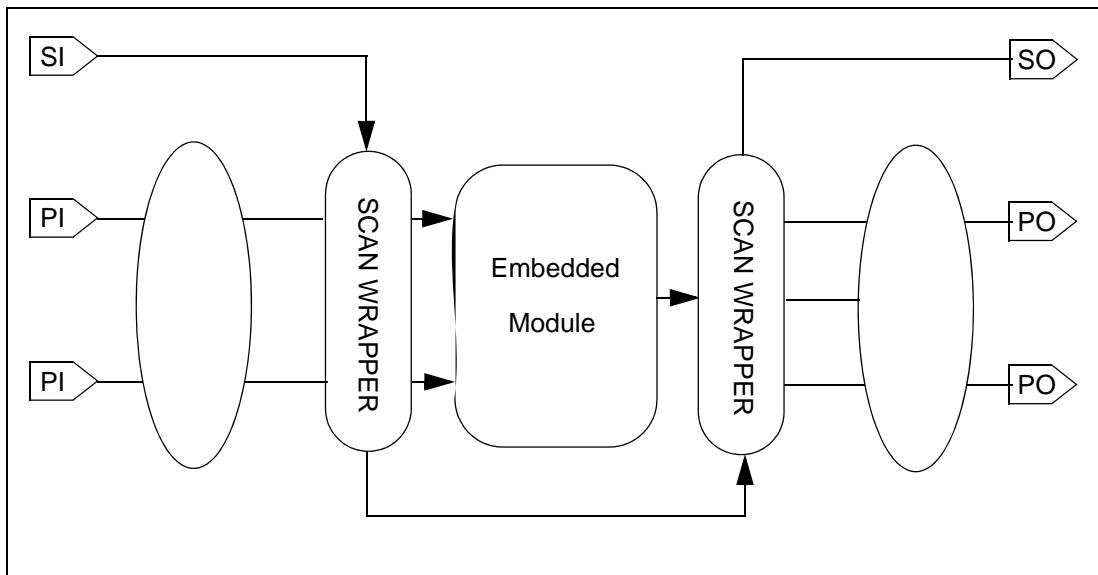


Working Example

This example demonstrates how to test an embedded module in a design using a predefined set of functional patterns. It uses the TetraMAX PatternMap independent algorithm to allow you to edit the vectors after mapping.

Using the independent algorithm will make your task easier since you must only edit a text file. Because a correspondence point can be either a primary input, a primary output, or a scan cell, your TetraMAX PatternMap success possibilities are improved if you add a scan wrapper around the embedded module (see [Figure 13-4](#)). The wrapper provides better access to the module ports.

Figure 13-4 Design Overview



The Embedded Module

Here is a simple example of an AND gate. The Verilog model for the embedded module is as follows:

```
 `celldefine
  module embedded (A, B, Z);
    input A, B;
    output Z;
    and U1 (Z, A, B);
  endmodule
 `endcelldefine
```

To map an embedded module, write a set of functional patterns:

```

A=1 ; B=1 ? Z=1
A=0 ; B=0 ? Z=0
A=1 ; B=0 ? Z=0

```

The following script is the TetraMAX command file for achieving the good machine simulation of the functional vectors on the submodule within TetraMAX.

```

Read netlist embedded.v -delete
Run build embedded
Set drc -nofile
Run drc
Set pattern external func_pat.wgl
Run simulation

```

Running TetraMAX PatternMap

Next, build the top-level design and to run TetraMAX PatternMap using the functional set of vectors. The following script goes through the steps of reading and building the design:

```

Read netlist embedded.v -library -delete
Read netlist design.v
Run build design

Set drc design.spf
Run drc

Run mapping design/emb_module -pattern \
    pat_mod.wlg -verbose -delete
Run simulation
Write patterns after_map.wgl -format wgl -replace

```

Before running PatternMap, you need to run the design rule checker with a STIL protocol file. This protocol file defines the scan wrapper as a scan chain in the design and sets up the various signals associated with it (clocks, scan enable, and so on).

Run Mapping Command. The syntax of this command is:

```
run mapping design/emb_module file_name -verbose -delete
```

Argument	Definition
<i>design/emb_module</i>	The instance name of the module involved in pattern mapping. If your design contains several identical modules and you want to map the same functional vectors for each of them, you can use the <i>-module</i> option instead of specifying an instance name.
<i>file_name</i>	The name of the file containing the functional patterns. The functional patterns can only be described in Extended VCD (VCDE), STIL, or WGL.

Argument	Definition
-verbose	Forces TetraMAX to display the list of the correspondence points for each port of the embedded module. This list is useful if you plan to modify a few vectors by hand.
-delete	Deletes all previous vectors in the internal pattern buffer before running TetraMAX PatternMap.

Analyzing the Report

You can analyze the report and data TetraMAX generates when running TetraMAX PatternMap, for example:

```
//
// RUN MAPPING USING
// INDEPENDENT ALGORITHM
//
run mapping design/emb_module -pattern mod_pat.wgl -delete -
verbose
Begin pattern mapping for instance em_module using patterns in
file mod_pat.wgl.
End parsing WGL file mod_pat.wgl with 0 errors;
End reading 6 patterns, CPU_time = 0.00 sec, Memory = 0MB
Control pattern generation successful: #forces=0(load).
Observe pattern generation successful: #forces=1(unload),
#clocks=1, obs_proc=none.
-----
-----
List of correspondence points for pins of instance embedded
-----
type      pin_name          gate_id  correspondence_point
-----
input      A                  7        9 (DFF)
input      B                  7        10 (DFF)
output     Z                  7        11 (DFF)
-----
Output conditioning pattern: #forces=1, clock=CK, obs_proc=none
  1(PI)=0
-----
Warning:Force_all_PIs was preceded by an ignored force_all_PIs
in pattern 1. (M246)
Warning:Force_all_PIs was preceded by an ignored force_all_PIs
in pattern 3. (M246)
Warning: Force_all_PIs was preceded by an ignored force_all_PIs
in pattern 5. (M246)
```

Report Description

TetraMAX first analyzes the WGL file containing the patterns. It reads six patterns as specified in the pattern file. TetraMAX then tries to find a way to control and observe all the ports of the module. TetraMAX then produces the list of all the correspondence points in the design. The left side of the report lists the ports of the embedded module; the right side of the report lists the correspondence points. All module ports are mapped on scan cells.

The “verbose” mode displays the correspondence point for every module signal (not displayed in the preceding example).

The numbers given in the output report are the gate IDs in the TetraMAX simulation model. For example, 7 is the gate ID for the example module (an AND gate); 9, 10, and 11 are the gates IDs of scan cells belonging to the wrapper.

Note:

The “output conditioning patterns” portion of the report shows that TetraMAX issues a clock pulse to observe the output ports of the embedded module. This is because the design has scan wrappers.

Here is how TetraMAX translates the first vector:

```
Pattern 0 (full_sequential)
Time 0: load c0 = X11
Time 1: force_all_PIs = X0X0
Time 2: pulse clocks /CK (3)
Time 3: unload c0 = 1XX
```

Time 0: load c0 = X11

TetraMAX loads the scan wrapper with the values required as input for the embedded module (in this case, A=1 and B=1).

Time 1: force_all_PIs = X0X0

TetraMAX forces the primary inputs of the design. The first 0 stands for scan enable line; it turns to 0 for nonscan mode, the second 0 stands for CK; it sets the clock to its off-state value.

Time 2: pulse clocks /CK (3)

Because the module is a combinational AND gate, the module’s output port should be set to a logical 1. But since this value needs to be checked through the scan wrapper, TetraMAX needs to issue a clock pulse on CK (the clock line that drives the scan wrapper cells) to load emb_module/Z in its correspondence point (DFF 11).

Time 3: unload c0 = 1XX

TetraMAX unloads the scan chain. The logical 1 corresponds to the output of the module. An AND gate with both inputs is set to 1, and the values in the two other cells are not important.

Once the patterns have been translated successfully, you need to verify them using the `run simulation` command and write the resulting pattern file for the whole design. You should use your timing annotated file to perform this final verification step.

Enhancing the Quality of Results

The overall number of vectors required to test a device is a critical parameter in the cost of manufacturing test. TetraMAX has an option to enable some pattern merging during the mapping process. TetraMAX can merge the pattern of several instances of the same module as well as merge patterns of different modules. In the event that TetraMAX cannot merge some vectors, the vectors will be appended to the existing set of vectors.

In some cases, you may not want to use all module pins for functional testing. However, since by default TetraMAX accesses all the ports during pattern mapping, you could end up with no vector at all due to a controllability or observability problem on unused port. To circumvent this problem, use the `-NOMap_pin pin_name` option. TetraMAX won't attempt to map this line and you might be able to complete the pattern translation, thus getting some coverage on your module.

Modifying the PatternMap Results

If you use the independent pattern mapping algorithm, you can modify the test vectors without having to run TetraMAX PatternMap again.

The following example shows how to handle this:

```
...
{ scan_test }
  c0U0 := c0G(XXX);
  c0L0 := c0G(X11);
  c0U1 := c0G(1XX);
end
...
pattern group_ALL ("D", "SE", "SI", "CK", "Z")
{ pattern 0
  { load_unload }
  vector("_default_WFT_") := [ X 1 X 0 X ];
  scan("_default_WFT_") := [ X 1 - 1 - ],
  output [c0:c0U0], input [c0:c0L0];
  { vector }
  vector("_default_WFT_") := [ X 0 X 0 X ];
  { vector }
  vector("_default_WFT_") := [ X 0 X 1 X ];
}
end
```

The preceding example is the first pattern to be found in the resulting vector file. Notice the first unload of the scan chain is XXX and the first load is X11. This means that A=1 and B=1 are applied on the embedded module. If you prefer testing for A=0, B=1, then the only items you need to change are:

- The values loaded in the scan chain
- The values unloaded for the next vector (c0U1)

The following file is generated, with the changes shown in bold:

```
...
{ scan_test }
  c0U0 := c0G(XXX);
  c0L0 := c0G(X01);
  c0U1 := c0G(0XX);
end
...
pattern group_ALL ("D", "SE", "SI", "CK", "Z")
{ pattern 0
  { load_unload }
  vector("_default_WFT_") := [ X 1 X 0 X ];
  scan("_default_WFT_") := [ X 1 - 1 - ],
  output [c0:c0U0], input [c0:c0L0];
  { vector }
  vector("_default_WFT_") := [ X 0 X 0 X ];
  { vector }
  vector("_default_WFT_") := [ X 0 X 1 X ];
}
end
```

You can now load your pattern file and run a machine simulation on it. You do not have to run TetraMAX PatternMap again.

Fault Grading Your Vectors from TetraMAX PatternMap

The vectors generated with TetraMAX PatternMap might detect faults other than those in the embedded module. Thus, you should run TetraMAX fault simulator with your new set of vectors on the whole fault list as shown in the following example. If the fault simulator detects other faults, you can choose to drop them from the fault list before running ATPG.

```
set patterns -delete
set patterns external after_mapping.wgl
run simulation

add faults -all
run fault_sim -store
write fault_list fault_map.gz -compress gzip -all \
-uncollapsed
```

This example shows the results after fault simulation. The circuit has 52 faults, 34 are detected by implication and 18 remain. Fault simulation is performed on 18 faults and leads to a fault coverage of 75.86 percent.

```
Simulation performed for 18 faults on circuit size of 13 gates.  
-----  
#patterns      #faults      test      process  
stored        detect/active  coverage   CPU time  
-----  
Begin PROOFS fault simulation of 18 faults on 3 external patterns  
-----  
#patterns      #faults      cum. #faults      fault      test      process  
simulated     detect/total    detect/active  coverage   coverage  
CPU time  
-----  
      3       10       18       10       8       75.86      84.62      0.01  
Fault simulation completed: faults simulated = 18,  
      fault coverage = 75.86%, test coverage = 84.62%, CPU time  
= 0.01
```

Tips for Using PatternMap

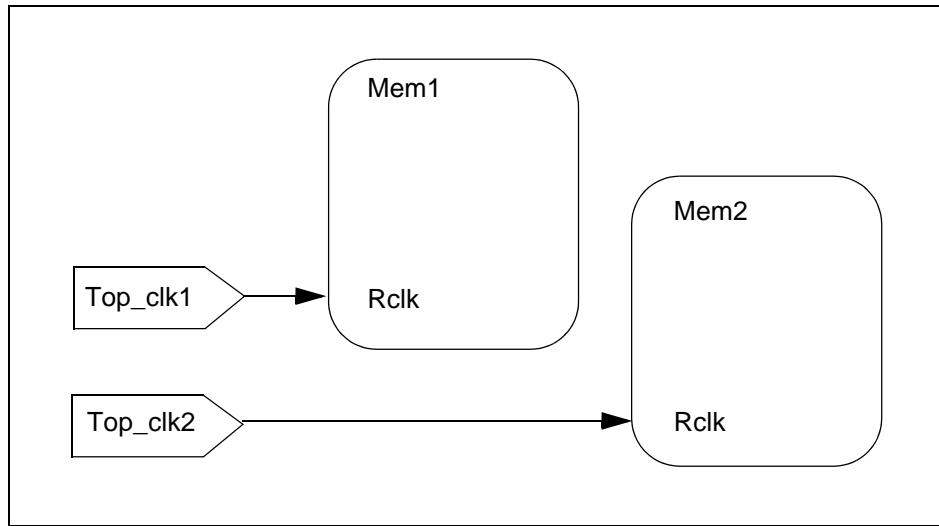
Following are a few tips for using PatternMap.

Applying a Set of Vectors to Several Instances

- If you have several instances of a memory module in your design, you might want to be able to apply a set of functional vectors on all those memories. If your modules are clocked with different top-level clock lines, then you will not be able to merge the vectors between the various modules because of the way the clocks are handled. TetraMAX does not merge patterns whose clocks are not identical. To make the clocks identical, you must define them as equivalent at the top level. Thus, when TetraMAX pulses one clock, it also pulses all the clocks that are equivalent to this one, allowing the merging of patterns.

For example, the design shown in [Figure 13-5](#) has two memory modules being clocked by different clock lines.

Figure 13-5 Two Instances of a Memory Module



To merge the patterns generated for Mem1 and Mem2, you need to define Top_clk1 equivalent to Top_clk2. TetraMAX provides two ways for doing this:

- Use the add pi equivalence command
- Use the E statement in your STIL protocol file (be sure to include this statement in all capture procedures)

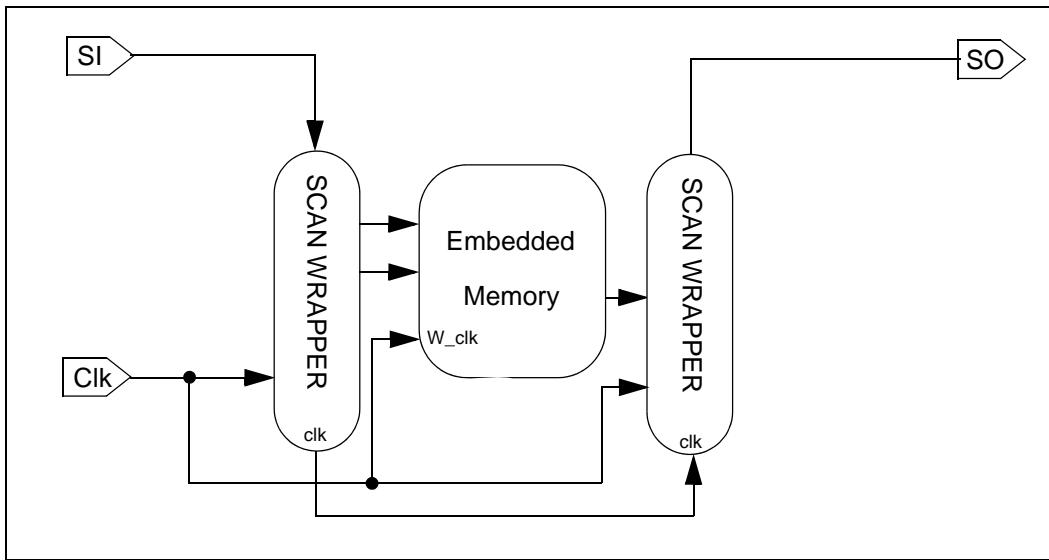
```
add pi equivalence Top_clk1 Top_clk2  
run mapping Top/Mem1 -pattern pat.wgl -delete  
run mapping Top/Mem2 -pattern pat.wgl -merge
```

TetraMAX PatternMap first creates a set of vectors containing the test vectors for Mem1. Then it merges the patterns dedicated to Mem2 with the first set of vectors. This is possible because the memory clock lines are considered identical. The clock pulse for reading the memories is distributed on both clock lines.

Using Clock As Scan Chain and Memory Write Clocks

If your embedded module is a memory, your correspondence points are scan cells, and the same clock is used for driving the scan chain and the memory write port (or any set/reset port). This configuration can cause some stability problems with the memory content.

Figure 13-6 Scan Chain and Memory Clock



If your design looks like the one shown in [Figure 13-6](#), you need to ensure that the contents of the memory will not be affected by test procedures. Typically in this case, you would need to put the W_clk port of the embedded module to its off state during loading and unloading. Usually, this is done by making W_clk a function of Ck and a scan enable line of the design.

If this problem exists in your design, TetraMAX issues an S30 rule violation. If you ignore this rule and run pattern mapping, you cannot ensure good values on the outputs of your module; you might have overwritten the memory content with bad values during the scan load and thus have mismatches with respect to your vectors.

In ATPG mode (no pattern mapping), TetraMAX will not be able to use the values on output ports of a memory that has an S30 violation, resulting in a loss of test coverage.

14

Transition Delay Fault ATPG

The transition delay fault model is used to generate test patterns to detect single-node slow-to-rise and slow-to-fall faults. For this model, TetraMAX launches a logical transition upon completion of a scan load operation, and a pulse on capture clock procedure is used to observe the transition results.

This chapter contains the following sections:

- [Transition Delay Fault Model](#)
- [Specifying Transition Delay Faults](#)
- [Pattern Generation for Transition Delay Faults](#)
- [Pattern Formatting for Transition Delay Faults](#)
- [Specifying Setup Timing Exceptions From an SDC File](#)
- [Small Delay Defect Testing](#)

You need a Test-Fault-Max license to use the transition delay fault ATPG feature. This license is also checked out if you read an image that was saved with the fault model set to transition.

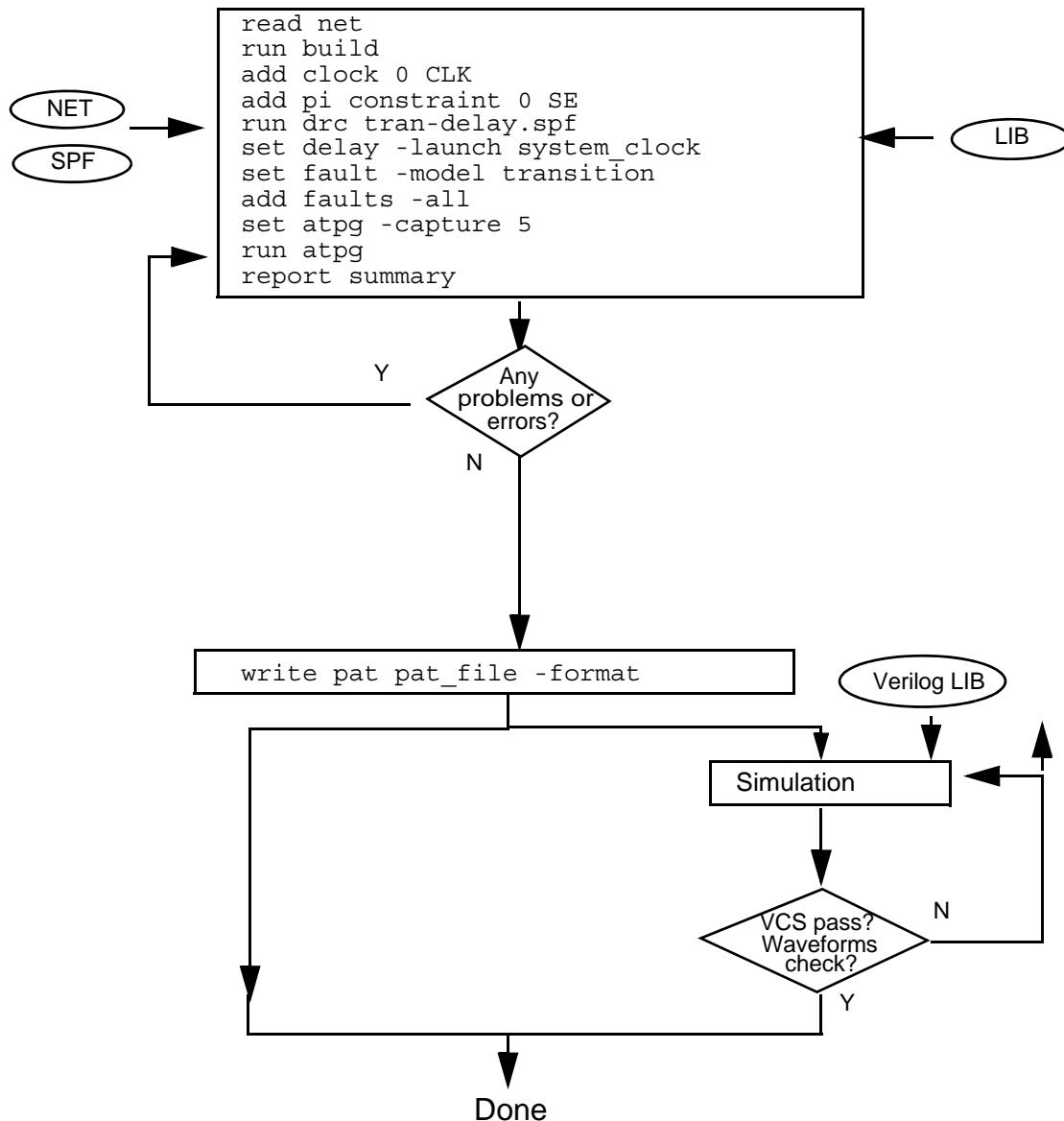
Transition Delay Fault Model

The transition delay fault model is similar to the stuck-at fault model, except that it attempts to detect slow-to-rise and slow-to-fall nodes, rather than stuck-at-0 and stuck at-1 nodes. A slow-to-rise fault at a defect means that a transition from 0 to 1 on the defect does not produce the correct results at the maximum operating speed of the device. Similarly, a slow-to-fall fault means that a transition from 1 to 0 on a node does not produce the correct results at the maximum operating speed of the device.

To detect slow-to-rise or slow-to-fall fault, the ATPG algorithm launches a transition with one clock edge, and then captures the effect of that transition with another clock edge. The amount of time between the launch and capture edges should test the device for correct behavior at the maximum operating speed.

The ATPG process for transition delay faults is similar to the process for stuck-at faults. [Figure 14-1](#) shows the typical steps done for transition delay fault ATPG.

Figure 14-1 Transition Delay Fault Test Flow



Transition Fault ATPG Modes

TetraMAX transition delay fault ATPG supports three ATPG modes for applying transition delay tests: `last_shift`, `system_clock`, and `any`. You select the desired mode with the set delay `-launch_cycle` command.

In the `last_shift` mode, TetraMAX initiates launching a logic value in the last scan load cycle when the scan enable is active (in scan shift mode). It exercises target transition faults and then captures new logic values in a system clock cycle when the scan enable is inactive (in capture mode). [Figure 14-2](#) shows the clock and scan enable timing for this mode.

In the `system_clock` mode, TetraMAX initiates launching a logic value using a normal system clock. It exercises target transition faults and then captures the new logic values with a subsequent system clock. [Figure 14-3](#) shows the clock and scan enable timing for this mode.

In the `any` mode, TetraMAX attempts the `last_shift` mode first, and then uses the `system_clock` mode to test any faults that the `last_shift` mode is unable to detect. This is the default ATPG mode for transition delay faults.

Figure 14-2 Last Shift Launch Timing

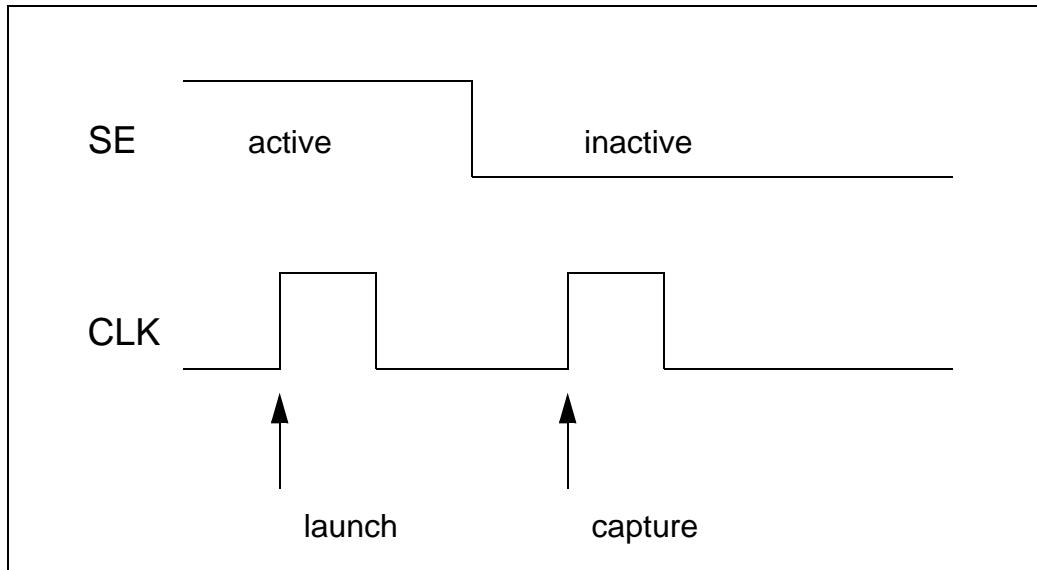
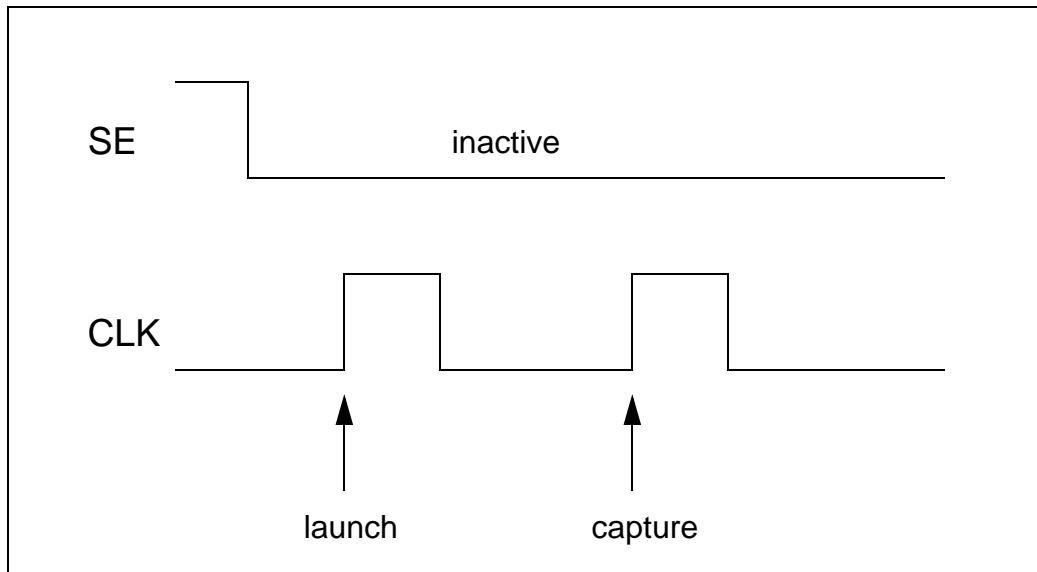


Figure 14-3 System-clock Launch Timing



One of the major differences between these modes is that for the `last_shift` mode, the scan enable signal must switch between a launch and capture cycle, which may or may not be possible depending on the design and cycle time. For details, see [“DRC for Transition Faults” on page 14-8](#). [Figure 14-2](#) and [Figure 14-3 on page 14-5](#) show the clock waveform for a typical target transition fault that is between registers. If the target fault is between primary inputs and registers or if the target fault is between registers and primary outputs, then you can expect just one clock pulse (either launch or capture) or no clock pulse.

The `last_shift` mode generates only Basic-Scan patterns (using a single capture procedure between scan load and scan unload). The `system_clock` mode generates both types of patterns.

By default, when using a `run atpg -auto_compress` command for the `system_clock` mode, TetraMAX uses a highly optimized two-clock ATPG algorithm that has some features of both the Basic-Scan and Fast-Sequential engines. The patterns generated by this algorithm will be only two clock cycles long and listed as Fast Sequential patterns in the TetraMAX pattern summary.

To use the `system_clock` mode, Fast-Sequential ATPG must be enabled prior to starting the ATPG process. You enable Fast-Sequential ATPG and specify its effort level with the `-capture_cycles` option of the `set atpg` command. For details, see [“Set ATPG Command” on page 14-11](#). The two-clock algorithm, used with the `run atpg -auto_compress` command by default, will automatically set the `-capture_cycles` to two.

If there is a need for more than two capture cycles, for example if there are memories in the circuit, you can set the capture cycles to a number larger than two prior to issuing the `run atpg -auto_compress` command. In this case, TetraMAX will first run the optimized two-clock algorithm for all the faults that can be detected in two capture cycles and then run Fast-Sequential ATPG with the larger number of capture cycles for any remaining undetected faults.

STIL Protocol for Transition Faults

TetraMAX, by default, generates a three-event capture procedure: force PI, measure PO, and pulse clock. As a result, a force PI or measure PO event can be inserted between a launch and capture cycle. This will negatively impact an overall quality of a transition test because an extra time delay is added between launch and capture. Therefore, it is recommended that you use a single-event capture procedure containing only the pulse clock event.

If there are scan postamble vectors (that is, vectors following the scan shift in the `load_unload` procedure) in the STIL procedure file (SPF), the extra time delay for the postamble is inserted between the launch and capture cycle in the `last_shift` mode. The extra time delay for `last_shift` will negatively impact the overall quality of the test, but will not impact test quality for `system_clock` mode. If such a scan postamble exists in the `last_shift` or any mode during the ATPG process (when you execute the `run atpg` command), a warning message is reported (M237).

There can be primary inputs initialized to known values in the scan load and unload procedure of a STIL protocol file. This can cause faults between primary inputs and registers to be ATPG untestable in the `last_shift` mode.

Creating Transition Fault Waveform Tables

For transition fault delay paths, you can control the clock speed with different waveform tables: one for the `load_unload` procedure “`_default_WFT_`” and one for the capture procedure “`_fast_WFT_`” (as shown in the following example).

```

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "all_inputs" {01Z {'0ns' D/U/Z;}}
            "all_bidirectionals" {01XZ {'0ns' D/U/X/Z;}}
            "all_bidirectionals" {THL {'0ns' X; '40ns' T/H/L;}}
            "all_outputs" {X {'0ns' X;}}
            "all_outputs" {HLT {'0ns' X; '40ns' H/L/T;}}
            "Pixel_Clk" {P {'0ns' D; '45ns' U; '55ns' D; } }
        }
    }
    WaveformTable "_fast_WFT_" {
        Period '20ns';
        Waveforms {
            "all_inputs" {01Z {'0ns' D/U/Z;}}
            "all_bidirectionals" {01XZ {'0ns' D/U/X/Z;}}
            "all_bidirectionals" {THL {'0ns' X; '8ns' T/H/L;}}
            "all_outputs" {X {'0ns' X;}}
            "all_outputs" {HLT {'0ns' X; '8ns' H/L/T;}}
            "Pixel_Clk" {P {'0ns' D; '9ns' U; '11ns' D; } }
        }
    }
}

```

Here is a load_unload procedure defined in the SPF file for the preceding “_default_WFT_” waveform table:

```

"load_unload" {
W "_default_WFT_";
Shift { W "_default_WFT_";
    V { "BPCICLK"=P; "Pixel_Clk"=P; "Test_mode"=1; "nReset"=1;
        "test_sei"=0; "_so"=###; "_si"=###; }
    }
}

```

The following is an example three-event capture procedure (the default) followed by its recommended single-event capture procedure for the above “_fast_WFT_” waveform table:

```

"capture_Pixel_Clk" {
    W "_default_WFT_";
    F { "CSC_test_mode"=0; "Test_mode"=1; }
    "forcePI": V { "_pi"=\r587 # ; "_po"=\j \r101 X ; }
    "measurePO": V { "_po"=\r101 # ; }
    "pulse": V { "Pixel_Clk"=P; "_po"=\j \r101 X ; } }

"capture_Pixel_Clk" {
    W "_fast_WFT_";
    F { "CSC_test_mode"=0; "Test_mode"=1; }
    V { "_pi"=\r587 # ; "_po"=\r101 # ; "Pixel_Clk"=P; } }

```

Notice that the three pattern events ForcePI, MeasurePO, and PulseClock are in separate vectors in the first capture procedure, but have been combined into a single vector in the second capture procedure.

There is another way to do waveform timing for transition fault testing in TetraMAX. TetraMAX allows the use of special waveform tables for at-speed testing; both transition fault testing and path delay fault testing. There are separate waveform tables for the clock cycle in which a transition is launched (`_launch_WFT_`), for the clock cycle in which a transition is captured (`_capture_WFT_`), and for cycles in which a transition is both launched and captured (`_launch_capture_WFT_`).

The use in transition fault testing is different from the use in path delay testing. For path delay testing, these special waveform tables are always used. If they are not present in the SPF file, they are first created and then used.

The difference for transition faults is that these special waveform tables must be present in the SPF file in order to be used; TetraMAX will not create them for transition fault ATPG.

Several additional options for timing support are available. For information about waveform tables, see “[Defining Basic Signal Timing](#)” on page 9-14. To get more details about specialized timing support for both transition and path delay environments, see

- “[Pattern Formatting for Transition Delay Faults](#)” on page 14-14
- “[Creating DSMTTest WaveformTables](#)” on page 15-15
- “[MUXClock Support for Transition Patterns](#)” on page 14-16

DRC for Transition Faults

In the `last_shift` mode, the scan enable signal must have a transition between launch and capture. In the `system_clock` mode, the scan enable signal must be inactive between launch and capture, so the `add pi` constraints command (or a constraint in the STIL procedure file) must be used to set the scan enable signal to inactive. Otherwise, you might get patterns in the `system_clock` mode with the scan enable signal switching between launch and capture. This transition fault ATPG requirement does not normally apply to stuck-at ATPG.

Note:

In the case of At-speed ATPG, the ScanEnable, Set, and Reset signals should not pulse during capture because they are typically slow signals.

You can use the `-clock port_name` option of the `set drc` command to enable a specific clock and to disable other clocks in a design. This option can be useful for transition delay fault ATPG if you want to target only those faults that can be launched and captured from a

specific clock (for example, to prevent skew between different clock domains). However, this option works only in the `last_shift` mode. In the `system_clock` mode, you can use the `add pi constraints` command to disable the clocks that you do not want to be used.

Limitations of Transition Delay Fault ATPG

The following limitations apply to transition delay fault ATPG:

- For a target fault between a register and an output, only a launch clock is needed to test. In TetraMAX, an output strobe occurs prior to a clock pulse (when using a single-cycle capture procedure). This adds an extra capture cycle without a clock pulse just to strobe an output, which might negatively impact the overall quality of the transition delay fault test. For this type of fault to be tested effectively, an output strobe after a clock pulse (end of cycle measure) should be used, which is not supported in the current release.
- For pattern formatting, the FAST_MUXCLOCK (also called MUXClock) technique is not supported unless you set the options:

```
- set faults -model_transition  
      set delay -launch_type system_clock  
- set delay -nopi_changes  
- add po masks -all  
- set atpg -capture_cycles > 1
```

These constraints are necessary to generate patterns appropriate for MUXClock operation. The FAST_CYCLE technique is not supported in the `system_clock` mode if the launch and capture clock are the same.

- The Verilog testbench written out by TetraMAX only supports a single period value for all cycle operations. This implies that a single waveform table can be taken into account when writing out the testbench. The delay waveform tables are not supported with the Verilog testbench. The flow is to write out the STIL vectors and then use the Verilog DPV PLI with VCS. Refer to the *Test Pattern Validation User Guide*.

Specifying Transition Delay Faults

To start the transition delay fault ATPG process, you need to select the transition fault model with the `set faults` command. Then you can add faults to the fault list using the `add faults` or `read faults` command. You can select all fault sites, a statistical sample of all fault sites, or individually specified fault sites for the fault list.

Selecting the Fault Model

The transition fault model is selected with the `set faults -model transition` command. The fault model can be changed during a TetraMAX session so that patterns produced with one fault model can be fault-simulated with another fault model. To do this, you need to remove faults and use the `set patterns external` command prior to the fault simulation run.

The three available transition delay fault ATPG modes (`last_shift`, `system_clock`, and `any`) can be selected by the `-launch_cycle` option to the `set delay` command. The default is the `any` mode. This option selection is valid only if the transition model is selected with the `-model transition` option. In the `any` mode, TetraMAX

- Attempts to detect all faults using `last_shift` mode
 - Applies `system_clock` mode to target faults left undetected by the `last_shift` mode
-

Adding Faults to the Fault List

The `add faults` command adds stuck-at or transition faults to fault sites in the design. The faults added to the fault list are targeted for detection during test pattern generation.

To add a specific transition fault to the design, use the `pin_pathname -slow` option and specify R, F, or RF to add a slow-to-rise fault, a slow-to-fall fault, or both types of faults, respectively. To add faults to all potential fault sites in the design, use the `-all` option.

To add a statistical sample of all faults to the fault list,

1. Add all possible transition faults:

```
add faults -all
```

2. Remove all but the desired percentage (10 percent in this example):

```
remove faults -retain_sample 10
```

Reading a Fault List File

To read a list of faults from a file, use the `read faults` command.

A fault file can be read into TetraMAX and should have the format shown below. Each node of the design has two associated transition faults: slow-to-rise (str) and slow-to-fall fault (stf). Attempting to read a fault list containing stuck-at fault notation (sa1 and sa0) results in an “invalid fault type” error message (M169).

```
str  NC  /TOP/EMU_FLK/SYNTOP_GG/NR1/A  
stf  NC  /TOP/EMU_FLK/SYNTOP_GG/U123/Z
```

Pattern Generation for Transition Delay Faults

The TetraMAX commands for transition fault ATPG are the same as the commands for stuck-at fault ATPG. You should be aware of how the command options affect the operation of ATPG for the transition delay fault model.

Set ATPG Command

The `set atpg` command sets the parameters that control the ATPG process.

The `-merge` option is effective in reducing a number of transition patterns in the `last_shift` mode.

You can enable Fast-Sequential ATPG by using the command `set atpg -capture_cycles d`, where d is a nonzero value. However, the `last_shift` mode is based strictly on the Basic-Scan ATPG engine. Therefore, when you use `run atpg` in the `last_shift` mode, TetraMAX uses Basic-Scan ATPG only and generates Basic-Scan patterns, even if Fast-Sequential ATPG has been enabled. No warning or error message is reported to indicate that Fast-Sequential ATPG has been skipped.

The `system_clock` mode is based on Fast-Sequential ATPG engine. If Fast-Sequential ATPG is not enabled when you use `run atpg` in the `system_clock` mode, TetraMAX reports an error message (M236).

When you enable Fast-Sequential ATPG with the `set atpg -capture_cycles d` command, you must set the effort level d to at least 2 for the `system_clock` mode. If you try to set it to 1 in the `system_clock` mode, the `set atpg` command returns an “invalid argument” error. For full-scan designs, you can set the effort level d to 2. For partial-scan designs, a number greater than 2 might be necessary to obtain satisfactory test coverage.

For the complete syntax and option descriptions, see the online help for the `set atpg` command.

Set Delay Command

The `set delay` command determines whether the primary inputs are allowed to change between launch and capture.

The default setting is `-pi_changes`, which allows the primary inputs to change between launch and capture. With this setting, slow-to-transition primary inputs can cause the transition test to be invalid.

The `-nopi_changes` setting causes all primary inputs to be held constant between launch and capture, thus preventing slow-to-transition primary inputs from affecting the transition test. This setting is useful only in the `system_clock` mode. The `-nopi_changes` characteristic must be set before you use the `run atpg` command.

The `-nopi_changes` option causes an extra unclocked tester cycle to be added to each generated transition fault or path delay pattern. The use of a `set drc -clock -one_hot` command may interfere with the addition of this unclocked cycle and is not recommended for use when the `-nopi_changes` option is in effect.

The primary outputs can still be measured between launch and capture. To mask all primary outputs, use the `add po masks -all` command.

For the complete syntax and option descriptions, see the online help for the `set delay` command.

Run ATPG Command

The `run atpg` command starts the ATPG process. The `-auto_compression` option should be used.

The `-auto_compression` option works for transition delay fault ATPG, but it is not as effective as it is for stuck-at. Also, when you use the `-auto_compression` option, you must enable the appropriate ATPG mode using the `capture_cycles d` option of the `set atpg` command for the transition ATPG mode in effect: Basic-Scan ATPG for the `last_shift` mode, or Fast-Sequential ATPG for the `system_clock` or `any` mode.

The `run atpg` command has three additional ATPG options: `basic_scan_only`, `fast_sequential_only`, and `full_sequential_only`. Under normal conditions, you should not attempt to use these options to start transition delay fault ATPG. If you do so, be aware of the following:

- If you use the Full-Sequential ATPG engine with transition faults, you should be aware that its behavior is not controlled by `set delay -launch_cycle` command options. If you wish to avoid last-shift launch patterns and generate only system-clock launch patterns with the Full-Sequential engine, you must constrain all scan enable signals to their inactive values. Conversely, if you want to generate only last-shift launch patterns and avoid all system-clock launch patterns, you should be aware that there is no way to guarantee that you will get only last-shift launch patterns with the Full-Sequential engine. Even those last-shift launch patterns that it may generate will not be identical in form to those generated by the Basic-Scan ATPG engine.

- The `basic_scan_only` and `fast_sequential_only` options work for transition delay fault ATPG when used correctly: `basic_scan_only` for the `last_shift` mode, or `fast_sequential_only` for the `system_clock` mode. If you use the wrong command option, no patterns are generated and no warning or error message is reported.

For the complete syntax and option descriptions, see online Help for the `run atpg` command.

Pattern Compression for Transition Faults

Dynamic pattern compression specified by the `set atpg` command works for transition faults as it does for stuck-at faults.

Report Faults Command

The `report faults` command provides various types of information on the faults in the design.

You can use the `-slow` option to report a specific transition fault.

The fault classes for transition delay fault ATPG are the same as for stuck-at ATPG. There are no specific fault classes that apply only to transition delay faults. The faults classified as DI (Detected by Implication) prior to the ATPG process for transition delay fault ATPG are the same as for stuck-at ATPG.

The total number of transition faults in a design is the same as the total number of stuck-at faults.

For the complete syntax and option descriptions, see the online help for the `report faults` command.

Write Faults Command

The `write faults` command writes fault data to an external file. The file can be read back in later to specify a future fault list.

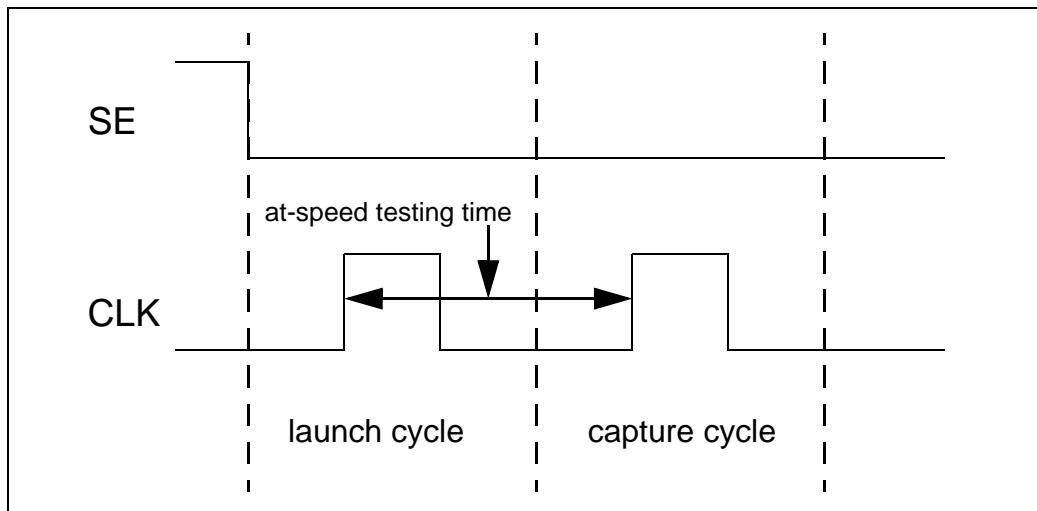
You can use the `-slow` option to write out a specific transition fault to a file.

For the complete syntax and option descriptions, see the online help for the `write faults` command.

Pattern Formatting for Transition Delay Faults

For a transition test to be effective, the time delay between launch and capture should be an at-speed value, as illustrated in [Figure 14-4](#).

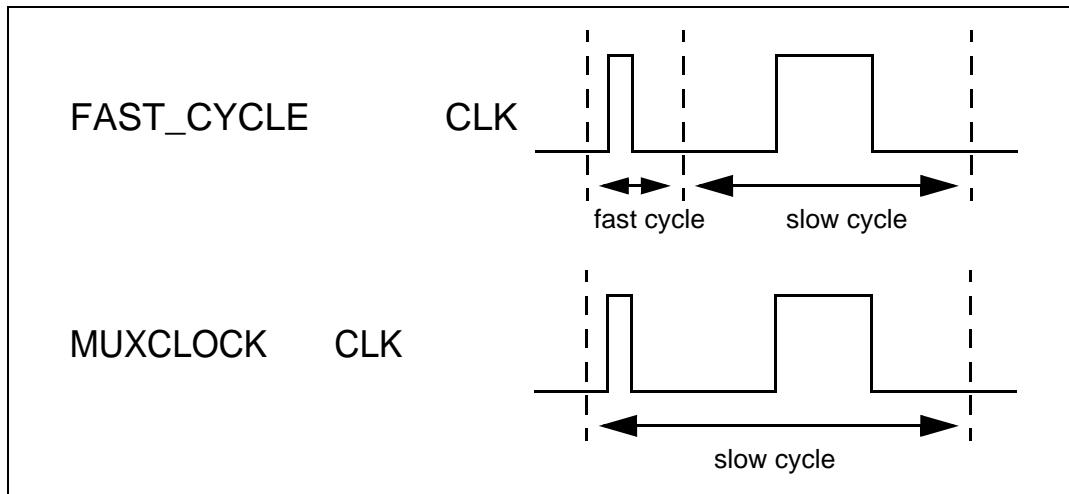
Figure 14-4 At-Speed Transition Test Timing



A fast tester might be able to generate two clock pulses (cycles) at the desired at-speed value without dynamic cycle-time switching or other special timing formatting. For these testers, TetraMAX can generate ready-for-tester transition patterns.

A slow tester might need dynamic cycle-time switching or a special timing format to test a transition fault at the desired at-speed value. In general, two pattern formatting techniques are available for slow testers. In TestGen terminology, these two techniques are called FAST_CYCLE and MUXClock. [Figure 14-5](#) illustrates these techniques.

Figure 14-5 Pattern Formatting Techniques



Using the FAST_CYCLE technique, the cycle time is switched dynamically from fast time to slow time. Using the MUXClock technique, two tester timing generators are logically ORed to produce two clock pulses in one cycle.

The FAST_CYCLE technique is supported for the following cases:

- The `last_shift` mode is being used. The waveform format of the scan load and scan unload procedure in a STIL protocol file can be different from that of a capture clock procedure.
- The `system_clock` mode is being used and the launch clock is different from the capture clock. In this case, each capture clock procedure can have its own waveform format.

The FAST_CYCLE operation is supported by defining specific WaveformTables to apply to the launch and capture vectors. The constructs necessary to support the creation of these test cycles are the same constructs used for path delay test generation. See the section “[Creating DSMTTest WaveformTables](#)” on page 15-15. The constructs presented in that section are also used to identify the launch and capture timing for transition delay tests.

The testgen FAST_MUXCLOCK operation is supported by defining TetraMAX MUXClock constructs. However, to apply MUXClock behavior to transition tests requires the following set of options to be specified when transition tests are developed:

- `set faults -model_transition`
- `set delay -launch_type system_clock`

- set delay -nopi_changes
- add po masks -all
- set atpg -capture_cycles > 1

These options will support the creation of patterns that may merge the launch and capture operations into a single test vector necessary to support MUXClock application. To create MUXClock-based patterns use the same constructs defined for MUXClock path delay definitions. For information on these constructs, see “[Creating DSMTTest WaveformTables](#)” on page 15-15.

MUXClock Support for Transition Patterns

The following limitations apply to MUXClock support for transition patterns:

- Output pattern files containing MUXClock waveforms are not yet readable in TetraMAX.
- Bidirectional clocks in a design are not supported in WGL output when MUXClock definitions are present. STIL output supports bidirectional clocks in the design.
- MUXclock is not supported with clock_grouping.

To disable multiple clocks and dynamic clocking and use only a single clock for launch and capture, exclude these commands from your command file:

```
#set delay -common_launch_capture_clock
#set delay -NOAllow_Multiple_common_clocks
```

Specifying Setup Timing Exceptions From an SDC File

TetraMAX can read setup timing exceptions directly from an SDC (Synopsys Design Constraints) file. You can use an SDC file written by PrimeTime or create one independently, but it must adhere to standard SDC syntax. This section describes the flow associated with reading an SDC file. Note that this flow is supported only in Tcl mode.

Note: For more information on PrimeTime timing exceptions, see “[Translating PrimeTime Timing Exceptions](#)” in [Appendix D, “Utilities](#) and “[Using the Synopsys Design Constraints Format Application Note](#)” on [Docs on the Web \(DOW\)](#):

This section includes the following subsections:

- [Reading an SDC File](#)
- [Interpreting an SDC File](#)
- [How TetraMAX Interprets SDC Commands](#)

- [Controlling Clock Timing](#)
 - [Controlling ATPG Interpretation](#)
 - [Controlling Timing Exceptions Simulation for Stuck-at Faults](#)
 - [Reporting SDC Results](#)
 - [Limitations](#)
-

Reading an SDC File

You use the `read_sdc` command to read in an SDC file. Note that you must be in DRC mode (after you successfully run the `run_build` command, but before running the `run_drc` command) in order to use the `read_sdc` command.

The syntax for the `read_sdc` command is as follows:

```
read_sdc  file_name
[-echo]
[-syntax_only]
[-version sdc_version]
```

Argument	Description
<code>file_name</code>	The name of the SDC file to be read
<code>-echo</code>	Echoes all commands
<code>-syntax_only</code>	Only performs syntax/semantic checks
<code>-version sdc_version</code>	SDC version — default is the latest; values: 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, latest

Note that SDC commands cannot be entered on the command line; they must be specified in an SDC file and can only be executed via the `read_sdc` command.

Interpreting an SDC File

To control how TetraMAX interprets an SDC file, you can specify the `set_sdc` command. This command will only work if you specify it before the `read_sdc` command. As is the case with the `read_sdc` command, you must be in DRC mode in order to use the `set_sdc` command.

The syntax for the `set_sdc` command is as follows:

```
set_sdc
    [-verbose | -noverbose]
    [-case_paths | -nocase_paths]
    [-disable_paths | -nodisable_paths]
    [-false_paths | -nofalse_paths]
    [-multicycle_paths | -nomulticycle_paths]
    [-instance instance_path]
    [-environment < sdc_case_analysis | tmax_drc | none >]
    [-show_all_matches | -noshow_all_matches]
```

Argument	Description
<code>-verbose -noverbose</code>	The <code>-verbose</code> switch causes messages to be printed for SDC commands and options not supported by TetraMAX. The <code>-noverbose</code> switch restores the default messaging.
<code>-case_paths -nocase_paths</code>	The <code>-case_paths</code> option creates <code>-through</code> exceptions for paths blocked by <code>set_case_analysis</code> settings in the SDC file. Note that <code>set_case_analysis</code> is still used for clock tracing no matter how this switch is set. The <code>-nocase_paths</code> option restores the default.
<code>-disable_paths -nodisable_paths</code>	The <code>-disable_paths</code> option (the default) uses <code>set_disable_timing</code> commands in the SDC file for exceptions. The <code>-nodisable_paths</code> option makes sure that these commands are not used for exceptions.
<code>-false_paths -nofalse_paths</code>	The <code>-false_paths</code> option (the default) creates timing exceptions for a false path. The <code>-no_false_paths</code> option makes sure that timing exceptions are not created for a false path.

Argument	Description
-multicycle_paths -nomulticycle_paths	The <code>-multicycle_paths</code> option (the default) uses <code>set_multicycle_path</code> commands in the SDC file for exceptions. The <code>-nomulticycle_paths</code> option makes sure that timing exceptions are not created for multicycle paths.
-instance instance_path	Prepends the specified argument to all instances in the SDC file. This option should be used in case the SDC was written for a submodule of the top-level design. For example:
	<pre># First SDC is for instance "a" inside # top-level instance "top" set_sdc -instance /top/a read_sdc my.sdc</pre>
	<pre># Second SDC is for the top-level design set_sdc -instance "/" read_sdc second.sdc</pre>
-environment < sdc_case_analysis tmax_drc none >	Controls clock timing. For details, see the “ Controlling Clock Timing section on page 14-21 ”.
-show_all_matches -noshow_all_matches	The <code>-show_all_matches</code> option specifies that for each SDC command, all TetraMAX database objects that match the specified points will be shown. This results in an extremely verbose listing. The <code>-noshow_all_matches</code> option (the default) makes sure that these database objects will not be shown.

Note that the `set_sdc` command settings are cumulative; this command may be run multiple times to prepare for a `read_sdc` command. If multiple `read_sdc` commands are required, you can also specify an `set_sdc` command before each `read_sdc` command to specify its verbosity and instance.

How TetraMAX Interprets SDC Commands

TetraMAX creates timing exceptions for transition delay testing based on a specific set of SDC commands. Some SDC commands also identify the clock or the generated clock source, and assist in tracing clocks to specific registers. All SDC commands must be specified in an SDC file.

The following list describes the specific set of SDC commands that are used by TetraMAX, and how they are interpreted:

- `set_disable_timing` — This command creates a timing exception by disabling timing arcs to and/or from the specified points. TetraMAX does not support library cells in the `object_list`.
- `set_false_path` — This command creates a timing exception for a false path according to the specified from, to, or through points. TetraMAX does not distinguish between edges; this means, for example, that `-rise_from` is interpreted the same as `-from`. Only setup exceptions can become timing exceptions for TetraMAX, so if `-hold` is specified without `-setup`, the command is ignored.
- `set_multicycle_path` — This command creates a timing exception for a multicycle path according to the specified from, to, or through points. TetraMAX does not distinguish between edges. This means, for example, that `-rise_from` is interpreted the same as `-from`. Only setup exceptions can become timing exceptions for TetraMAX, so if `-hold` is specified without `-setup`, the command is ignored. Setup path multipliers of 1 are ignored.
- `create_clock` and `create_generated_clock` — For both of these timing exception commands, the `-name` argument and the `source_objects` are used to identify either the clock or the generated clock sources. Clocks must be traced to specific registers so there are some support limitations. “Virtual clock” definitions without a `source_object` are ignored. Multiple clocks that are defined with the same `source_objects` cannot be distinguished from each other. Note that clocks defined in the SDC file are only used to identify timing exceptions and only clocks defined by the TetraMAX `add_clocks` command or in the STIL protocol are used for pattern generation.
- `set_case_analysis` — This command is used to assist in tracing clocks to specific registers. Only static logic values (0 or 1, not rising or falling) are supported. This information is used based on the value set by the `set_sdc_environment` command (see “[Controlling Clock Timing on page 14-21](#) for details).
- `set_clock_groups` — This command creates a timing exception that specifies exclusive or asynchronous clock groups between the specified clocks. It works only if the `-asynchronous` switch is used and the `-allow_paths` switch is not used. All other usages are ignored.

Controlling Clock Timing

You can control clock tracing using the `set_sdc -environment` switch. The syntax for this switch is as follows:

```
set_sdc -environment < sdc_case_analysis | tmax_drc | none >
```

Argument	Description
<code>sdc_case_analysis</code>	Traces clocks based on the SDC <code>set_case_analysis</code> commands. This is useful when SDC is created for the functional mode of a design when the test mode clocks are different.
<code>tmax_drc</code>	Traces clocks based on the TetraMAX <code>run_drc</code> results. This is useful for SDC created specifically for test mode, which will give the most accurate results.
<code>none</code>	The default, <code>none</code> , causes clocks to be traced without reference to logic values that are defined either in the SDC <code>set_case_analysis</code> commands or found by the TetraMAX <code>run_drc</code> command. As a result, the clock will propagate to the maximum possible number of registers.

Controlling ATPG Interpretation

In some cases, you may want to treat multicycle paths below a certain number as if they are single-cycle paths. To do this, use this `set_delay -multicycle_length` switch:

```
set_delay -multicycle_length <N>
```

Based on this option, all `set_multicycle_path` exceptions with numbers of N or less will be ignored. The default is to treat all multicycle paths of length 2 or greater as exceptions.

Controlling Timing Exceptions Simulation for Stuck-at Faults

You can use the following `set_simulation` switch to control timing exceptions simulation for stuck-at faults:

```
set_simulation [-timing_exceptions_for_stuck_at |  
-notiming_exceptions_for_stuck_at]
```

The default is `-notiming_exceptions_for_stuck_at`.

Reporting SDC Results

There are several ways you can report SDC results. You can report specific types of results using the `report_sdc` command (note that this command can only be run in TEST mode). The syntax for this command is as follows:

```
report_sdc
  [-clocks]
  [-to_cells]
  [-groups]
  [-case_analysis]
  [-false_paths]
  [-multicycle_paths]
  [-case_paths]
  [-disable_paths]
  [-all_paths]
```

Argument	Description
<code>-clocks</code>	Reports SDC clocks specified by the <code>create_clock</code> and <code>create_generated_clock</code> commands. This does not include clocks specified only by regular TetraMAX commands.
<code>-to_cells</code>	Reports SDC clocks and the registers driven by these clocks. This report is useful for debugging clock-to-clock exceptions, but can be very long.
<code>-groups</code>	Reports exceptions specified by the <code>set_clock_groups</code> command.
<code>-case_analysis</code>	Reports constant signals specified by the <code>set_case_analysis</code> command.
<code>-false_paths</code>	Reports exceptions specified by the <code>set_false_path</code> command.
<code>-multicycle_paths</code>	Reports exceptions specified by the <code>set_multicycle_path</code> command.

Argument	Description
-case_paths	Reports exceptions specified by the set_case_analysis command (with set_sdc -case_paths).
-disable_paths	Reports exceptions specified by the set_disable_timing command.
-all_paths	Reports all exceptions.

You can also use the `report_slow_path` command to report exceptions from an SDC file, but with a different format than used to report exceptions that are added by the `add_slow_path` command.

In addition, the `report_settings` command has a switch that reports the current settings specified by the `set_sdc` command:

```
report_settings sdc
```

A pindata type related to SDC is available. You can control the display of this pindata type by specifying the following command:

```
set_pindata -sdc_case_analysis
```

The format of the data is N/M

where N is the case analysis setting from the SDC, and M is the TetraMAX constraint value. Unconstrained values are printed as x. You can also specify the display of this pindata type directly from the GSV Setup menu.

Limitations

The following limitations apply to SDC support in TetraMAX:

- The input SDC file must contain only SDC commands — not arbitrary PrimeTime commands. Constraints files comprised of arbitrary PrimeTime commands interspersed with SDC commands are unreliable. If the SDC file can be read into PrimeTime using its `read_sdc` command, then it can be read into TetraMAX. If it must be read into PrimeTime using the `source` command, then it cannot be read into TetraMAX. PrimeTime can write SDC, and this output is valid as SDC input for TetraMAX.

- Slow path commands cannot be mixed with SDC. Once the SDC file is read into TetraMAX, none of the slow path commands can be used (with the exception of `report_slow_path`).
- Multicycle 1 paths cannot be used. In some applications, a `set_multicycle_path` command is used for one set of paths, but is followed by another `set_multicycle_path` command — with a `path_multiplier` of 1 on a subset of these paths. This is used to set that subset back to single-cycle timing. TetraMAX does not support this usage.
- Full Sequential ATPG, including Path Delay Testing, is not supported. The only types of timing exceptions supported by Full Sequential ATPG are the `add_slow_cells` and `add_capture_masks` commands.

Small Delay Defect Testing

As geometries shrink, there has been an increasing need to test for small delay defects. Standard transition fault test generation is often inadequate because it focuses on finding the easier (and shorter) paths. Unfortunately, this methodology usually overlooks small delay defects. TetraMAX attempts to remedy this issue by providing special functionality to test for small delay defects.

The TetraMAX approach to small delay defect testing is basically a subset of the transition fault model. When the small delay defect testing functionality is activated, TetraMAX will generate a specific set of transition fault tests that systematically seek out the longest paths.

This section describes how to use TetraMAX to test for small delay defects. It includes details on how to extract slack data from PrimeTime, how to read that data into TetraMAX, and how to use various TetraMAX commands, command options, and flows related to testing small delay defects.

Note:

When this feature is activated, tests are generated for both small delay defects and regular transition faults in a single ATPG run.

Small delay defect testing is described in the following sections:

- [Basic Usage Flow](#)
- [Special Elements of Small Delay Defect Testing](#)
- [Limitations](#)

Basic Usage Flow

The basic flow for testing small delay defects in TetraMAX includes the following steps:

- [Extracting Slack Data from PrimeTime](#)
- [Understanding the Slack Data File Format](#)
- [Utilizing Slack Data in the TetraMAX Flow](#)
- [Command Support](#)

Extracting Slack Data from PrimeTime

TetraMAX utilizes a specific set of timing data extracted from PrimeTime. To obtain this information, you need to call a function that extracts slack data for all pins from PrimeTime. This function, `write_timing_slacks`, is defined in the `pt2tmax.tcl` script that is included

in the TetraMAX release tree. As shown in the syntax description below for `write_timing_slacks`, you need to specify a name for the output file to which the slack information is written.

Syntax:

```
write_timing_slacks <output_filename>
```

Understanding the Slack Data File Format

The script example in the previous section creates a slack data file that TetraMAX can read and correctly parse the data. The following example shows a typical slack data file with proper formatting:

```
SPARE10/VLO1X/Z INFINITY INFINITY
U54/A 21 INFINITY
U73/A 22.286619 23.575638
U113/A1 -23.400898 -21.773178
U113/B1 22.020908 0
U113/B2 23.755539 INFINITY
U113/C 22.045767 21.629681
U_IO_BUF/U_D0/A 1.552831 0.214887
U_IO_BUF/U_D0/EN 3.700592 3.938488
```

Note the following:

- The first column is the node on which the slack is applied.
- The middle column is the rising transition slack.
- The last column is the falling transition slack.
- “INFINITY” is a valid slack data value.
- Scientific notation (for example, 3.7E-2) is also accepted; zero (0) and negative numbers are also accepted.
- Extra white space (spaces) between characters is acceptable.

Utilizing Slack Data in the TetraMAX Flow

After producing a slack data file, you need to read it into TetraMAX using the `read_timing` command. Make sure you specify this command after entering DRC or TEST mode (after a successful `run_build` or `run_drc`).

When TetraMAX reads in a slack data file, it uses a set of small delay defect testing algorithms to construct a pattern for the target fault. If TetraMAX doesn’t read in the slack file, regular transition delay ATPG is performed.

How TetraMAX Integrates Slack Data

During ATPG, TetraMAX selects the first available fault from the list of target faults. TetraMAX then uses the available slack data for the selected fault, and attempts to construct a delay test that makes use of the longest sensitizable path available. Secondary target faults for that same pattern may not have their longest testable path sensitized because some values have already been set in the test for the primary target fault. Faults that are detected only by fault simulation without being targeted by test generation will not necessarily be detected along a long path. However, the fault simulator will use the slack data to determine the size of defect that could be detected at that fault site by the pattern.

ATPG algorithms typically obtain their efficiency by first targeting the easiest solution. This means that transition faults are more likely detected along the shorter paths or paths with larger slack. The concept of fault simulation, along with ATPG, adds to the efficiency by accounting for transition faults that are randomly detected by the tests generated for the targeted fault. Those transition faults that are detected only by fault simulation represent a large fraction of the detected faults and they are usually detected along paths with slacks that are random with respect to all the paths on which the faults could be detected.

Command Support

[Table 14-1](#) lists the key commands available to help validate the flow and pattern content.

Table 14-1 Key TetraMAX Commands

Command	Comment
read_timing <FILE> [-delete]	Reads in minimum slack data in defined format and optionally deletes previous data.
set_pindata slack	Sets the displayed pindata type to show slack data.
set_delay [-noslackdata_for_atpg -slackdata_for_atpg]	Turns on and off the small delay defect testing function during ATPG. If slack data exists, the default is -slackdata_for_atpg.
set_delay [-noslackdata_for_faultsim -slackdata_for_faultsim]	Turns on and off the small delay defect testing function during fault simulation. If slack data exists, the default is -slackdata_for_faultsim.
set_delay -max_tmgn <float defect%>	Defines the cutoff for “faults of interest” in the small delay defect test generation. Faults with minimum slacks larger than the -max_tmgn parameter will not be targeted by the test generator, but will be fault-simulated.

Table 14-1 Key TetraMAX Commands

Command	Comment
<code>set_delay -max_delta_per_fault <float></code>	Sets a “level” between the longest path and the path on which the fault is detected. This will still give full detection credit, with the fault being dropped from further consideration. Default is zero (full credit only when detected on the minimum slack path).
<code>report_faults [-slack tmgn [integer float]]</code>	Reports histogram of faults based on the minimum slack numbers read in by <code>read_timing</code> command..The reported histogram can be either fixed in number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.
<code>report_faults -slack tdet [integer float]</code>	Reports a histogram of faults based on the slack numbers for the detection path for each fault (detection slacks)..The reported histogram can be either fixed in number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.
<code>report_faults -slack delta [integer float]</code>	Reports a histogram of faults based on the difference between detection slacks and minimum slacks. The reported histogram can be either fixed in number of buckets or fixed in the slack interval between two consecutive buckets. The fixed number of buckets is specified by an integer and the fixed bucket interval is specified with a float. The default is an integer of 10.
<code>report_faults -slack effectiveness</code>	Reports a measure of the effectiveness of the small delay defect test set. The measure varies from 0% (no faults of interest with detection slacks smaller than the <code>-max_tmgn</code> parameter) to 100% (all faults of interest detected on the minimum-slack path).
<code>report_faults -slack sdql</code>	Reports the SDQL value for the pattern set.

Table 14-1 Key TetraMAX Commands

Command	Comment
<code>set_delay -sdql_coefficient [A B C D E]</code>	Specifies the values to be used in either the power function or the exponential function when computing the SDQL number. If you don't specify this option, the default values of A, B, C, D, and E are 1, 1, 0, 0, and infinity.
<code>set_delay -sdql_histogram -sdql_power_function -sdql_exponential_function</code>	Specifies the type of probability distribution function that is to be used in computing the Statistical Delay Quality Level (SDQL).

Special Elements of Small Delay Defect Testing

The process of creating small delay defect tests produces some unique functionality that you may find useful throughout the flow. This section describes some of the special elements that may be present in the small delay defect testing flow, and relates them to the commands from [Table 14-1](#). These special elements include the following:

- [Allowing Variation From the Minimum-Slack Path](#)
- [Defining Faults of Interest](#)
- [Reporting Faults](#)

Allowing Variation From the Minimum-Slack Path

When creating tests for small delay defects, the transition fault test generator targets the path with minimum slack for the primary target fault. As with regular transition fault ATPG, there may be secondary target faults that are targeted following the successful generation of a test for the primary target fault.

It is likely that many faults detected during fault simulation will not be targeted faults for the test generator. As such, you will need to decide whether you are willing to accept any test that detects a transition fault, or only a test that detects the fault along a path with small slack. To specify what type of test you are willing to accept, you use the `set_delay -max_delta_per_fault` command.

If you are unwilling to accept a fault unless it has been detected along the path that has the absolute smallest slack for the fault, then you can use a setting of 0 for the `-max_delta_per_fault` parameter (this is also the default setting). If you are willing to accept any test that comes within 0.5 time units of the minimum slack for the fault, then set `-max_delta_per_fault` to 0.5. This allows you to control when faults can be dropped from simulation in a small delay defect ATPG run.

When a fault is detected with a slack that exceeds the minimum slack by more than the `-max_delta_per_fault` parameter, the fault goes into a special sub-category of Detected (DT). This sub-category is called Transition Partially-detected (TP). A fault that has gone into the TP category may continue to be simulated in hopes of getting a better test for the fault.

When a fault is detected with a slack that is equal to or smaller than the `-max_delta_per_fault` parameter, that fault is placed in the DS category that is normally used for detected transition faults. A DS category fault will always be dropped from further simulation.

It is important to note that a `-max_delta_per_fault` specification of 0 is likely to produce the highest quality test set. However, this specification is also likely to produce the longest run-times and the largest test sets. The `-max_delta_per_fault` setting allows you to choose an acceptable tradeoff point for test set quality versus run-time and test set size.

Defining Faults of Interest

While small delay defects are targeted in this flow, you might find that certain faults that would not be susceptible to such small defects are not targeted for test generation. TetraMAX includes an option that enables you to specify how small the slack needs to be for TetraMAX to target the fault for small delay defect test generation. If you specify `set_delay -max_tmgn`, the test generator will target only those faults with a slack smaller than the `-max_tmgn` parameter. All of the faults will be fault simulated even if they are not designated as “faults of interest” to be targeted in test generation.

In some cases, you might want to examine the distribution of slacks in order to determine a reasonable value of `-max_tmgn`. You can do this using the `report_faults -slack tmgn` command, which will print a histogram of the slack values that are read in by the read timing command. Note that there is an optional parameter to the `report_faults -slack tmgn` command that allows you to specify how many categories will be used in this report.

Reporting Faults

Small delay defect testing is intended to produce tests along paths with smaller slack than those typically activated in regular transition fault test generation. There are several options to the `report_faults` command that facilitate examination of this data:

- The `report_faults -slack tdet` command will print a histogram that gives the slack of the detection paths. This can be compared directly against the output of the `report_faults -slack tmgn` command to see how close TetraMAX got to the minimum slack paths.

- The `report_faults -slack delta` command more clearly shows the slack of the detection paths. The reporting histogram associated with this command is based on the difference between the slacks for the detection paths and the minimum slack that was read in from the slack data file. A distribution that is heavily skewed toward the zero end of the continuum would indicate a highly successful small delay defect test generation.
 - The `report_faults -slack effectiveness` command reports a measure of delay effectiveness that is based on how close the slacks for the fault detection paths came to the minimum slacks. If every fault defined to be of interest is detected on its minimum slack path, the delay effectiveness measure would be 100%. If none of the faults of interest are detected on paths that have slack smaller than the `-max_tmgn` parameter used to define faults of interest, the delay effectiveness measure would be 0%.
-

Limitations

The following limitations are currently apply to small delay defect testing.

- [Engine and Flow Limitations](#)
- [Algorithmic Limitations](#)
- [Limitations in Support for TetraMAX Primitives](#)

Engine and Flow Limitations

Last-shift launch and two-clock transition fault testing are supported. There is currently no support for general Fast-Sequential, or support for Full-Sequential mode.

Algorithmic Limitations

There are two known limitations in the algorithm used for small delay defect ATPG:

- **Second Smallest Slack**

If the path with the smallest slack for a given fault is untestable, TetraMAX will begin normal back-tracking in an attempt to find a test along some other path. There is no guarantee that the second path TetraMAX will try will be the path with the second smallest slack. For now, the only guarantee is that the first path tried is the path with the smallest slack.

- **Test May End Prematurely at PO**

When propagating a fault effect along the minimum-slack propagation path, the fault effect might propagate to a primary output. If this occurs, and the fault can be considered detected at the primary output, TetraMAX will stop trying to propagate the fault effect along the minimum-slack path. This can produce fault detection on a path with larger slack than desired.

In this case, the detection slack is measured accurately and will reflect the detection along the path with greater slack to the primary output. This is not normally a problem for transition fault test generation, because the `-nopo_measures` option is commonly set for transition faults. If that option is set, then the fault cannot be detected at a primary output so the propagation along the minimum-slack path will continue uninterrupted.

Limitations in Support for TetraMAX Primitives

Full small delay defect testing support is not currently available for some selected TetraMAX primitive and some pins of another primitive. In all these cases, the limitations apply to test generation only. The computation of the detection slack and the slack delta is done accurately in all cases. The affected primitives are as follows:

- **Bus Drivers**

TetraMAX will not choose the minimum slack path when back-tracing through a bus driver if that path goes through the enable input. TetraMAX will always choose the path through the data input to the driver.

- **Memories**

TetraMAX does not currently back-trace through a memory or propagate a fault effect forward through a memory in small delay defect testing. There is no support for small delay defect testing for either general fast-sequential or full-sequential. No tests through memories can be produced in any event, so the absence of support for the memory primitive is not yet a crucial omission.

15

Path Delay Fault Testing

With the TetraMAX DSMTTest option you can use path delay fault testing to perform test generation to detect critical path delay faults. This option generates the most effective tests possible while providing the highest coverage of critical paths. TetraMAX also includes features to read, manage, and analyze paths from static timing analysis tools such as PrimeTime.

This chapter contains the following sections:

- [Path Delay Fault Theory](#)
- [Path Delay Testing Flow](#)
- [Obtaining Delay Paths](#)
- [Generating Path Delay Tests](#)
- [Untested Paths](#)

You will need a Test-Fault-Max license to use this feature. This license is also checked out if you read an image that was saved with the fault model set to path delay.

Path Delay Fault Theory

The single stuck-at fault model (stuck-at-0 or stuck-at-1) plays an important part in manufacturing test. However, you can achieve higher quality testing when you target other fault models, such as the path delay fault model, in addition to the single stuck-at model.

The path delay fault model is useful for testing and characterizing critical timing paths in your design. Path delay fault tests exercise the critical paths at-speed (the full operating speed of the chip) to detect whether the path is too slow because of manufacturing defects or variations.

Path delay fault testing targets physical defects that might affect distributed regions of a chip. For example, incorrect field oxide thicknesses could lead to slower signal propagation times, which could cause transitions along a critical path to arrive too late. By comparison, stuck-at, IDDQ, and transition delay faults are generally targeted at single-point defects.

Path delay faults are tested using the following sequence:

- The first vector initializes the path before applying the launch event, typically a clock pulse.
- The launch event generates the second vector, which propagates a logic transition along the entire path.
- A second clock pulse, occurring one at-speed cycle after the launch clock, captures the resulting transition at the end of the path.

Definitions

[Table 15-1](#) lists definitions for terms used in this chapter.

Table 15-1 Definitions of Terms

Terms	Definitions
at-speed clock	A pair of clock edges applied at the same effective cycle time as the full operating frequency of the device.
capture clock capture clock edge	The clock used to capture the final value resulting from the second vector at the tail of the path.
capture vector	The circuit state for the second of the two delay test vectors.
critical path	A path with little or no timing margin.

Table 15-1 Definitions of Terms (Continued)

Terms	Definitions
delay path	A circuit path from a launch node to a capture node through which logic transition is propagated. A delay path typically starts at either a primary input or a flip-flop output, and ends at either a primary output or a flip-flop input.
detection, robust (of a path delay fault)	A path delay fault detected by a pattern providing a robust test for the fault.
detection, non-robust (of a path delay fault)	A path delay fault detected by a pattern providing a non-robust test for the fault.
false path	A delay path that does not affect the functionality of the circuit, either because it is impossible to propagate a transition down the path (combinatorially false path) or because the design of the circuit does not make use of transitions down the path (functionally false path).
launch clock launch clock edge	The launch clock is the first clock pulse; the launch clock edge creates the state transition from the first vector to the second vector.
launch vector	The launch vector sets up the initial circuit state of the delay test.
off-path input	An input to a combinational gate that must be sensitized to allow a transition to flow along the circuit delay path.
on-path input	An input to a combinational gate along the circuit delay path through which a logic transition will flow. On-path inputs would typically be listed as nodes in the Path Delay definition file.
path	A series of combinational gates, where the output of one gate feeds the input of the next stage.
path delay fault	A circuit path that fails to transition in the required time period between the launch and capture clocks.
scan clock	The clock applied to shift scan chains. Typically, this clock is applied at a frequency slower than the functional speed.
test, non-robust	A pair of at-speed vectors that test a path delay fault; fault detection is not guaranteed, because it depends on other delays in the circuit.

Table 15-1 Definitions of Terms (Continued)

Terms	Definitions
test, robust	A pair of at-speed vectors that test a path delay fault independent of other delays or delay faults in the circuit.

Models for Manufacturing Tests

Path delay fault ATPG targets individual path delay faults and then simulates each test generated against the remaining undetected faults in the fault list using both robust and non-robust path delay fault models suitable for pass/fail manufacturing tests. By default, TetraMAX uses an auto-relaxation scheme that provides both efficient ATPG and the flexibility of multiple path delay fault models.

The manufacturing test off-path inputs of various gates, for both the on-path input rising and falling, are shown in the following:

```
set delay -nodiagnostic_propagation (default manufacturing tests)
```

Path Delay Fault Class	AND Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	X1	S1
Non-robust (DS)	X1	X1

Note:

X1 : initial state don't care; final state is 1

S1: steady 1 state (hazard-free)

Path Delay Fault Class	OR Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	S0	X0
Non-robust (DS)	X0	X0

Note:

X0 : initial state don't care; final state is 0

S0: steady 0 state (hazard-free)

Path Delay Fault Class	XOR Gate Off-path Inputs			
	Rising On-path Input (rising output)	Rising On-path Input (falling output)	Falling On-path Input (falling output)	Falling On-path Input (rising output)
Robust (DR)	S0	S1	S0	S1
Non-robust (DS)	X0	X1	X0	X1

Path Delay Fault Class	MUX Gate Select Off-path Inputs		
	Rising/Falling Data0 On-path Input	Rising/Falling Data1 On-path Input	
Robust (DR)	S0	S1	
Non-robust (DS)	X0	X1	

Path Delay Fault Class	MUX Gate Data Off-path Inputs			
	Rising Select On-path Input (rising output)	Rising Select On-path Input (falling output)	Falling Select On-path Input (falling output)	Falling Select On-path Input (rising output)
Robust (DR)	S0, X1	S1, X0	X0, S1	X1, S0
Non-robust (DS)	X0, X1	X1, X0	X0, X1	X1, X0

Models for Characterization Tests

TetraMAX can also generate single-path sensitization tests that have unambiguous diagnostic results. Such tests are useful to measure individual path delays on a physical device for design characterization purposes. With these tests, any failure can be directly related to a specific path delay fault. You can determine the maximum operating frequency of each testable critical path by varying the at-speed test cycle time and associating failures to the paths being tested.

The characterization test off-path inputs of various gates, for both the on-path input rising and falling, are shown in the following:

```
set delay -diagnostic_propagation (characterization tests)
```

Path Delay Fault Class	AND Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	S1	S1
Non-robust (DS)	11	11

Note:

11 : initial and final states are a 1

Path Delay Fault Class	OR Gate Off-path Inputs	
	Rising On-path Input	Falling On-path Input
Robust (DR)	S0	S0
Non-robust (DS)	00	00

Note:

00 : initial and final states are a 0

XOR Gate Off-path Inputs				
Path Delay Fault Class	Rising On-path Input (rising output)	Rising On-path Input (falling output)	Falling On-path Input (falling output)	Falling On-path Input (rising output)
Robust (DR)	S0	S1	S0	S1
Non-robust (DS)	00	11	00	11

MUX Gate Select Off-path Inputs		
Path Delay Fault Class	Rising/Falling Data0 On-path Input	Rising/Falling Data1 On-path Input
Robust (DR)	S0	S1
Non-robust (DS)	00	11

MUX Gate Data Off-path Inputs				
Path Delay Fault Class	Rising Select On-path Input (rising output)	Rising Select On-path Input (falling output)	Falling Select On-path Input (falling output)	Falling Select On-path Input (rising output)
Robust (DR)	S0, S1	S1, S0	S0, S1	S1, S0
Non-robust (DS)	00, 11	11, 00	00, 11	11, 00

Testing I/O Paths

You can also use TetraMAX to generate test patterns that exercise paths from an input pin to a flip-flop or from a flip-flop to an output pin. Unlike internal paths, physical at-speed testing of I/O paths generally requires the following:

- High-speed, high-bandwidth ATE equipment
- A low-skew test fixture
- Very accurate placement of input signal edges
- Very accurate placement of output strobe delays

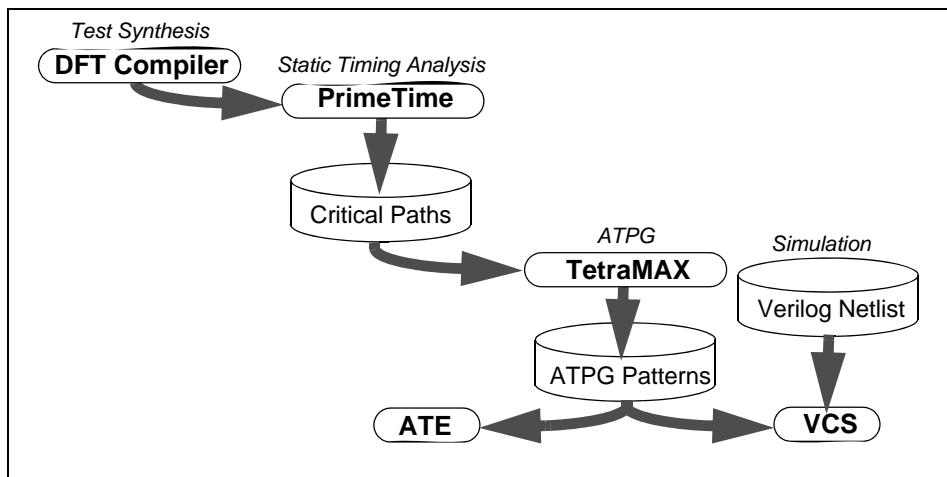
It is also important to be aware that the electrical environment of the test fixture may differ significantly from the system in which the device was designed to operate. Consequently, issues such as poorly terminated transmission lines and output driver simultaneous-switching current may cause excessive ringing on the input pins and additional delays on the output pins.

For these reasons, at-speed testing is not recommended for I/O paths unless ATE expertise exists for general high-speed testing issues and the electrical requirements for test fixtures are well understood in advance of their design.

Path Delay Testing Flow

PrimeTime generates the critical path information you need to input for a path delay ATPG test run as shown in [Figure 15-1](#).

Figure 15-1 Path Delay Test Flow

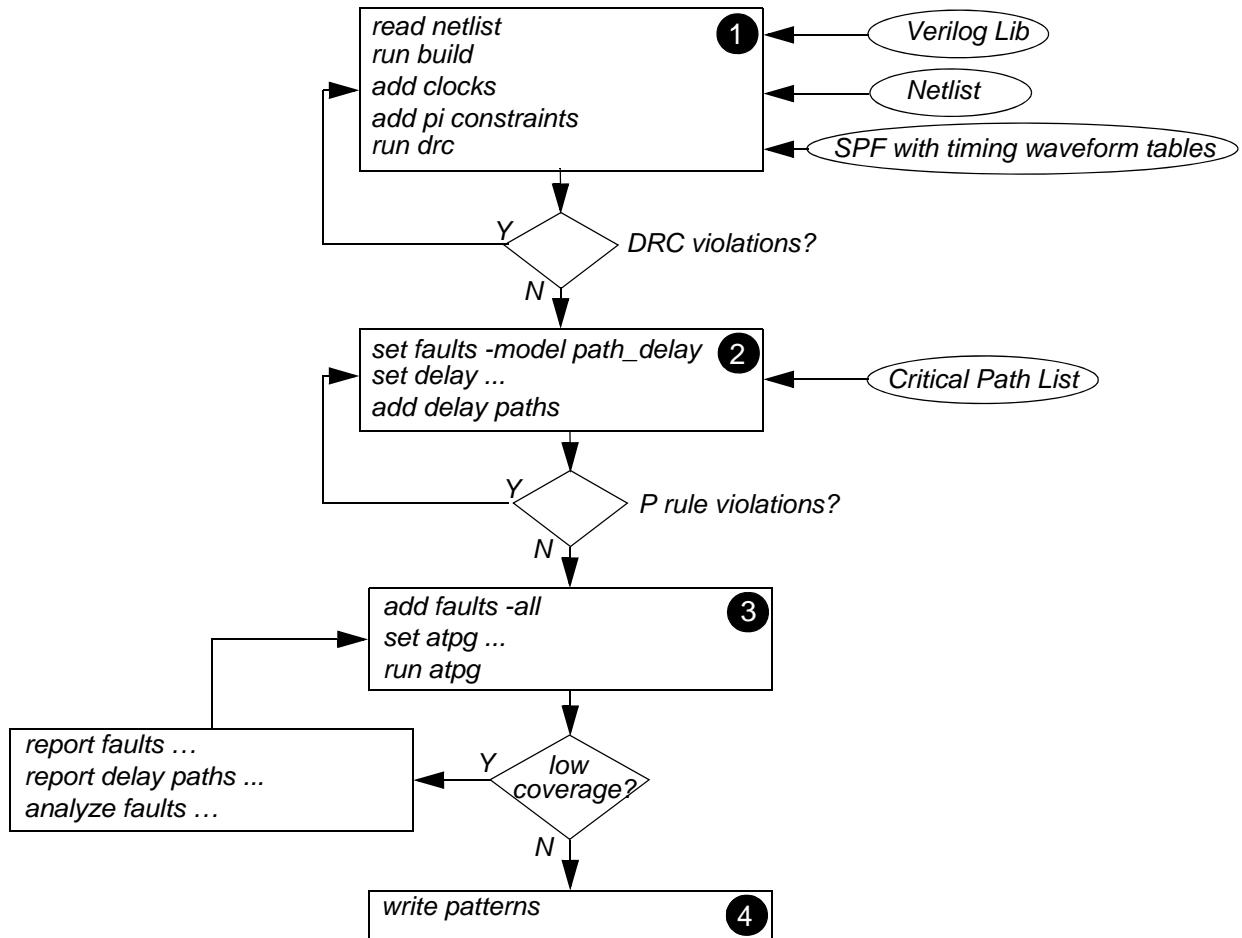


TetraMAX supports ATPG and fault simulation for scan-based path delay fault testing with the following features:

- Reads critical paths reported by PrimeTime
- Supports a comprehensive set of path (P) rules
- Most rule violations can be analyzed and debugged in the GSV
- Clock waveforms in the STIL protocol file are checked to ensure they match static timing analysis conditions
- Identifies combinational false paths and other untestable paths
- Generates a full range of tests supporting both robust and non-robust path delay fault models

[Figure 15-2](#) shows the basic TetraMAX steps and checkpoints to generate an effective set of path delay tests.

Figure 15-2 Path Delay Test Generation Flowchart



Launch and capture events are pertinent only to transition and path delay fault environments. If the fault model is set to the default model (stuck), then the launch and capture events are likely to be dropped. TetraMAX will attempt to maintain this information when possible. However, because of the variety of flows and the ability to process patterns generated for one fault model under a different model (for instance, regrading transition patterns under a stuck model), care must be exercised if this information needs to be maintained. Before the write pattern operation is executed in the file that reads-back the binary patterns, add the set faults -model transition command. Then, the launch and capture events will remain across all outputs.

Obtaining Delay Paths

TetraMAX requires a list of critical paths to target for path delay fault test generation. TetraMAX can read an ASCII file containing the critical paths reported by a static timing analysis tool, such as PrimeTime, or you can specify these paths manually in an ASCII file.

Importing PrimeTime Path Lists

The pt2tmax.tcl file included with the TetraMAX software consists of a Tcl procedure (`write_delay_paths`) for both internal and I/O path selection. This Tcl procedure generates a list of critical paths in the required DSMTTest format according to the criteria you specify.

Note:

Set the case analysis in PrimeTime to correspond with the device in test mode and operating on the tester.

The syntax for `write_delay_paths` is

```
write_delay_paths -max_paths num_paths
[-slack crit_time] [-nworst num_per]
[-clock clock_name | -launch clock_name |
 -capture clock_name] [-IO [-each]]
[-group group_name] [-version] filename
```

Argument	Definition
<code>-capture clock_name</code>	Selects paths ending at <code>clock_name</code> domain
<code>-clock clock_name</code>	Selects paths in <code>clock_name</code> domain
<code>-each</code>	Selects paths for each I/O
<code>filename</code>	Name of file where paths are written
<code>-group group_name</code>	Selects paths from existing <code>group_name</code>
<code>-IO</code>	Writes I/O paths instead. The default is to only write internal paths
<code>-launch clock_name</code>	Selects paths starting from <code>clock_name</code> domain
<code>-max_paths num_paths</code>	Specifies the maximum number of paths to be written. The default is 1
<code>-nworst num_per</code>	Specifies the number of paths to each endpoint. The default is 1

Argument	Definition
<code>-slack crit_time</code>	Writes paths with slack less than the specified <i>crit_time</i> . The default is infinite
<code>-version</code>	Reports version number

The pt2tmax.tcl file, found under \$SYNOPSYS/auxx/syn/tmax, must first be sourced:

```
pt_shell> source pt2tmax.tcl
```

To select a set of target critical paths, use the write delay paths command:

```
pt_shell> write_delay_paths -slack 6 -max_paths 100 \
    paths.import
```

Path Definition Syntax

The following syntax is used to define critical delay paths. Keywords are shown in bold and arguments are shown in italics. Brackets ([]) enclose optional blocks, and a vertical bar (|) indicates that one of several fields must be specified.

```
$path {
    [ $name path_name ; ]
    [ $cycle required_time ; ]
    [ $slack slack_time ; ]
    [ $launch clock_name ; ]
    [ $capture clock_name ; ]
    $transition {
        pin_name1 ^ | v | = | ! ;
        pin_name2 ^ | v | = | ! ;
        ...
        pin_nameN ^ | v | = | ! ;
    }
    ]+
}
[ $condition {
    pin_name1 0 | 1 | Z | 00 | 11 | ZZ | ^ | v ;
    pin_name2 0 | 1 | Z | 00 | 11 | ZZ | ^ | v ;
    ...
    pin_nameN 0 | 1 | Z | 00 | 11 | ZZ | ^ | v ;
}
}
```

Where:

- \$name - Assigns a name to the delay path
- \$cycle - Time between launch and capture clock edges

- \$slack - Available time margin between the \$cycle time and calculated delay of the path
- \$launch - Launch clock primary input to be used
- \$capture - Capture clock primary input to be used
- \$transition - (Required) Describes the expected transitions of path_startpoint, output pins of path cells, and path_endpoint.
- \$condition - (Optional) allows the user to add more constraint for testing the associated path.
- Argument signal notation:
 - V - falling transition
 - ^ - rising transition
 - = - transition same as previous node
 - ! - transition inverted with respect to previous node
 - 0 - node must be set to "0" during V2
 - 1 - node must be set to "1" during V2
 - Z - node must be set to "Z" during V2
 - 00 - node must be set to "0" during V1 and remain during V2
 - 11 - node must be set to "1" during V1 and remain during V2
 - ZZ - node must be set to "Z" during V1 and remain during V2

The following example of a path definition file shows two path delay faults that might be created manually or by a third-party timing analysis tool:

```
$path {
    $name path_1 ;
    $transition {
        P201/C4/DESCTL/EN_REG/Q      ^ ;
        P201/C4/DESCTL/C0/U62/CO     ^ ;
        P201/C4/DESCTL/C0/U66/X      v ;
        P201/C4/DESCTL/C0/Q2_REG/D   v ;
    }
}
$path {
    $name path_2 ;
    $transition {
        . ;
        . ;
        . ;
    }
}
```

Translating Timing Exceptions

If you have `set_false_path` or `set_multicycle_path` exceptions defined in PrimeTime, you can translate them into TetraMAX commands with `write_timing_exceptions` utility. This utility creates a file called "tmax_exceptions.cmd" will be created that you can source into TetraMAX. For details, see section "["Translating PrimeTime Timing Exceptions" on page D-13.](#)

Generating Path Delay Tests

Note:

Many of the commands shown have other options that may be useful to adjust TetraMAX to your unique requirements. The online help describes these options in complete detail.

To generate a set of path delay tests,

1. Start TetraMAX, as you would for any fault model.
2. Read in libraries and netlists.
3. Build a circuit model.
4. Run DRC (possibly with delay waveform tables in the STIL protocol file (SPF)):

```
run drc filename.spf
```

5. Set desired delay testing options (depending on ATE functionality):

```
set delay -NOPI_changes  
set delay -NOPO_measures
```

6. Read in delay paths:

```
add delay paths filename
```

Analyze any P-rule errors or warnings.

To remove any of paths you have read in:

```
remove delay paths pathname
```

7. (Optional) Display a delay path by issuing the command:

```
report delay paths path_name -Display -Pindata
```

8. Add path delay faults by issuing these commands:

```
set faults -Model Path_delay
```

```
add faults -All
```

9. Set desired ATPG options:

```
set atpg -full_seq_merge medium
```

10.Run ATPG:

```
run atpg
```

11.(Optional) Analyze low path delay coverage:

```
report faults -Class AU  
analyze faults path_name -slow rise -display \  
-verbose -fault_simulation
```

12.Write out path delay test patterns:

```
write patterns pathname.stil -format stil  
write patterns pathname.wgl -format wgl
```

Set Delay Options

Once DRC has passed and before reading in your list of critical paths, specify any options related to path delay testing. You can do this by entering the `set delay` command at the command line. For the complete syntax and option descriptions, see the online help for the `set delay` command.

Note that the launch cycle setting has no effect on path delay fault ATPG. By default, ATPG can use a last-shift or a system clock for the launch cycle. To prevent last-shift launch behavior, constrain the scan enable signal to its inactive value using the `add pi` constraint command.

Reading and Reporting Path Lists

After you have set delay options set using the `set delay` command, you can read a set or sets of delay faults into TetraMAX using the `add delay paths` command. This command reads in a path delay definition file. See “[Obtaining Delay Paths](#)” on page 15-9. You can remove paths from memory with the `remove delay paths` command. To display paths in text format, use the `report delay paths` command. By using the `-verbose` option, you can include in the report information regarding launch and capture clocks and nodes, transition direction of faults, fault status, and the vector in which detection took place.

Analyzing Path Rule Violations

Like other warnings and errors reported by TetraMAX, many of the P-rule violations can be analyzed using the GSV. Use the online help topics for further debugging strategies. For example, to find out more information on P20 violations that were flagged, enter:

```
report violations P20
analyze violations P20-3
```

Viewing Delay Paths

You can view delay paths using the GSV. The command for doing this is `report delay paths path_name -display -pindata`. This displays the named path in the GSV with any path requirements (transitions and conditions) annotated to the wires of the design or primitive elements in the path.

Path Delay ATPG Options

Prior to ATPG, you can set options to improve vector generation or pattern compression. These options include

- `set atpg -FULL_SEQ_ABORT_limit seq_max_remade_decs`
- `set atpg -FULL_SEQ_Time max_secs_per_fault`
- `set atpg -FULL_SEQ_Merge [low | medium | high]`

If the fault report printed after ATPG indicates that some faults were aborted (undetected), you may want to increase the time limit above 10 seconds (the default) and rerun ATPG on the remaining faults. Raising the merge effort will allow TetraMAX to generate fewer vectors for the same fault coverage. Note that the default is to not merge patterns.

Internal Loopback and False/Multicycle Paths

You can generate transition and path delay tests while ensuring that you will not get tests that "loopback" through a bidirectional port or tests for false/multicycle paths that begin at a specific start point. The following six commands implement this capability:

- `add slow bidis port_name | -all>`
- `remove slow bidis port_name | -all>`
- `report slow bidis`
- `add slow cells instance_path | gate_id`

- `remove slow cells instance_path | gate_id | -all`
- `report slow cells`

The `add slow bidis` command modifies the associated BUS primitives to output an X if any tristate driver (TSD) or switch (SW) primitives are not driving a Z onto the BUS primitive. The value observed on the primary inout (PIO) primitive continues to be the resolved value of the BUS primitive prior to this masking operation. If all TSD and SW primitives are driving a Z onto the BUS primitive, the BUS behavior is not modified. This includes the behavior if the PIO primitive is also driving a Z, or if there are weak input values.

An error message is issued if the `add slow bidis` command is specified for a port that is not an inout or does not exist. The `add slow bidis -all` command issues a message showing the number of ports modified.

The `add slow cells` command modifies the simulation behavior of DFF or DLAT cells in two ways:

- For Basic-Scan patterns, the DFF/DLAT gets loaded with an X if the adjacent scan cell (closer to the scan out) is being loaded with a different value (that is, if the last scan shift creates a transition on the DFF/DLAT output). The capture and unload behavior of the DFF/DLAT is not modified. When setting a scan cell value with this attribute, Basic-Scan ATPG also attempts to set the adjacent scan cell with this same value prior to pattern merging, if it has not already been set.
- For Fast-Sequential and Full-Sequential patterns, the DFF/DLAT outputs an X if data captured by a clock changes the state of the DFF/DLAT, or if a set/reset changes the state of the DFF/DLAT. The DFF or DLAT continues to output an X until the next load operation. However, the capture and internal state behavior is not modified and this internal state value, not an X, will be observed by an unload operation. Full-sequential ATPG will continue to apply the “robust fill” algorithm prior to random fill. This decreases the probability that the launch clock will create a transition from scan cells feeding off-path inputs, including any with this attribute.

See online Help for additional information about these commands.

You can use the `write_timing_exceptions` utility to translate timing exceptions from PrimeTime to TetraMAX. For details, see “[Translating PrimeTime Timing Exceptions](#)” on page [D-13](#).

Creating DSMTTest WaveformTables

Path delay tests are generated during the Full-Sequential test mode. These tests conform to user constraints through defined clocks and specified primary input constraints. The timing for these vectors adhere to one of several timing WaveformTables in the SPF.

If there are no additional waveform tables in the SPF, then the default timing (_default_WFT_) will be used for all path delay test vectors. However, special timing can be defined for the launch and capture events in ancillary timing waveform tables. These tables are called:

- _launch_WFT_
- _capture_WFT_
- _launch_capture_WFT_

Each of these tables can have different timing defined for inputs, clocks and output strobes. The path delay test vectors can use these timing definitions when applied to the device under test to detect faults defined in the path definition file.

The following is an example of a “_capture_WFT_” timing WaveformTable in the context of a STIL protocol file:

```
Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "TxClk" { 01Z { '0ns' D/U/Z; } }
            "TxClk" { P { '0ns' D; '50ns' U; '80ns' D; } }
            "_default_In_Timing_" "{01ZN {'0ns' D/U/Z/N; } }"
            "_default_Out_Timing_" "{X {'0ns' X; } }"
            "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
        }
    }
    WaveformTable "_capture_WFT_" {
        Period '20ns';
        Waveforms {
            "TxClk" { 01Z { '0ns' D/U/Z; } }
            "TxClk" { P { '0ns' D; '5ns' U; '10ns' D; } }
            "_default_In_Timing_" "{01ZN {'0ns' D/U/Z/N; } }"
            "_default_Out_Timing_" "{X {'0ns' X; } }"
            "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
        }
    }
}
```

A path delay test cycle uses the same order of events as for other fault models:

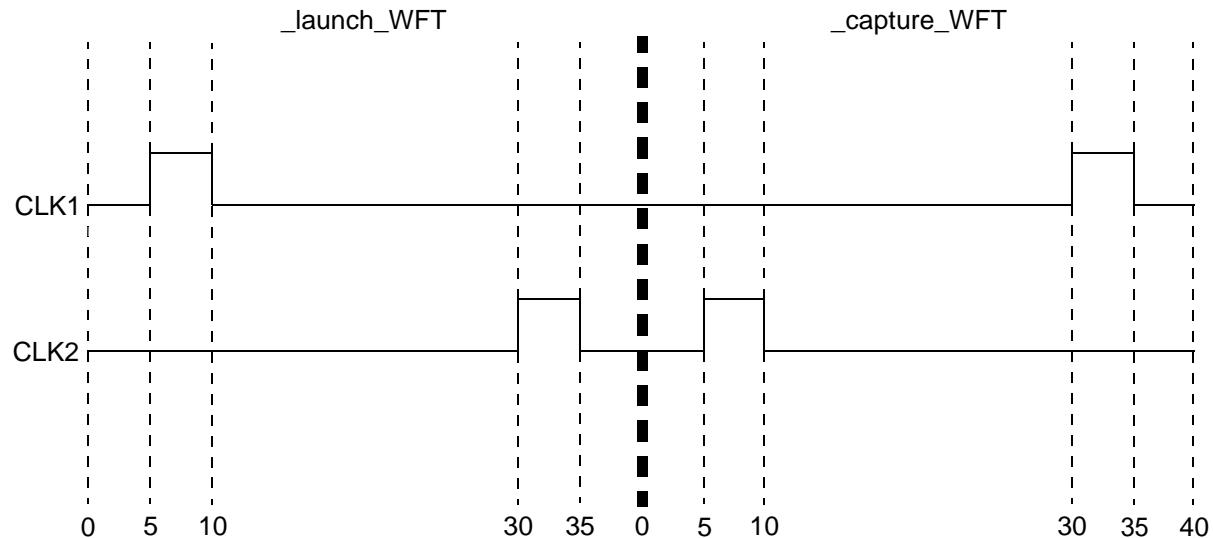
- Force primary inputs
- Measure primary outputs (optional)
- Pulse a clock

Given this order of events, one or two test cycles are required to launch and capture a path delay fault. For most paths, a two-cycle test is generated to apply a launch clock pulse and a capture clock pulse. However, a path delay fault test requiring a launch on the rising (leading) edge of a clock and a capture on the falling (trailing) edge of the same clock will generate a one-cycle test that uses the “_launch_capture_WFT_”. For a delay path fault test that requires a launch in one clock domain and a capture in another clock domain, two vectors are generated, and thus use “_launch_WFT_” for the launch vectors, and “_capture_WFT_” for the capturing vector.

If two or more different at-speed frequencies need to be used for different clock domains within your design, you might consider the following example WaveformTable definition. This example shows two input clocks with their launch and capture timing defined (see [Figure 15-3](#)).

```
WaveformTable "_launch_WFT_" {
    Period '40ns';
    Waveforms {
        "CLK1" { 01Z { '0ns' D/U/Z; } }
        "CLK1" { P { '0ns' D; '5ns' U; '10ns' D; } }
        "CLK2" { 01Z { '0ns' D/U/Z; } }
        "CLK2" { P { '0ns' D; '30ns' U; '35ns' D; } }
        "_default_In_Timing_" {01ZN {'0ns' D/U/Z/N; } }
        "_default_Out_Timing_" {X {'0ns' X; } }
        "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
    }
}
WaveformTable "_capture_WFT_" {
    Period '40ns';
    Waveforms {
        "CLK1" { 01Z { '0ns' D/U/Z; } }
        "CLK1" { P { '0ns' D; '30ns' U; '35ns' D; } }
        "CLK2" { 01Z { '0ns' D/U/Z; } }
        "CLK2" { P { '0ns' D; '5ns' U; '10ns' D; } }
        "_default_In_Timing_" {01ZN {'0ns' D/U/Z/N; } }
        "_default_Out_Timing_" {X {'0ns' X; } }
        "_default_Out_Timing_" { HLT { '0ns' X; '4ns' H/L/T; } }
    }
}
```

Figure 15-3 Two Different At-Speed Times



If a specific sequence of vectors should be applied to your design for path delay test generation, then a `sequential_capture` procedure can be used. The `sequential_capture` procedure is described in [Chapter 9, “STIL Procedure Files.”](#) To select this procedure, use the `set drc -clock -seq_capture` command before running DRC.

Maintaining DSMTTest Waveform Table Information

The presence of launch and capture operations is pertinent only under transition and path delay environments. To ensure this information remains in a pattern set through various flows, such as importing patterns into TetraMAX (see [“Selecting the Pattern Source” on page 4-21](#)), specify the appropriate fault model for these patterns. See [“Specifying Transition Delay Faults” on page 14-9](#) for transition patterns, or [“Generating Path Delay Tests” on page 15-12](#) for the appropriate `set faults -model` command.

Limitations of Waveform Table Support for Path Delay Patterns

The Verilog testbench written out by TetraMAX only supports a single period value for all cycle operations. This implies that a single waveform table can be taken into account when writing out the testbench. The delay waveform tables are not supported with the Verilog testbench. The flow is to write out the STIL vectors and then use the Verilog DPV PLI with VCS. Refer to the *Test Pattern Validation User Guide*.

MUXClock Support for Path Delay Patterns

Testing of internal paths in DSMTTest requires that the system clock be applied at-speed to the device under test. MUXClock, a common technique for applying the system clock at-speed, merges (or multiplexes) two patterns within a single, uniform cycle to create the at-speed clock.

For MUXClock vector formatting, two additional clock waveforms D (double) and E (early) need to be defined for the at-speed test. Definitions of the waveforms used during scan chain shifting and normal (slow) system cycles are contained in the SPF file.

MUXClock is a single waveform table/timeset construct that eliminates switching waveform tables between the default path delay waveform tables for launch, capture, and launch_capture operations and reduces requirements on ATE to support this timing flexibility.

By overlapping both the launch and capture events in one tester cycle, it is possible for ATE timing accuracy to be higher than across multiple vectors. Also, it is possible to place the launch and capture events closer together in a single vector than normally permitted when separate vectors were required. This feature, however, requires testers to support flexible double-pulse definitions in STIL, and relies on MUX constructs in WGL that tie multiple tester channels together to generate a flexible double-pulse waveform.

The following formats are supported by the MUXClock technique:

- WGL (using the WGL ":mux" construct)
- STIL (using multiple pulsed waveforms P, E, and D)
- MUXClock is not supported with clock_grouping

Enabling MUXClock Functionality

The waveform table sections in the SPF need to be modified to support MUXClock behavior for delay test vectors. The typical waveform table section specifies values that are applied during the scan shift and normal system tester cycles. Two additional waveform definitions are required to specify the at-speed clock.

Delay Test Vector Format

Here is an example WaveformTable section for the MUXClock technique:

```

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "all_inputs" { 0 { '0ns' D; } }
            "all_inputs" { 1 { '0ns' U; } }
            "all_inputs" { Z { '0ns' Z; } }
            "all_outputs" { X { '0ns' X; } }
            "all_outputs" { H { '0ns' X; '40ns' H; } }
            "all_outputs" { T { '0ns' X; '40ns' T; } }
            "all_outputs" { L { '0ns' X; '40ns' L; } }
            "CK" { P { '0ns' D; '75ns' U; '85ns' D; } }
            "CK" { D { '0ns' D; '45ns' U; '55ns' D; '75ns' U;
'85ns' D; } }
            "CK" { E { '0ns' D; '45ns' U; '55ns' D; } }
        }
    }
}

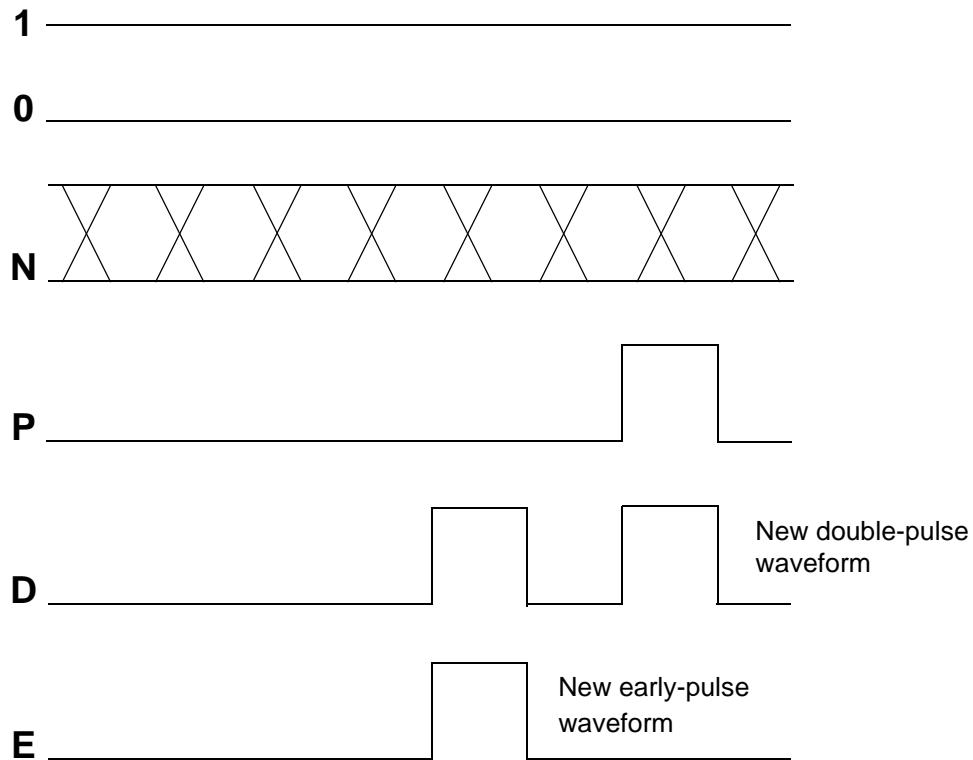
```

In the waveform table, all signals identified as clocks in the design must have two additional waveforms present. These waveforms use the WaveformCharacters D (double-pulse), and E (early-pulse). This brings the count of pulsed-waveforms for clocks up to 3: P, D, and E. These pulses have the following requirements:

- The edges of the E pulse must align with the edges of the first D pulse
- The edges of the P definition must align with the edges of the second D pulse

Also, the timing of all pulses, including the E pulse, must occur after the timing of the input edges and the output measures. In MUXClock mode, all path delay launch and capture operations are performed in a single cycle (described next); therefore, the timing of all events must follow the forcePI/measurePO/clock-pulse sequence. Because there is only one cycle, an option to define multiple cycles does not exist. Visually, the set of waveforms for an active-high clock to define an MUXClock operation appear similar to that shown in [Figure 15-4](#).

Figure 15-4 MUXClock: Active-High Clock Waveforms



The D and E waveforms must be present for all clocks in the design and in all waveform tables. If a waveform is missing, a warning is generated and the behavior reverts to generating separate launch, capture, and launch_capture WaveformTables. If the D and E waveforms are not defined, path delay operations are supported with multiple WaveformTable definitions as is done today.

When these definitions are present in the WaveformTable and when path delay faults are enabled, the tests will use the E waveform timing if only a launch clock pulse is required, and the P waveform timing if only a capture clock pulse is required. If the same clock is used for both launch and capture, the vector will use the D waveform. These STIL waveforms will be converted into the WGL MUX constructs necessary to represent these behaviors.

When MUXClock waveforms have been defined, the WGL output will contain references to two WGL muxparts for each clock signal in the design. An example of this construct is shown below for the WGL signals and timeplate sections.

```

"CK" [ "CK_Epulse", "CK_Ppulse" ] :mux input;
-
-
-
timeplate "_default_WFT_" period 100ns
  "CK_Ppulse" := input [0ps:D, 75ns:S, 85ns:D];
  "CK_Epulse" := input [0ps:D, 45ns:S, 55ns:D];
-
-
-

```

Limitations of MUXClock Support for Path Delay Patterns

The following limitations apply to MUXClock support for path delay patterns:

- Output pattern files containing MUXClock waveforms are not yet readable in TetraMAX.
- Bidirectional clocks in a design are not supported in WGL output when MUXClock definitions are present. STIL output supports bidirectional clocks in the design.

ATPG Requirements to Support MUXClock

For MUXClock to function, the `set delay` command options `-nopi_changes` and `-nopo_measures` must be used. In MUXClock mode, there can be no change of PI state or detectable PO information between the end-of-launch and the start-of-capture: the only event that can happen in the capture operation is a clock pulse.

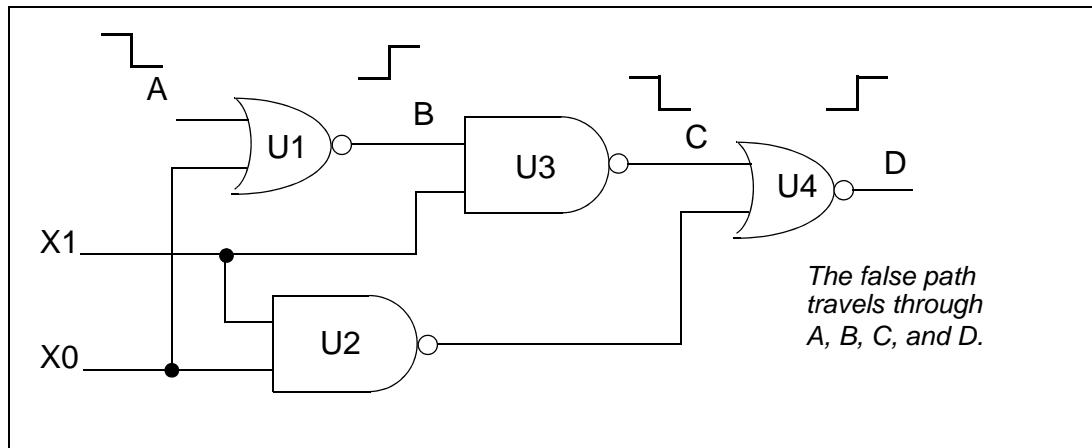
Untested Paths

The following paragraphs explain false and untestable paths, and describe how you might handle them.

False Paths

A false path may be caused by a portion of combinational logic that is configured so a path can never be fully exercised. In other words, the path can never propagate a high-to-low or low-to-high transition from the startpoint to the endpoint. [Figure 15-5](#) illustrates a false path, ABCD.

Figure 15-5 False Path Example



The transition cannot be propagated to output D because of a blockage created by the X0 pin driving U2 and subsequently U4. TetraMAX identifies组合ally false paths when reading paths and classifies the associated path delay fault as undetectable-redundant (UR). False paths will also be flagged with a P21 rule violation (on-path values not satisfiable).

Note:

PrimeTime has an optional ability to identify and eliminate组合ally false paths. However, TetraMAX appropriately handles组合ally false or sequentially false paths that static analysis tools might not identify.

Untestable Paths

TetraMAX DSMTest may prove a path delay fault to be untestable for one of the following reasons:

- It is a sequentially false path. Such paths cannot be tested in a functional mode, because logic prevents the required state transitions.
- ATPG constraints or tester limitations may prevent some true paths from being tested. DSMTest restrictions must adhere to all ATPG constraints.
- Redundant logic (for circuit speed) prevents a single path from being independently tested. Multiplier arrays are a good example of such circuits.

Note:

If there are reconverging paths, you may want to use the `-allow_reconverging_paths` option to the `set_delay` command. The default is `-noallow_reconverging_paths`.

- Paths that require multiple launch or capture events are not usually supported by TetraMAX and are declared untestable.

Note:

Multicycle paths can often be tested if the appropriate clock timing is applied.

- Other TetraMAX restrictions may cause paths to be declared untestable. These paths are usually flagged with a P-rule violation.
- Paths through RAMs or ROMs modeled with memory primitives are not supported by TetraMAX and are declared untestable.

Reporting Untestable Paths

A specific path delay fault may not be testable due to either a path rule violation or a failure of path delay ATPG to find a test for the path. You can generate a list of P-rule violations using the `report violations P` command. For analyzing undetectable and untestable paths, check the results of rules P19, P20, P21, P22, P23, and P24.

To review untestable paths after ATPG:

```
TEST> report faults -class AU
str AN path8
str AN path9
```

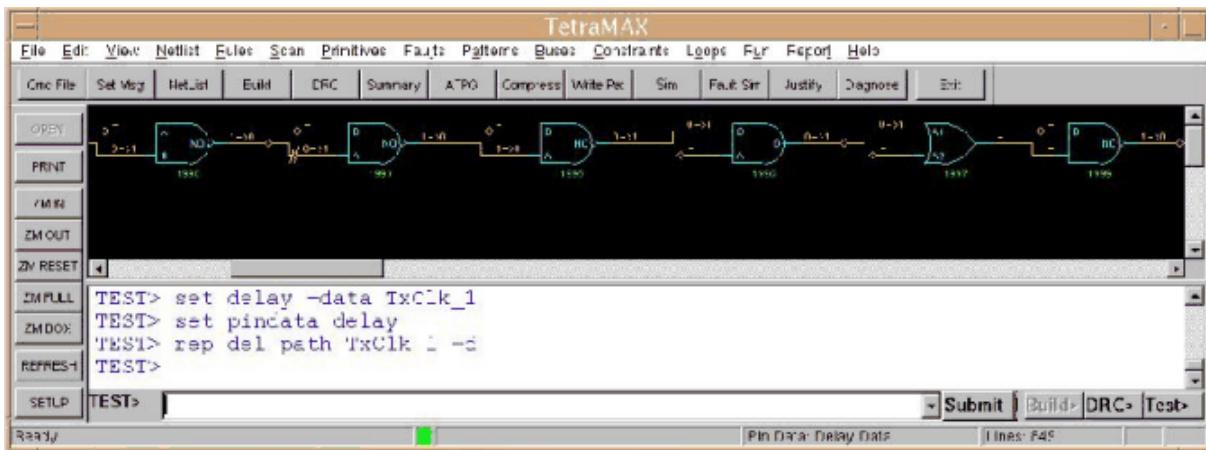
To display the delay for a particular path use this command:

```
TEST> report delay path path_name -verbose -display
```

Adding the `-display` option to the command displays the path in the GSV, where you can annotate ports with delay path data by selecting delay data from the pindata list in the Setup dialog box or by including the `-pindata` option.

[Figure 15-6](#) shows a path being displayed in the GSV.

Figure 15-6 Path Display Example

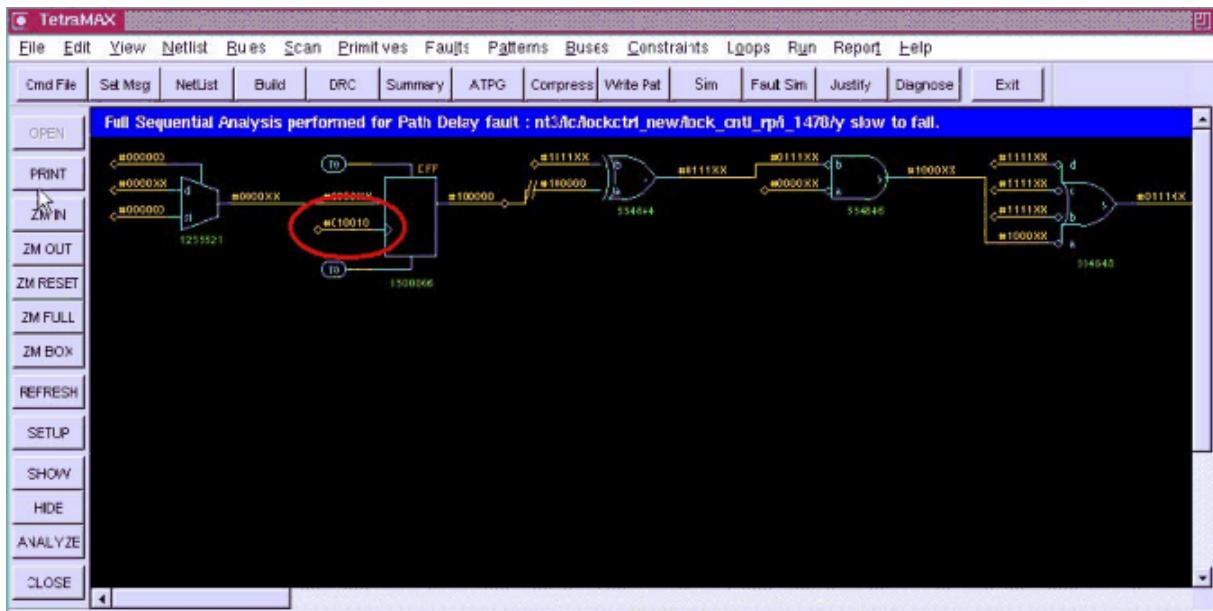


Analyzing Untestable Faults

If a fault has been classified as ATPG Untestable, you can use the `analyze faults path_name -slow r | f` command to display the path in the GSV so you can analyze it. For example, [Figure 15-7](#) shows the displayed result of entering

```
analyze faults CLK_0 -slow f -display
```

Figure 15-7 Analyze Untestable Faults Display



TetraMAX performs fault analysis for the specified path. An incremental approach to test generation is pursued in which the sensitization requirements for the path are attempted one node at a time, until a path node is added that causes a justification failure or a test is generated. With the -display option, pattern values for the last successful justification are shown in the GSV for the specified path.

TetraMAX Commands for Path Delay Fault Testing Example

In this example, the scan enable signal is constrained as in the transition fault testing using system clock launch. However, this step is not needed if the circuit can support last shift launch.

This example also uses commands specific to path delay testing such as

- The `set delay -mask_nontarget_paths` command, which ensures that TetraMAX does not generate expected values on multi-cycle or false paths
- The `set delay -relative_edge` command, which causes TetraMAX to inject both a slow-to-rise and a slow-to-fall fault for each path when you run the `add faults -all` command

The following example also shows the pattern reporting commands that are unique to path delay testing:

```

read netlist ckt.v

run build test_ckt

set delay -nopi_changes -nopo_measures // if needed
set delay -mask_nontarget_paths
set delay -common_launch_capture_clock // if needed
set delay -relative_edge // if desired

add capture mask dff0 // if needed
add slow cell dff1 // if needed
add slow bidi -all

add pi constraint 0 scan_enable

run drc ckt.spf

add delay path ckt.paths

set fault -model path
add fault -all

run atpg -auto

report patterns -all -path_delay // if desired
report patterns -all -slack // if desired

// You can optionally run the following command
analyze fault path0 -slow r -verbose -display -fault_sim

```


16

Quiescence Test Pattern Generation

TetraMAX allows you to generate test patterns specifically targeted for quiescence, or IDDQ, testing. You can also verify IDDQ test patterns and choose IDDQ strobe points in existing patterns for maximum fault coverage.

This chapter contains the following sections:

- [Why Do IDDQ Testing?](#)
- [About IDDQ Pattern Generation](#)
- [Limitations](#)
- [Fault Models](#)
- [DRC Rule Violations](#)
- [Generating IDDQ Test Patterns](#)
- [IDDQ Commands](#)
- [Design Principles for IDDQ Testability](#)

Why Do IDDQ Testing?

IDDQ testing can detect certain types of circuit faults in CMOS circuits that are difficult or impossible to detect by other methods. IDDQ testing, when used to supplement standard functional or scan testing, provides an additional measure of quality assurance against defective devices.

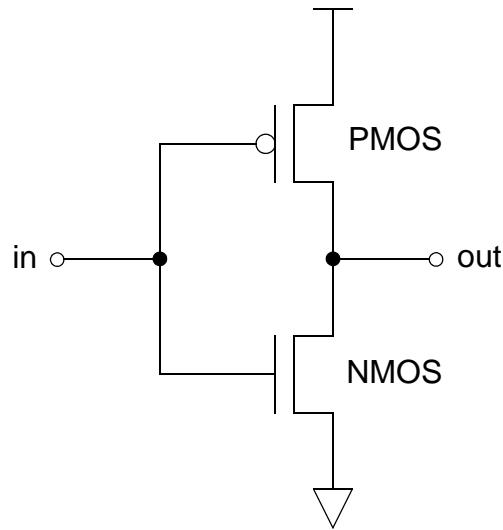
IDDQ testing detects circuit faults by measuring the amount of current drawn by a CMOS device in the quiescent state (a value commonly called “ I_{ddQ} ”). If the circuit has been designed correctly, this amount of current is extremely small. A significant amount of current indicates the presence of one or more defects in the device.

CMOS Circuit Characteristics

An important characteristic of CMOS circuits is that they draw almost no current in the quiescent state. “Quiescent” means that the inputs are stable and the circuit is inactive. System designers sometimes take advantage of this characteristic by having a power down or sleep mode in which the device stops operating, but retains its internal state and memory contents, thus conserving battery charge while the device is idle.

[Figure 16-1](#) shows a schematic diagram of a typical CMOS inverter. The inverter has two MOS transistors, one NMOS and the other PMOS. The two transistor gates are tied together to make the inverter input, and the two drains are tied together to make the inverter output.

Figure 16-1 CMOS Inverter Schematic Diagram



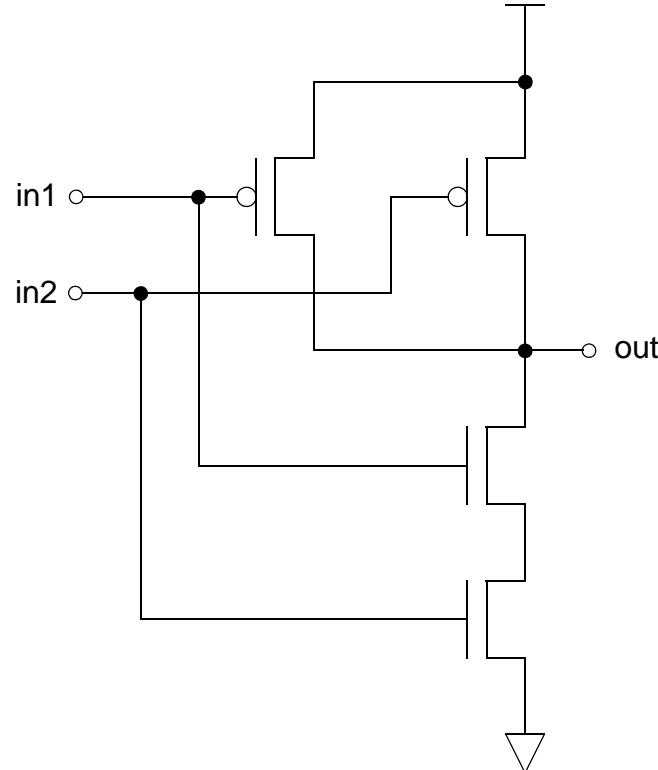
When the input is low, the upper transistor is on and the lower transistor is off, which pulls the output up to the supply voltage (VDD). When the input is high, the upper transistor is off and the lower transistor is on, which pulls the output to ground.

During a logic transition, a significant amount of current can flow while the capacitive load on the output node is charged up to VDD or discharged to ground. However, in the quiescent state, the only current that flows is the very small leakage current through the transistor that is off.

To ensure that no current flows in the quiescent state, every node must be pulled either low or high, and not allowed to float. For example, if the input of the inverter is allowed to float, the voltage could drift to an intermediate value, putting both transistors into a partially on state. This would allow a steady-state current to flow from VDD through the two transistors to ground.

A logical NAND gate uses multiple PMOS transistors in parallel at the top and multiple NMOS transistors in series at the bottom, as shown in [Figure 16-2](#). For each combination of input values, the power supply current is extremely small in the quiescent state because the path from VDD to ground is blocked by at least one off transistor.

Figure 16-2 CMOS NAND Gate Schematic Diagram

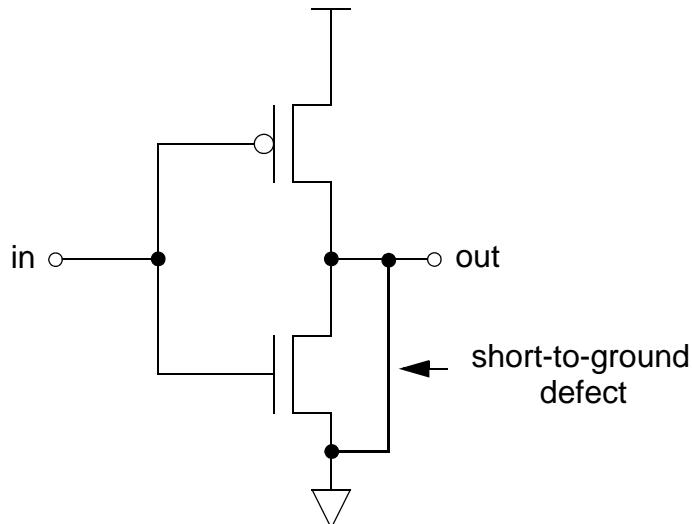


IDDQ Testing Methodology

IDDQ testing is different from traditional circuit testing methods such as functional or stuck-at testing. Instead of looking at the logical behavior of the device, IDDQ testing checks the integrity of the nodes in the design. It does this by measuring the current drain of the whole chip at times when the circuit is quiescent. Even a single defective node can easily cause a measurable amount of excessive current drain. In order to place the circuit into a known state, the IDDQ test sequence uses ATPG techniques to scan in data, but it does not scan out any data.

For example, consider the short-to-ground defect shown in [Figure 16-3](#). Depending on the controllability and observability characteristics of the defective node, this defect might be detectable as a stuck-at-0 fault using functional or scan testing.

Figure 16-3 Short-to-Ground Defect



With IDDQ testing, this defect can be detected even if the node is not observable. You only need to maintain the input of the inverter at logic 0, which turns on the upper transistor and places the output of the inverter at logic 1.

It is normal for current to flow during switching, but after the device has settled for a period of time, no more current should flow. At this point, an IDDQ strobe detects the excessive current drain through the upper transistor and the short to ground. The current drain of a single defect such as this can be orders of magnitude larger than the normal current drain of the entire device in the quiescent state.

Similarly, an IDDQ strobe can detect a short to VDD. For example, in the inverter circuit shown in [Figure 16-3](#), you only need to maintain the input of the inverter at logic 1, which turns on the lower transistor and places the output of the inverter at logic 0. After the device has settled, an IDDQ strobe detects the current drain through the short from VDD to the node and the lower transistor.

Types of Defects Detected

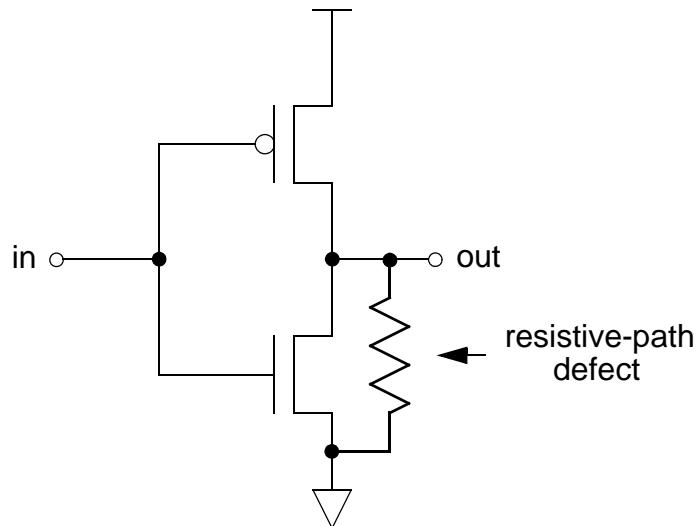
IDDQ testing can detect many kinds of circuit defects that are difficult or impossible to detect by functional or stuck-at testing, such as three-state enable nodes, redundant logic, high-resistance faults, scan chain control/data paths, undetectable faults, possibly detected faults, ATPG untestable faults, and bridging faults.

For example, consider the defect shown in [Figure 16-4](#), a resistive path to ground. This node might pass initial stuck-at testing, but fail after burn-in or during actual use by the customer. IDDQ testing can immediately detect this type of fault due to the excessive current drain when the node is at logic 1, even if the node is not observable by stuck-at testing.

IDDQ testing can partially or completely replace costly burn-in testing. Burn-in means testing the device using functional or scan testing, operating the device for a period of time under normal conditions, and then running the same tests to find any early failures in the lifetime of the device. IDDQ testing can detect many burn-in type defects.

IDDQ testing can also detect bridging faults. A bridging fault is a short between two different functional nodes in the design. An IDDQ strobe detects a fault of this type if one node is at logic 0 while the other is at logic 1.

Figure 16-4 Resistive Path to Ground



Number of IDDQ Strobes

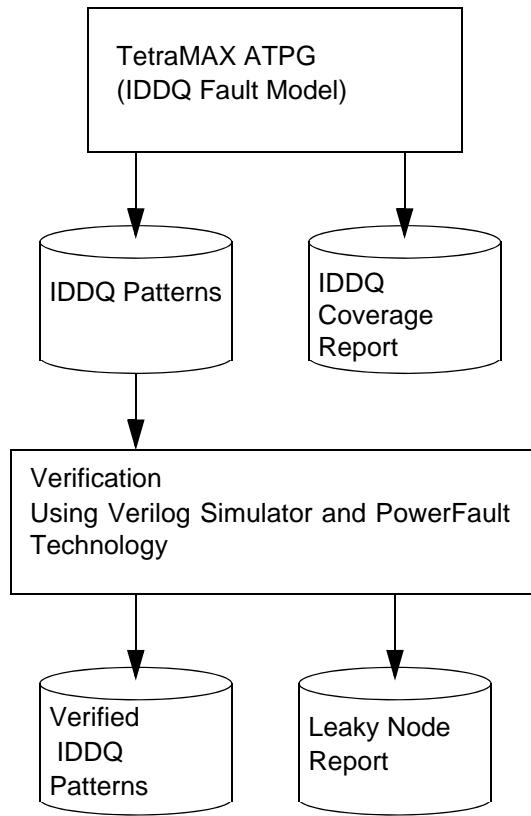
IDDQ testing can provide very high fault coverage with just a few strobes. The first IDDQ strobe typically detects half of all short-to-ground and short-to-VDD faults. IDDQ test patterns attempt to change or toggle as many nodes as possible in subsequent patterns to quickly increase fault coverage.

After the circuit nodes are forced to a known state, a certain amount of inactive time is required to allow the nodes to settle before the IDDQ measurement. The required settling time depends on the CMOS technology used and the desired testing threshold. A tester time “budget” of 10 or 20 IDDQ strobes is typically allowed for testing each device. This number of strobes is usually enough to achieve satisfactory fault coverage.

About IDDQ Pattern Generation

[Figure 16-5](#) shows the IDDQ testing flow using Tetramax test-pattern generation. The ATPG algorithm attempts to sensitize all IDDQ faults and apply IDDQ strobes to test all such faults. TetraMAX compresses and merges the IDDQ test patterns, just like ordinary stuck-at patterns.

Figure 16-5 IDDQ Testing Flow



While generating IDDQ test patterns, by default TetraMax avoids any condition that could cause excessive current drain, such as strong or weak bus contention or floating buses.

TetraMax generates an IDDQ test pattern and an IDDQ fault coverage report. It generates quiescent strobes by using ATPG techniques to avoid all bus contention and float states in every pattern it generates. The resulting test pattern has an IDDQ strobe for every ATPG test cycle. In other words, the output is an IDDQ-only test pattern.

After the test pattern has been generated, you can use PowerFault simulation to verify the test pattern for quiescence at each strobe. The simulation does not need to perform strobe selection or fault coverage analysis because these tasks are handled by TetraMAX. Refer to the *Test Pattern Validation User Guide* for details about PowerFault.

TetraMAX supports IDDQ testing in the following ways:

- It lets you generate test patterns that are targeted for IDDQ testing.
- It adds IDDQ verification and analysis capabilities into your Verilog simulator.

If you use the TetraMAX stuck-at model to generate standard test patterns, you can then use PowerFault technology to select the best strobe times in the resulting test patterns.

Note:

An alternative approach is to use an existing set of stuck-at ATPG patterns and have the Verilog/PowerFault simulation select appropriate IDDQ strobe times from those patterns. This is described in section “Selecting Strobes in TetraMAX Stuck-At Patterns” in the *Test Pattern Validation User Guide*.

Limitations

Note the following limitations:

- The Parallel Verilog or Parallel STIL testbenches for scan compression mode patterns will not contain the iddq_capture annotations. The Serial Verilog and Serial STIL testbenches for scan compression mode will contain the iddq_capture annotations.
- The VerilogDPV flow does not support mixed STIL and IDDQ grading using PowerFault IDDQ. Refer to the Test Pattern Validation User Guide for details about PowerFault.

Fault Models

TetraMAX offers a choice of fault models: stuck-at, IDDQ, transition, bridging, and path delay faults. You specify the IDDQ fault model to generate test patterns specifically for IDDQ testing.

With an IDDQ fault model, TetraMAX does not attempt to observe the logical behavior of the device at the outputs. Instead, it tries to toggle as many nodes as possible into both states while avoiding conditions that violate quiescence. Any node defects can be detected by the excessive current drain that they cause. That is, TetraMAX attempts to sensitize each node in the design, but does not try to propagate faults to the device outputs.

TetraMAX supports two IDDQ fault models: the pseudo-stuck-at model and the toggle model. The pseudo-stuck-at model is the default IDDQ model.

Pseudo-Stuck-At Fault Model

The pseudo-stuck-at model considers the functionality of each individual cell. It is similar to the standard stuck-at ATPG model, except that every cell output is considered observable by IDDQ testing. The fault site at a gate input requires sensitization and propagation to an

output of the same gate (but not to an output of the device) in order to be given credit for IDDQ fault detection. In other words, to be considered detected, a fault must cause an incorrect value at the output of the cell.

Toggle Fault Model

The toggle model is a simpler, net-only model that does not consider gate functionality. Each fault site only needs to have its state controlled in order to be given credit for IDDQ fault detection. The toggle model is less computationally intensive than the pseudo-stuck-at model, but it is not guaranteed to detect as wide a range of faults inside cells.

DRC Rule Violations

TetraMAX performs a wide range of test design rule checking (DRC) when you use the `run drc` command. Some DRC rule violations indicate that your design might not be IDDQ testable or not fully modeled for IDDQ quiescence checking, or might require additional ATPG effort to achieve circuit quiescence. To help avoid DRC violations, follow the design guidelines in “[Design Principles for IDDQ Testability](#)” on page 16-19

To view a list of rule violations after you perform design rule checking, use the `report rules` command. [Example 16-1](#) shows a typical DRC violation report.

Example 16-1 DRC Violation Report

```
TEST> report rules -fail
rule  severity #fails description
----  -----  -----
B6    warning     2 undriven module inout pin
B7    warning   178 undriven module output pin
B10   warning    32 unconnected module internal net
B13   warning     2 undriven instance input pin
S23   warning    64 unobservable potential TLA
S29   warning     1 invalid dependent slave operation
C3    warning    32 no latch transparency when clocks off
C6    warning     1 TE port captured data affected by new capture
Z1    warning   289 bus contention ability check
Z2    warning   289 Z-state ability check
Z4    warning   360 bus contention in test procedure
```

[Table 16-1](#) lists TetraMAX design rule violations that warrant investigation if you plan to generate IDDQ test patterns.

Table 16-1 DRC Rule Violations and IDDQ Significance

Rule	Description, severity	Significance for IDDQ testing
B5	Undefined module referenced, error	Incomplete model; nonquiescent circuitry could be missing
B7	Undriven module output pin, warning	Possible floating net; could be just an unused net
B9	Undriven module internal net, warning	Possible floating net; could be just an unused net
B12	Undriven instance input pin, error	Likely to be a floating net
B18	Three-state and non-three-state drivers combined, warning	Might require more ATPG effort to avoid bus contention
N2	Unsupported construct, warning	Incomplete model; nonquiescent circuitry could be missing
Z1	Bus capable of contention, warning	Might require more ATPG effort to avoid bus contention
Z2	Bus capable of holding Z state, warning	Might require more ATPG effort to avoid floating buses
Z3	Wire capable of contention, error	Likely to be a wired-net contention
Z7	Unable to prevent contention for circuit, error	ATPG cannot find nonquiescent circuit state
Z8	Unable to prevent contention for bus, warning	ATPG cannot avoid bus contention
X1	Sensitizable feedback path, warning	Possible circuit oscillation

For more information about TetraMAX design rule checking, see [“Performing Test Design Rule Checking” on page 4-9](#).

Generating IDDQ Test Patterns

After you check the design rule violations in TetraMAX, you can proceed to test pattern generation. To generate IDDQ test patterns,

1. Set the fault type to IDDQ with the `set faults` command.
2. Select the appropriate IDDQ fault model, either pseudo-stuck-at or toggle model, with the `set iddq` command.
3. Create the fault list with the `add faults` or `read faults` command.
4. Set the maximum number of IDDQ strobes with the `set atpg -patterns` command.
5. Run pattern generation with the `run atpg` command.

For example, here is a typical IDDQ ATPG session:

```
TEST> set faults -model iddq
TEST> set iddq -toggle      // pseudo-stuck-at is the default
TEST> add faults -all
TEST> set atpg -patterns 20  // budget of 20 IDDQ strobes
TEST> run atpg -auto_compression
```

The order of the steps is important. You cannot create the fault list until you have selected the IDDQ fault model.

After you generate the IDDQ test patterns, you can use PowerFault simulation technology to verify the patterns for quiescence. For more information, refer to the *Test Pattern Validation User Guide*.

If you generate stuck-at patterns and you want to use PowerFault to select IDDQ strobes from the pattern set, see “Selecting Strobes in TetraMAX Stuck-At Patterns” in the *Test Pattern Validation User Guide*.

iddq_capture

When you create IDDQ patterns, TetraMAX defines an additional procedure called `iddq_capture` in the pattern output file, which is used whenever an IDDQ measure is performed. Here is an example:

```
"iddq_capture" {
    W "_default_WFT_";
    F {"testmode"= 1; }
    V { "_pi"=\r379 # ; "_po"=\j \r276 X ; }
    IddqTestPoint;
    V { "_po"=\r276 # ; }
```

```
    }  
}
```

You can define your own `iddq_capture` routines in the SPF file and pass them into the flow.

Off-Chip IDDQ Monitor Support

This section describes how to transfer information into off-chip IDDQ monitors as part of your IDDQ test data. Typically, an off-chip IDDQ monitor is an additional hardware unit placed physically adjacent to the device under test (DUT). The monitor is used to perform current measurements and typically has extra signals that you use to control when and how IDDQ measurements are performed.

Off-chip IDDQ monitors require two fundamental constructs in order to be supported at test. One construct is to support the definition of additional signals present on the monitors as part of the test flow. The second construct is the application of specific procedure calls at the IDDQ measurement points.

This is the flow for including off-chip IDDQ monitor signals in your testing:

1. Specify additional signals in the netlist.
2. Define a `iddq_capture` routine so it supports operating the additional signals during test.

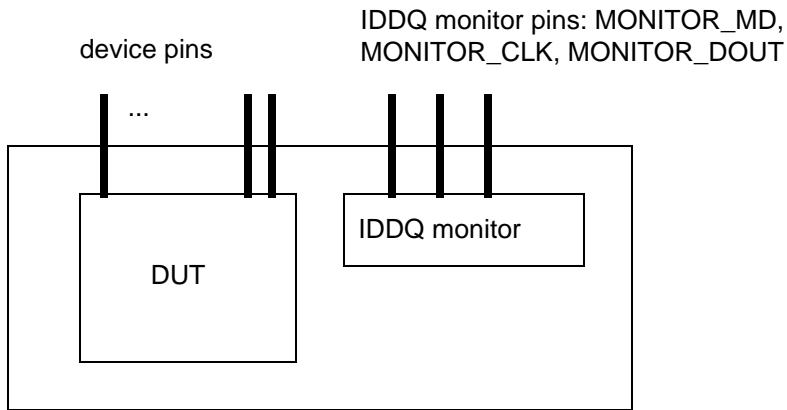
The next two sections describe the flow in detail.

Specifying Additional Signals in the Netlist

Monitor signals are not part of the DUT nor are they part of the tester. They exist adjacent to the DUT on the loadboard or some location near the DUT for signal measurement integrity. While not part of the DUT, these signals are required because they must toggle during IDDQ testing.

Define these signals in an additional hierarchical level that conceptually represents the DUT and off-chip IDDQ monitor as a single unit, as shown in [Figure 16-6](#). The only requirement in the flow is the presence of the additional monitor signals; a representation of the monitor itself is not required or expected.

Figure 16-6 Hierarchical Design With DUT and IDDQ Monitor



The following Verilog netlist shows how references to these signals could look, where the prefix *MONITOR_* is used on all the off-chip IDDQ monitor signals for easy identification:

```
// new top_module of design and MONITOR signals
module AAA_W_QSTAR ( MONITOR_MD, MONITOR_CLK, MONITOR_DOUT,
    ... other design signals ... );
input MONITOR_MD, MONITOR_CLK ;
output MONITOR_DOUT ;
...
AAA DUT ( ... other design signals ... );
endmodule; // AAA_W_MONITOR

// top design module
module AAA ( ... other design signals ... );
...
```

Defining `iddq_capture` to Support Additional Signals

Using off-chip IDDQ monitors affects how you define the `iddq_capture` procedure because the DUT needs to be controlled in particular during the capture operation. You can expand the `iddq_capture` template to support operation of the additional IDDQ monitor pins. The `iddq_capture` procedure will vary depending on operations present to manipulate the IDDQ monitor or to return measurement data.

Because the monitor control signals are part of the netlist sent to TetraMAX, these signals need to be specified as “Fixed” signals in the flow for all other applications; that is, held at their inactive states except during IDDQ testing.

The following example represents an application where the IDDQ measurement/settling time is defined in a WaveformTable with minimal functionality, supporting only the maintenance of the input states during this period. There are no requirements on the name of this WaveformTable. The prefix *MONITOR_* is used on all the off-chip IDDQ monitor signals for easy identification.

```

Timing {
    WaveformTable "_default_WFT_" {
        Period '100ns';
        Waveforms {
            "_default_In_Timing_" { 0 { '0ns' D; } }
            "_default_In_Timing_" { 1 { '0ns' U; } }
            "_default_In_Timing_" { Z { '0ns' Z; } }
            "_default_In_Timing_" { N { '0ns' N; } }
            "_default_Clk0_Timing_" { P { '0ns' D; '50ns' U;
'80ns' D; } }
            "_default_Out_Timing_" { X { '0ns' X; } }
            "_default_Out_Timing_" { H { '0ns' X; '40ns' H; } }
            "_default_Out_Timing_" { T { '0ns' X; '40ns' T; } }
            "_default_Out_Timing_" { L { '0ns' X; '40ns' L; } }
        }
    }
    WaveformTable "_IDDQ_MEASUREMENT_WFT_" {
        Period '100us';
        Waveforms {
            "_default_In_Timing_" { 0 { '0us' D; } }
            "_default_In_Timing_" { 1 { '0us' U; } }
            "_default_In_Timing_" { Z { '0us' Z; } }
            "_default_In_Timing_" { N { '0us' N; } }
            "_default_Out_Timing_" { X { '0us' X; } }
        }
    }
}
Procedures {
    "load_unload" {
        W "_default_WFT_";
        // establish inactive states on the monitor during
Shift
        V { "sdo"=X; "CLK"=0; "MONITOR_MD"=1; "MONITOR_CLK"=0; \
            "MONITOR_DOUT"=X; }
        Shift { V { "__si"=#; "__so"=#; "CLK"=P; } }
    }
    "capture*" { // All Capture Routines
Except iddq_capture
        // Hold monitor inactive
        F { "MONITOR_MD"=1; "MONITOR_CLK"=0; "MONITOR_DOUT"=X;
    }
        W "_default_WFT_";
        V { ... }
    }
    "iddq_capture" {
        W "_default_WFT_";
        V { "__pi"=\r15 # ; "__po"=\j \r7 # ; }
        IddqTestPoint;
        W "_IDDQ_MEASUREMENT_WFT_";

        V { "MONITOR_MD"=0; "_out"=XXX ; } // Activate monitor
measurement
        W "_default_WFT_";
    }
}

```

```

        V { "MONITOR_DOUT"=H; }      // Detect successful
measurement (pass)
    }
}      // end Procedures

```

The following example merges the `_po` measure operation into the IDDQ measurement vector. It requires a more complete WaveformTable to support the measure operation on the outputs but reduces vector count in the `iddq_capture` procedure by one. Only the WaveformTable and `iddq_capture` changes are shown here. The prefix `MONITOR_` is used on all the off-chip IDDQ monitor signals for easy identification.

```

Timing {
    WaveformTable "_IDDQ_MEASUREMENT_WFT_" {
        Period '100us';
        Waveforms {
            "_default_In_Timing_" { 0 { '0us' D; } }
            "_default_In_Timing_" { 1 { '0us' U; } }
            "_default_In_Timing_" { Z { '0us' Z; } }
            "_default_In_Timing_" { N { '0us' N; } }
            "_default_Out_Timing_" { X { '0us' X; } }
            "_default_Out_Timing_" { H { '0us' X; '98us' H; } }
            "_default_Out_Timing_" { T { '0us' X; '98us' T; } }
            "_default_Out_Timing_" { L { '0us' X; '98us' L; } }
        }
    }
}

Procedures {
    "iddq_capture" {
        W "_default_WFT_";
        V { "_pi"=\r15 # ; "_out"= XXX ; }
        IddqTestPoint;
        W "_IDDQ_MEASUREMENT_WFT_";
        V { "MONITOR_MD"=0; "_po"=\r7 # ; } // Activate monitor
measurement
        W "_default_WFT_";
        V { "MONITOR_DOUT"=H; } // Detect successful measurement
(pass)
    }
}

```

The following example maintains the current IDDQ test sequence with the addition of the extra cycles for the monitor's operation at the end. While consistent with current IDDQ constructs, this operation requires the most total cycles per IDDQ test. This construct can operate with a minimal measure WaveformTable, or a larger WaveformTable, depending on whether the outputs are masked in the monitor's measure cycle. Because no state changes are occurring, these outputs can remain in their previous measured state in the next two

vectors (requiring a more complete WaveformTable), or can be masked (requiring less definitions in the monitor's measure WaveformTable). The prefix *MONITOR_* is used on all the off-chip IDDQ monitor signals for easy identification.

```
Procedures {
    "iddq_capture" {
        W "_default_WFT";
        V {"_pi"=\r15 # ; "_out"= XXX ; }
        IddqTestPoint;
        V {"_po"=\r7 # ; }
        W "_IDDQ_MEASUREMENT_WFT";
        V {"MONITOR_MD"=0; } // Activate monitor measurement.
                                // Note outputs still tested
        W "_default_WFT";
        V {"MONITOR_DOUT"=H; } // Detect successful measurement
    (pass)
    }
}
```

IDDQ Commands

The TetraMAX command for setting the fault model is `set faults`. If you select the IDDQ fault model, you can specify the quiescence constraints and toggle/no-toggle model type by using the `set iddq` command. The `add atpg constraints` command lets you set IDDQ-specific ATPG constraints on nodes in the design.

Set Faults

To generate IDDQ-only test patterns, use the `set faults -model iddq` command. You can specify the quiescence constraints and toggle/no-toggle model with the `set iddq` command.

To generate standard stuck-at test patterns, use the `set faults -model stuck` command. This is the default model.

For the complete syntax and option descriptions, see the online help for the `set faults` command.

Set IDDQ

The `float`, `strong`, `weak`, and `write` options of the `set iddq` command allow you to specify the conditions required for quiescence. TetraMAX will not generate a pattern that fails to meet an enabled restriction.

The assertive option `float`, `strong`, `weak`, or `write` means that the restriction is enforced. The restrictions minimize conditions that could cause excessive current drain, such as strong or weak bus contentions or floating buses. The negative option `nofloat`, `nostrong`, `noweak`, or `nowrite` means that the restriction is removed and the condition is allowed. By default, all the assertive options are in effect and all restrictions are enforced. To allow a condition for IDDQ test pattern generation, use the appropriate negative option.

By default, the individual restrictions operate as follows:

- The `float` restriction means that every BUS gate must not be at the Z state during an IDDQ measure.
- The `strong` restriction means that the IDDQ measure must be contention-free for strong drivers of BUS gates.
- The `weak` restriction means that BUS gates with weak inputs must not compete with other strong or weak BUS inputs during an IDDQ measure.
- The `write` restriction means that RAMs must not have an active write port during an IDDQ measure.

The `atpg` or `noatpg` option determines whether the test generator attempts to satisfy all the IDDQ constraints during pattern generation (`atpg`), or only checks and discards patterns that fail to meet these constraints after completion of pattern generation (`noatpg`). The default setting is `-noatpg`.

The option `toggle` or `notoggle` option selects the type of IDDQ fault model. This selection is valid only if you have selected the IDDQ fault model with the `set faults -model iddq` command. The default selection is `notoggle`, which selects the pseudo-stuck-at fault model. To select the toggle model instead, use the `toggle` option. These two models are described in ["Pseudo-Stuck-At Fault Model" on page 16-8](#).

For the complete syntax and option descriptions, see online Help for the `set iddq` command.

Add ATPG Constraints

The `add atpg constraints` command lets you define constraints that apply during the generation of test patterns. For example, you can use this command to force a particular internal node to the value 1 at the clock-on time for all test patterns.

In this command, you specify an arbitrary name to identify the constraint, the value of the constraint (0, 1, or Z), and the place in the design where the constraint is to be applied. You can optionally specify when the constraint must be satisfied by using the `drc` or `iddq` option.

By default, the constraint must be satisfied only at clock-on time for test pattern generation. Using the `drc` option means that the constraint must also be satisfied during DRC procedures and ATPG analyses.

Using the `iddq` option means that the constraint only has to be satisfied during IDDQ measure strobes, and only if the IDDQ fault model has been selected with the `set faults -model iddq` command. An IDDQ measure strobe corresponds to the time in the tester cycle when outputs are measured, as specified by the `WaveformTable` block in the `run drc` test protocol file.

Design Principles for IDDQ Testability

The following design principles apply to designing your circuits for IDDQ testability. Paying attention to these requirements will result in more efficient IDDQ testing and reliable PowerFault simulation. For details about PowerFault, see the *Test Pattern Validation User Guide*.

I/O Pads

Put I/O pads on a separate power rail, if possible. Then you can test the I/O and core logic separately as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If I/O pads and core logic share the same power rail, use I/O pads that have controllable pullups rather than passive pullups. This will allow the pullups to be gated out during IDDQ testing.

Slew control for I/O pins must be disabled or I/O pins must be put on a separate rail. If I/O pads and core logic share the same power rail, all DC paths from power to ground (such as slew control) must be disabled during IDDQ testing. There are two strategies to achieve this:

- Use controllable pullups/pulldowns so that they can be gated out during IDDQ testing. This is the preferred method.
- Drive pads so that pullups/pulldowns not active (for example, drive a pad with a pullup to VDD). Have the testbench drive pads that have both pullups and pulldowns to VDD (or to VSS if you are measuring ISSQ).

Buses

Use fully multiplexed bus drivers so that only one driver can be active at a time. Furthermore, always drive a bus if possible, as described in the “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If buses cannot always be driven, gate buses at the receivers as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

If buses can’t be driven or gated, use keeper latches. Model keeper latches structurally as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

Avoid internal pullups and pulldowns. If possible, either drive or gate a bus to prevent it from floating. If pullups and pulldowns must be used, model them structurally as described in “Using PowerFault Technology” in the *Test Pattern Validation User Guide*.

Avoid tri1, tri0, wor, and wand wire types. Use pullup/pulldown primitives instead.

RAMs and Analog Blocks

Check your databook to make sure you do not hardwire your RAM into a high-current state. RAMs and analog blocks that have high-current states require either a sleep mode or a separate power supply.

If your chip uses a sleep mode for RAM or analog blocks, prevent IDDQ strobing when the blocks are not in sleep mode by doing one of the following:

- Avoid invoking the `strobe_try` command when the chip is in a high-current state.
- Use the `disallow` command to tell PowerFault when RAM or analog blocks are in high-current states.

If RAM or analog blocks are on a separate power rail and PowerFault reports them as leaky, use the `allow` command to have PowerFault ignore them.

Free-Running Oscillators

Avoid free-running oscillators, if possible, because they draw current. If you must use a free-running oscillator, disable it during IDDQ testing, or put the affected circuitry on a separate power rail. Use the `disallow` command to tell PowerFault when the oscillator is running.

Circuit Design

To prevent current drain through the substrate, connect the bulk node for n-type transistors to VSS and the bulk node for p-type transistors to VDD.

Avoid degraded voltages. For example, avoid using an NMOS transistor to serve as a pass gate.

Avoid circuits that put the gate and drain or source nodes of a transistor in the same transistor group.

Avoid circuits that create control loops among transistor groups. Obviously, control loops must exist in order to implement flip-flops and latches, but using certain flip-flop and scan chain design rules can make bridging faults more testable.

Avoid circuits that use charge sharing or charge retention. Bridging faults within dynamic (domino) logic cells are difficult to detect with IDDQ testing. Furthermore, the output voltage of dynamic logic cells might degrade during an IDDQ measurement, causing the inputs to the following static logic block to float.

Power and Ground

Declare supply0, supply1, tri0, and tri1 nets fed in from the testbench so that they have the same type in the DUT. For example, if tbench.VDD1 is a supply1 net in the testbench and it is connected to tbench.dut.vdd1, make sure that tbench.dut.vdd1 is also declared as a supply1 net.

If you are using Verilog-XL, do not use cell ports for VSS/VDD. Use a local supply0 or supply1 net, or a 'b0 or 'b1 constant to connect terminals and ports inside cells to VSS/VDD.

If you are using VCS, connect terminals and ports to supply0 or supply1 nets instead of using 'b0 or 'b1 constants.

Models With Switch/FET Primitives

Try to limit switch modeling to three-state cells.

Use user-defined primitives (UDPs) or standard logic gates to build models for multiplexers, flip-flops, and latches.

Avoid `tran`, `tranif0`, and `tranif1` primitives. Instead, use `cmos`, `nmos`, and `pmos` primitives.

Avoid having channels of switch primitives in series extend between module scopes.

Do not pass three-state values (Zs) through switches or field effect transistors (FETs). If a net can take on a three-state value, make the receivers (loads) strength-restoring gates, not switch primitives.

Connections

Maintain cell-level hierarchy and avoid creating a very large cell containing many Verilog primitives at the same level. Limit each bottom-level cell to a few hundred primitives at most. (Most ASIC libraries have only a few primitives per bottom-level cell.)

Do not use continuous assignments to connect nets to nets.

Do not use continuous assignments to implement three-state drivers.

Do not use mismatched drivers to model latches and flip-flops. If possible, use UDPs.

Do not connect registers directly to gate terminals. Connect registers to wires (via continuous assignments or module ports), and then connect the wires to gate terminals.

All internal buses should have gate loads. Each internal bus should fan out to at least one gate input, instead of fanning out to only behavioral statements (such as continuous assignments and event control for `always` blocks).

PowerFault is most accurate at identifying leaky states when used with gate-level models and libraries. Avoid using RTL models because they might not contain enough structural information to allow identification of floating nodes and drive contention.

IDDQ Design-for-Test Rule Summary

The following design-for-test (DFT) rules summarize the design principles for IDDQ testing:

1. Define an IDDQ test mode signal that does not contend with the scan test mode signal.
2. Use separate power rails for the I/O and core modules.
3. Use fully complementary, fully static CMOS.
4. Use separate power rails for analog and nonstatic CMOS modules.
5. Use separate power rails for unknown or otherwise IDDQ-untestable cores.
6. For RTL modules, specify any known input conditions and sequences that cause internal contention. Use ATPG constraints or IDDQ `allow` or `disallow` statements (or both).
7. For RTL modules, specify any known input conditions and sequences that cause internal floating.
8. Use transistors to enable and disable pullups and pulldowns. Disable them with the IDDQ test mode signal.
9. Using the IDDQ test mode signal, disable three-state and bidirectional outputs that require pullups or pulldowns.

10. Each internal three-state nets requires one of the following: a bus holder, one-hot enable logic, or logic to gate off all bits of the bus except for the least significant using the IDDQ test mode signal.
11. Each compiled SRAM or ROM requires one of the following: 100 percent CMOS circuitry, a separate power rail, or a defined condition controlled by the IDDQ test mode signal that guarantees quiescence.
12. Do not allow SRAM and DRAM outputs to go to the Z state unless a bus holder is present on the output.
13. Each compiled data path cell must either be 100 percent static CMOS or allow quiescence control with the IDDQ Test Mode signal.
14. Do not allow any unconnected module or cell inputs.

Additional System-on-a-Chip Rules

The following rules apply to system-on-a-chip (SOC) applications:

1. Each core must have a test isolation mode. Each core must not be affected by other cores or user-defined logic, and must not affect other cores or user-defined logic. Each core must not be allowed or required to propagate contention or float conditions.
2. All cores and user-defined logic sharing a power rail must be quiescent during the time each core is being IDDQ-tested.
3. Built-in self-test (BIST) must have a mode that gives the tester control of the BIST clock.
4. It must be possible to stop the clock. The core must have a bypass clock signal from the tester (a primary I/O).

17

Bridging Fault ATPG

This chapter describes the bridging fault model and fault simulation ATPG flow.

This chapter contains the following sections:

- [Understanding Bridging Defects](#)
- [Detecting Bridging Faults](#)
- [Bridging Fault Flows](#)
- [Using Star-RCXT to Generate a Bridge Fault List](#)
- [Bridging Fault Model Limitations](#)

You will need a Test-Fault-Max license to use this feature. This license is also checked out if you read an image that was saved with the fault model set to bridging.

Understanding Bridging Defects

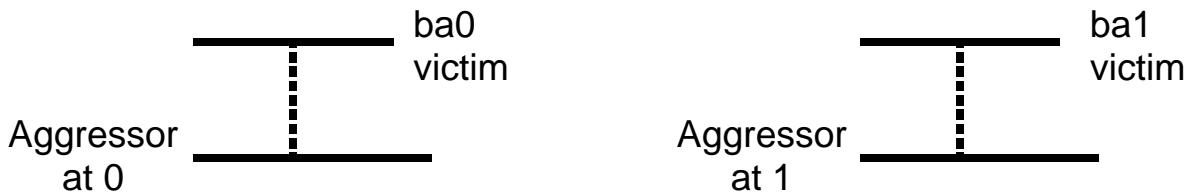
A common defect in semiconductor devices is the bridging defect (or short). This defect causes two normally unconnected signal nets in a device to become electrically connected due to extra material or incorrect etching.

Such defects can be detected if one of the nets (the aggressor) causes the other net (the victim) to take on a faulty value, which can then be propagated to an observable location. Although there is a strong correlation between stuck-at coverage and bridging coverage, there is no guarantee that a set of patterns generated to target stuck-at faults will achieve similar coverage for a set of bridge faults.

Detecting Bridging Faults

TetraMAX defines a bridging fault by type and a set of two nodes (which in this discussion can be instance pins or net names). The type is either bridging fault at 0 (ba0) or bridging fault at 1 (ba1) (see [Figure 17-1](#)). The first node is called the victim node and the second node is called the aggressor node.

Figure 17-1 Bridging Fault Types ba0 and ba1



A ba0 bridging fault is considered detected if the stuck-at-0 fault at the victim node is detected at the same time the fault-free value of the aggressor node is at 0. Similarly, a ba1 bridging fault is considered detected if the stuck-at-1 fault at the victim node is detected at the same time the fault-free value of the aggressor node is at 1.

Bridge Locations

The victim and aggressor nodes are specified by *bridge location*. A bridge location is a hierarchical path to a

- Cell instance input pin
- Cell instance output pin
- Net name

Faults on bidirectional pins are ignored.

Although a net can have many names as it traverses the hierarchy of a design, TetraMAX does not store them all. If you specify a net name as a bridge location that TetraMAX recognizes (those accepted by the `report primitives` command), it will be used to map the fault to the single output pin connected to that net.

Net names are internally translated to an instance pin. This pin path must be a valid stuck-at fault site. Instances dropped during the build process with a B22 warning message cannot be used. A warning is given if you specify an invalid bridge location.

Strength-Based Patterns

A bridge defect has complex analog effects due to parameters such as the strength of the driver, resistance of the bridge, and wire characteristics. Therefore, it is not always clear that a bridge will be detected by the pattern generated considering only logical behavior. Some researchers have speculated that patterns can be adjusted to improve the odds of detecting bridging faults. The basic premise is that forcing the aggressor to drive stronger and the victim to drive weaker increases the chance of the bridge being detected.

Patterns that use this principle can be generated when the victim or aggressor is on the output pin of a primitive gate having a dominant value (AND, OR, NAND, or NOR). A more stringent detection criteria can then be imposed. The ATPG process can be given additional soft constraints to optimize the drive strengths after the normal bridging fault detection requirements are met. Soft constraints are those that the ATPG algorithm attempts to meet on a best-effort basis. If the soft constraints are not met, the pattern is still retained for detection of bridging faults.

With the addition of strength-based patterns, bridge fault detection can be classified into these types:

- Minimal detection – the minimum condition for the detection of ba0 & ba1 faults, as previously described
- Fully optimized detection – a detection where the conditions specified in [Table 17-1](#) are met. For maximizing inputs with a specific value, all inputs of the driving gate must be at the specified value. To minimize the inputs at a specific value, only one of the driving gate's inputs must be at the specified value.

- Partially optimized detection – a detected bridging fault that is neither minimal nor fully optimized.

Table 17-1 Strength-Optimized Detection of Bridging Faults

Driving Gate	ba0		ba1	
	Driver of victim	Driver of aggressor	Driver of victim	Driver of aggressor
AND		Maximize driver inputs with 0s	Minimize driver inputs with 0s	
NAND	Minimize driver inputs with 0s			Maximize driver inputs with 0s
OR	Minimize driver inputs with 1s			Maximize driver inputs with 1s
NOR		Maximize driver inputs with 1s	Minimize driver inputs with 1s	

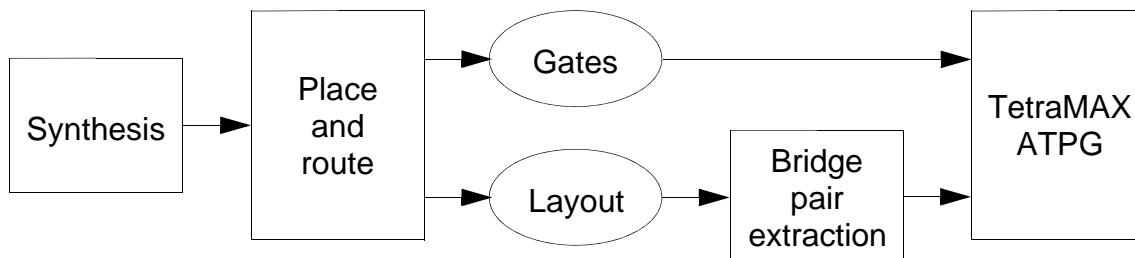
Bridging Fault Flows

The following subsection describes how bridging faults fit into an overall flow and in TetraMAX.

Bridging Faults and the Overall TetraMAX Flow

The overall flow for bridging faults is shown in [Figure 17-2](#). Stuck-at fault ATPG is run immediately after synthesis, and before place and route in some flows. Bridging fault ATPG and fault simulation is usually run following the completion of place and route on full-chip designs.

Figure 17-2 Overall Flow for Bridging Faults



There are multiple ways you can generate bridge pairs. Two possibilities are

- extract bridging pairs from the layout using an IFA-based scheme
- use an extracted coupling capacitance report

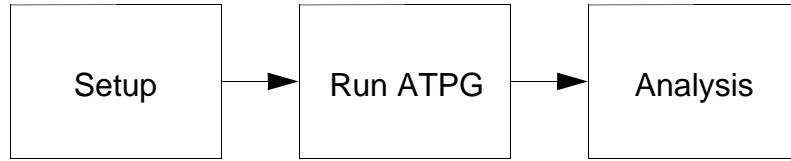
A flow based on capacitance extraction is shown in “[Using Star-RCXT to Generate a Bridge Fault List](#)” on page 17-9. TetraMAX can read the coupling capacitance report directly from Star-RCXT. Third-party capacitance extraction tools can be used to generate coupling capacitance reports as long as the node list is in the TetraMAX format.

For greater accuracy, eliminate bridges involving clocks and asynchronous sets and resets. Also, filter out bridges producing combinational loops.

Bridging Fault Flow in TetraMAX

[Figure 17-3](#) shows the bridging fault flow in TetraMAX. The typical commands used in this flow are identified in the subsections that follow. For details regarding command usage and option descriptions, see online Help.

Figure 17-3 Bridging Fault Model in TetraMAX



Setup

These commands are typically used at the beginning of a command file, because they have an effect on subsequent commands:

- `set faults -model bridging`
This command is mandatory for bridging faults.
- `set atpg -optimize_bridge_strengths`
This command can be used based on whether or not TetraMAX should optimize drive strength on the driving gates of the victim and aggressor nodes.
- `set fault -bridge_inputs`
This command can be used to accept input pins of instances as bridge locations.

Input Faults

The list of bridging pairs can be supplied in any combination of the following three ways:

- Command file – A set of `add faults` commands that can be sourced by a script:
`add faults [-BRIDGE_Location <bridge_location1>
[bridge_location2>] [-bridge <0|1|01>]
[-aggressor_node <first | second | both>]`
This command can be used to add bridging faults.
- Fault list file – Generated by a `report faults` or `write faults` command and read by the `read faults` command. Only ba0 and ba1 fault types are expected.
- Node file – A list of bridging node pairs:
`add faults <-node_file <name>>[-bridge <0|1|01>]
[-aggressor_node <first | second | both>]`

In its simplest form, the node file format is a pair of bridge locations per line, separated by a space. An unmodified coupling capacitance report from Star-RCXT can also be used. For details, see topic “Node File Format for Bridging Pairs” in the online help. A suggested flow of using the Synopsys Star-RCXT capacitance extraction tool for bridge fault list generation is described in [“Using Star-RCXT to Generate a Bridge Fault List” on page 17-9](#).

A bridge fault list should not include clocks and asynchronous set or reset signals. Proper detection status cannot be guaranteed for these faults.

Manipulating the Fault List

These commands and options are useful for manipulating the fault list:

- `addnofaults`
This command can be used to set “no fault” status on victim nodes to prevent the associated bridging fault from being added to the fault list.
- `removefaults`
`<[-BRIDGE_Location <bridge_location1>
<bridge_location2>]
| -all | -retain_sample <d>
| -class <fault_class>> [-bridge <0|1|01>]
[-clocks] [-aggressor_node <first | second | both>]
[-non_strength_sensitive]`

This command can be used to remove bridging faults.

Examining the Fault List

These options of the `report faults` command can be used to examine a fault list, both before and after fault ATPG and simulation:

```
<[bridge_location1 [bridge_location2]]  
[-bridge <0|1|01>]  
[-aggressor_node <first | second | both>]  
[-bridge_feedback] [-bridge_strong]
```

The format of the report is in four columns as follows

- Fault type (ba0 or ba1)
- Fault detection status code
- Bridge location of the victim node
- Bridge location of the aggressor node

For example,

```
ba0  NC  nodeA  nodeB  
ba1  NC  nodeA  nodeB  
ba0  NC  nodeB  nodeA  
ba1  NC  nodeB  nodeA
```

Fault Simulation

Fault simulation of bridging faults is usually done to determine which bridges are detected by other existing patterns, such as those generated for stuck-at faults. Typically, many bridges will be detected by patterns targeting other fault models.

For bridging faults with either or both nodes driven by gates with dominant values (AND, OR, NAND, or NOR), use the `run fault_sim -strong_bridge` command to require a fully optimized detection. When this option is used, the fault is marked as detected only if the criteria for fully optimized bridging fault detection is met.

Running ATPG

Bridging fault ATPG attempts to set the victim and aggressor bridge locations at opposite values, while attempting to detect the value of the victim net.

If you plan on issuing a `run atpg -auto_compression` command, you first need to create an explicit fault list by either issuing an `add faults` or `read faults` command.

If you issue a `set atpg -optimize_bridge_strengths` command, ATPG attempts to generate patterns with fully optimized detections on a best effort basis. This assumes that the TetraMAX libraries are modeled in a manner that would produce meaningful strength-based patterns. For example, gates with dominant values should be instantiated so that the correct transistors are activated or deactivated.

Analysis

After running ATPG or fault simulation, you can use the `report faults` and `write faults` commands to analyze fault detection status. You can invoke automated analysis and schematic display by using the `analyze faults <bridge_location1 bridge_location2 -bridge <0|1>` command.

Example Script

[Example 17-1](#) shows a script for bridging fault support. This script generates tests for bridging faults followed by stuck-at faults. You may want to experiment with the reverse order as well to see which method produces better results.

Example 17-1 Script for Bridging Faults

```
# read netlist and libraries, build, run drc
read netlist design.v -delete
run build_model design
run drc design.spf

# bridging faults
set faults -model bridging

# allow instance input pins to be valid victim sites
set faults -bridge_inputs

# to optimize strengths during atpg
set atpg -optimize_bridge_strengths

# read in fault list
add fault -node_file nodes.txt

# run atpg with merging
set atpg -merge high
run atpg -auto_compression

# write the bridging patterns out
write patterns bridge_pat.bin -format binary -replace

# now fault simulate bridge patterns with stuck-at faults
# this part is intended to reduce the set of patterns by not
generating
# patterns for stuck-at faults detected by the bridging patterns
remove faults -all
set faults -model stuck
add faults -all

# read in bridging pattern
```

```

set patterns external bridge_pat.bin

# fault simulate
run fault_sim

# generate additional stuck-at patterns
set atpg -merge high
set patterns internal
run atpg -auto_compression

```

Using Star-RCXT to Generate a Bridge Fault List

The bridging fault model requires a pair of locations to bridge. Because all pairs of nets would result in an intractably large fault set for ATPG or fault simulation, a set of bridge pairs that are likely candidates are used. This information is generated using layout information, because nets in close proximity are more likely to be bridged.

The flow currently supported by Synopsys uses finding nets with high coupling capacitance using Star-RCXT. Some studies have shown there is a correspondence between capacitance and likelihood of bridging.

This section describes the process of generating a bridging fault list. It assumes you have some knowledge of Star-RCXT and that your environment is properly set up and licensed. See the Star-RCXT documentation for more details. If Star-RCXT is not available, any capacitance extraction tool that generates a coupling capacitance report can be used. Synopsys does not support third-party tools or flows.

If high accuracy is a requirement, the coupling capacitance report from a normal run of Star-RCXT can be used as a bridge pair list. If a small set of bridge faults will be used, or if a higher accuracy bridge fault list is required, a separate run of TCAD characterization and extraction is required to remove the effect of varying dielectric constants across layers.

If a coupling capacitance report from a normal Star-RCXT run will be used, ensure that the report has enough net pairs for bridging fault ATPG and for fault simulation. The remainder of this section may be skipped if this is the flow you choose.

The process involves the following main steps:

- TCAD characterization using grdgenxo, a part of the Star-RCXT toolset. This is done only once for a given fabrication process and bridge probability.
- Capacitance extraction and coupling capacitance report generation using StarXtract. This is a separate run from the normal capacitance extraction run.
- Running TetraMAX.

TCAD Characterization

TCAD characterization is run once per process and bridging probability. It need not be run multiple times for each process corner. Only the layer locations are important. The process parameters themselves play almost no role in this procedure.

Edit a copy of the Interconnect Technology Format (ITF) file supplied by your foundry. It will be used by Star-RCXT to compute bridging likelihood. You edit the ITF file so the dielectric constants have less of an effect in generating the bridging pairs.

Generating a resistance and capacitance (GRD) model.

The following steps assume that inter-layer bridging is much less likely than intra-layer bridging.

1. Change all DIELECTRIC statements between the top metal and poly layers as follows, depending on each dielectric's location.

To determine if a dielectric is between layers or between conductors on the same layer, create a diagram of the layers. See the section on DIELECTRIC statements in Star-RCXT documentation for more information.

2. For dielectrics between different layers, change "ER" to 0.1.
3. For dielectrics between conductors on the same layer, change "ER" to some constant value; for example 1.

These numbers can be adjusted based on data from the foundry on bridge probabilities. For example, if some layers have higher defect densities, the intra-layer ER values can be increased for that layer. This effectively sets the probability of intralayer bridges to be 10X the probability of interlayer bridges, based on the simplified parallel-plate capacitance equation ($C = \epsilon * A/d$, where ϵ is the dielectric constant, A is the area of the conductors facing each other, and d is the distance between the conductors).

4. Do TCAD characterization by running the field solver on the ITF file; for example, and create the .nxtgrd file.

```
% grdgenxo my.itf
```

This produces a .nxtgrd (resistance and capacitance model) file that will be used by Star-RCXT.

This step can be done parallel to starting other jobs in the same directory on different machines. The speedup is almost linear relative to the number of processors. See the Star-RCXT documentation for details.

Extracting Capacitance

Before beginning this process, you need to supply these files:

- Post-layout data in the form of a Milkyway database, LEF/DEF files, or Calibre files.
- The .nxtgrd file from the TCAD characterization step in the preceding section.
- A layer mapping file that maps the layer names in the ITF file to the layout layer names. This layer mapping file is created only once per process and should already exist for normal extraction.

Make sure the SYNOPSYS environment variable (\$SYNOPSYS) is set to the Synopsys root directory for Star-RCXT.

Running Star-RCXT in GUI or Batch Mode

To run Star-RCXT in GUI Mode:

1. Start Star-RCXT with the GUI option; for example,

```
% StarXtract -clean -gui &
```

Milkyway database is assumed to exist. Other layout database formats have a similar flow.

2. Set up the extraction run.

3. Choose Setup > Timing. In the Timing Wizard, enter:

Drop down – Set to the layout data format (Milkyway is assumed).

Data format –

If Milkyway or Hercules data:

BLOCK – Typically the design name. Most often there is a file with this name under the CEL view if you are using Milkyway. Check the CEL directory under the Milkyway directory.

MILKYWAY DATABASE – The directory name containing the Milkyway database.

If LEF/DEF or Calibre, specify the appropriate input values.

TCAD GRD FILE – Path to the .nxtgrd file.

MAPPING FILE – Name of mapping file.

EXTRACTION – RC

COUPLE_TO_GROUND – NO

COUPLING_MULTIPLIER – 1

Then click OK.

4. Choose Setup > Noise. In the Noise Wizard, enter:

COUPLING REPORT FILE – Name of output file containing coupling report.
COUPLING ABS THRESHOLD – 0.1
COUPLING REL THRESHOLD – 1FF
COUPLING REPORT NUMBER – Number of the top net pair to report.

A rule of thumb for the number of pairs is that it should be on the order of magnitude of the stuck-at faults in the design.

Then click OK.

The default extraction should not consider power and ground signals as coupling partners. The preceding settings should prevent “smart decoupling,” because all coupling capacitances must be preserved.

5. Choose File > Run to begin the extraction.

Run Star-RCXT in Batch Mode:

1. Star-RCXT can be run in batch mode without a GUI. An example command file follows:

```
BLOCK: <design name>
MILKYWAY_DATABASE: <milkyway db>
TCAD_GRD_FILE: <nxtgrd file>
MAPPING_FILE: <mapping file>
EXTRACTION: C
COUPLE_TO_GROUND: NO
COUPLING_MULTIPLIER: 1
COUPLING_REPORT_FILE: <output coupling capacitance
report file>
COUPLING_ABS_THRESHOLD: 1e-15
COUPLING_REL_THRESHOLD: 0.1
COUPLING_REPORT_NUMBER: <number of nets to include in
report>
```

2. Specify the command file on the command line; for example,

```
StarXtract [options] star_cmd
```

Coupling Capacitance Report

An example coupling capacitance report follows. Note that some net pairs might be repeated in the opposite order with the victim and aggressor switched, as seen in the last 2 pairs. TetraMAX will add the faults twice.

```
* 632 worst couplings list in decreasing order:  
* % coupling victim aggressor  
100 3.33e-18 timer/se_7_cnt/K2370 timer/se_7_cnt/I2370  
61.2 1.66e-15 io_c0[7] io_c1[9]  
60.5 9.26e-17 timer/m_cnt/L2865 timer/m_cnt/T2128  
60.4 2.77e-15 io_c3[5] io_c3[3]  
59.7 2.29e-15 io_c4[7] io_c4[12]  
57.5 2.33e-15 io_c4[14] io_c4[15]  
54.9 7.97e-15 io_c2[14] io_c2[0]  
54.1 7.97e-15 io_c2[0] io_c2[14]  
...
```

Running TetraMAX

The Star-RCXT coupling capacitance report can be directly read in by TetraMAX using the `add faults -node_file` command. TetraMAX automatically recognizes a Star-RCXT report and adds the necessary faults.

Do not use the netlist before place and route as is sometimes used. Use only the postroute netlist for TetraMAX. Otherwise, the signal names might not match.

Do not use an image file when adding faults that include net names, because net names are not preserved in the image file. An alternative means is to

- Read the netlist
- Build
- Perform DRC
- Add the net name-based bridge faults
- Write out the fault list

The output fault list will be pin-path based that can be read in subsequent TetraMAX runs that use the image file.

Bridging Fault Model Limitations

Use of the bridging fault model currently has the following limitations:

- No oscillation effects are considered. The aggressor remains at the fault-free value. Fault effects from a victim in the fanin cone will be dropped at the aggressor.
- Full-Sequential ATPG and Full-Sequential fault simulation are not supported.
- Bidirectional pins cannot be faulted.
- Basic-Scan ATPG and fault simulation assumes clocks and asynchronous sets/resets are at constant values per pattern.
- There is no fault collapsing for bridging faults.
- No detection by implication (DI) credit is given.
- No method for generating bridging node pairs is provided within TetraMAX.
- Net names cannot be used for bridging locations if the `read image` command was used. Only net names given by the `report primitives` command are supported.

18

AUTOMATIC TEST PATTERN GENERATION Distributed Processing

This chapter describes distributing compute tasks across several CPUs.

This chapter contains the following sections:

- [Command Summary](#)
- [Distributed Process Flow](#)
- [Verifying Your Environment](#)
- [Using Distributed Processing: Step By Step](#)
- [Licensing Schemes](#)
- [Limitations](#)

To run ATPG distributed processing on any number of slave machines, you will need a Test-Accelerate-Max license, or each distributed processor will check out one Test-ATPG-Max and one Test-Faultsim license.

Command Summary

Additional commands are available in order to set up a distributed processing environment. Options of the `run atpg` and `run fault_simulation` commands control this feature.

Identifying a Work Directory

The master and slave machines have to share a common directory for exchanging data. The `set distributed -work_dir` command specifies which directory to use. This directory has to be visible to each machine involved in distributed processing. The log files from the slaves are also saved in the work directory. It is also required that the relevant permissions (read and write) be set for each machine.

For the complete syntax and option descriptions, see online Help for the `set distributed` command.

Adding Machines to the Distributed Processor List

The `add distributed processors` command adds one or more machines to the pool of distributed processors. At least one machine name needs to be passed as an argument to this command.

Note:

You can execute multiple processes on a uniprocessor machine, but the various processes would have to share processor time. Therefore, you will not be able to take full advantage of the parallelization.

You can specify the processors directly by their host names or you can request the processors from load balancing software, like LSF or GRID but you cannot combine both of these types. For the complete syntax and option descriptions, see online Help for the `add distributed processors` command.

Removing a Machine From the Distributed Processor List

The `remove distributed processors` command removes one or more machines from the distributed processor list. If your distributed processor list is populated with load sharing facility jobs, you cannot remove just one processor. Use the `-all` option if you want to remove all the processors.

For the complete syntax and option descriptions, see online Help for the `remove distributed processors` command.

Controlling Timeouts

The `set distributed` command allows you to control the amount of time the system is given to perform various distributed processor-related communications. This includes checking the status of the distributed processors (using an UNIX `rsh` command), printing statistics regarding the current job, and waiting for load sharing software to schedule the jobs. This command is provided for your convenience and is not mandatory to start a distributed job.

For the complete syntax and option descriptions, see online Help for the `set distributed` command.

Reporting Current Slave Machines

The `report distributed processors` command reports all machines currently in the list of distributed processors and identifies the distributed working directory.

```
report distributed processors
```

Starting Distributed ATPG

These options of the `run atpg` command add additional controls to start distributed processing and ATPG modes:

```
auto  
basic_scan_only | fast_sequential_only |  
    full_sequential_only  
distributed
```

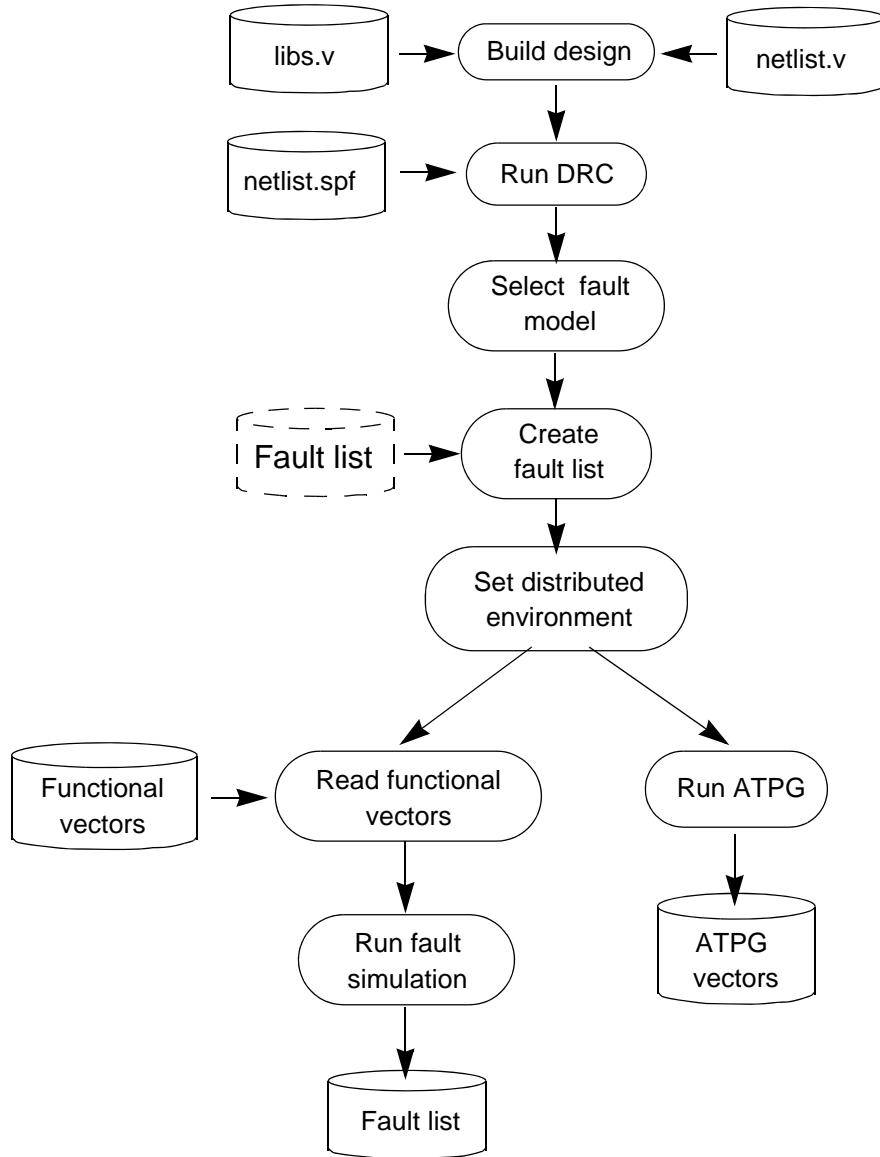
Like a uniprocessor flow, TetraMAX works with one or more pattern generation engines. In TetraMAX versions before 2004.12, you could only specify one ATPG engine mode at a time. In TetraMAX 2004.12, you can use the same ATPG options as the uniprocessor mode.

For the complete syntax and option descriptions, see online Help for the `run atpg` command.

Distributed Process Flow

[Figure 18-1](#) shows the steps necessary to complete a distributed fault simulation or ATPG task. The methodology and flow remains the same as in single-processor fault simulation or ATPG tasks.

Figure 18-1 Distributed Processing Flow



Verifying Your Environment

Whenever TetraMAX starts a distributed process, it issues the `tmax` command. As a consequence, be sure you have the `tmax` script directly accessible from each distributed processor machine. Do not alias this script to another name or TetraMAX will not be able to spawn distributed processes. The safest approach is to have the path to this script added in your PATH environment variable; for example,

```
setenv SYNOPSYS /softwares/synopsys/2004.12
set path = ( $SYNOPSYS/bin $path )
```

For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see “[Specifying the Location for TetraMAX Installation](#)” on page 2-2.

Remote Shell Considerations

If you are running distributed processing by directly specifying the host names, TetraMAX relies on the `rsh` (`remsh` for HP platforms) UNIX command to start a process on a distributed processor machine. This command is very sensitive to the user environment, so you could experience some problems because of your UNIX environment settings. In case you get an error message while adding a distributed processor, refer to the message list at the end of this document to find out the reason and some advice on how to solve the problem.

You need to have special permissions to start a distributed process with a `rsh` (or `remsh`) command. In a classical UNIX installation, those permissions are given by default, however your system administrator may have changed them. If you experience any issue with starting slaves and suspect it is due to this command, enter the following:

```
rsh distributed_processor_machine "tmax -shell"
```

If you get an error message, it is related to your local UNIX environment. Contact your system administrator to solve this issue.

Tuning Your .cshrc File

You should pay special attention to what you put in your `.cshrc` file. Avoid putting commands that exercise the following behavior:

- Are interactive with the user (that is, the system asking the user to enter something from the keyboard). Because you will not have the ability to answer (distributed processes are transparent to the user), it is likely that the process will hang waiting for an answer to a question you will never get.
- Require some GUI display. Your `DISPLAY` environment variable will not point to your master machine when the `rsh` (or `remsh`) command starts a distributed process. As a consequence, the system will put the task “`tty output stopped mode`” on the distributed processor machine, and your master will sit there waiting for a process that has quit running.

If you have any trouble using the `add distributed_processors` command, you may want to have a dedicated `.cshrc` file for running your distributed tasks. A basic configuration should help you get through these issues.

Checking the Load Sharing Setup

If you are planning to run distributed ATPG with load sharing software, first make sure you have the following information available to you.

- Path to the load sharing software (bsub for LSF and qsub for GRID)
- Required options (like project name or queue name)

The TetraMAX session for the master must be run on a machine that is a valid submit host capable of submitting jobs to load sharing software.

Using Distributed Processing: Step By Step

The following sections guide you in using distributed ATPG.

Building the Design and Running DRC

Just as you would do for a uniprocessor run, you have to build an internal representation of the design to start running fault simulation or test pattern generation. This step is exactly the same as the one you are already familiar with.

Sample Script

```
BUILD> read netlist Libs/*.v -delete -library -noabort  
BUILD> run build top_level  
DRC> set drc top_level.spf  
DRC> run drc
```

Selecting the Fault Model and Creating the Fault List

Note:

N-detect ATPG and fault simulation are not supported for distributed ATPG.

Distributed Fault Simulation

Everything supported during uniprocessor fault simulation is also supported during distributed fault simulation. There is no limitation added by this feature. The fault list can be created either with the add faults command or read from a file.

Distributed ATPG

Every fault model and ATPG engine is supported the same way it is in uniprocessor mode.

The following is a quick summary of the various fault models and their related ATPG engines:

- Stuck-at fault model is supported by Basic-Scan, Fast-Sequential, and Full-Sequential engines
- Path delay fault model is supported by Basic-Scan, Fast-Sequential, and Full-Sequential engines
- Transition fault model is supported by Basic-Scan, Fast-Sequential, and Full-Sequential engines
- IDDQ fault model is supported by Basic-Scan and Fast-Sequential engines
- Bridging fault model is supported by Basic-Scan and Fast-Sequential engines

Sample Scripts for Selecting Fault Models.

Example 1:

```
TEST> set fault -model stuck
TEST> addnofaults top_level/module1/sub_mod
TEST> add faults -all
```

Example 2:

```
TEST> set fault -model transition
TEST> set delay -nopi_change -nopo_measure
TEST> set delay -launch last_shift
TEST> read fault myFaultList.flt
```

Setting Up the Distributed Environment

The first thing you have to do is to define a working directory. This is where all the files required for exchanging data between the various machines and their log files will be stored. This directory must be accessible by each machine involved in the distributed process and they must be able to read from and write to this directory. An example:

```
TEST> set distributed -work_dir /home/dist/work_dir
```

The working directory must be specified using an absolute path name starting from the root of the system. Relative paths are not supported in the current TetraMAX release. If you do not specify a working directory, the current directory is used as the default work directory.

After you set the working directory, you might want to populate the distributed processors list; for example:

```
TEST> add distributed processors zelda nalpari
      Arch: sparc-64, Users: 22, Load: 2.18 2.14 2.17
      Arch: sparc-64, Users: 1, Load: 1.45 1.41 1.40
```

These commands help you maintain this list:

- add distributed processors
- remove distributed processors
- report distributed processors

For every machine, you automatically get the type of platform (Architecture), as well as the number of users currently logged on that machine and the processor load. You can add as many distributed processes as you want on one machine. However, you should know in advance the number of processors on that machine in order not to start more distributed processes than the number of available processors. Even if it is technically possible, the various processes would have to share time on the processors; thus you will not be able to take full advantage of the parallelization.

TetraMAX supports heterogeneous machine architectures (sparcOS5, Linux, and HP-UX). For example,

```
TEST> add distributed processors proc1_sparcOS5 proc2_Linux
      proc3_HPUX
```

You can visualize the current list of machines in the list of distributed processors with the report distributed processors command; for example:

```
TEST> report distributed processors
      Working directory ==> "/remote/dtg654/atpg/dfs" (32bit)
      -----
      MACHINE: zelda [ARCH: sparc-64]
      MACHINE: nalpari [ARCH: sparc-64]
      -----
```

You get both the name of the machine and its architecture. If you see the same machine name several times, this means that several distributed processes were launched on this machine. The working directory is also displayed in the report along with the type of files in use (32- or 64-bit). This type of file is automatically determined by the master machine. If the master machine is a 32-bit machine, then distributed processes will have to be 32-bit also. If the master is a 64-bit machine, then everything has to follow 64-bit conventions.

You may want to remove some machines from this list (for example because of an overloaded machine). In this case, you can use the remove distributed processors command; for example:

```

TEST> remove distributed processors zelda
TEST> report distributed processors
  Working directory ==> "/remote/atpg/dfs" (32bit)
  -----
  MACHINE:    nalpari [ARCH: sparc-64]
  -----

```

You can use the `report settings distributed` command to get a list of the current timeout and shell settings; for example:

```

BUILD> report settings distributed
distributed =      shell_timeout=30, slave_timeout=100,
                  print_stats_timeout=30, verbose=-noverbose,
                  shell=rsh;

```

Setting Up a Distributed Environment With Load Sharing

TetraMAX also supports the load sharing facility (LSF) and GRID network management tools. When you are using load sharing, jobs are submitted to a queue instead of to specific machines. The load sharing system manager then decides on which machine the job will be started. This allows you to maximize the usage efficiency of your network.

To populate the distributed processor list, you need to use the `add distributed processors` command to specify the absolute path to the LSF and GRID submission executables (`bsub`), as well as the number of slaves to be spawned and additional options. For using LSF to launch the slaves, all of these options must be specified. For using GRID to launch the slaves, all of these options must be specified as well as the `-script` option of the `set distributed` command. If you do not have any additional options to pass to `bsub`, you can pass empty options using `-options " "`. For descriptions of these options, see the online help for the `add distributed processors` command.

Here is an example:

```

BUILD> add distributed processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 4 \
-options "-q lb0202"
BUILD> report distributed processors
  Working directory ==> "/remote/dtgnat/Distributed"
(32bit)
  -----
  MACHINE    **lsf** [ARCH:      linux]
  MACHINE    **lsf** [ARCH:      linux]
  MACHINE    **lsf** [ARCH:      linux]
  MACHINE    **lsf** [ARCH:      linux]
  -----

```

Notice that instead of getting some distributed processors in the report, you see ****lsf****. This is because no job has been started yet, and thus, no distributed processor has been assigned to the job. Once you issue the `run atpg -distributed` command (or the `run fault_sim -distributed` command), four jobs will be assigned to four distributed processors. However, this will be transparent to you.

You cannot remove only one distributed processor from the list when you are using the LSF environment. If you simply want to change the current number of distributed processors in the pool, you have to issue a new `add distributed processors` command with the correct value for the `-nslaves` option. Every time you issue an `add distributed processors` command under the LSF environment, it overrides the previous definition of your distributed processor list. Here is an example:

```
BUILD> add distributed processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 3 \
-options "-q lb0202"
BUILD> report distributed processors
Working directory ==> "/remote/dtgnat/Distributed"
(32bit)
-----
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
MACHINE    **lsf** [ARCH:      linux]
-----
```

Starting Distributed Fault Simulation

Once the functional vectors are read into the TetraMAX external pattern buffer, you simply need to add the `-distributed` option to the `run fault_simulation` command to trigger the parallelization of the algorithm; for example:

```

TEST> run fault_sim -distributed
Master: Saving patterns for slaves ...
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 2 slaves .

..
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Slaves: About to read in patterns ...
Master: Removing temporary files ...
Master: Sending 98 faults to slaves ...
Master: End sending faults. Time = 0.00 sec.

-----
#patterns #collapsed faults test process
simulated inactive/active coverage CPU time
-----
Fault simulation completed: #patterns=32
Uncollapsed Path_delay Fault Summary Report
-----
fault class code #faults
-----
Detected DT 61
detected_by_simulation DS (2)
    detected_robustly DR (59)
    Possibly detected PT 0
    Undetectable UD 0
    ATPG untestable AU 33
        atpg_untestable-not_detected AN (33)
    Not detected ND 4
        not-controlled NC (4)
-----
total faults 98
test coverage 62.24%
fault coverage 62.24%
ATPG effectiveness 95.92%
-----
Pattern Summary Report
-----
#internal patterns 0
#external patterns (pat.bin) 32
    #full_sequential patterns 32
-----

```

Events After Starting A Distributed Run

First, the master machine writes an image of the database in the working directory. This image is a binary file containing everything the distributed processors need to know to process the fault simulation. This file can be rather large, because it is based on the size of your design, so as soon as the database is read by the slaves, it is deleted from the disk.

Next, the distributed processes are started (see the message in the example report shown in the next section). If something goes wrong at this step (problem with starting the slave processors), TetraMAX will notify you and stop.

After the distributed machines read the database, they are in the same state as the master with respect to the information about the design. The fault list is then split among the various processors and they all start to run concurrently. Whenever a slave processor finishes its job, it sends some information back to the master machine and then it shuts down. If any of the slaves unexpectedly dies during the process, the master machine will detect it and that process stops. An error message is issued to notify you. After every slave processor finishes, the master machine computes the fault coverage and prints out the final results.

Interpreting Distributed Fault Simulation Results

The transcript that follows shows the relevant information displayed during distributed fault simulation.

```

TEST> run fault_sim -distributed
Master: Saving patterns for slaves ...
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Slaves: About to read in patterns ...
Master: Removing temporary files ...
Master: Sending 98 faults to slaves ...
Master: End sending faults. Time = 0.00 sec.
-----
#patterns #collapsed faults test process
simulated inactive/active coverage CPU time
-----
Fault simulation completed: #patterns=32
Uncollapsed Path_delay Fault Summary Report
-----
fault class code #faults
-----
Detected DT 61
detected_by_simulation DS (2)
detected_robustly DR (59)
Possibly detected PT 0
Undetectable UD 0
ATPG untestable AU 33
atpg_untestable-not_detected AN (33)
Not detected ND 4
not-controlled NC (4)
-----
total faults 98
test coverage 62.24%
fault coverage 62.24%
ATPG effectiveness 95.92%
-----
Pattern Summary Report
-----
#internal patterns 0
#external patterns (pat.bin) 32
#full_sequential patterns 32
-----

```

Where:

#patterns simulated = The number of patterns simulated

#collapsed faults inactive = The number of faults TetraMAX has already processed

Starting Distributed ATPG

Distributed ATPG works in a similar way as distributed fault simulation. You simply have to add the `-distributed` switch on the `run atpg` command line to trigger a distributed job; for example:

```
TEST> run atpg -distributed -auto
```

The master process sends the fault information to the distributed processors in a collapsed format. Thus, all reports refer to the collapsed fault list.

If you have some vectors in the external buffer before starting distributed ATPG, they will be automatically transferred into the internal buffer. The new vectors created during distributed ATPG will be added to this existing set of vectors. After the run is complete, you will have both the external vectors and the ATPG created vectors in the internal pattern buffer. If you do not want to merge those sets, you have to clean the external buffer before starting distributed ATPG by issuing a `set pattern -delete` command.

As the following transcript shows, TetraMAX starts the various processes and issues some informational messages to keep you informed at the beginning of the run. A warning or error message is issued if TetraMAX cannot proceed. Then, TetraMAX starts generating the vectors.

```

run_atpg -auto -dist
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 2 slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Master: Removing temporary files ...
Master: Sending 5918 faults to slaves ...
Master: End sending faults. Time = 1.00 sec.
-----
#patterns #collapsed faults test process
    total      inactive/active coverage     CPU time
-----
Compressor unload adjustment completed:
#patterns_adjusted=241,
#patterns_added=0,
CPU time=1.00 sec.
    Uncollapsed Stuck Fault Summary Report
-----
fault class           code #faults
-----
Detected              DT   8109
Possibly detected     PT   0
Undetectable          UD   10
ATPG untestable       AU   28
Not detected          ND   11
-----
total faults          8158
test coverage         99.52%
fault coverage        99.40%
ATPG effectiveness    99.87%
-----
    Pattern Summary Report
-----
#internal patterns    243
-----

```

Where:

#patterns total = The total number of patterns generated up to this point.

#collapsed faults inactive = The number of collapsed faults already processed by TetraMAX.

#collapsed faults active = The number of collapsed faults not yet processed.

process CPU time = The time consumed up to this point.

At the end of the pattern generation process, TetraMAX automatically prints a summary for the faults and the vectors.

Saving Results

A sample script:

```
TEST> set fault -summary verbose
TEST> report faults -summary
TEST> report pattern -summary
TEST> write fault final.flt -all -collapsed \
-compress gzip -replace
TEST> write patterns final_pat.bin.gz \
-format binary -compress gzip -replace
TEST> write patterns final_patv \
-format verilog_single_file -replace
```

Distributed Processor Log Files

When you run distributed ATPG or distributed fault simulation, the tool creates a log file in the work directory for each slave. The name of this log file is derived from the name of the master log file appending a number to it. For example, if the master log file is defined with a `set message log run.log -replace` command, a command that indicates you are running distributed ATPG with four slaves, the log files that will be created would be called "run.log.1," "run.log.2," "run.log.3," and "run.log.4."

The tool creates the slave log files to give you visibility to the activity happening on the slaves.

Note that if you run distributed ATPG multiple times in the same session, the slave log files are overwritten by each run. If you want to prevent the slave log files from being overwritten, you can either save a copy or redefine the work directory by issuing a `set distributed -work_dir` command before starting a new distributed run.

Licensing Schemes

There are two different licensing schemes and ways to be run TetraMAX Distributed ATPG and TetraMAX Distributed Fault Simulation.

Scheme 1: You have a Test-Accelerate-Max license key.

A Test-Accelerate-Max license key allows you to run as many distributed processes as you want. Whenever you issue an `add distributed processors` command, TetraMAX looks for this key. If a Test_Accelerate-Max key is found, run `atpg` automatically looks for one Test-ATPG-Max key and a Test-Faultsim key. If TetraMAX cannot get a Test_Accelerate-Max key, the site is either not licensed for it, or all Test-Accelerate-Max keys are currently checked out by other TetraMAX sessions), TetraMAX switches to scheme 2.

Scheme 2: You do not have an available Test-Accelerate-Max key.

If a Test-Accelerate-Max key is not available, TetraMAX will checkout as many Test-ATPG-Max and Test-Faultsim licenses as the number of slave processors in the pool.

Licensing Examples

Example 1:

You have three processors in your pool and you want to run distributed ATPG. A Test-Accelerate-Max license key is available in your license pool. TetraMAX will checkout these license keys with respect to the command listed:

```
TEST> add distributed processors proc1 proc2 proc3  
      => 1 Test-Accelerate-Max  
TEST> run atpg -distributed  
      => 1 Test-ATPG-Max (or 1 Test-ATPG)  
      => 1 Test-Faultsim
```

Example 2:

You have three processors in your pool and you want to run distributed fault simulation. You do not have access to a Test-Accelerate-Max key. TetraMAX will checkout these license keys with respect to the command listed:

```
TEST> add distributed processors proc1 proc2 proc3  
      => Nothing checked-out, but a warning is issued  
TEST> run fault_sim -distributed  
      => 3 Test-Faultsim
```

Limitations

The following are not supported for distributed ATPG are

- The `-analyze_untestable_faults` option of the `set atpg` command
- N-detect ATPG and fault simulation

19

Diagnosing Manufacturing Test Failures

When a device fails testing, you can use TetraMAX to determine what caused the failure. To do this, you put the failure information into a file (following a specified format) and then invoke the `run diagnosis` command. This command analyzes the failure information and reports the locations and types of faults that could have caused the failure.

This chapter contains the following sections:

- [Providing Tester Failure Log Files](#)
- [Pattern File](#)
- [Diagnosing Tester Failure](#)
- [Understanding the Diagnosis Summary](#)
- [Reading Physical Data](#)

Providing Tester Failure Log Files

Before you use the `run diagnosis` command, you need to have the failure information in a failure log file. Some testers can directly create the failure log file in TetraMAX format.

TetraMAX supports two types of failure log files: pattern-based or a cycle- or vector-based file. Both of these log files are described in the following subsections.

Note the following:

- The `set diagnosis` command includes an option, `-failure_memory_limit <d>`, that helps ease the failure log file truncation task. This option enables you to specify the maximum number of failures that can be captured by the tester; it then lets TetraMAX automatically truncate the patterns considered during diagnosis.
- A pattern-based failure log file can also be created using Verilog DPV simulation using the pre-defined VCS option `+tmax_diag`. For details, refer to “Additional Verilog Options” section in the “Using Verilog DPV Testbench” chapter of the *Test Pattern Validation User Guide*.

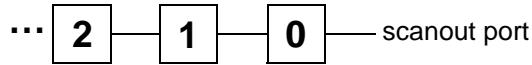
Providing a Pattern-Based Failure Log File

The failure log file is an ASCII text file. Each line in the file describes a pattern in which the output values detected by the test equipment did not match the expected values. This is the format of each line

```
<pattern_num> <output_port> [cell_position] [expected_data]
```

Where:

- `pattern_num` is the TetraMAX pattern number in which the failure occurred, starting with 0 for the first pattern.
- `output_port` is the name of the output port at which the failure was detected, or the scan chain name when the pin name is shared among scan groups.
- `cell_position` must be provided if the failure occurred during a scan shift cycle. It is the number of tester shift cycles that have occurred since the start of the scanout process. From this value, TetraMAX determines the position of the scan chain cell that captured the erroneous data. The cell position of the scan chain cell closest to the output port is 0, the next one is 1, and so on; for example:



- `expected_data` is an optional argument that describes the expected value of the measured failure. It is used with the `-check_expected_data` option of the `set diagnosis` command. Its value is either 0 or 1.

Note: In order to indicate the expected data during a capture of a scan chain output and use `-check_expected_data` option, then you must use the `exp=` syntax. For example:

```
103 scan_out3 1 //invalid
103 scan_out3 (exp=1, got=0) //valid
103 scan_out3 exp=1 //valid
```

Any line in the failure log file that begins with two slash characters is considered a comment line.

[Example 19-1](#) shows a tester data file. In this example, five failing patterns are reported: pattern numbers 3, 4, 10, 11, and 12.

Example 19-1 Tester Failure Data

```
// pattern 3, port REQRDYO
3 REQRDYO

// pattern 4, port MA[9], scan chain 'c9', 30 shifts
4 MA[9] 30

//pattern 10-12, port NRD, scan chain 'c29', 3 shifts
10 NRD 3
11 NRD 3
12 NRD 3
```

Providing a Cycle-Based Failure Log File

The preceding section described the pattern-based failure log file. Most ATE vendors generate failure data directly in this format. However, some older testers are not supported. For such testers, you can use a cycle-based (or vector-based) failure log file format (TetraMAX patterns contain multiple cycles/vectors). Cycle-based failure log files in TetraMAX format should be simpler to generate than pattern-based failure log files.

The pattern file must be in STIL or WGL format. TetraMAX does not support binary pattern format for cycle-based failure log file diagnosis (but, it supports the binary pattern format for pattern-based failure log files).

If the external pattern buffer contains an unsupported pattern format, TetraMAX displays an error message when you execute the `run diagnosis` command with a cycle based failure log file.

TetraMAX supports Basic-Scan and Fast-Sequential patterns.

Cycles (V statements in STIL, vector statements in WGL) are counted when you read patterns using the `set patterns -external` command. This count identifies the vectors at pattern boundaries as well as when shift cycles start within each pattern. If you or the tester make adjustments that cause the failing cycle/vector to deviate from the corresponding vectors in the STIL/WGL patterns used for diagnosis (such as combining multiple STIL/WGL vectors into a single tester cycle), you must make a corresponding change in the cycle-based failure log to map back to the vectors in the pattern file.

The `set diagnosis` options associated with the cycle-based failure log file are:

- `-CYcle_offset <integer>` — You can use this option to adjust the cycle count when the cycle numbering does not start at 1.
- `-Verbose` — This option causes the translated pattern-based failure log file to be reported by the `run diagnosis` command.

For more information on these options, see TetraMAX online Help for the `set diagnosis` command.

The following example shows sample output. Comments indicate the failure cycle used to generate the pattern-based failure. It also shows whether the cycle was a capture or a shift cycle.

```
4 po0  # Cycle conversion from cycle 34; fail in capture
4 so 2  # Cycle conversion from cycle 38; fail in
shift
```

The following is an example flow:

```
run drc ...
set patterns -external pat.stil
set diagnosis -cycle_offset 1
run diagnosis fail.log
```

Cycle-based Failure Log File Format

The failure log file still contains only failed cycles. The format of each line is as follows:

```
C <output_name> <cycle> [<expected_value>]
```

Where

- `C` is the first character on the line, indicating that the line specifies tester cycles and not TetraMAX pattern numbers. It helps you identify the type of failure log file (pattern- or cycle-based).
- `output_name` is a string that can be a PO or a scan chain name (no output compressor).
- `cycle` is an integer indicating the cycle in the external pattern set that failed on the tester; when the first cycle is cycle 1, not 0. TetraMAX expects failures only in cycles where measurements occur; for example, during shift or capture cycles. Invalid failure cycles can provide inaccurate diagnosis results.
- `expected_value` is an optional 0 or 1, depending on what the pattern specified as the expected value.

TetraMAX ignores all other characters in the line. It treats them as comments.

Limitations

TetraMAX supports only STIL and WGL patterns with cycle-based failure log files. It does not support binary patterns.

Only WGL patterns written by TetraMAX releases X-2005.06 and later support proper indexing of failures for the last shift. Pre-existing WGL patterns require manual modification.

TetraMAX does not support the `-truncate` option of the `run diagnosis` command with cycle based diagnosis.

Pattern File

Diagnosis requires either a single pattern file corresponding to the patterns that were run on the tester when the device failed or the entire set of split patterns file along with their failures files as explained in the “[Split Patterns File](#)” section below.

The binary pattern format is best, but STIL or WGL are ok, with the risk that the pattern set might be misinterpreted due to language limitations.

Split Patterns File

There may be times when you need to create multiple TetraMAX pattern files (split after ATPG) to run on the tester for a particular design. Each pattern file is typically run individually, in separate test programs, on the tester. When a device fails, the tester generates one failure log file per TetraMAX pattern file. At the end of all tests, you will have multiple pattern files and its corresponding failure log file.

TetraMAX diagnosis can read multiple pattern files and multiple fail data files so that you can get a single result from a single diagnosis run.

There can be as many failure log files as there are pattern files. A failure log file is expected to contain the failures for only the patterns in the corresponding pattern file. Otherwise, an error is generated.

If there are no failures for any of the patterns in a particular pattern file, the corresponding failure log file can be non-existent.. The correspondence between pattern files and failure log files is specified by a required directive in the failure log file, as explained “Failure Data File Format” topic in TetraMAX OLH. An error is generated otherwise.

To read multiple patterns file, use the following command (see TetraMAX On-Line Help for more details).

```
set patterns external <file> -split_patterns
```

When split pattern files are read, you will need to supply multiple failure log files for the diagnosis run.

For native mode, the command is shown bellow:

```
TEST> run diagnosis fail1.log fail2.log ... [options]
```

In Tcl mode, the second and following failure log files should be specified using the -file option.

By default, TetraMAX considers that the cycle count recorded in the failures file in cycle-based format is reset to 1 from the execution of one pattern set to the next one. The directive .cycle_offset could be used to change this behavior, as explained in the “Failure Data File Format” topic in TetraMAX OLH.

Translating Adaptive Scan Patterns Into Normal Scan Patterns

If your design has Adaptive Scan technology, you can perform diagnosis on the patterns that include compression, or create patterns that bypass the output compression. Diagnosing patterns in compressor mode could reduce diagnostic resolution due to compressor effects.

After a device in compressor mode fails on the tester, if the diagnostic resolution is not high enough, you can retest it in the scan mode. The translated patterns detect the same defects, but diagnostic resolution is higher because the compressor no longer affects the unloaded values.

The process involves writing out a special netlist-independent version of the pattern in binary format along with the Adaptive Scan pattern set. The netlist-independent pattern file contains a mapping of the scan cells and primary inputs to their ATPG generated values. This pattern set can be read back into TetraMAX after design is put into the reconfigured scan mode by reading the scan mode SPF. After the patterns are read back, a simulation is done internally to compute the expected values to complete the translation process. The internal patterns can then be written out for the tester to be used in scan mode for diagnosis.

Example Flow

The following example illustrates the translation steps:

1. Read the design with Adaptive Scan in compressor mode and write out the netlist independent pattern format.

```
run build ...
# read SPF for adaptive scan mode
run drc scan_compression.spf
run atpg -auto
# write out adaptive scan mode patterns
write patterns compressed_pat.bin -format binary
# write netlist independent patterns that can be translated
set pat -netlist_independent
write patterns compressed_pat.net_ind.bin
```

2. Read the design with Adaptive Scan but in scan mode. Translate the patterns into scan mode.

```
run build ...
# read SPF for normal scan mode
run drc scan.spf
# read netlist independent patterns
set pat ext compressed_pat.net_ind.bin
# optional sanity check to verify that simulation passes
run simulation
# write out translated patterns to be re-run on the tester
write patterns scan_pat.pats -external -format <any format>
# write out translated patterns in binary format for
diagnostics
write patterns scan_pat.bin -external -format binary
```

Limitations

The following are the limitations of the Adaptive Scan compression to normal scan mode translation feature:

- Translation is one-way. You cannot translate scan patterns to compression mode.
- This feature does not support PLL.
- It supports only Basic-Scan and Fast-Sequential pattern (similar to diagnosis).
- Configuration differences between compressor mode and scan mode might result in slightly different coverage numbers.

Diagnosing Tester Failure

To diagnose the tester failure data in TetraMAX, do the following:

1. Establish the original ATPG environment under which the patterns were created, such as reading in the design and running DRC. For more information, see “[Setting Up Advanced ATPG Features](#)” on page 4-35.
2. Read in the original patterns used to detect the failure. You can use patterns generated by the Basic-Scan and Fast-Sequential modes, but not the Full-Sequential mode.

When reading patterns into TetraMAX, use binary formats whenever possible. Other pattern format such as WGL and STIL have limited facilities to accurately store all data about the patterns. Therefore, upon reading back a STIL or WGL pattern file, errors such as a Fast-Sequential pattern being interpreted as a Full-Sequential pattern are possible.

An optional sanity check to verify that simulation passes with the patterns read can be performed. In this case, simply add the `run simulation` command before doing the diagnosis.

Note:

To avoid bad patterns on the tester, the expected response in later releases of TetraMAX may produce more pessimistic simulation results than earlier versions. If TetraMAX diagnostics are run with patterns from an earlier version of TetraMAX ATPG, the recommended flow is to use the `run simulation -update -store` command to update the patterns by masking the mismatching measures. (See the description of `run simulation` command in TetraMAX online Help for further details on these options.) Then, save the new patterns and use them for the diagnostic.

3. Start the diagnosis by using the Run Diagnosis dialog box, or by entering the `run diagnosis` command at the command line.

TetraMAX determines the cause of the failing patterns and generates a diagnosis report. By default, the diagnostics search for defects in functional logic. If pattern 0 is the chain test pattern and it fails, then chain diagnosis is performed. See [“Performing Scan Chain Diagnosis” on page 19-10](#). for details.

Diagnose tester failure with Adaptive Scan compression technology is also supported for logic diagnosis only. One of the first actions performed by the Diagnosis is to map each individual failure to a scan cell. You can use the `-mapping_report` option of the `set diagnosis` command to produce a detailed failure mapping report. For details, see the description of the `set diagnosis` command in TetraMAX Online Help. Nevertheless, note that in some cases the failures might not be mapped successfully.

If your diagnostic resolution is low, you can create patterns that bypass the output compressor. For more information, see [“Translating Adaptive Scan Patterns Into Normal Scan Patterns” on page 19-6](#).

Note:

When using Adaptive Scan Diagnosis, the chain diagnosis is not supported in compression mode.

Using the Run Diagnosis Dialog Box

To start the diagnosis using the Run Diagnosis dialog box, follow these steps:

1. Click the Diagnosis button in the command toolbar at the top of the TetraMAX main window. The Run Diagnosis dialog box appears.
2. Fill in the dialog box.

For descriptions of these controls, see online Help for the `run diagnosis` (and `set diagnosis`) command(s).

3. Click OK.

Using the `run diagnosis` Command

You can also start the diagnosis using the `run diagnosis` command. For example:

```
TEST> run diagnosis /project/mars/lander/ \
    chipA_failure.dat -display
```

For the complete syntax and option descriptions, see the online help for the `run diagnosis` command.

Performing Scan Chain Diagnosis

Functional logic diagnosis assumes the data load and unload are working correctly. But if your patterns show failures during the chain test, then you might have a device failure that prevents the proper loading and unloading of the scan chains. The scan chain diagnosis engine is designed to detect defects that affect scan chain shifting.

Both standard and adaptive scan patterns can be used for scan chain diagnosis. ATPG automatically produces additional chain test patterns for DFT MAX X-tolerant designs if lower diagnostics accuracy is expected because of an R10 violation.

You run scan chain diagnosis engine using the `-chain_failure` option of the `run_diagnosis` command. Any faults that affect shifting can be diagnosed by this engine, including non-operating clock pins or reset lines stuck-at an active value.

The diagnostic algorithm identifies the location of stuck-at, slow-to-rise, slow-to-fall, fast-to-rise or fast-to-fall fault. The latter two types cover hold-time problems affecting the scan chain shift operation.

For best accuracy, failure data from ten or more patterns is required. It is recommended that you provide TetraMAX as many failures as possible — not just the failures that occur during chain test pattern.

In order to isolate the location of the defect, the basic approach of scan chain diagnosis is to compare the control and observe ability of scan cells. For example, assume that a stuck-at fault prevents the shifting from scan cell A to B. In this case, scan cell A, and all cells located before it, will drive valid values to functional logic; however, they will appear as tied cells when unloaded, and thus are unobservable. Scan cell B and the cells that follow it will drive invalid values to functional logic, but they may capture valid values that can be observed.

The output displays a set of possible defect locations (chain, cell position, and instance name), along with a score, which indicates the confidence of each location. This score is a percentage that measures the degree to which a failure seen on the tester matches a simulated chain defect at that location. It also shows the predicted type of defect. For example:

```
fail.log scan chain diagnosis results: #failing_patterns=79
-----
defect type=fast-to-rise
match=100% chain=c0 position=178 master=CORE/c_rg0 (46)
match=100% chain=c0 position=179 master=CORE/c_rg2 (57)
match= 98% chain=c0 position=180 master=CORE/c_rg6 (54)
```

The example report indicates that a fast-to-rise defect is most likely the root cause of the failures. Three scan cells locations that have an output with the physical defect are then listed.

Understanding the Diagnosis Summary

[Example 19-2](#) shows a typical diagnosis summary produced by the `run diagnosis` command.

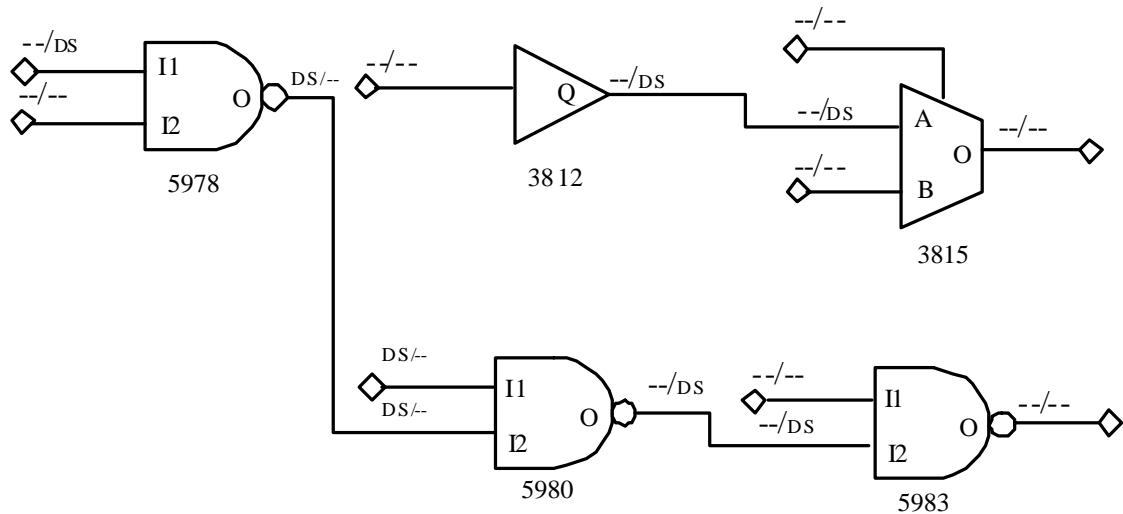
Example 19-2 Diagnosis Summary

```
TEST> run diagnosis /project/mars/lander/chipA_failure.dat -display
Diagnosis summary for failure file /project/mars/lander/chipA_failure.dat
#failing_pat=4, #failures=5, #defects=2, #faults=3, CPU_time=0.05
Simulated : #failing_pat=4, #passing_pat=35, #failures=5
-----
Fault candidates for defect 1: stuck fault model, #faults=1, #failing_pat=3,
#passing_pat=36, #failures=3
-----
match=100.00%, #explained patterns: <failing=3, passing=36>
sal   DS   de_d/data3_reg_0/Q    (S003)
sal   --   de_d/U211/A    (SELX2)
-----
Fault candidates for defect 2: stuck fault model, #faults=2, #failing_pat=2,
#passing_pat=37, #failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=37>
sal   DS   de_encrypt/C264/U36/O    (L434ND)
sa0   --   de_encrypt/C264/U36/I1    (L434ND)
sa0   --   de_encrypt/C264/U36/I2    (L434ND)
sa0   --   de_encrypt/C264/U28/O    (L434ND)
sal   --   de_encrypt/C264/U26/I2    (L434ND)
-----
match=50.00%, #explained patterns: <failing=1, passing=37>
sal   DS   de_encrypt/C264/U28/I1    (L434ND)
```

The summary shows that the four failing patterns in the failure log file were resolved to two defects. The first defect came from three failing patterns and was resolved to one fault location and its fault-equivalent location. The second defect came from two failing patterns and was resolved to two fault locations. The first fault location has a 100% match score and has four faults-equivalents. The second fault location of second defect has a 50% match score.

By using the `-display` option of the `run diagnosis` command, or by checking the Display Results in Viewer check box in the Run Diagnosis dialog box, you can display the instances and fault locations graphically, as shown in [Figure 19-1](#). For this schematic, the pin display data format has been set to Fault Data, where the format is stuck-at-0/stuck-at-1.

Figure 19-1 Diagnosis Data Displayed Graphically



You can identify each of the defect locations by the DS (detected by simulation) code on the pin corresponding to either a fault site or a fault equivalent. The DS notation marks all potential fault sources that could cause the same failing data pattern. The notation **DS/-** indicates that a stuck-at-0 fault at that point in the design would cause the failure, and the notation **--/DS** indicates that a stuck-at-1 at that point in the design would cause the failure. TetraMAX shows all potential failure sites that would cause the same failure data patterns.

In this example, the diagnosis by TetraMAX finds two independent areas of failure in the design. The GSV display of [Figure 19-1](#) shows the two corresponding independent groups of logic. According to the diagnosis, the faulty circuit location for each failure is along the path displayed.

Displaying Composite Fault Types

You can use the `-composite` option of the `set diagnosis` command to report the following composite fault types:

- **sa01** — The fault location can behave as a stuck-at 0 on some patterns and a stuck-at 1 on others. This could be a coupled open defect or a bridge type defect. On nets with fanout branches, it is possible for it to appear as stuck-at 0 on some patterns and stuck-at 1 on others. For ranking, this fault model can produce optimistic scores.
- **strf** — The fault location can cause a delay on both rising and falling transitions (slow-to-rise-fall). The traditional fault models of **str** and **stf** are unidirectional.
- **bAND** or **bOR** — The defect location behaves as a wired-AND or wired-OR type bridge. Both nodes of the bridging fault are simulated and reported.

- bDOM — The defect location behaves as the victim node of a dominant bridge. The fault is simulated at the fault site only for failing patterns. Note that this may result in optimistic ranking scores since this model will always match the tester for passing patterns. Only the victim node is reported.
- bUN — An unknown bridging type relationship exists. It indicates that there is some signal correlation which does not behave as the well known bridging fault models. An example can be where two bridged nodes fail, however the corresponding nodes in the TetraMAX design do not necessarily have opposite values for all failing pattern. This phenomenon can result in situations with cell defects, analog effects, or build optimizations. Both nodes are reported.

Reading Physical Data

TetraMAX can read a Milkyway database (Synopsys physical database) associated with the design that passed DRC. After it reads the database, TetraMAX displays the corresponding physical data for the resulting fault candidates in its diagnosis output. The Milkyway database consists of a library that holds all the cell views of the design. There is a cell in the library matching the top-level module in TetraMAX. The library cells in TetraMAX also have a corresponding cell in Milkyway. TetraMAX can identify the missing correspondences. Physical data for corresponding objects is stored for each TetraMAX primitive.

To read the Milkyway database, use the `read layout` command.

```
READ LAYOUT <lib_name> <-Cell_name <cell_name>> [-Version <d>]
[-MAX_mismatch_report <d>] [-Debug]
```

Note:

The `read layout` command is supported only on sparcOS5, sparc64, hp64, amd64 and linux platforms.

- `lib_name` — Milkyway library name. This is the name of the directory in which the Milkyway files are stored.
- `-Cell_name <cell_name>` — Top-level cell name. The CEL directory of the Milkyway database has at least one file with this prefix.
- `-Version <d>` — Version of the cell to be loaded. The default is 0, which indicates the latest version. This argument is optional.
- `-MAX_mismatch_report <d>` — Maximum number of layout and TetraMAX instance mismatches to report. The default is 0. This argument is optional.

- `-Debug` — Causes additional debug data to be stored when running the `read layout` command. It can be used if you find that some TetraMAX design objects were not mapped to objects in the layout. You can use the `report layout` command to help debug the cause, if this option is used beforehand. Debug data is not stored in the image file.

The `read layout` command sends any messages from the Milkyway database parser to the standard output of the TetraMAX process. These messages are related to the physical implementation and might not be useful for TetraMAX users. TetraMAX does not store the output in its log file.

Understanding the `read layout` Command Output

An example of the TetraMAX output after reading the Milkyway database is the following:

```
-----
End read_layout: CPU_time=1.37 Memory=115.36MB
Milkyway : #modules=293 (missing=3), #instances=71142
(missing=12), #module_pins=1209 (missing=11)
TetraMAX : #modules=293 (missing=1), #instances=67182
(missing=7), #primitives=99006 (missing=23)
-----
```

The following are the parameters and descriptions in the `read layout` command output.

TetraMAX shows the number of objects for which it does not find any correspondence as the "missing" number. The Master cell is the cell definition; not a cell instance.

`modules` — Master cells in the Milkyway database or module definitions in Verilog.

`instances` — Cell instances.

`module_pins` — Total number of signal pins used in TetraMAX modules.

`primitives` — TetraMAX primitives.

If you use the `-max_mismatch_report` option, TetraMAX displays additional output that indicates the layout or TetraMAX objects that do not have a correspondence such as:

```
Corresponding TetraMAX object of layout object cpu/f2
(fetch_u) not found
Corresponding layout object of TetraMAX instance i0/f432
(DFSX3) not found
Corresponding layout object of TetraMAX module AN3X4 not
found
```

Viewing Physical Data in the Diagnosis Report

To see physical data in the diagnosis output (when available), use the following commands:

```
set diagnosis -LAyout_data  
run diagnosis
```

The physical data is listed immediately after each fault candidate. All coordinates are in nanometers (nm) and relative to the origin for the cell; usually the lower-left corner.

```
sa0 DS tlb8/tlbhit/U374/A1 (OAIX2)  
Pin_data: X=1830005 Y=1776660, Layer: METAL1 (31)  
Cell_boundary: L=1829880 R=1833560 B=1774890 T=1778580
```

Where:

Pin_data: X — Horizontal coordinate of one of the vertices of the pin associated with the location of the fault candidate.

Pin_data: Y — Vertical coordinate of one of the vertices of the pin associated with the location of the fault candidate.

Pin_data: Layer — Physical layer where the pin object is defined.

Cell_boundary: L — Horizontal coordinate of the leftmost boundary of the cell identified with the fault candidate.

Cell_boundary: R — Horizontal coordinate of the rightmost boundary of the cell identified with the fault candidate.

Cell_boundary: B — Vertical coordinate of the bottommost boundary of the cell identified with the fault candidate.

Cell_boundary: T — Vertical coordinate of the topmost boundary of the cell identified with the fault candidate.

If you run the `write image` command after the `read layout` command, TetraMAX stores the physical data in the image file. If you run the `read image` command and then run the `run diagnosis` command, TetraMAX generates physical data (you need not run the `read layout` command). You can use this feature to provide physical data to a foundry without the complete Milkyway database, thereby protecting your intellectual property. The following is a sample flow for creating the image:

```
build -force
read netlist
run drc
# read library "mw_cpu" and top cell "cpu"
read layout mw_cpu -cell cpu
# store physical data in TetraMAX image file
write image image.gz -compress gzip -replace
```

Adding Netlist Data In Image Files

To add the nets connected to fault sites to image files, use the following command:

```
write image <file> -NETlist_data
```

The recipient of the image can enter the `read image` command to restore the net names in the design. Commands involving net names such as `report nets` or `set diagnosis -report_net_data` are functional after TetraMAX reads in an image with netlist data. Image file size increases to store the netlist data.

20

Troubleshooting

This chapter describes some troubleshooting tips and techniques.

This chapter contains the following sections:

- [Reporting Port Names](#)
- [Reviewing a Module Representation](#)
- [Rerunning Design Rule Checking](#)
- [Troubleshooting Netlists](#)
- [Troubleshooting STIL Procedures](#)
- [Analyzing the Cause of Low Test Coverage](#)
- [Completing an Aborted Bus Analysis](#)

Reporting Port Names

To verify the names of top-level ports, you can obtain a list of the inputs, outputs, or bidirectional ports for the top level of the design using these commands:

```
DRC> report primitives -pis
DRC> report primitives -pos
DRC> report primitives -pios
DRC> report primitives -ports
```

To obtain the names of ports for any specific module, use the following command:

```
DRC> report modules module_name -verbose
```

[Example 20-1](#) shows a verbose report produced by the `report modules` command. The names of the pins are listed in the Inputs and Outputs sections.

Example 20-1 Verbose Module Report

```
TEST> report modules INC4 -verbose
          pins
module name      tot( i/ o/ io)    inst     refs(def'd)   used
-----  -----
INC4           11( 5/ 6/ 0)      10        1 (Y)         1
  Inputs : A0 ( ) A1 ( ) A2 ( ) A3 ( ) CI ( )
  Outputs: S0 ( ) S1 ( ) S2 ( ) S3 ( ) CO ( ) PR ( )
  PROP1  : and conn=( O:PROP I:A0 I:A1 I:A2 I:A3 )
  HADD0S  : xor conn=( O:S0 I:A0 I:CI )
  HADD1S  : xor conn=( O:S1 I:A1 I:C0 )
  HADD2S  : xor conn=( O:S2 I:A2 I:C1 )
  HADD3S  : xor conn=( O:S3 I:A3 I:C2 )
  HADD0C  : and conn=( O:C0 I:A0 I:CI )
  HADD1C  : and conn=( O:C1 I:A1 I:C0 )
  HADD2C  : and conn=( O:C2 I:A2 I:C1 )
  CARRYOUT: and conn=( O:CO I:PROP I:CI )
  buf9   : buf conn=( O:PR I:PROP )
```

Reviewing a Module Representation

To review the internal representation of a module definition, you use the `report modules` command with the name of the module and the `-verbose` option. Alternatively, you can use the `run build_model` command and specify the name of the module as the top-level design.

You might want to review the internal representation of a library module in TetraMAX if errors or warnings are generated by the `read netlist` command. For example, suppose that you use the `read netlist` command to read in the module `csdff`, whose truth table definition is shown in [Example 20-2](#), and the command generates the warning messages shown in [Example 20-3](#).

Example 20-2 Truth Table Logic Model

```
primitive csdff (Q, SDI, SCLK, D, CLK, NOTIFY);
    output Q; reg Q;
    input SDI, SCLK, D, CLK, NOTIFY;
    table
        // SDI SCLK D  CLK  NR  : Q-  : Q+
        // --- --- --- --- : --- : ---
        ? 0 0 (01) ? : ? : 0 ; // clock D=0
        ? 0 1 (01) ? : ? : 1 ; // clock D=1
        0 (01) ? 0 ? : ? : 0 ; // scan clock SDI=0
        1 (01) ? 0 ? : ? : 1 ; // scan clock SDI=1

        ? 0 * 0 ? : ? : - ; // hold
        * 0 ? 0 ? : ? : - ;
        ? 0 ? 0 ? : ? : - ;
        ? 0 ? (?0) ? : ? : - ;
        ? (?0) ? 0 ? : ? : - ;
        ? 0 ? ? * : ? : x ; // force to X
    endtable
endprimitive
```

Example 20-3 Read Netlist Showing Warnings

```
BUILD> read netlist csdff.v
Begin reading netlist ( csdff.v )...
Warning: Rule N15 (incomplete UDP) failed 64 times.
Warning: Rule N20 (underspecified UDP) failed 2 times.
End parsing Verilog file test.v with 0 errors;
End reading netlist: #modules=1, top=csdff, #lines=25,
CPU_time=0.01 sec
```

To review the model,

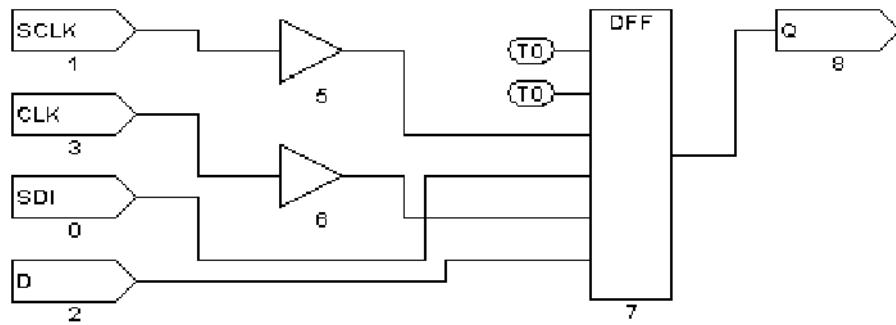
1. Execute the following command:

```
BUILD> run build_model csdff
```

2. Click the SHOW button in the Graphical Schematic Viewer (GSV) toolbar, and from the pop-up menu, choose ALL.

A schematic similar to [Figure 20-1](#) appears, allowing you to examine the ATPG model.

Figure 20-1 Module Showing Correct Interpretation



Do not be concerned if the schematic shows extra buffers. During the model building process, TetraMAX inserts these buffers wherever there is a direct path to a sequential device from a top-level port. These buffers are not present in instantiations of the module in the design.

Rerunning Design Rule Checking

The file specified in the `run drc` command is read each time the design rule checking (DRC) process is initiated. You can quickly test any changes that you make to this file by issuing another `run drc` command, as follows:

```
DRC> run drc myfile.spf
// pause here for edits to DRC file
TEST> drc -force
DRC> run drc
```

Troubleshooting Netlists

Here are some tips for troubleshooting problems TetraMAX might encounter while reading netlists:

- For severe syntax problems, start troubleshooting near the line number indicated by the TetraMAX error message.
- Focus on category N rules; these cover problems with netlists.
- To see the number of failures in category N, execute the `report rules n -fail` command.
- To see all violations in a specific category such as N9, execute the `report violations n9` command.

- To see violations in the entire category N, execute the `report violations n` command.
- Netlist parsing stops when TetraMAX encounters 10 errors. To increase this limit, execute the `set netlist -max_errors` command.
- When reading multiple netlist files using wildcards in the `read netlist` command, to determine which file had a problem, reread the files with the `-verbose` option and omit the `-noabort` option.
- Extract the problematic module definition, save it in a file, and attempt to read in only that file.
- Consider the effect of case sensitivity on your netlist, and explicitly set the case sensitivity by using the `-sensitive` or `-insensitive` option with the `read netlist` command.
- Consider the effect of the hierarchical delimiter. If necessary, change the default by using the `-hierarchy_delimiter` option of the `set build` command. Then reread your netlists.

Troubleshooting STIL Procedures

Problems in the procedures defined in the STIL procedure file (SPF) can be either syntax errors or DRC violations. Syntax errors usually result in a category V (vector rule) violation message, and TetraMAX reports the line number near the violation.

To fix the problem, open the SPF with an editor, make any necessary changes, and use the `run drc` command again to verify that the problem was corrected. For detailed descriptions and examples of the STIL procedures, see [Chapter 9, “STIL Procedure Files.”](#)

A general tip for troubleshooting any of the SPF procedures is to click the ANALYZE button in the GSV toolbar and select the applicable rule violation from the Analyze dialog box. TetraMAX draws the gates involved in the violation and automatically selects an appropriate pin data format for display in the schematic. To specify a particular pin data format, click the SETUP button and select the Pin Data Type in the Setup dialog box. For more information on pin data types, see [“Displaying Pin Data” on page 7-17.](#)

STIL load_unload Procedure

When you analyze DRC violations TetraMAX encountered during the `load_unload` procedure, the GSV automatically sets the pin data type to Load. With the Load pin data type, strings of characters such as `10X11{ }11` are displayed near the pins. Each character corresponds to a simulated event time from the vectors defined in the `load_unload` procedure. The curly braces indicate where the `Shift` procedure is

inserted as many times as necessary. Thus, the last value before the left curly brace is the logic value achieved just before starting the `Shift` procedure. The values following the right curly brace are the simulated logic values between the last `Shift` procedure and the end of the `load_unload` procedure.

Here are some guidelines for effectively using the `load_unload` procedure:

- Set all clocks to their off states before the `Shift` procedure.
- Enable the scan chain path by asserting a control port (for example, `scan_enable`).
- Place any bidirectional ports that operate as scan chain inputs into input mode.
- Place any bidirectional or three-state ports that operate as scan chain outputs into output mode, and explicitly force the ports to Z.
- Set all constrained ports to values that enable shifting of scan chains.
- Place all bidirectional ports into a nonfloating input mode if this is possible for the design.

STIL Shift Procedure

When you analyze DRC violations encountered during the `Shift` procedure, the GSV automatically sets the displayed pin data type to Shift. In the Shift pin data type, logic values such as 010 are displayed. Each character represents a simulated event time in the `Shift` procedure defined in the SPF.

Here are some guidelines for the test cycles you define in the `Shift` procedure:

- Use the predefined symbolic names `_si` and `_so` to indicate where scan inputs are changed and scan outputs are measured.
- If you want to save patterns in Waveform Generation Language (WGL) format, describe the `Shift` procedure using a single cycle.
- Remember that state assignments in STIL are persistent for a multicycle `Shift` procedure. Therefore, when you place a `CLOCK=P` to cause a pulse, that setting continues to cause a pulse until `CLOCK` is turned off (`CLOCK=0` for a return-to-zero port, or `CLOCK=1` for a return-to-one port).

[Example 20-4](#) shows a `Shift` procedure that contains an error. The first cycle of the shift applies `MCLK=P`, which is still in effect for the second cycle. As the `Shift` procedure is repeated, both `MCLK` and `SCLK` become set to P, which unintentionally causes a pulse on each clock on each cycle of the `Shift` procedure.

Example 20-4 Multicycle Shift Procedure With a Clocking Error

```
"load_unload" {
    V { MCLK = 0; SCLK = 0; SCAN_EN = 1; }
    Shift {
        V { _si=##; _so=##; MCLK=P; }
        V { SCLK=P; } // PROBLEM: MCLK is still on!
    }
}
```

[Example 20-5](#) shows the same Shift procedure with correct clocking. As the Shift procedure is interactively applied, MCLK and SCLK are applied in separate cycles. An additional SCLK=0 has been added after the Shift procedure, before exiting the load_unload, to ensure that SCLK is off.

Example 20-5 Multicycle Shift Procedure With Correct Clocking

```
"load_unload" {
    V {
        MCLK = 0; SCLK = 0; SCAN_EN = 1;
    }
    Shift {
        V { _si=##; _so=##; MCLK=P; SCLK=0; }
        V { MCLK=0; SCLK=P; }
    }
    V { SCLK=0; }
}
```

[Example 20-6](#) shows the same Shift procedure converted to a single cycle. The procedure assumes that timing definitions elsewhere in the test procedure file for MCLK and SCLK are adjusted so that both clocks can be applied in a non-overlapping fashion. Thus, the two clock events can be combined into the same test cycle.

Example 20-6 Multicycle Shift Converted to a Single Cycle

```
"load_unload" {
    W "TIMING";
    V { MCLK = 0; SCLK = 0;; SCAN_EN = 1; }
    Shift {
        V { _si=##; _so=##; MCLK=P; SCLK=P; }
    }
    V { MCLK=0; SCLK=0; }
}
```

STIL test_setup Macro

When you analyze DRC violations encountered during the test_setup macro, the graphical schematic viewer automatically sets the displayed pin data type to Test Setup. In the Test Setup pin data type, logic values in the form xx1 are displayed. Each character represents a simulated event time in the test_setup macro defined in the SPF.

Here are some rules for the test cycles you define in the `test_setup` macro:

- Force bidirectional ports to a Z state to avoid contention.
- Initialize any constrained primary inputs to their constrained values by the end of the procedure.
- Pulse asynchronous set/reset ports or clocking in a synchronous set/reset only if you want to initialize specific nonscan circuitry.
- Place clocks and asynchronous sets and resets at their off states by the end of the procedure. Note that it is not necessary to stop Reference clocks (including what DFT Compiler refers to as ATE clocks). All other clocks still must be stopped.

Correcting DRC Violations by Changing the Design

If you cannot correct a DRC violation by adjusting one of the SPF procedures, defining a primary input constraint, or changing a clock definition, the violation is probably caused by incorrect implementation of ATPG design practices, and a design change might be necessary. Note that a design can be testable with functional patterns and still be untestable by ATPG methods.

If you have scan chains with blockages and you cannot determine the right combination of primary input constraints, clocks, and SPF procedures, the problem might involve an uncontrolled clock path or asynchronous reset. Try dropping the scan chain from the list of known scan chains. This will increase the number of nonscan cells and decrease the achievable test coverage, but it might let you generate ATPG patterns without a design change.

If you still cannot correct the violation, you must make a design change. Examine the design along with the design guidelines presented in [Chapter 6, “Working With Design Netlists and Libraries,”](#) to determine how to change your design to correct the violation.

Analyzing the Cause of Low Test Coverage

When test coverage is lower than expected, review the AN (ATPG untestable), ND (not detected), and PT (possibly detected) faults, and answer these questions:

- Where are the faults located?
- Why are the faults untestable or difficult to test?

Where Are the Faults Located?

To find out where the faults are located, choose **Faults > Report Faults** to access the Report Faults window, which displays a report in a separate window. Alternatively, you can use the **report faults** command with the **-class** and **-level** options.

The following command generates a report of modules that have 256 or more AN faults:

```
TEST> report faults -class an -level 4 256
```

[Example 20-7](#) shows the report generated by this command. The first column shows the number of AN faults for each block. The second column shows the test coverage achieved in each block. The third column shows the block names, organized hierarchically from the top level downward.

Example 20-7 Fault Report of AN Faults Using the Level Option

```
TEST> report faults -class AN -level 4 256
#faults  testcov  instance name (type)
----- -----
22197  91.70%  /my_asic  (top_module)
2630   83.00%  /my_asic/born (born)
2435   28.00%  /my_asic/born/fpga2 (fpga2)
788    5.35%   /my_asic/born/fpga2/avge1 (avge)
1647   3.28%   /my_asic/born/fpga2/avge2 (yavge)
5226   0.00%   /my_asic/dac (dac)
5214   0.00%   /my_asic/dac/dual_port (dual_port)
11098  66.46%  /my_asic/video (video)
11098  66.24%  /my_asic/video/decipher (vdp_cyphr)
11027  60.00%  /my_asic/video/decipher/dpreg (dpreg)
426    96.97%  /my_asic/gex (gex)
260    93.89%  /my_asic/gex/fifo (gex_fifo)
799    94.56%  /my_asic/vint (vint)
798    54.29%  /my_asic/vint/vclk_mux (vclk_mux)
1514   94.80%  /my_asic/crtc_1 (crtc)
476    96.79%  /my_asic/crtc/crtc_sub (crtc_sub)
465    94.20%  /my_asic/crtc/crtc_sub/attr (attr)
1004   77.68%  /my_asic/crtc/crap (crap)
```

The report shows that the two major contributors to the high number of AN faults are these hierarchical blocks:

- `/my_asic/dac/dual_port` (with 5,214 AN faults and 0.00 percent test coverage)
- `/my_asic/video/decipher/dpreg` (with 11,027 faults and 60.00 percent test coverage)

You can also review other classes of faults and combinations of classes of faults by using different option settings in the **report faults** command.

Why Are the Faults Untestable or Difficult to Test?

To find out why the faults cannot be tested, you can use the `analyze faults` command or the `run justification` command.

The following example uses the `analyze faults` command to generate a fault analysis summary for AN faults:

```
TEST> analyze faults -class an
```

[Example 20-8](#) shows the resulting fault analysis summary, which lists the common causes of AN faults. In this example, the three major causes are constraints that interfered with testing (7,625 faults), blockages as a secondary condition of constraints (5,046 faults), and faults downstream from points tied to X (1,500 faults). As with the `report faults` command, you can specify other classes of faults or multiple classes.

Example 20-8 Fault Analysis Summary of AN Faults

```
TEST> analyze faults -class an
Fault analysis summary: #analyzed=13398, #unexplained=257.
7625 faults are untestable due to constrain values.
5046 faults are untestable due to constrain value blockage.
11 faults are connected to CLKPO.
11 faults are connected to DSLAVE.
210 faults are connected to TIEX.
233 faults are connected to TLA.
129 faults are connected to CLOCK.
50 faults are connected to TS_ENABLE.
26 faults are connected from CLOCK.
128 faults are connected from TLA.
1500 faults are connected from TIEX.
114 faults are connected from CAPTURE_CHANGE.
```

To see specific faults associated with each classification cause (for example, to see a specific fault connected from TIEX), use the `-verbose` option with the `analyze faults` command.

The following command generates an AN fault analysis report that gives details of the first three faults in each cause category:

```
TEST> analyze faults -class an -verbose -max 3
```

Example 103 shows the result of this command.

You can redirect this report to a file by using the output redirection option:

```
TEST> analyze faults -class an -verb > an_faults_detail.txt
```

You can examine each fault in detail by using the `analyze faults` command and naming the specific fault. For example, the following command generates a report on a stuck-at-0 fault on the module /gcc/hclk/U864/B:

```
TEST> analyze faults /gcc/hclk/U864/B -stuck 0
```

[Example 20-9](#) shows the result of this command. The report lists the fault location, the assigned fault classification, one or more reasons for the fault classification, and additional information about the source or control point involved.

Example 20-9 Fault Analysis Report of a Specific Fault

```
-----  
Fault analysis performed for /gcc/hclk/U864/B stuck at 0 \  
    (input 2 of MUX gate 58328).  
Current fault classification = AN \  
    (atpg_untestable-not_detected).  
-----  
Connection data: to=DSLAVE  
Fault is blocked from detection due to constrained values.  
    Blockage point is gate /gcc/hclk/writedata_reg0 (91579).  
-----
```

For additional examples, see ["Example: Analyzing an AN Fault" on page 7-41](#).

Using Justification

The `run justification` command provides another troubleshooting tool. Use it to determine whether one or more internal points in the design can be set to specific values. This analysis can be performed with or without the effects of user-defined or ATPG constraints.

If there is a specific fault that shows up in an NC (not controlled) class, you can use the `run justification` command to determine which of the following conditions applies to the fault:

- The fault location can be identified as controllable if TetraMAX is given more CPU time or a higher abort limit and allowed to continue.
- The fault location is uncontrollable.

In [Example 20-10](#), the `run justification` command is used to confirm that an internal point can be set to both a high and low value.

Example 20-10 Using run justification

```
TEST> run justification -set /my_asic/gex/hclk/U864/B 0
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 127 bus contentions (ID=37039,
pat1=0). (M181)

TEST> run justification -set /my_asic/gex/hclk/U864/B 1
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 127 bus contentions (ID=37039,
pat1=0). (M181)
```

For additional examples of the `run justification` command, see “[Checking Controllability and Observability](#)” on page 7-30.

Completing an Aborted Bus Analysis

During the DRC analysis, TetraMAX identifies the multidriver nets in the design and attempts to determine whether a pattern can be created to do the following:

- Turn on multiple drivers to cause contention.
- Turn on a single driver to produce a noncontention state.
- Turn all drivers off and have the net float.

TetraMAX automatically avoids patterns that cause contention. However, it is important to determine whether each net needs to be constantly monitored. The more nets that must be monitored, the more CPU effort is required to create a pattern that tests for specific faults while avoiding contention and floating conditions.

When TetraMAX successfully completes a bus analysis, it knows which nets must be monitored. However, if a bus analysis is aborted, nets for which analysis was not completed are assumed to be potentially problematic and therefore need to be monitored. Usually, increasing the ATPG abort limit and performing an `analyze buses` command completes the analysis, allowing faster test pattern generation.

For an example of interactively performing a bus analysis, see “[Analyzing Buses](#)” on page 7-37.

21

Using Tcl With TetraMAX

This chapter describes how to use the TetraMAX Tcl command interface. It includes a description of the process for obtaining a conversion script, general Tcl concepts as they apply to TetraMAX, and a cross-reference between native and Tcl commands.

For a general guide on how to use Tcl with Synopsys tools, see *Using Tcl With Synopsys Tools*, available through SolvNet at:

https://solvnet.synopsys.com/dow_retrieve/A-2008.03/tclug/tclug.html

In Tcl Mode, it is possible to use Tcl API commands to access, and then manipulate TetraMAX data. For a complete description, see “An Introduction to the TetraMAX Tcl API” in TetraMAX Online Help.

This chapter contains the following sections:

- [Converting TetraMAX Command Files to Tcl](#)
- [Tcl Syntax and TetraMAX Commands](#)
- [Redirecting Output](#)
- [Using Command Aliases](#)
- [Interrupting Commands](#)
- [Using Command Files](#)
- [Cross-Reference List](#)

Converting TetraMAX Command Files to Tcl

Synopsys provides a TetraMAX command translation script, native2tcl.pl, to convert existing native-mode TetraMAX command files to Tcl-mode TetraMAX command files.

You can find the TetraMAX command translation script in the installation tree at the following location:

```
$SYNOPSYS/auxx/syn/tmax/native2tcl.pl
```

Translating Native-mode Scripts

Two database files are provided with the `tmax_cmd.perl` script: `tmax_cmd.grm` and `tmax_cmd.db`.

Usage

```
native2tcl.pl [-t <ext>] [- | -r <dir>]
```

Argument	Description
<code>[-t <ext>]</code>	Identifies the file extension to assign the converted files; for example, <code>TCL</code> .
<code>[- -r <dir>]</code>	Specifies to accept input from STDIN or from the directory path specified.

For example, assuming that the native mode script to be converted is located under `/user/TMAX`, the command-line entry would look like this:

```
native2tcl.pl -t .TCL -r /user/TMAX
```

Tcl Syntax and TetraMAX Commands

The TetraMAX user interface is based on Tcl version 8.2. Using Tcl, you can extend the TetraMAX command language by writing reusable procedures.

The Tcl language has a straightforward syntax. Every Tcl script is viewed as a series of commands, separated by a new-line character or semicolon. Each command consists of a command name and a series of arguments.

For a general guide to using Tcl with TetraMAX, see the documentation on Using Tcl With Synopsys Tools at the URL:

https://solvnet.synopsys.com/cgiservlet/aban/dow/Y-2006.03/tclug/tclug_toc.html

There are two types of TetraMAX commands:

- Application commands
- Built-in commands

Each type is described in the following sections. Other aspects of Tcl version 8.2 are also described.

If you need more information about the Tcl language, consult books on the subject in the engineering section of your local bookstore or library.

Abbreviating Commands and Options

Application commands are specific to TetraMAX. You can abbreviate application command names and options to the shortest unambiguous (unique) string. For example, you can abbreviate the `add_pi_constraints` command to `add_pi_c` or the `report_faults` command option `-collapsed` to `-co`. Conversely, you cannot abbreviate most built-in commands.

Command abbreviation is meant as an interactive convenience. You should not use command or option abbreviations in script files, however, because script files are then susceptible to command changes in subsequent versions of the application. Such changes can make abbreviations ambiguous.

The variable `sh_command_abbrev_mode` determines where and whether command abbreviation is enabled. Although the default value is `Anywhere`, in the site startup file for the application, you can set this variable to `Command-Line-Only`. To disable abbreviation, set `sh_command_abbrev_mode` to `None`.

If you enter an ambiguous command, TetraMAX attempts to help you find the correct command.

Example

The `report_scan_c` command as entered here is ambiguous:

```
> report_scan_c
Error: ambiguous command 'report_scan_c' matched 2 commands:
(report_scan_cells, report_scan_chains) (CMD-006).
```

TetraMAX lists up to three of the ambiguous commands in its error message. To list all the commands that match the ambiguous abbreviation, use the help function with a wildcard pattern. For example,

```
> help report_scan_c_*
report_scan_cells      # Reports scan cell information for
selected
scan cells
report_scan_chains     # Reports scan chain information.
```

Using Tcl Special Characters

The characters listed in [Table 21-1](#) have special meaning for Tcl in certain contexts.

Table 21-1 Special Characters

Character	Meaning
\$	Dereferences a variable.
()	Used for grouping expressions.
[]	Denotes a nested command.
\	Used for escape quoting.
""	Denotes weak quoting. Nested commands and variable substitutions still occur.
{ }	Denotes rigid quoting. There are no substitutions.
;	Ends a command.
#	Begins a comment.

Using the Result of a Command

TetraMAX commands return a result, which is interpreted by other commands as strings, Boolean values, integers, and so forth. With nested commands, the result can be used as

- A conditional statement in a control structure
- An argument to a procedure
- A value to which a variable is set

Example

Here is an example using a result:

```
if {[expr $a + 11] <= $b} {  
    echo "Done"  
    return $b  
}
```

Using Built-In Commands

Most built-in commands are intrinsic to Tcl. Their arguments do not necessarily conform to the TetraMAX argument syntax. For example, many Tcl commands have options that do not begin with a dash, but do have a value argument.

Example

The Tcl `string` command has a `compare` option that you use as follows:

```
string compare string1 string2
```

Log File

A log file of the TetraMAX session can be created using the `set_messages -log <file>` command as with native mode. However, some Tcl built-in commands may not be able to write to the log file. For example, the `puts` command cannot write to the TetraMAX log file; use the `echo` command instead.

TetraMAX Extensions and Restrictions

Generally, TetraMAX implements all the Tcl built-in commands. However, TetraMAX adds semantics to some Tcl built-in commands and imposes restrictions on some elements of the language. Here are the differences:

- The Tcl `rename` command is limited to procedures you have created.
- The Tcl `load` command is not supported.
- You cannot create a command called `unknown`.
- The auto-exec feature found in tclsh is not supported. However, autoload is supported.
- The Tcl `source` command has additional options: `-echo` and `-verbose`, which are nonstandard to Tcl.
- The `history` command has additional options, `-h` and `-r`, nonstandard to Tcl, and the form `history n`.
For example, `history 5` lists the last five commands.

- The TetraMAX command processor processes words that look like bus (array) notation (words that have square brackets, such as `a [0]`), so that Tcl does not try to execute the index as a nested command. Without this processing, you would need to rigidly quote such array references, as in `{a [0]}`.
- Always use braces (`{ }`) around all control structures and procedure argument lists. For example, quote the `if` condition as follows:

```
if { ! ($a > 2) } {  
    echo "hello world"  
}
```

Redirecting Output

You can direct the output of a command, procedure, or a script to a specified file in two ways:

- Using the `redirect` command
- Using the traditional UNIX redirection operators (`>` and `>>`)

The UNIX style redirection operators cannot be used with built-in commands. You must use the `redirect` command when using built-in commands. See “[Using the redirect Command](#)” on page 21-6.

You can use either of the following two commands to redirect command output to a file:

```
redirect temp.out {report_nets n56}  
report_nets n56 > temp.out
```

You can use either of the following two commands to append command output to a file:

```
redirect -append temp.out {report_nets n56}  
report_nets n56 >> temp.out
```

Note:

The Tcl built-in command `puts` does not respond to redirection of any kind. Instead, use the TetraMAX command `echo`, which responds to redirection.

Using the `redirect` Command

In an interactive session, the result of a redirected command that does not generate a Tcl error is an empty string. For example:

```
> redirect -append temp.out { history -h }
> set value [redirect blk.out {plus 12 34}]
> echo "Value is <$value>"
Value is <>
```

Screen output from a redirected command occurs only when there is an error. For example:

```
> redirect t.out { report_commands -history 5.0 }

Error: Errors detected during redirect
      Use error_info for more info. (CMD-013)
```

This command had a syntax error because 5.0 is not an integer. The error is in the redirect file.

```
> exec cat t.out
Error: value '5.0' for option '-history' not of type
'integer'
(CMD-009)
```

The `redirect` command is more flexible than traditional UNIX redirection operators. The UNIX style redirect operators `>` and `>>` are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

For example, you can redirect `expr $a > 0` only with

```
redirect file {expr $a > 0}
```

With `redirect` you can redirect multiple commands or an entire script. As a simple example, you can redirect multiple `echo` commands:

```
redirect e.out {
    echo -n "Hello"
    echo "world"
}
```

Getting the Result of Redirected Commands

Although the result of a successful `redirect` command is an empty string, you can get and use the result of the command you redirected. You do this by constructing a `set` command in which you set a variable to the result of your command, and then redirecting the `set` command. The variable holds the result of your command. You can then use that variable in a conditional expression.

For example:

```

redirect p.out {
    set rnet [catch {read_netlist h4c.lib}]
}
if {$rnet == 1} {
    echo "read_netlist failed! Returning..."
    return
}

```

Using the Redirection Operators

Because Tcl is a command-driven language, traditional operators usually have no special meaning unless a particular command (such as `expr`) imposes some meaning. TetraMAX commands respond to `>` and `>>` but, unlike UNIX, TetraMAX treats the `>` and `>>` as arguments to the command. Therefore, you must use white space to separate these arguments from the command and the redirected file name. For example:

```

echo $my_variable >> file.out; # Right
echo $my_variable>>file.out; # Wrong!

```

Keep in mind that the result of a command that does not generate a Tcl error is an empty string. To use the result of commands you are redirecting, you must use the `redirect` command.

The UNIX style redirect operators `>` and `>>` are not part of Tcl and cannot be used with built-in commands. You must use the `redirect` command with built-in commands.

Using Command Aliases

You can use aliases to create short forms for the commands you commonly use. For example, this command duplicates the function of the `dc_shell include` command when using TetraMAX:

```
> alias include "source -echo -verbose"
```

After creating the above alias, you can use it by entering this command:

```
> include commands.cmd
```

When you use aliases, keep the following points in mind:

- TetraMAX recognizes an alias only when it is the first word of a command.
- An alias definition takes effect immediately, but only lasts until you exit the TetraMAX session.

- You cannot use an existing command name as an alias name; however, aliases can refer to other aliases.
 - Aliases cannot be syntax checked. They look like undefined procedures.
-

Interrupting Commands

If you enter the wrong options for a command or enter the wrong command, you can usually interrupt command processing by pressing Control-c.

The time the command takes to respond to an interrupt (to stop what it is doing and return to the prompt) depends on the size of the design and the function of the command being interrupted.

Some commands may take awhile before responding to an interrupt request, but TetraMAX commands will eventually respond to the interruption.

If TetraMAX is processing a command file (see “[Using Command Files](#)” on page 21-9), and you interrupt one of the file’s commands, script processing is interrupted and TetraMAX does not process any more commands in the file.

If you press Control-c three times before a command responds to your interrupt, TetraMAX is interrupted and exits with this message:

Information: Process terminated by interrupt.

There are a few exceptions to this behavior, which are documented with the applicable commands.

Using Command Files

You can use the `source` command to execute scripts in TetraMAX. A script file, also called a command file, is a sequence of commands in a text file.

The syntax is:

```
> source [-echo] [-verbose] cmd_file_name
```

By default, the `source` command executes the specified command file without showing the commands or the system response to the commands. The `-echo` option causes each command in the file to be displayed as it is executed. The `-verbose` option causes the system response to each command to be displayed.

Within a command file you can execute any TetraMAX command. The file can be simple ASCII or gzip compressed.

Adding Comments

You can add block comments to command files by beginning comment lines with the pound sign (#).

Add inline comments using a semicolon to end the command, followed by the pound sign to begin the comment. For example:

```
#  
# Set the new string  
#  
set newstr "New"; # This is a comment.
```

Controlling Command Processing When Errors Occur

By default, when a syntax or semantic error occurs while executing a command in a command file, TetraMAX discontinues processing the file. There are two variables you can use to change the default behavior: `sh_continue_on_error` and `sh_script_stop_severity`.

To force TetraMAX to continue processing the command file no matter what, set `sh_continue_on_error` to true. This is usually not recommended, because the remainder of the file may not perform as expected if a command fails due to syntax or semantic errors (for example, an invalid option).

Note:

The `sh_script_stop_severity` variable has no effect if the `sh_continue_on_error` variable is set to true.

To get TetraMAX to stop the command file when certain kinds of messages are issued, use the `sh_script_stop_severity` variable. This is set to `none` by default. Set it to `E` in order to get the file to stop on any message with error severity. Set it to `W` to get the file to stop on any message with warning severity.

Using a Setup Command File

You can use a command file as a setup file so that TetraMAX will automatically execute it at startup. To use a setup command file in the Tcl interface, you must name it either `.tmaxtclrc` or `tmaxtcl.rc`, and place it in the directory where TetraMAX was started or in your home directory.

Cross-Reference List

[Table 21-2](#) shows the correspondence between TetraMAX native commands and their Tcl equivalents.

Table 21-2 Native Command to Tcl Command Cross-Reference

Native command	Tcl Equivalent
Add Atpg Constraints <name> <0 1 Z> [-Module <name>] <atpg_gate_name gate_id module_pinname pin_pathname> [-Drc -Iddq]	add_atpg_constraints name <[0 1 z]> [-module name] <atpg_gate_name gate_id module_pinname pin_pathname> [-drc -iddq]
Add Atpg Primitives <name> <And Or SEL1 SEL01 Equiv> [-Module <name>] <[~]input_connection...>	add_atpg_primitives name <and or sel1 sel01 equiv> [-module name] {input_connection_list}
Add CApture Masks <instance_name gate_id>...	add_capture_masks <instance_name gate_id>...
Add CCell Constraints <0 1 X OX XX> <<chain_name <<cell_pos1 SCI> [cell_pos2 SCI]> -All> instance_name>	add_cell_constraints <[0 1 x ox xx]> <[chain_name instance_name> [-position <cell_pos1 sci [cell_pos2 sci]> -all]
Add CLocks <off_state> <pin_name>... [<ext.pin>+ -refclock] [<int.net>+ -pllclock] [<int.net>+ -intclock {-cycle {<n> {-always {on off} <int.net> {0 1} }+ } }+ {-pll_source <node_name>}] [-Timing <period> <LE> <TE> <measure_time> [-Unit <ps ns>]] [-Shift]	add_clocks <0 1> {pin_name_list} [<ext.pin>+ -refclock] [<int.net>+ -pllclock] [<int.net>+ -intclock [-Cycle <<cycle_id> ->-ALWAYS_ON -ALWAYS_OFF <int_node_name <0 1>>+ >]+ (-pll_source <node_name>)] [-timing {period le te measure_time}] [-unit <ps ns>]] [-shift]
Add DElay Paths <file_name>	add_delay_paths file_name

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
Add Display Gates <pin_pathname gate_id net_name instance_name -All>	add_display_gates <pin_pathname gate_id net_name instance_name -all>
Add Distributed Processors [-32bit] <hostname> [*<d>] -<Lsf <exec> -Nslaves <d> -Options <str>> -<GRd <exec> -Nslaves <d> -Options <str>> -<GENeric <exec> -Nslaves <d> -Options <str>>	add_distributed_processors [-32bit] <-{hostnames namelist} -lsf bsub_exec_name -nslaves d -options {options_list} -grd bsub_exec_name -nslaves d -options {options_list} -generic exec_name -nslaves d -options {options_list}
Add Equivalent Nofaults	add_equivalent_nofaults
Add Faults [<instance_name <pin_pathname> -Module <name> -All -CLocks -SCan_enable -BRIDGE_Location <bridge_location1> <bridge_location2> -Node_file <name>] [-Stuck <0 1 01> -Slow <R F RF>] [-Bridge <0 1 01>] [-AGgressor_node <First Second Both>] [-Launch <launch_clock>] [-Capture <capture_clock>] [-EXclusive] [-SHared] [-INTER_clock_domain] [-INTRA_clock_domain]	add_faults [instance_name] [pin_pathname] [-module name] [-all] [-clocks] [-scan_enable] [-bridge_location <bridge_location1> <bridge_location2>] [-node_file name] [-stuck <0 1 01>] [-slow <r f rf>] [-bridge <0 1 01>] [-aggressor_node <first second both>]>] [-launch launch_clock] [-capture capture_clock] [-exclusive] [-shared] [-inter_clock_domain] [-intra_clock_domain]
Add Net Connections < PI PO PIO TIE0 TIE1 TIEX TIEZ > <net_name pin_pathname>... [-Port <name>] [-Module <name>... -All] [-Disconnect -Remove]	add_net_connections < pi po pio tie0 tie1 tiex tiez > <net_name pin_pathname> [-port name] [-module name -all] [-disconnect -remove]

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
Add NofAults <instance_name pin.pathname [-Stuck <0 1 01>] -Module <name> [-Stuck <0 1 01>] [pin_names]... >	add_nofaults instance_name pin.pathname [-stuck <0 1 01>] -module name [-stuck <0 1 01>] [-pin_names {pin_name_list}]
Add PI Constraints <0 1 X Z [-ALL_Bidis -ALL_BIDIS_Except_clocks]> <pin_name -All -ALL_Except_clocks>...	add_pi_constraints <0 1 x z [-all_bidis -all_bidis_except_clocks]> <pin_name -all -all_except_clocks>
Add PI Equivalences <pin_name> [-Invert -Differential] <pin_name>...	add_pi_equivalences pin_name [-invert -differential] pin_name
Add PO Masks <pin_name -All>	add_po_masks pin_name -all
Add Scan Chains <chain_name> <input_pin> <output_pin>	add_scan_chains chain_name input_pin output_pin
Add Scan Enables <0 1 Z> <pin_name>...	add_scan_enables <0 1 z> pin_name
Add SLow Bidis <port_name -All>...	add_slow_bidis port_name -all
Add SLow Cells <instance_name gate_id>...	add_slow_cells <instance_name gate_id>
ALias [alias_name [alias_text]]	alias [alias_name [alias_text]]
ANalyze Buses <gate_id... -All> [-Exclusive [First All] -Prevention -Zstate] [-Update]	analyze_buses <gate_id... -all> [-exclusive [first all] -prevention -zstate] [-update]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
ANalyze Faults << pin.pathname -Stuck <0 1> [-Observe <gate_id>] [-Display]> < delay.pathname -Slow <R F> [-Observe <gate_id>][-Display]> <-Class fault_class>...> <bridge_location1 bridge_location2 -Bridge <0 1>> [-FAULT_simulation] [-FASt_sequential -FULL_sequential] [-Max <d>] [-Verbose]	analyze_faults pin.pathname -stuck <0 1> [-observe gate_id] [-display] delay.pathname -slow <R F> [-observe gate_id] [-display] -class {fault_class_list} <bridge_location1 bridge_location2 -bridge <0 1>> [-fault_simulation] [-fast_sequential -full_sequential] [-max d] [-verbose]
ANalyze FEedback Path <loop_id> [-Verbose]	analyze_feedback_path loop_id [-verbose]
ANalyze Simulation Data <file_name [-FASt_sequential <d>] [-FULL_sequential <d> <d>]> [-Top <top_instance_name>] [-Sensitive -INSensitive] [-Bidi_events] [-MAX_Values_reported <d>] [-MAX_Fails_reported <d>] [-Diag]	analyze_simulation_data [file_name] [-fast_sequential d] [-full_sequential {d d}]> [-top top_instance_name] [-sensitive -insensitive] [-bidi_events] [-max_values_reported d] [-max_fails_reported d] [-diag]
ANalyze Violation <violation_id>	analyzeViolation violation_id
ANalyze Wires <gate_id -All> [-Update]	analyze_wires <gate_id -all> [-update]
BUILD [-Force]	build [-force]
CAT <filename> [-Max <n>] [-Line_numbers]	cat filename [-max n] [-line_numbers]
CD <filename>	cd <filename>
CLEAR	clear

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
CP <source_filename> <destination_filename>	cp source_filename destination_filename
DRC [-Force]	drc [-force]
EXIt [-Force]	exit [-force]
Get Licenses <TEST-ANalysis TEST-Faultsim TEST-ATPG-Max TEST-Diagnosis TEST-Map> TEST-Fault-max TEST-LBIST-ATPG TEST-Core-Wrapper>	get_licenses <test-analysis test-faultsim test-atpg-max test-diagnosis test-map> test-fault-max test-lbist-atpg test-core-wrapper>
Help [command_name -All] [-Usage]	help [command_name -all] [-usage]
LS [-L] [filename]	ls [-l] [filename]
Man [<command> <rule_id> <ruleViolation_id> <message_id> COMmands GETting_started FAUlt_classes STIL_examples RAM_examples]	man [command rule_id ruleViolation_id message_id commands getting_started fault_classes stil_examples ram_examples]
MKDIR <pathname>	mkdir pathname
MV <source_filename> <destination_filename>	mv source_filename destination_filename
PWD	pwd
QUIT [-Force]	quit [-force]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
REAd Faults <file_name> [-Retain_code -Force_retain_code] [-Add -Delete]	read_faults file_name [-retain_code -force_retain_code] [-add -delete]
REAd Image <file_name> [-Password <string>]	read_image file_name [-password string]
REAd Memory_file <gate_id instance_name> <file_name> [-Binary -Hex] [-Range <first_address> <last_address>]	read_memory_file [gate_id instance_name] file_name [-binary -hex] [-range {first_address last_address}]
REAd NElist <file_name> [-Format <Edif VErilog VHdl>] [-Sensitive -INSensitive] [-Delete] [-Library] [-Master_modules] [-Noabort] [-Verbose]	read_netlist file_name [-format <edif verilog vhdl>] [-sensitive -insensitive] [-delete] [-library] [-master_modules] [-noabort] [-verbose]
REAd NOfaults <file_name>	read_nofaults file_name
REFresh Schematic	refresh_schematic
REMove Atpg Constraints <atpg_constraint_name -All>	remove_atpg_constraints <atpg_constraint_name -all>
REMove Atpg Primitives <name id -All>	remove_atpg_primitives <name id -all>
REMove CApture Masks <instance_name gate_id -All>...	remove_capture_masks <instance_name gate_id -all>
REMove CEll Constraints < <chain_name <<cell_pos1 SCI> [cell_pos2 SCI]> -All> instance_name -All>	remove_cell_constraints <chain_name instance_name -all> [-position <cell_pos1 sci [cell_pos2 sci]>]

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
REMove CLocks <pin_name... -All>	remove_clocks <{pin_name_list} -all>
REMove DElay Paths <-All path_name -MIN_Cycle <d> -MIN_Slack <d> -MAX_Cycle <d> -MAX_Slack <d>>	remove_delay_paths <-all path_name -min_cycle d -min_slack d -max_cycle d -max_slack d>
REMove Display Gates <pin_pathname gate_id -All>	remove_display_gates <pin_pathname gate_id -all>
REMove Distributed Processors < <hostname>... -Lsf -GRd -GENeric -All>	remove_distributed_processors <{hostname_list} -lsf -grd -generic -all>
REMove Faults <pin_pathname instance_name -Module <name> <[-BRIDGE_Location bridge_location1> <bridge_location2>] -CLass <fault_class> -All -Retain_sample <d>> [-Stuck <0 1 01> -Slow <R F RF>] [-Bridge <0 1 01>] [-AGgressor_node <First Second Both>] [-NON_Strength_sensitive] [-CLOcks] [-SCan_enable] [-COMBINATIonal_feedback] [-LAunch <launch_clock>] [-CApture <capture_clock>] [-EXclusive] [-SHared] [-INTER_clock_domain] [-INTRA_clock_domain]	remove_faults pin_pathname instance_name -module name [[-bridge_location bridge_location1 bridge_location2 -class fault_class -all -retain_sample d] [-stuck <0 1 01> -slow <r f rf>] [-bridge <0 1 01>] [-aggressor_node <first second both>] [-non_strength_sensitive] [-clocks] [-scan_enable] [-combinational_feedback] [-launch <launch_clock>] [-capture <capture_clock>] [-exclusive] [-shared] [-inter_clock_domain] [-intra_clock_domain]
REMove Licenses <TEST-ANalysis TEST-Faultsim TEST-ATPG-Max TEST-Diagnosis TEST-Map> TEST-Fault-max TEST-LBIST-ATPG TEST-Core-Wrapper>...	remove_licenses <test-analysis test-faultsim test-atpg-max test-diagnosis test-map> test-fault-max test-lbist-atpg test-core-wrapper>

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
REMove Net Connections <net_name... -All >	remove_net_connections <{net_name_list} -all >
REMove Nofaults <instance_name pin.pathname [-Stuck <0 1 01> -All> -Module <name> [-Stuck <0 1 01>] [pin_names]... >	remove_nofaults instance_name pin.pathname [-stuck <0 1 01> [-all] -module name [-stuck <0 1 01>] [-pin_names { pin_name_list}]
REMove PI Constraints <pin_name... -All>	remove_pi_constraints <{pin_name_list} -all>
REMove PI Equivalences <pin_name... -All>	remove_pi_equivalences <{pin_name_list} -all>
REMove PO Masks <pin_name -All>	remove_po_masks <pin_name -all>
REMove Scan Chains <-All chain_name...>	remove_scan_chains <-all {chain_name_list}>
REMove Scan Enables <pin_name -All>...	remove_scan_enables <pin_name -all>
REMove SLow Bidis <port_name -All>...	remove_slow_bidis <port_name -all>
REMove SLow Cells <instance_name gate_id -All>...	remove_slow_cells <instance_name gate_id -all>
RM <filename>	rm filename
REPort Distributed Processors	report_distributed_processors
REPort Atpg Constraints [-Summary -All] [-Max <d>]	report_atpg_constraints [-summary -all] [-max d]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
REPort Atpg Primitives <name -Summary -All> [-Max <d>] [-Verbose]	report_atpg_primitives <name -summary -all> [-max d] [-verbose]
REPort Blst [-Lfsr_chain -Prpg_shadow_chains]	report_bist [-lfsr_chain -prpg_shadow_chains]
REPort Buses < gate_id -BEhavior <Buf Inv And Or Xor Mux TIE0 TIE1 TIEZ> -Bldis -Clock -Contention <Fail Pass Abort Bidi Ignore> -Keepers -Pull -Weak -Zstate <Fail Pass Abort Bidi Ignore> -Summary -All > [-Max <d>] [-Verbose]	report_buses < gate_id -behavior <buf inv and or xor mux tie0 tie1 tiez> -bidis -clock -contention <fail pass abort bidi ignore> -keepers -pull -weak -zstate <fail pass abort bidi ignore> -summary -all > [-max d] [-verbose]
Report CApture Masks	report_capture_masks
REPort CELl Constraints	report_cell_constraints
REPort CLOCKS [-Command_report -Verbose -Matrix] [-Intclocks] [-Pllclocks]	report_clocks [-command_report -verbose -matrix] [-intclocks] [-pllclocks]
REPort COMmands [command_name] [-All] [-Usage] [-History] [-HISTORY_Depth <d>] [-Secure]	report_commands [command_name -all] [-usage] [-history] -history_depth d] [-secure]
REPort DElay Paths <-All path_name> [-Verbose] [-Display] [-Pindata]	report_delay_paths <-all path_name> [-verbose] [-display] [-pindata]
REPort DIplay Gates	report_display_gates

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
REPort FAults <instance_name pin_pathname [-Stuck <0 1 01> -Slow <R F RF>] [bridge_location1 [bridge_location2]] [-Bridge <0 1 01>] [-AGressor_node <First Second Both>] [-BRIDGE_Feedback] [-BRIDGE_Strong]> -<Class fault_class>... -UNsuccessful -Summary -Profile -All > [-COllapsed -UNCollapsed] [-Max <d>] [-Verbose] [-Level <depth> <min_count>] [-Pattern_id <d>] [-PER_clock_domain]	report_faults instance_name pin_pathname [-stuck <0 1 01> -slow <r f rf>] [bridge_location1 [bridge_location2]] [-bridge <0 1 01>] [-aggressor_node <first second both>] [-bridge_feedback] [-bridge_strong]> {-class fault_class} -unsupported -summary -profile -all > [-collapsed -uncollapsed] [-max d] [-verbose] [-level {depth min_count}] [-pattern_id d] [-per_clock_domain]
REPort FEedback Paths <path_id -Summary -All -Gate <gate_id pin_pathname instance_name> > [-Verbose]	report_feedback_paths <path_id -summary -all -gate <gate_id pin_pathname instance_name> > [-verbose]
REPort INstances <instance_name> [-Module <name>] [-Netnames]	report_instances instance_name [-module name] [-netnames]
REPort Licenses	report_licenses
REPort MEmory <gate_id instance_name -All -Summary> [-Contents <address All>] [-Max <d>] [-Verbose]	report_memory <gate_id instance_name -all -summary> [-contents <address all>] [-max d] [-verbose]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
REPort MOdules [<name> -UNReferenced -UNDefined -UNSpecified -Behavioral -Errors -Summary -All] [-Verbose]	report_modules [name -unreferenced -undefined -unspecified -behavioral -errors -summary -all] [-verbose]
REPort NET Connections	report_net_connections
REPort NETs <net_name> [-Module <name>]	report_nets net_name [-module name]
REPort NOFaults <instance_name pin.pathname [-Stuck <0 1 01>] -All -Summary> [-Max <d>]	report_nofaults <instance_name pin.pathname [-stuck <0 1 01>] -all -summary> [-max d]
REPort NONscan Cells <-Summary -All C0 C1 CU L0 L1 TLA LE TE LS Ram_out Unstable_set_resets> [-Max <d>] [-Verbose]	report_nonscan_cells <-summary -all c0 c1 cu l0 l1 tla le te ls ram_out unstable_set_resets> [-max d] [-verbose]
REPort PAtterns <start_pattern [stop_pattern] -All -Summary> [-Internal -External] [-Chain <name>] [-CHain_mapping} [-Types] [-Path_delay] [-SLack]	report_patterns { <start_pattern [stop_pattern] all summary> } [-internal -external] [-chain <name>] [-chain_mapping] [-types] [-path_delay] [-slack]
REPort PI Constraints [-Command_report]	report_pi_constraints [-command_report]
REPort PI Equivalences [-Command_report]	report_pi_equivaleces [-command_report]
REPort PO Masks	report_po_masks

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
REPort PRimitives < id instance_name net_name pin_pathname -PORts -PIS -POS -PIOs -Type <type> -Summary > [-Max <d>]	report_primitives < id instance_name net_name pin_pathname -ports -pis -pos -pios -type <type> -summary > [-max d]
	report_proc_data [-name procname -id procid]
REPort RUles [rule_id rule_type -All] [-Fail]	report_rules [rule_id rule_type -all] [-fail]
REPort SCan Ability [-Max <d>]	report_scan_ability [-max d]
REPort SCan CElls <chain_name [position] -Shadows -Unstable_set_resets -All> [-Pins] [-MAX <d>] [-MASter_only] [-Reverse_order] [-Verbose -SHift_clocks -Clocks]	report_scan_cells chain_name -position position -shadows -unstable_set_resets -all [-pins] [-max d] [-master_only] [-reverse_order] [-verbose -shift_clocks -clocks]
REPort SCan CHains [-Verbose] [-Command_report]	report_scan_chains [-verbose] [-command_report]
REPort Scan Enables	report_scan_enables
REPort SCan Path <chain_name> <SCO cell_position> <SCI cell_position> [-Reverse] [-Verbose]	report_scan_path chain_name {sco cell_position sci cell_position} [-reverse] [-verbose]
REPort SEttings [command_type -All] [-Command_report]	report_settings [command_type -all] [-command_report]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
REPort SLow Bidis	report_slow_bidis
REPort SLow Cells	report_slow_cells
REPort SUMmaries [PRimitives] [Faults] [PAtterns] [Library_cells] [Memory_usage] [Optimizations] [Sequential_depths] [Cpu_usage] [-LAunch <launch_clock>] [-CApture <capture_clock>] [-EXclusive] [-SHared] [-INTER_clock_domain] [-INTRA_clock_domain] [-PER_clock_domain]	report_summaries [primitives] [faults] [patterns] [library_cells] [memory_usage] [optimizations] [sequential_depths] [cpu_usage] [-launch <launch_clock>] [-capture <capture_clock>] [-exclusive] [-shared] [-inter_clock_domain] [-intra_clock_domain] [-per_clock_domain]
REPort VErsion [-Full -Short -Address] [-Banner] [-Verbose]	report_version [-full -short -address] [-banner] [-verbose]
REPort VViolations <violation_id rule_id rule_type -All> [-max <d>]	report_violations <violation_id rule_id rule_type -all> [-max d]
REPort WIres [gate_id -Summary -All] [-Contention <Fail Pass Abort>] [-Max <d>] [-Verbose]	report_wires [gate_id -summary -all] [-contention <fail pass abort>] [-max d] [-verbose]
RESet AU Faults	reset_au_faults
RESet State	reset_state

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
RUn Atpg [-Auto_compression] [Basic_scan_only FAst_sequential_only FUll_sequential_only] [-DIstributed] [-DNotify <low medium high>] [-DPartition <low medium high>] [-NDetects <d>] [-Observe_file <filename>] [-Random]	run_atpg [basic_scan_only fast_sequential_only full_sequential_only] [-auto_compression] [-random] [-distributed] [-observe_file file_name] [-ndetects d]
RUn Build_model [top_module] [-Weakgates]	run_build_model [top_module] [-weakgates]
RUn Commands <file_name>	run_commands file_name
RUn Diagnosis <file_name> [-Auto -Chain_failure] [-CHEck_expected_data] [-COne_analysis] [-Display] [-LAyout_data] [-Max_report_failures <d>] [-Only_report_failures] [-Post_analysis] [-RANK_faults][-REPOrt_net_data <d>] [-Sample <d1> <d2>] [-Truncate <pat#>] [-Verbose] [-Write_fault_list <out_file>] [-Replace] [-COMpress <Bin Gzip>]]	run_diagnosis [-filename name [-chain_failure] [-check_expected_data] [-cone_analysis] [-display] [-max_report_failures d] [-only_report_failures] [-post_analysis] [-report_net_data d] [-sample d1 d2] [-truncate pat#] -incomplete_failures [-verbose] [[-write_fault_list [-replace] [-compress <bin gzip>]]]
RUn DRc [file_name] [-Test -Delete]	run_drc [file_name] [-test -delete]
RUn FAult_sim [-First_pattern <d>] [-Last_pattern <d>] [-STore] [-CHeckpoint <fault_filename>] [-SEquential [-NODrop_faults]] [-Distributed] [-DETected_pattern_storage] [-NDetects <d>] [-STrong_bridge]	run_fault_sim [-first_pattern d] [-last_pattern d] [-store] [-checkpoint <fault_filename>] [-sequential [-nodrop_faults]] [-distributed] [-detected_pattern_storage] [-ndetects d] [-strong_bridge]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Run Justification [-Set <>gate_id pin_pathname> <0 1 Z>>...] [-Verbose] [-STore] [-Previous] [-NOConstraints] [-NOPrevention] [-Full_sequential]	run_justification -set gate_id -set pin_pathname -states {0 1 z...} [-verbose] [-store] [-previous] [-noconstraints] [-noprevention] [-full_sequential]
RUN Mapping <instance_name -Module <module_name>> -<Pattern_file <file_name>> [-DElete] [-DEPendent_pattern_mapping] [-NOMAp_pin <pin_name>] [-Sensitive -INSensitive] [-Verbose] [-Clock_merge] [-MAX_Errors <number>] [-Analyze <pattern number>] [-STORE_Debug_data -NOSTORE_Debug_data] [-Merge -MIXed_event_merge -NOMerge] [-READ_write_overlapping -NOREAD_write_overlapping] [-Reload_after_unload -NOREload_after_unload] [-STRobe <Offset Period> <d> <FS PS NS US MS S>] [-STRobe <Rising Falling Event> <port_name>] [-STRobe <Comments>] [-VCd_clock <0 1> <port>]...	run_mapping instance_name -module module_name -<pattern_file <file_name>> [-delete] [-dependent_pattern_mapping] [-nomap_pin pin_name]> [-sensitive -insensitive] [-verbose] [-clock_merge] [-max_errors number] [-analyze pattern number] [-store_debug_data -nostore_debug_data] [-merge -mixed_event_merge -nomerge] [-read_write_overlapping -noread_write_overlapping] [-reload_after_unload -noreload_after_unload] [-strobe_offset { d <fs ps ns us ms s>}] [-strobe_period { d <fs ps ns us ms s>}] [-strobe_rising { event port_name}] [-strobe_falling { event port_name}] [-strobe_comments { ccomments}][br/> [-vc_clock { 0 1 port ...}]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
RUn Observe Analysis <-Observe_file <file_name>> [-Replace] [-MAX_Observe_points <d>] [-Num_observe_points_per_cell <d>] [-Type_observe <Pos Scancells>]	run.observe.analysis <-observe_file file_name> [-replace] [-max_observe_points d] [-num_observe_points_per_cell d] [-type_observe <pos scandells>]
RUn Simulation [-SEquential] [-SEQUENTIAL_Update]] [-STore] [-NOCompare] [-NOX_difference] [[-Pin <pathname> <0 1>] [-Chain <name> <position>] [-Max_fails <d>] [-Failure_file <file_name>] [-Replace] [-Last_pattern <d>]	run.simulation [-sequential] [-sequential_update]] [-store] [-nocompare] [-nox_difference] [-pin {<pathname> <0 1>...}] [-chain {chain_name position}]] [-max_fails number] [-failure_file file_name] [-replace] [-last_pattern number] [-stuck <0 1>] [-slow <r f rf>] [-fast <r f rf>]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Atpg [-Abort_limit comb_max_remade_decs] [-ADJacent_fill_range <d>] [-ALLOW_clockon_measures -NOALLOW_clockon_measures] [-ANalyze_untestable_faults -NOANalyze_untestable_faults] [-BASIC_Min_detects_per_pattern <d> [d]] [-CApture_cycles <d>] [-CHAin_test <OFf 0011 0101 1000 0111 <string><R C>>] [-CHeckpoint time fault_file pattern_file -NOCHeckpoint] [-COverage max_percent] [-DECision Random NORandom*] [-DI_analysis* -NODI_analysis] [-FAST_Min_detects_per_pattern <d> [d]] {-FIII <Random 0 1 X Adjacent>} {-FIII <Random 0 1 X Adjacent>} [-FULL_Min_detects_per_pattern <d> [d]] [-FOrce_global_analysis] [-FULL_SEQ_ATpg -NOFULL_SEQ_ATpg] [-FULL_SEQ_Time max_secs_per_fault [max_secs_per_run]] [-FULL_SEQ_Merge Off Low Medium High <d>] [-MErge Off Low Medium High <d> [Low Medium High <d>]>] -MIN_Ateclock_cycles <d>	set_atpg [-adjacent_fill_range d] [-allow_clockon_measures -noallow_clockon_measures] [-analyze_untestable_faults -noanalyze_untestable_faults] [-abort_limit comb_max_remade_decs] [-basic_min_detects_per_pattern { d [d] }] [-capture_cycles d] [-chain_test { OFf 0011 0101 1000 0111 <bit_string><R C> }] [-checkpoint {interval_time fault_filename pattern_filename} -nocheckpoint] [-coverage <max_percent>] [-decision <random norandom>] [-DI_analysis -noDI_analysis] [-fast_min_detects_per_ pattern { d [d] }] [-fault_analysis_ thresholds d1 d2] [-fill <random 0 1 X adjacent>] [-full_min_detects_per_ pattern { d [d] }] [-full_seq_abort_limit seq_max_remade_decs] [-full_seq_atpg -nofull_seq_atpg] [-full_seq_merge { Off Low Medium High d }] [-full_seq_time { max_secs_per_fault [max_secs_per_run] }] [-merge < Off Low Medium High d > [low medium high d >]] [-min_patterns_threshold d] [-min_ateclock_cycles d]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Atpg (continued) [-NEw_capture -NONew_Capture] [-OPTIMIZE_Bridge_strengths -NOOPTIMIZE_Bridge_strengths] [-PATterns max_patterns] [-POST_Capture_contention_prevention -NOPOST_Capture_contention_prevention] [-PREvention Random NORandom] [-SINGLe_load_per_pattern -NOSINGLe_load_per_pattern] [-STor* -NOSTor] [-SUMmary -NOSUMmary] [-Time max_secs_per_fault[max_secs_per_run]] [-Verbose -NOVerbose]	set_atpg (continued) [-new_capture -none_new_capture] [-new_auto_thresholds d1 d2 d3 f4] [-optimize_bridge_strengths -nooptimize_bridge_strengths] [-patterns <max_patterns>] [-prevention <random norandom>] [-post_capture_contention_ prevention -nopost_capture_contention_ prevention] [-single_load_per_pattern -nosingle_load_per_pattern] [-store -nostore] [-summary -nosummary] [-time { max_secs_per_fault [max_secs_per_run] }] [-verbose -noverbose]
Set Blst [-ASSUME_Bist_setup] [-CHAin_test -NOCHAin_test] [-DBist -NODBist] [-DEbug <pattern_number> -NODEbug] [-DUMP <interval_id None>] [-Hex -NOHex] [-Hex_Diag_data] [-MAX_intervals <d>] [-MAX_Routing <d>] [-MAX_Seed_patterns <d>] [-NUM_Dbist_patterns_per_interval <d>] [-NUM_Patterns_per_interval <d>] [-OPtimize_fault_sim -NOOPtimize_fault_sim] [-Randomize_pis -NORandomize_pis] [-USE_CELL_constraints -NOUSE_CELL_constraints] [-USE_COnstant_value_cells -NOUSE_COnstant_value_cells] [-Verbose -NOVerbose]	set_blist [-chain_test -nochain_test] [-dbist -nobdbist] [-debug pattern_number -nodebug] [-dump <interval_id none>] [-hex -nohex] [-max_intervals d] [-max_routing d] [-max_seed_patterns d] [-num_dbist_patterns_per_ interval d] [-num_patterns_per_interval d] [-optimize_fault_sim] [-nooptimize_fault_sim] [-randomize_pis -norandomize_pis] [-use_cell_constraints -nouse_cell_constraints] [-use_constant_value_cells -nouse_constant_value_cells] [-verbose -noverbose]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set BUILD [-DElete_unused_gates -NODElete_unused_gates] [-Merge < Equivalent_dlat_dff EQUIVALENT_ Initialized_dlat_dff NOEquivalent_dlat_dff> <FLipflop_from_dlat FLIPFLOP_Cell_from_dlat NOFLipflop_from_dlat> <Dlat_from_flipflop NODlat_from_flipflop> <Wire_to_buffer NOWire_to_buffer > <_mux_from_gates MUXPins_from_gates MUXX_from_gates NOMux_from_gates> <Xor_from_gates XORPins_from_gates NOXor_from_gates> <Cascaded_gates_with_pin_loss NO Cascaded_gates_with_pin_loss> <Tied_gates_with_pin_loss NOTied_gates_with_pin_loss> <Bus_keepers NOBus_keepers> <FEedback_paths NOFEedback_paths <Global_tie_propagate NOGlobal_tie_propagate>]... [-Undriven_bidi < PIO PI PO >] [-Hierarchical_delimiter <c>] [-Coerce_port_directions -NOCoerce_port_directions] [-Fault_boundary < Lowest Hierarchical>] [-Add_buffer -NOAdd_buffer] [-Limit_fanout <d>] [-Black_box <module_name> -Empty_box <module_name>	set_build [-delete_unused_gates -nodelete_unused_gates] [-merge <equivalent_dlat_dff equivalent_ initialized_dlat_dff noequivalent_dlat_dff flipflop_from_dlat flipflop_cell_from_dlat noflipflop_from_dlat dlat_from_flipflop nodlat_from_flipflop wire_to_buffer nowire_to_buffer mux_from_gates muxpins_from_gates muxx_from_gates nomux_from_gates xor_from_gates xorpins_from_gates noxor_from_gates cascaded_gates_with_pin_loss nocascaded_gates_with_pin_loss tied_gates_with_pin_loss notied_gates_with_pin_loss bus_keepers nobus_keepers feedback_paths nofeedback_paths global_tie_propagate noglobal_tie_propagate >] [-undriven_bidi <pio pi po>] [-hierarchical_delimiter c] [-coerce_port_directions -nocoerce_port_directions] [-fault_boundary <lowest hierarchical>] [-add_buffer -noadd_buffer] [-limit_fanout d] [-black_box module_name -empty_box module_name

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
Set BUlld (continued) -DESign_box <module_name> -Portfault_box <module_name> -NOBox <module_name> -Reset_boxes]... [-INStance_modify <instance_name> <gate_type> -NOINStance_modify <instance_name -All>] [-Net_connections_change_netlist -NONet_connections_change_netlist]	set_build (continued) -design_box module_name -portfault_box module_name -nobox <module_name> -reset_boxes] [-instance_modify <instance_name gate_type> -noinstance_modify instance_name -noinstance_modify_all] [-net_connections_change_netlist -nonet_connections_change_netlist]
Set BUSes [-External_z <X 0 1 Z>] [-Fault_contention <And Or X>] [-Contention_status <Ignore NOIgnore> <gate_id pin_pathname All>...] [-Zstate_status <<Ignore NOIgnore> <gate_id> pin_pathname All>...]	set_buses [-external_z <x 0 1 z>] [-fault_contention <and or x>] [-contention_status { <ignore noignore> <gate_id pin_pathname all> ... }] [-zstate_status { <ignore noignore> <gate_id pin_pathname all> }]
Set COLORs <Error Warning User_commands File_commands> <red_number> <green_number> <blue_number>	set_colors <error warning user_commands file_commands> <red_number> <green_number> <blue_number>
Set COMMands [History NOHistory] [NOAbort Abort Exit] [-SECure <command>] [-NOSECure <command -All>]	set_commands [history nohistory] [noabort abort exit] [-secure command] [-nosecure <command -all>]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set CONTENTION [BIdi NOBIdi] [Bus NOBus] [Dff_dlat NODff_dlat] [Float NOFloat] [Ram NORam] [Wire NOWire] [-Atpg -NOAtpg] [-Capture -NOCapture] [-Multiple_on -NOMultiple_on] [-REtain_bidi -NOREtain_bidi] [-Severity <Error WArning Ignore>] [-Preclock -NOPreclock] [-Verbose -NOVerbose]	set_contention [bidi nobidi] [bus nobus] [dff_dlat nodff_dlat] [float nofloat] [ram noram] [wire nowire] [-atpg -noatpg] [-capture -nocapture] [-multiple_on -nomultiple_on] [-retain_bidi -noretain_bidi] [-severity <error warning ignore>] [-preclock -nopreclock] [-verbose -noverbose]
Set DElay [-PI_changes -NOPI_changes] [-PO_measures -NOPO_measures] [-Allow_reconverging_paths -NOAllow_reconverging_paths] [-ALLOW_Multiple_common_clocks -NOALLOW_Multiple_common_clocks] [-Diagnostic_propagation -NODIagnostic_propagation] [-DISturb -NODISturb] [-Mask_nontarget_paths -NOMask_nontarget_paths] [-RElative_edge -NORElative_edge] [-RObust_fill -NORObust_fill] [-Launch_cycle <Last_shift System_clock Any>] [-DAta <path_name>] [-Simulate_hazards -NOSimulate_hazards] [-COmmon_launch_capture_clock -NOCOmmon_launch_capture_clock] [-TWO_clock_transition_optimization -NOTWO_clock_transition_optimization] [-SSlow_equivalence -NOSSlow_equivalence] [-OPTimize_atpg -NOOPTimize_atpg]	set_delay [-pi_changes -nopi_changes] [-po_measures -nopo_measures] [-allow_reconverging_paths -noallow_reconverging_paths] [-allow_multiple_common_clocks -noallow_multiple_common_clocks] [-diagnostic_propagation -nodiagnostic_propagation] [-disturb -nodisturb] [-mask_nontarget_paths -nomask_nontarget_paths] [-relative_edge -norelative_edge] [-robust_fill -norobust_fill] [-launch_cycle <last_shift system_clock any>] [-data <path_name>] [-simulate_hazards -nosimulate_hazards] [-common_launch_capture_clock -nocommon_launch_capture_clock] [-two_clock_transition_optimization -notwo_clock_transition_optimization] [-slow_equivalence -noslow_equivalence] [-optimize_atpg -nooptimize_atpg]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set DIstributed [-SHell_timeout <d>] [-SLave_setup_timeout <d>] [-PRint_stats_timeout <d>] [-SCript <file_name>] [-Work_dir <dir_name>] [-SHELL REMSH RSH SSH] [-Verbose -NOVerbose]	set_distributed [-shell_timeout d] [-slave_setup_timeout d] [-print_stats_timeout d] [-script file_name] [-work_dir dir_name] [-shell remsh rsh ssh] [-verbose -noverbose]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Drc [<i>file_name</i> -NOFile] [-Oscillation <d>] [-TRace -Chain_trace <chain_name> -NOTRace] [-SHadows -NOSHadows -SErial_shadows_only] [-TLAs -NOTLas] [-STORE_SEtup -NOSTORE_SEtup] [-STORE_STability_patterns -NOSTORE_STability_patterns] [-Bidi_control_pin <0 1><name> None] [-SKew <d>] [-Remove_false_clocks -NORemove_false_clocks] [-Initialize_dff_dlat <0 1 X Random>] [-Allow_unstable_set_resets -NOAllow_unstable_set_resets] [-CLock <pin_name> -Any -Dynamic -One_hot -Seq_capture]> [-MULTi_captures_per_load -NOMulti_captures_per_load] [-DISturb_clock_grouping -NODISturb_clock_grouping] [-COntroller_clock <pin_name> -NOCOnntroller_clock] [-Z_Check_with_all_constraints -NOZ_Check_with_all_constraints] [-USE_Cell_constraints -NOUSE_Cell_constraints] [-Unstable_Israms -NOUnstable_Israms] [-Dslave_remodel -NODslave_remodel]	set_drc [<i>file_name</i> -nofile] [-oscillation <i>d</i>] [-trace -chain_trace <i>chain_name</i> -notrace] [-shadows -noshadows -serial_shadows_only] [-tlas -notlas] [-store_setup -nostore_setup] [-store_stability_patterns -nostore_stability_patterns] [-bidi_control_pin { 0 1 name } -bidi_control_none] [-skew <i>d</i>] [-remove_false_clocks -noremove_false_clocks] [-initialize_dff_dlat <0 1 x random>] [-allow_unstable_set_resets -noallow_unstable_set_resets] [-clock <pin_name> -any -dynamic -one_hot -seq_capture]> [-multi_captures_per_load -nomulti_captures_per_load] [-disturb_clock_grouping -nodisturb_clock_grouping] [-controller_clock <pin_name> -nocontroller_clock] [-z_check_with_all_constraints -noz_check_with_all_constraints] [-use_cell_constraints -nouse_cell_constraints] [-unstable_Israms -nounstable_Israms] [-dslave_remodel -nodslave_remodel]

**Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)**

Native command	Tcl Equivalent
Set Drc (continued) [-OBServer_procedure <Master Any>] [-Pipeline -NOPipeline] [-num_pll_cycles <d>] [-max_pll_simulation_passes <d>] [-pll_simulate_test_setup -nopll_simulate_test_setup] [-use_cell_constraints -no_use_cell_constraints]	set_drc (continued) [-observe_procedure <master any>] [-pipeline -nopipeline] [-num_pll_cycles d] [-max_pll_simulation_passes d] [-pll_simulate_test_setup -nopll_simulate_test_setup] [-use_cell_constraints -no_use_cell_constraints]
Set ENVironment Gui -Cmdbar -NOCmdbar] -History_order <Newest> Oldest> [-Max_history_length <d>] [-TOolbar <Text Bltmap_only>]	set_environment_gui -cmdbar -nocmdbar -history_order <newest oldest> [-max_history_length d] [-toolbar <text bitmap_only>]
Set ENVironment Info [-Always_on_top -NOAlways_on_top] [-Save_dimensions -NOSave_dimensions]	set_environment_info [-always_on_top -noalways_on_top] [-save_dimensions -nosave_dimensions]
Set ENVironment Reports [-MAX_lines <d>] [-Line_length <d>] [-Color <Text Background> <red_number> <green_number> <blue_number>...] [-Xpos <d>] [-Ypos <d>] [-Width <d>] [-Height <d>] [-Font_size <d>]	set_environment_reports [-max_lines d] [-line_length d] [-color_text <red_number green_number blue_number>...] [-color_background <red_number green_number blue_number>] [-xpos d] [-ypos d] [-width d] [-height d] [-font_size d]
Set ENVironment Transcript [-MAX_lines <d>] [-Line_length <d>] [-Font_size <d>]	set_environment_transcript [-max_lines d] [-line_length d] [-font_size d]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set ENVironment Viewer [-Buf_inv -NOBuf_inv] [-DEsign_symbols <Block Primitives>] [-DFf_dlat_symbol <MODE1 MODE2 MODE3>] [-Font <System Hershey>] [-Instance_names -NOInstance_names] [-Left_justify_pis -NOLeft_justify_pis] [-Max_gates <d>] [-Primitive -NOPrimitive] [-Redraw <Always If_necessary>] [-Scheme <Default Black_on_white Synopsys>] [-Tlme_out <d>] [-TRuncate -NOTRuncate] [-Color <BAckground BIdi BLock BUfinv_removed EDge_connector ID INstance_name HIGHLIGHT NET PINData_text PINName_text RUbberband TIE TITLE_Background TITLE_Text TYpe_name_text> <red_number> <green_number> <blue_number>]...	set_environment_viewer [-buf_inv -nobuf_inv] [-design_symbols <block primitives>] [-dff_dlat_symbol <mode1 mode2 mode3>] [-font <system hershey>] [-instance_names -noinstance_names] [-left_justify_pis -noleft_justify_pis] [-max_gates d] [-primitive -noprimitive] [-redraw <always if_necessary>] [-scheme <default black_on_white Synopsys>] [-time_out d] [-truncate -notruncate] [-color { <background bidi block bufinv_removed edge_connector id instance_name highlight net pindata_text pinname_text rubberband tie title_background title_text type_name_text> <red_number> <green_number> <blue_number>... }]
Set Faults [-Model <Stuck Iddq Transition Path_delay Bridging>] [-Report <Collapsed Uncollapsed>] [-Pt_credit <d>] [-AU_credit <d>] [-BRIDGE_Inputs -NOBRIDGE_Inputs] [-ATpg_effectiveness -NOAtpg_effectiveness] [-Fault_coverage -NOFault_coverage] [-Equiv_code <code_name> -NOEquiv_code] [-Summary <Verbose NOVerbose>]	set_faults [-model <stuck iddq transition path_delay bridging>] [-report <collapsed uncollapsed>] [-pt_credit d] [-au_credit d] [-bridge_inputs -nobridge_inputs] [-atpg_effectiveness -noatpg_effectiveness] [-fault_coverage nofault_coverage] [-equiv_code code_name -noequiv_code] [-summary <verbose noverbose>]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Iddq [Float NOFloat] [Strong NOStrong] [WEak NOWEak] [WRite NOWRite] [-Atpg -NOAtpg] [-Toggle -NOToggle] [-Exclude_ports <None Bidis All>]	set_iddq [float nofloat] [strong nostrong] [weak noweak] [write nowrite] [-atpg -noatpg] [-toggle -notoggle] [-exclude_ports <none bidis all>]
Set Learning [-Atpg_equivalence -NOAtpg_equivalence] [-Common_input -NOCommon_input] [-Equivalent_latches -NOEquivalent_latches] [-Implication [Low Medium High]] -NOImplication] [-MAX_FFEedback_sources <d>] [-Sim_passes <d>] [-Test_passes <d>] [-Verbose -NOVerbose]	set_learning [-atpg_equivalence -noatpg_equivalence] [-common_input -nocommon_input] [-equivalent_latches -noequivalent_latches] [-implication [low medium high]] -noimplication] [-max_feedback_sources d] [-sim_passes d] [-test_passes d] [-verbose -noverbose]
Set Messages [Display NODisplay] [-Double_slash -NODouble_slash] [NOLog Log <filename> [-Replace -Append]] [-Level <Expert Standard>] [-Transcript_comments -NOTranscript_comments]	set_messages <-display -nodisplay> [-double_slash -nodouble_slash] [-nolog -log file] [-replace -append] [-level <expert standard>] [-transcript_comments -notranscript_comments]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Netlist [-Pin_assign -NOPin_assign] [-EScape < Cond All None >] [-SCalar_net -NOScalar_net] [-Max_errors <d>] [-REDefined_module < First Last >] [-Dominance_detection < ON OFF Boolean >] [-CEldefine -NOCEldefine] [-ENable_portfaults -NOENable_portfaults] [-SUppress_faults -NOSUppress_faults] [-Xmodeling -NOXmodeling] [-SEquential_modeling -NOSEquential_modeling] [-CHeck_only_used_udps -NOCHeck_only_used_udps] [-COnservative_mux < None Combinational_udp All >]	set_netlist [-pin_assign -nopin_assign] [-escape < cond all none >] [-scalar_net -noscalar_net] [-max_errors d] [-redefined_module < first last >] [-dominance_detection < on off boolean >] [-cellddefine -nocellddefine] [-enable_portfaults -noenable_portfaults] [-suppress_faults -nosuppress_faults] [-xmodeling -noxmodeling] [-sequential_modeling -nosequential_modeling] [-check_only_used_udps -nocheck_only_used_udps] [-conservative_mux < none combinational_udp all >]
Set PAtterns [-APPend] [-Delete] [-expand_xdbist_patterns] [<Internal Random External <filename>] [-misr_ends_zero] [-misr_starts_zero] [-SEnsitive -INSensitive] [-STrobe <Comments>] [-STrobe <Offset Period> <d> <FS PS NS US MS S>...] [-STrobe <Rising Falling Event> <port_name>] [-Histogram_summary -NOHistogram_summary] [-Load_summary -NOLoad_summary] [-NOVCD_clock -VCD_clock Auto -VCD_clock <0 1> <port>...] [translate_dbist_patterns] [-VERIlog_last_scan -NOVERIlog_last_scan]	set_patterns [-append] [-delete] [-expand_xdbist_patterns] [-internal -random -external filename] [-misr_ends_zero] [-misr_starts_zero] [-sensitive -insensitive] [-strobe_comments { comments }] [-strobe_period { d <fs ps ns us ms s> }] [-strobe_rising { event port_name }] [-strobe_falling {event port_name}] [-strobe_offset { d <fs ps ns us ms s> }] [-histogram_summary -nohistogram_summary] [-load_summary -noload_summary] [-novcd_clock -vcd_clock auto -vcd_clock { 0 1 port ... }] [-translate_dbist_patterns] [-verilog_last_scan -noverilog_last_scan]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set Plndata < None CLOCK_Cone <pin_name> CLOCK_Off Constrain_data Debug_sim_data DELAY_data Error_data FAULT_Sim_results <pin_pathname gate pin_number> <0 1> <pattern> Fault_data Good_sim_results <pattern> Load Master_observe Pattern <d All Fast_sequential> FULL_SEQ_Scoap_data FULL_SEQ_Tg_data SCoap_data SEQ_Sim_data SHAdow_observe SHlft STability_patterns TEst_setup Tle_data Bidi_control_value > [-Refresh] [-Shift_character <c>] [-Constrain_character <c>]	set_pldata -none -clock_cone pin_name -clock_off -constrain_data -debug_sim_data -delay_data -error_data -fault_sim_results {<pin_pathname gate pin_number> <0 1> <pattern> } -fault_data -good_sim_results pattern -load -master_observe -pattern {d all fast_sequential} -full_seq_scoap_data -full_seq_tg_data -scoap_data -seq_sim_data -shadow_observe -shift -stability_patterns -test_setup -tie_data -bidi_control_value [-refresh] [-shift_character c] [-constrain_character c]
Set PRimitive_report [-Max_fanout <d>] [-Interval <d>] [-Time <Clock PReclock POstclock Lete All>] [-Verbose -NOVerbose]	set_primitive_report [-max_fanout d] [-interval d] [-time <clock preclock postclock lete all>] [-verbose -noverbose]
Set RAndom_patterns [-Clock <pinname> -NOClock] [-Length <d>] [-Observe <Master SLave SHadow>]	set_random_patterns [-clock pinname -clock_none] [-length d] [-observe <master slave shadow>]
Set RUles <rule_id> [Error Warning Ignore] [-Mask -NOMask]	set_rules rule_id [error warning ignore] [-mask -nomask]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
Set SCan Ability <ON OFF> <nonscan_cell_gate -DFf -DLat -All>	set_scan_ability <on off> <nonscan_cell_gate -dff -dlat -all>
Set Simulation [-Bidi_fill -NOBidi_fill -STRong_bidi_fill -WEAk_bidi_fill] [-Data <first_pattern> <last_pattern>] [-Measure <Sim Pat>] [-Oscillation <num_passes> <num_additional_fault_passes>] [-Words_per_pass <d>] [-Verbose -NOVerbose] [-BASIC_scan -NOBASIC_scan] [-STORe_memory_ contents <pattern_id LAst>] [-NOSTORe_memory_contents] [-XClock_gives_xout -NOXClock_gives_xout]	set_simulation [-bidi_fill -nobidi_fill -strong_bidi_fill -weak_bidi_fill] [-data <first_pattern> <last_pattern>] [-measure <sim pat>] [-oscillation { num_passes num_additional_fault_passes }] [-words_per_pass d] [-verbose -noverbose] [-basic_scan -nobasic_scan] [-store_memory_ contents { pattern_id last }] [-nostore_memory_contents] [-xclock_gives_xout -noxclock_gives_xout]
Set WGI [-Bidi_map <ab> <cd>] [-Scan_map < Dash Bidi Keep None >] [-Macro_usage -NOMacro_usage] [-PAd -NOPad] [-Last_scan -NOLast_scan] [-Group_bidis -NOGroup_bidis] [-Tester_ready -NOTester_ready -PRe_measured] [-INversion_reference < Master Cell Omit >] [-FOrces_during_load < Allow_x Previous_values No_x >] [-Chain_list < All Shift >]	set_wgl [-bidi_map { ab cd }] [-scan_map < dash bidi keep none >] [-macro_usage -nomacro_usage] [-pad -nopad] [-last_scan -nolast_scan] [-group_bidis -nogroup_bidis] [-tester_ready -notester_ready -pre_measured] [-inversion_reference < master cell omit >] [-forces_during_load < allow_x previous_values no_x >] [-chain_list < all shift >]
Set WOrkspace Sizes < [-Atpg_gates <d>] [-CONnectors <d>] [-Decisions <d>] [-Line <d>] [-String <d> >	set_workspace_sizes < [-atpg_gates d] [-connectors d] [-decisions d] [-line d] [-string d] >

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
SOURCE <file_name>	source file_name
SYstem <system_instruction>	system <system_instruction>
TEST	test
UNALias <alias_name -All>	unalias <alias_name -all>
Write Drc_file <filename> [-Replace] [-Compress <Gzip Binary>] [-Scancells]	write_drc_file filename [-replace] [-compress <gzip binary>] [-scancells]
Write Faults <filename> <instance_name pin.pathname [-Stuck <0 1 01> -Slow <R F RF>] <Class fault_class>... -Summary -All > [-COMpress <Gzip Binary>] [-COllapsed -UNCollapsed] [-Max <d>] [-Replace] [-Verbose] [-Profile]	write_faults filename -name instance_name -name pin.pathname [-stuck <0 1 01> -slow <r f rf>] -class { fault_class ... } -summary -all [-compress <gzip binary>] [-collapsed -uncollapsed] [-max d] [-replace] [-verbose] [-profile]
Write Image <file_name> [-COMpress <Gzip Binary Off>] [-Password <string>] [-Replace] [-Schematic_view] [-Violations]	write_image file_name [-compress <gzip binary off>] [-password string] [-replace] [-schematic_view] [-violations]

*Table 21-2 Native Command to Tcl Command Cross-Reference
(Continued)*

Native command	Tcl Equivalent
<p>Write Netlist <code><filename></code> <code>[-Replace]</code> <code>[-Top top_name]</code> <code>[-Format <Edif [External] Verilog>]</code> <code>[-Compress <Gzip Binary>]</code> <code>[-STOp [Edif Verilog VHdl Design_level]]</code></p>	<code>write_netlist</code> <code>filename</code> <code>[-replace]</code> <code>[-top top_name]</code> <code>[-format { <edif [external] verilog > }]</code> <code>[-compress <gzip binary>]</code> <code>[-stop [edif verilog vhdl design_level]]</code>
<p>Write Patterns <code><filename></code> <code>[-CELLnames < Internal Verilog >]</code> <code>[-COMpress <Gzip Binary>] [-CORe -NOCORE]</code> <code>[-EXClude <Setup Repeat_setup Patterns All>...]</code> <code>[-EXPand_vector -NOEXPand_vector]</code> <code>[-FIrst <d>] [-Last <d>]</code> <code>[-FOrmat <Binary Ftdl Stil STIL99 TDI91 TStl2 VErlog_tables VERILOG_Single_file VHdl VHDL93 Wgl WGL_Flat>]</code> <code>[-CONfig_file <cfilename>]]</code> <code>[-Internal -EXTernal] [-Measure_forced_bidis]</code> <code>[-NOCompaction] [-NOOverlap_load]</code> <code>[-Order_pins]</code> <code>[-PAD_character < 0 1 X >]</code> <code>[-PATInfo -NOPATInfo]</code> <code>[-PATNAME <pat_block_name>] [-Replace]</code> <code>[-SERial -PARallel [n_shifts]]</code> <code>[-SOrted -REOrder <file>] [-Type <Basic_scan Clock_on_measure FAst_sequential Full_sequential>]</code> <code>[-SPLIT <d>] [-STl] [-VCS] [-MTI] [-XL] [-NC]</code> <code>[-Verilog_testbench_name <filename>]</code> </p>	<code>write_patterns</code> <code>filename</code> <code>[-cellnames <internal verilog>]</code> <code>[-config_file <cfilename>]</code> <code>[-compress <gzip binary>]</code> <code>[-core -nocore]</code> <code>[-exclude { <setup repeat_setup patterns all> ... }]</code> <code>[-first d]</code> <code>[-last d]</code> <code>[-format < binary stil stil99 verilog_tables verilog_single_file vhdl vhdl93 wgl tstd2 ftdl tdi91 wgl_flat >]</code> <code>[-internal -external]</code> <code>[-measure_forced_bidis]</code> <code>[-nocompaction]</code> <code>[-nooverlap_load]</code> <code>[-order_pins]</code> <code>[-pad_character < 0 1 x >]</code> <code>[-patinfo nopatinfo]</code> <code>[-replace]</code> <code>[-serial -parallel num_shifts]</code> <code>[-sorted -reorder <file_name>]</code> <code>-type <basic_scan clock_on_measure fast_sequential full_sequential></code> <code>[-split d]</code> <code>[-stil]</code> <code>[-verilog_testbench_name name]</code>

A

Test Concepts

When you perform manufacturing testing, you ensure high-quality integrated circuits by screening out devices with manufacturing defects. You can thoroughly test your integrated circuit when you adopt structured design-for-test (DFT) techniques. The DFT techniques currently supported by TetraMAX ATPG consist of internal scan (both full scan and partial scan) and boundary scan. This appendix covers the background you need to understand these techniques.

This appendix contains the following sections:

- [Why Perform Manufacturing Testing?](#)
- [What Are Fault Models?](#)
- [What Is Internal Scan?](#)
- [What Is Boundary Scan?](#)

Why Perform Manufacturing Testing?

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

Manufacturing testing verifies that your circuit does not have manufacturing defects by focusing on circuit structure rather than functional behavior. Manufacturing defects include problems such as the following:

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

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

A development team usually performs both functional and manufacturing testing of devices.

What Are Fault Models?

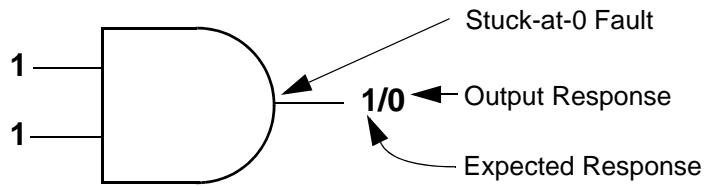
A manufacturing defect has a logical effect on the circuit behavior. An open connection can appear to float either high or low, depending on the technology. A signal shorted to power appears to be permanently high. A signal shorted to ground appears to be permanently low. Many of these manufacturing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

Stuck-At Fault Models

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

For example, [Figure A-1](#) shows a two-input AND gate that has a stuck-at-0 fault on the output pin. Regardless of the logic level of the two inputs, the output is always 0.

Figure A-1 Stuck-at-0 Fault on Output Pin of 2-input AND Gate



Detecting Stuck-At Faults

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

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

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

To detect a stuck-at fault on a target node, you must do the following:

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

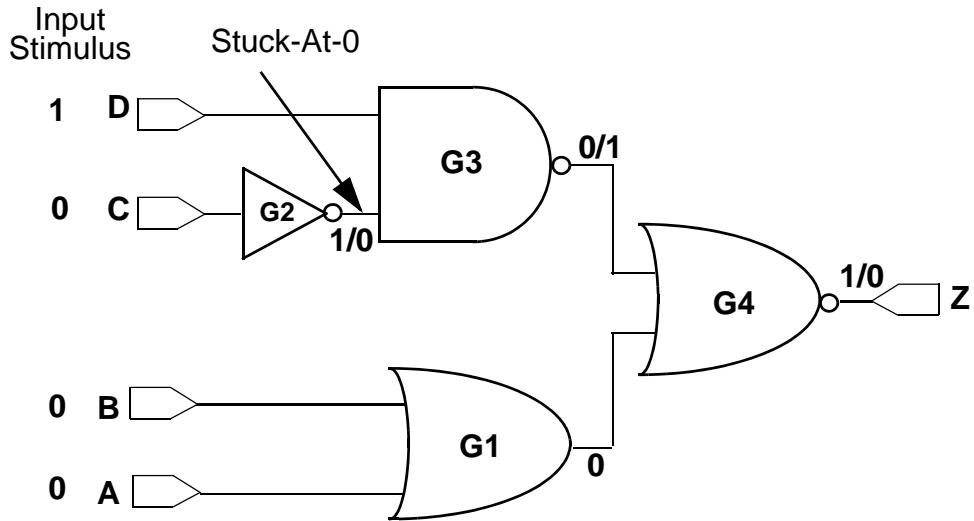
The set of logic 0s and 1s applied to the primary inputs of a design is called the input stimulus. The set of resulting values at the primary outputs, assuming a fault-free design, is called the expected response. The set of actual values measured at the primary outputs is called the output response.

If the output response does not match the expected response for a given input stimulus, the input stimulus has detected the fault. To detect a stuck-at-0 fault, you need to apply an input stimulus that forces that node to 1. For example, to detect a stuck-at-0 fault at the output of the two-input AND gate shown in [Figure A-1](#), you need to apply a logic 1 at both inputs. The expected response for this input stimulus is logic 1, but the output response is logic 0. This input stimulus detects the stuck-at-0 fault.

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

In a more complex situation, you may need to control all other nodes to ensure observability of a particular target node. [Figure A-2](#) shows a circuit with a detectable stuck-at-0 fault at the output of cell G2.

Figure A-2 Simple Circuit With Detectable Stuck-At Fault



To detect the fault, you need to control the output of cell G2 to logic 1 (the opposite of the faulty value) by applying a logic 0 value at primary input C. To ensure that the fault effect is observable at primary output Z, you need to control the other nodes in the circuit so that the response value at primary output Z depends only on the output of cell G2.

For this example, you can accomplish these goals as follows:

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

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

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

Using Fault Models to Determine Test Coverage

One definition of a design's testability is the extent to which that design can be tested for the presence of manufacturing defects, as represented by the single stuck-at fault model. Using this definition, the metric used to measure testability is test coverage. For a precise explanation of how test coverage is calculated in TetraMAX, see “[Test Coverage](#)” on page 10-10.

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

IDDQ Fault Model

For CMOS circuits, an alternative testing method is available, called IDDQ testing. IDDQ testing is based on the principle that a CMOS circuit does not draw a significant amount of current when the device is in the quiescent (quiet, steady) state. The presence of even a single circuit fault, such as a short from an internal node to ground or to the power supply, can be detected by the resulting excessive current drain at the power supply pin. IDDQ testing can detect faults that are not observable by stuck-at fault testing.

For the IDDQ testing, the ATPG algorithm uses an IDDQ fault model rather than a stuck-at fault model. The generated test patterns only need to control internal nodes to 0 and 1 and comply with quiescence requirements. The patterns do not need to propagate the effects of faults to the device outputs. The ATPG tool attempts to maximize the toggling of internal states and minimize the number of patterns needed to control each node to both 0 and 1 for IDDQ testing.

TetraMAX has an optional IDDQ pattern generation/verification capability called IddQTest. It uses the following criteria for IDDQ pattern generation:

- No current should flow through resistors.
- There must not be contention on any bus or node.
- No nodes can be allowed to float. A floating node could cause some CMOS transistors to turn on and draw current.
- RAM modules must be disabled so that they do not draw any current.

For more information on IddQTest, see the *Test Pattern Validation User Guide*.

Fault Simulation

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

Fault simulation determines all faults detected by a test vector. By fault simulating the test vector that is generated to detect the stuck-at-0 fault on the output of G2 in [Figure A-2 on page A-4](#), it becomes clear that this vector also detects the following single stuck-at faults:

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

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

Automatic Test Pattern Generation

ATPG generates test patterns and provides test coverage statistics for the generated pattern set. The difference between test vectors and test patterns is defined in [“What Is Internal Scan?” on page A-7](#). For now, consider the term “test vector” to be the same as “test pattern.”

ATPG for combinational circuits is well understood; it is usually possible to generate test vectors that provide high test coverage for combinational designs.

Combinational ATPG tools can use both random and deterministic techniques to generate test patterns for stuck-at faults. By default, TetraMAX only uses deterministic pattern generation; using random pattern generation is optional.

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

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

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

Structured DFT techniques (for example, internal scan) simplify the test pattern generation task for complex sequential designs, resulting in higher test coverage and reduced testing costs. For more information about internal scan techniques, see [“What Is Internal Scan?”](#) on page A-7.

Translation for the Manufacturing Test Environment

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

You might use more than one set of test vectors for manufacturing testing. The term “test program” means the collection of all test vector sets used to test a design.

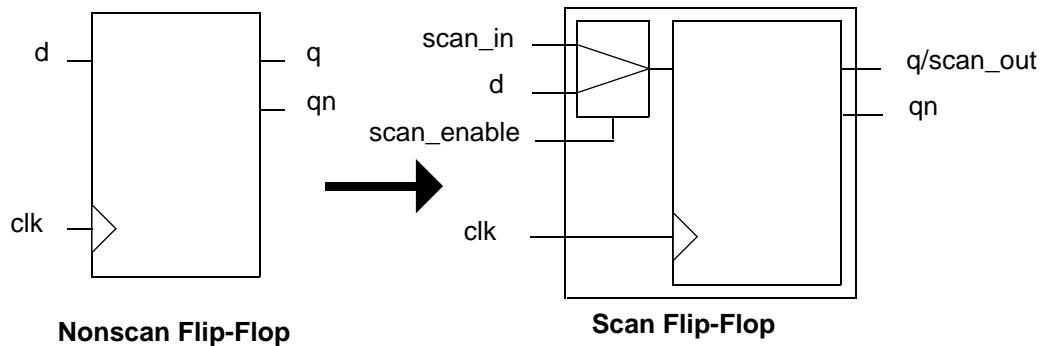
What Is Internal Scan?

Internal scan design is the most popular DFT technique and has the greatest potential for high test coverage. The principle of this technique is to modify the existing sequential elements in the design to support serial shift capability, in addition to their normal functions; and to connect these elements into serial chains to make, in effect, long shift registers.

Each scan chain element can operate like a primary input or primary output during ATPG testing, greatly enhancing the controllability and observability of the internal nodes of the device. This technique simplifies the pattern generation problem by effectively dividing complex sequential designs into fully isolated combinational blocks (full-scan design) or semi-isolated combinational blocks (partial-scan design).

[Figure A-3](#) shows an example of the multiplexed flip-flop scan style, where a D flip-flop has been modified to support internal scan by the addition of a multiplexer. Inputs to the multiplexer are the data input of the flip-flop (*d*) and the scan-input signal (*scan_in*). The active input of the multiplexer is controlled by the scan-enable signal (*scan_enable*). Input pins are added to the cell for the *scan_in* and *scan_enable* signals. One of the data outputs of the flip-flop (*q* or *qn*) is used as the scan-output signal (*scan_out*). The *scan_out* signal is connected to the *scan_in* signal of another scan cell to form a serial scan (shift) capability.

Figure A-3 D Flip-Flop With Scan Capability



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

Most semiconductor vendor libraries include pairs of equivalent nonscan and scan cells that support a given scan style. One special test cell is a scan flip-flop that combines a D flip-flop and a multiplexer. You can also implement the scan function of this special test cell with discrete cells, such as the separate flip-flop and multiplexer shown in [Figure A-3](#).

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

- Design size and power increases slightly because scan cells are usually larger than the nonscan cells they replace, and the nets used for the scan signals occupy additional area.

- Design performance (speed) decreases marginally because of changes in the electrical characteristics of the scan cells that replace the nonscan cells.
- Global test signals that drive many sequential elements might require buffering to prevent electrical design rule violations.

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

The Synopsys scan DFT synthesis capabilities fully optimize for the user's design rules and constraints (timing, area, and power) in the context of scan. These scan synthesis capabilities are available in DFT Compiler, the Synopsys test-enabled synthesis configuration. For information about how DFT Compiler minimizes the impact of inserting scan logic in your design, see the *DFT Compiler Scan Synthesis User Guide*.

For scan designs, an ATPG tool generates input stimuli for the primary inputs and pseudo-primary inputs and expected responses for the primary outputs and pseudo-primary outputs. The set of input stimuli and output responses is called a test pattern or scan pattern. This set includes the data at the primary inputs, primary outputs, pseudo-primary inputs, and pseudo-primary outputs.

A test pattern represents many test vectors because the pseudo-primary-input data must be serialized to be applied at the input of the scan chain, and the pseudo-primary-output data must be serialized to be measured at the output of the scan chain.

Applying Test Patterns

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

Scan cells operate in one of two modes: parallel mode or shift mode. In the multiplexed flip-flop scan style shown in [Figure A-3](#), the mode is controlled by the scan_enable pin. When the scan_enable signal is inactive, the scan cells operate in parallel mode; the input to each scan element comes from the combinational logic block. When the scan_enable signal is asserted, the scan cells operate in shift mode; the input comes from the output of the previous cell in the scan chain (or from the scan input port, if it is the first chain element). Other scan styles work similarly.

The target tester applies a scan pattern in the following sequence:

1. Select shift mode by setting the scan-enable port. This test signal is connected to all scan cells.
2. Shift in the input stimuli for the scan cells (pseudo-primary inputs) at the scan input ports.

3. Select parallel mode by disabling the scan-enable port.
 4. Apply the input stimuli to the primary inputs.
 5. Check the output response at the primary outputs after the circuit has settled and compare it with the expected fault-free response. This process is called parallel measure.
 6. Pulse one or more clocks to capture the steady-state output response of the nonscan logic blocks into the scan cells. This process is called parallel capture.
 7. Select shift mode by asserting the scan-enable port.
 8. Shift out the output response of the scan cells (pseudo-primary outputs) at the scan output ports and compare the scan cell contents with the expected fault-free response.
-

Scan Design Requirements

You achieve the best test coverage results when all nodes in your design are controllable and observable. Adding scan logic to your design enhances its controllability and observability. The rules governing the controllability and observability of scan cells are called test design rules.

Controllability of Sequential Cells

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

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

Using a partial-scan methodology, not all state elements need to be in the scan chain. As long as the nonscan state elements can be brought to any required state predictably through sequential operation, the circuit remains sufficiently controllable.

Observability of Sequential Cells

For sequential cells, test design rules require predictable capture of the next state of the circuit and visibility at the boundary of the design. In the context of scan design, you can ensure that sequential cells are observable if you successfully clock the scan cells in the circuit, and then shift their state to the scan outputs.

The following operations define circuit observability:

1. Observe the primary outputs of the circuit after scan-in.

Normally, this does not involve DFT and does not present problems.

2. Reliably capture the next state of the circuit.

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

3. Extract the next state through a scan-out operation.

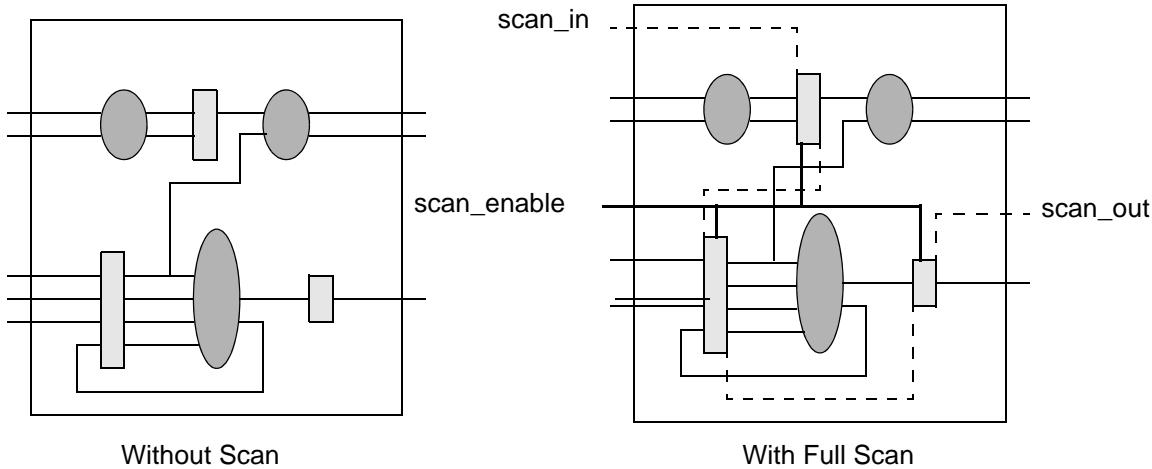
This process is similar to scan-in. The additional requirement is that the shift registers pass data reliably to the output ports.

Full-Scan Design

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

Full scan divides a sequential design into combinational blocks as shown in [Figure A-4](#). Ovals represent combinational logic; rectangles represent sequential logic. The full-scan diagram shows the scan path through the design.

Figure A-4 Scan Path Through a Full-Scan Design



Through pseudo-primary inputs, the scan path enables direct control of inputs to all combinational blocks. The scan path enables direct observability of outputs from all combinational blocks through pseudo-primary outputs. You can use the efficient combinational capabilities of TetraMAX to achieve high test coverage results on a full-scan design.

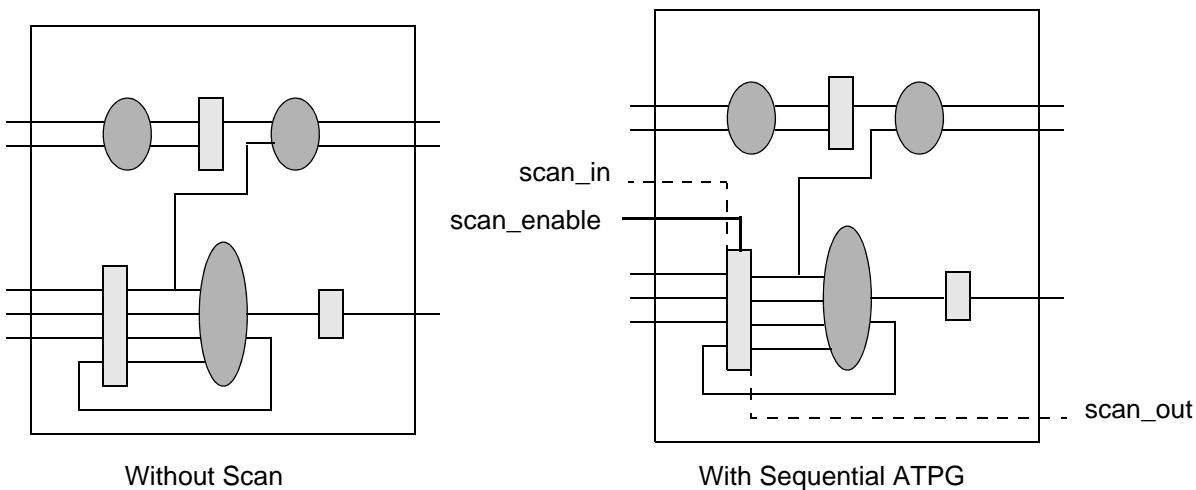
Partial-Scan ATPG Design

With a partial-scan design technique, the scan chains contain some, but not all, of the sequential cells in the design. A partial-scan technique offers a tradeoff between the maximum achievable test coverage and the effect on design size and performance.

The default ATPG mode of TetraMAX, called Basic-Scan ATPG, performs combinational ATPG. To get good test coverage in partial-scan designs, you need to use Fast-Sequential or Full-Sequential ATPG. The sequential ATPG algorithms perform propagation of faults through nonscan elements. For more information, see “[ATPG Modes](#)” on page 12-2.

Partial scan divides a complex sequential design into simpler sequential blocks as shown in [Figure A-5](#). Ovals represent combinational logic; rectangles represent sequential logic. The partial-scan diagram shows the scan path through the design after sequential ATPG has been performed.

Figure A-5 Scan Path Through a Partial-Scan Design



Typically, a partial-scan design does not allow test coverage to be as high as for a similar full-scan design. The level of test coverage for a partial-scan design depends on the location and number of scan registers in that design, and the ATPG effort level selected for the Fast-Sequential or Full-Sequential ATPG algorithm.

What Is Boundary Scan?

Boundary scan is a DFT technique that simplifies printed circuit board testing using a standard chip-board test interface. The industry standard for this test interface is the *IEEE Standard Test Access Port and Boundary Scan Architecture* (IEEE Std 1149.1).

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

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

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

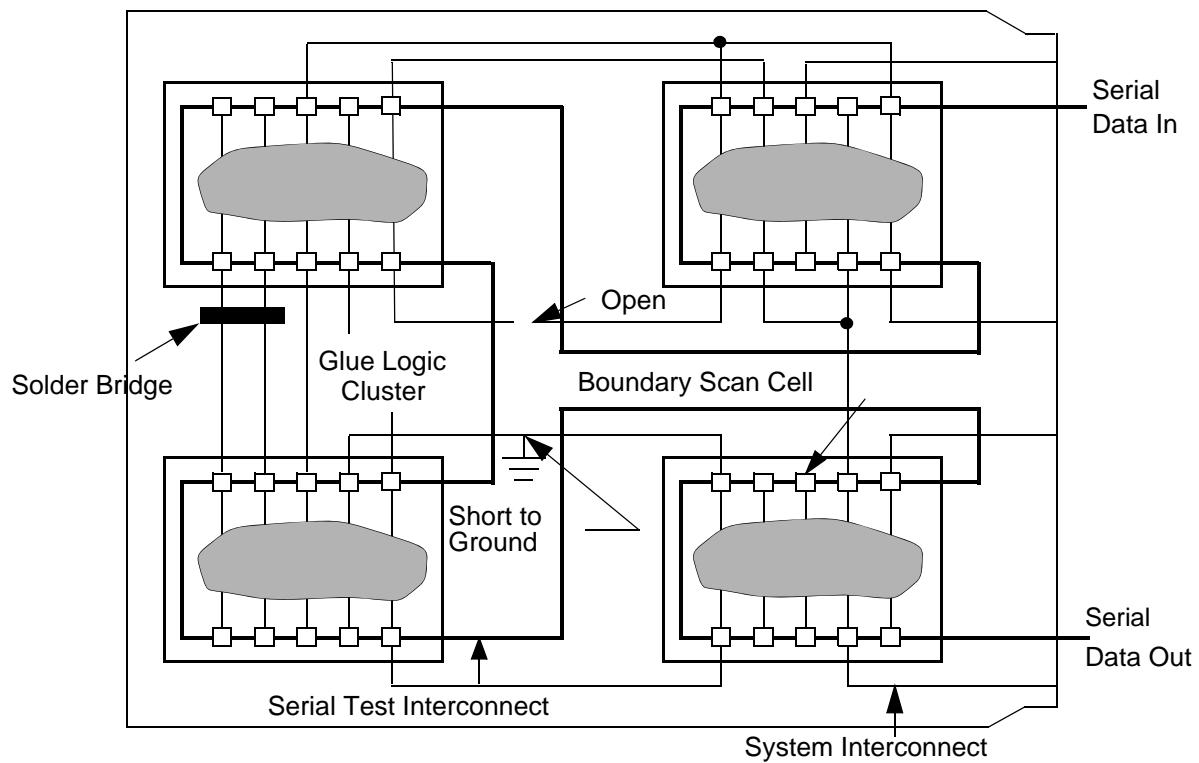
- Testing of the interconnect wiring on a printed circuit board for shorts, opens, and bridging faults
- Testing of clusters of non-boundary-scan logic
- Identification of missing, misoriented, or wrongly selected components
- Identification of fixture problems
- Limited testing of individual chips on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

Note:

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

[Figure A-6](#) shows a simple printed circuit board with several boundary scan ICs and illustrates some of the failures that boundary scan can detect.

Figure A-6 Board Testing With IEEE Std 1149.1 Boundary Scan



B

ATPG Design Guidelines

This appendix presents some design guidelines to facilitate successful ATPG and suggests sources of extra ports for test I/O. The design topics are discussed first in textual form. Next, selected design guidelines are illustrated with schematics. Finally, concise checklists for the design guidelines and port suggestions provide you with a quick reference as you implement your design.

This appendix contains the following sections:

- [ATPG Design Guidelines](#)
- [Ports for Test I/O](#)
- [Checklists for Quick Reference](#)

Synopsys DFT Compiler can help you implement some of the guidelines, as noted in the text. For more information, see the *DFT Compiler User Guide*.

ATPG Design Guidelines

This section provides guidelines for ATPG testing and offers suggestions for identifying ports to use for test I/O. The provided guidelines are not exhaustive, but if implemented, they can prevent many problems that commonly occur during ATPG testing.

Internally Generated Pulsed Signals

Guideline 1

While TetraMAX is in ATPG test mode, ensure that clocks and asynchronous set or reset signals come from primary inputs. Your design should not include internally generated clocks or asynchronous set or reset signals.

- Do not use clock dividers in ATPG test mode. If your design contains clock dividers, bypass them in ATPG test mode. A scan chain must shift one bit for one scan clock. Use the TEST signal to control the source of the internal clocks, so that in ATPG test mode you can bypass the clock divider and source the internal clocks from the primary CLK output.
- Do not use gated clocks such as the one shown in [Figure B-1](#) in ATPG test mode. If your design contains clock gating, constrain the control side of the gating element while in ATPG test mode.

[Figure B-2](#) and [Figure B-3](#) show two solutions. In [Figure B-2](#), the TEST input blocks the path from CLK to register B. However, B cannot be used in a scan chain.

Figure B-1 Gated Clock: A Problem

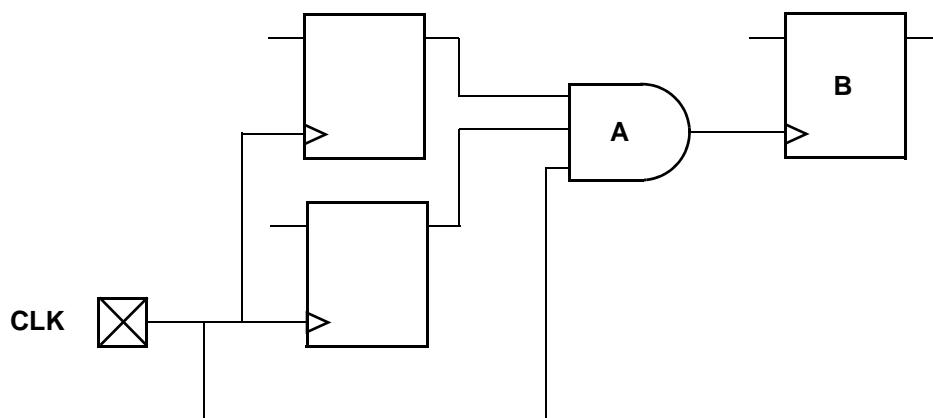
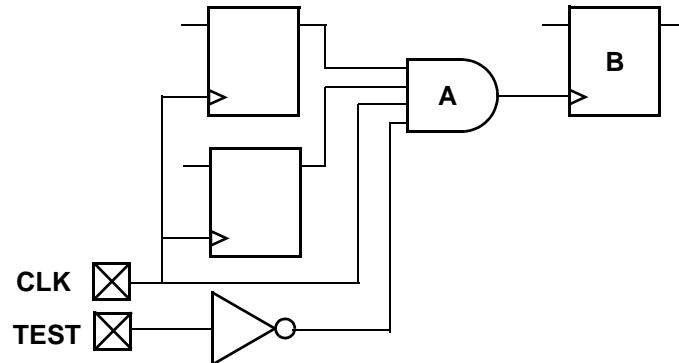
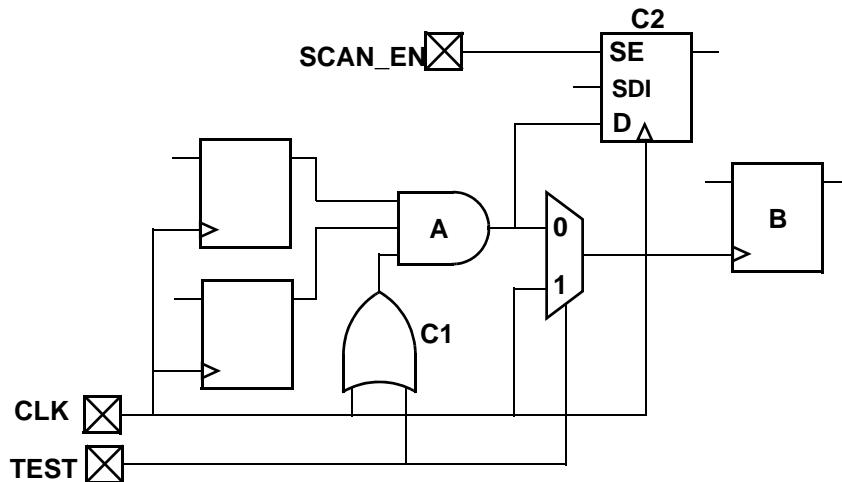


Figure B-2 Gated Clock: Solution 1



In Figure B-3, the TEST input controls a MUX that changes the clock source for register B. Optionally adding gates C1 and C2 provides observability for the output of gate A; otherwise, gate A is unobservable and all faults into A are ATPG untestable.

Figure B-3 Gated Clock: Solution 2



- Do not use phase-locked loops (PLLs) as clock sources in ATPG test mode. If your design contains PLLs, bypass the clocks while in ATPG test mode.
- Do not use pulse generators in ATPG test mode, such as the one shown in Figure B-4. If your design contains pulse generators, bypass them using a MUX with the select line constrained to a constant value or shunted with AND or OR logic so that the pulse generators do not pulse while in ATPG test mode, as shown in Solution 1 and Solution 2 in Figure B-5.

In Solution 1, the TEST input disables the pulse generator. However, using this solution, any sequential elements that use N2 as a clock source no longer have a clock source. In Solution 2, the TEST input multiplexes out the original pulse and replaces it with access from a top-level input port.

Figure B-4 Pulse Generators: A Problem

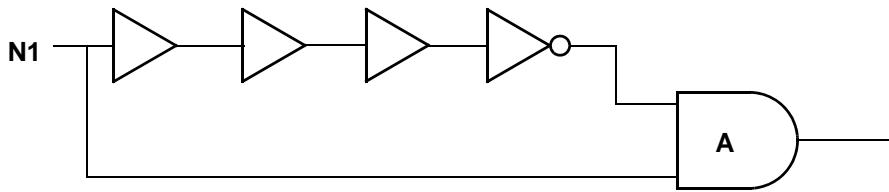
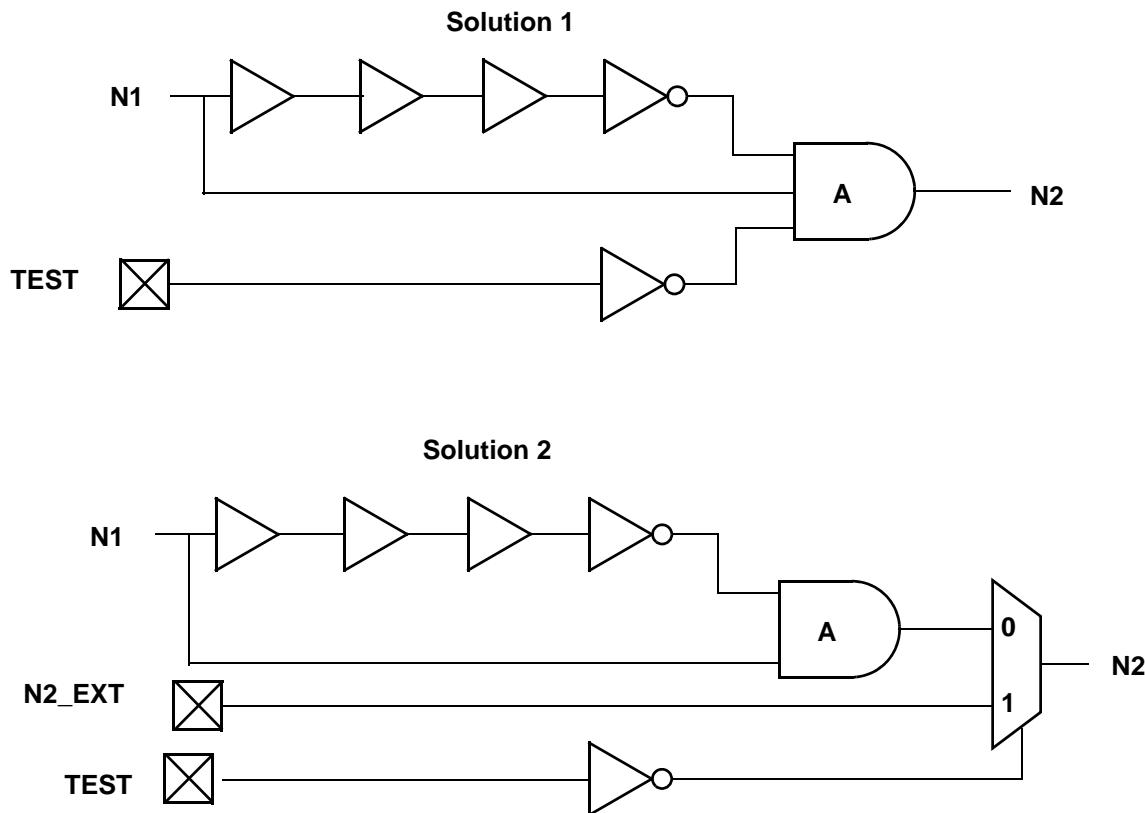


Figure B-5 Pulse Generators: Two Solutions



- Do not use a power-on reset circuit in ATPG test mode. A power-on reset circuit is essentially an uncontrolled internal clock source that operates when the power is initially applied to the circuit. See [Figure B-6](#).

To prevent a power-on reset circuit from operating during test, you can do either of the following:

- Use the test mode control signal to multiplex the power-on reset signal so that it comes from an existing reset input or some other primary input during test. See [Figure B-7](#).

- Use the test mode control signal to block the power-on reset source so that it has no effect during test. See [Figure B-8](#).

The first of these two methods is usually better because it is less likely to cause a reduction in test coverage.

Figure B-6 Power-On Reset Circuit Configuration to Avoid

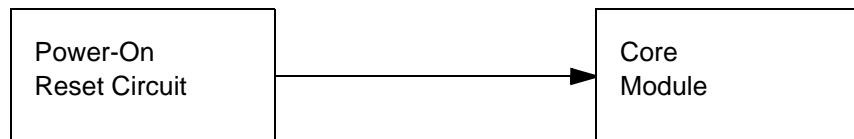


Figure B-7 Power-On Reset Circuit Test Method 1

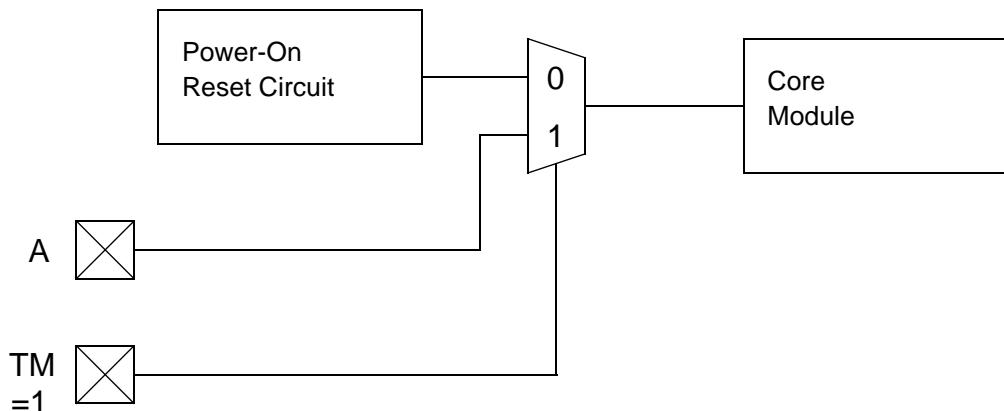
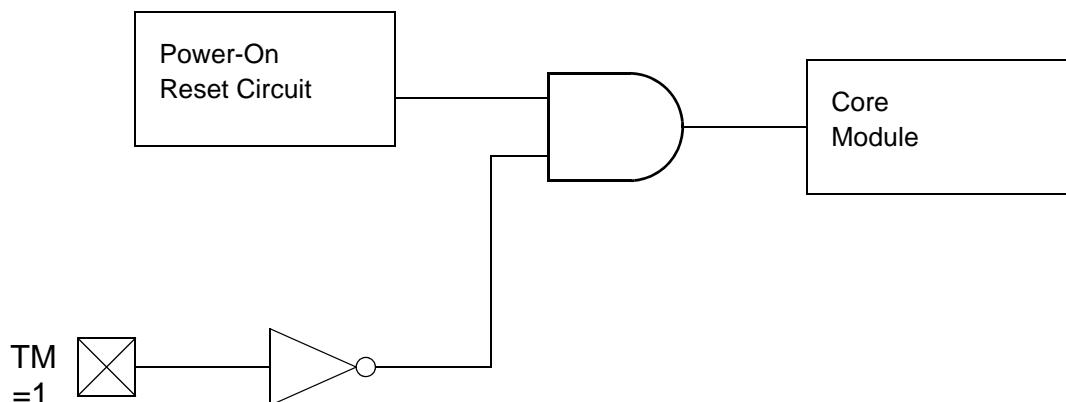


Figure B-8 Power-On Reset Circuit Test Method 2



Clock Control

Guideline 2

While TetraMAX is in ATPG test mode, provide complete control of clock paths to scan chain flip-flops. The clock/set/reset paths to scan chain elements must be fully controlled.

- If a clock passes through a MUX, constrain the select line of the MUX to a constant value while in ATPG test mode.
- If a clock passes through a combinational gate, constrain the other inputs of the gate to a constant value while in ATPG test mode. See [Figure B-9](#) and [Figure B-10](#).
- Pass clock signals directly through JTAG I/O cells without passing through a MUX, unless the MUX control can be constrained. This typically involves using a special JTAG input cell. [Figure B-11](#) shows a JTAG input cell with a MUX through which the signal passes; it is difficult to hold the MUX control constant. [Figure B-12](#) shows a modified JTAG input cell that has no MUX in the path.
- Avoid using bidirectional clocks or asynchronous set or reset ports while in ATPG test mode. If your design supports bidirectional clocks or asynchronous set or reset ports, force them to operate as unidirectional ports while in ATPG test mode. See [Figure B-13](#) and [Figure B-14](#).

Figure B-9 Problem: Clock Paths Through Combinational Gates

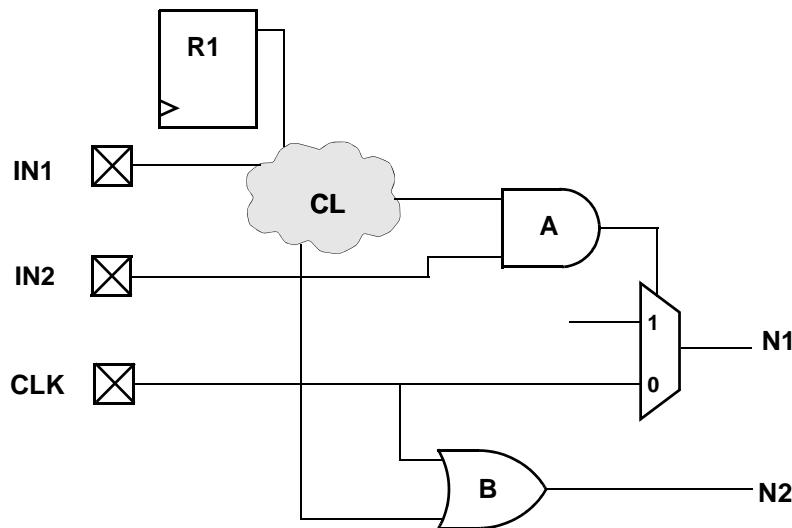


Figure B-10 Solution: Clock Paths Through Combinational Gates

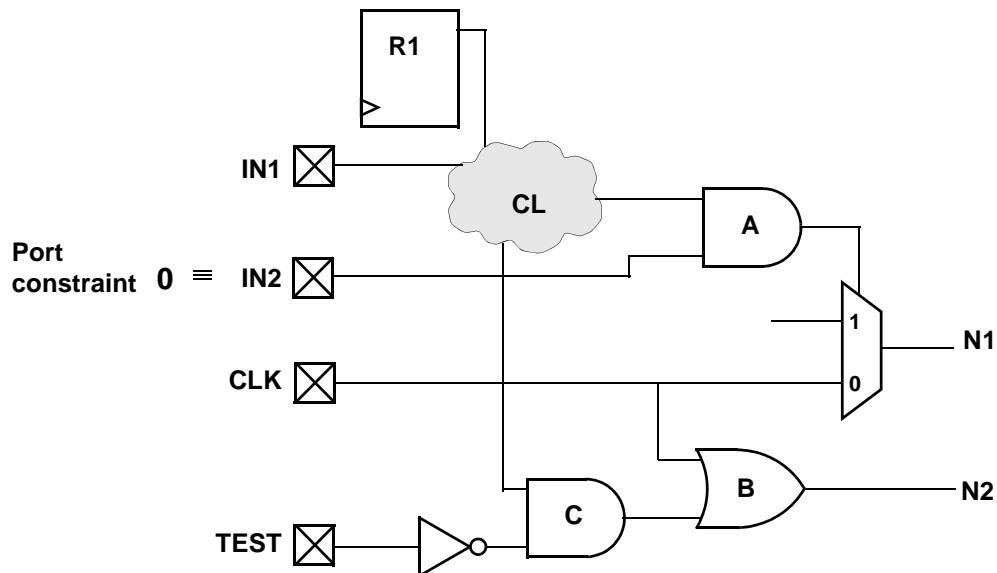


Figure B-11 Problem: Clock/Set/Reset Inputs and JTAG I/O Cells

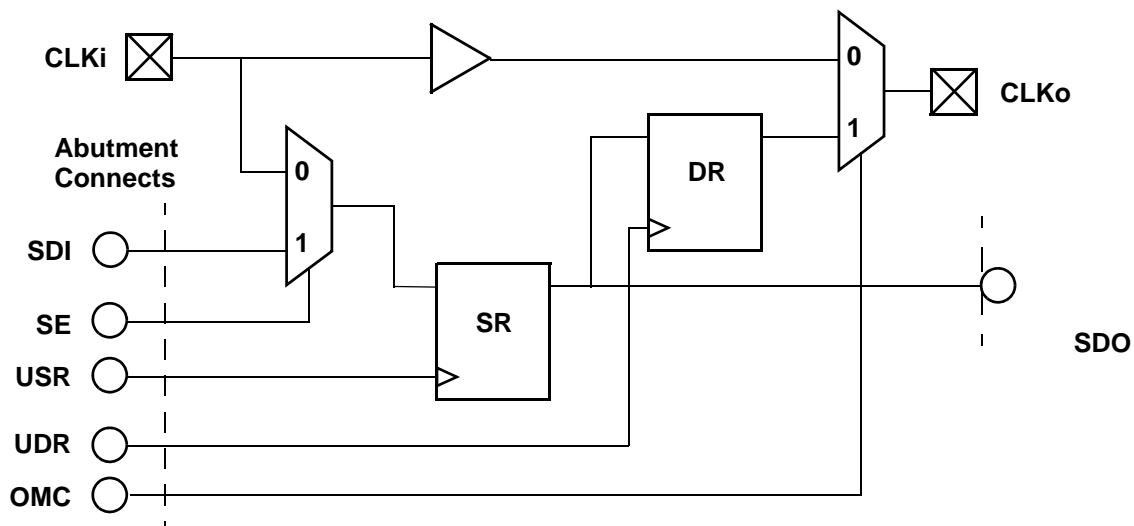


Figure B-12 Solution: Clock/Set/Reset Inputs and JTAG I/O Cells

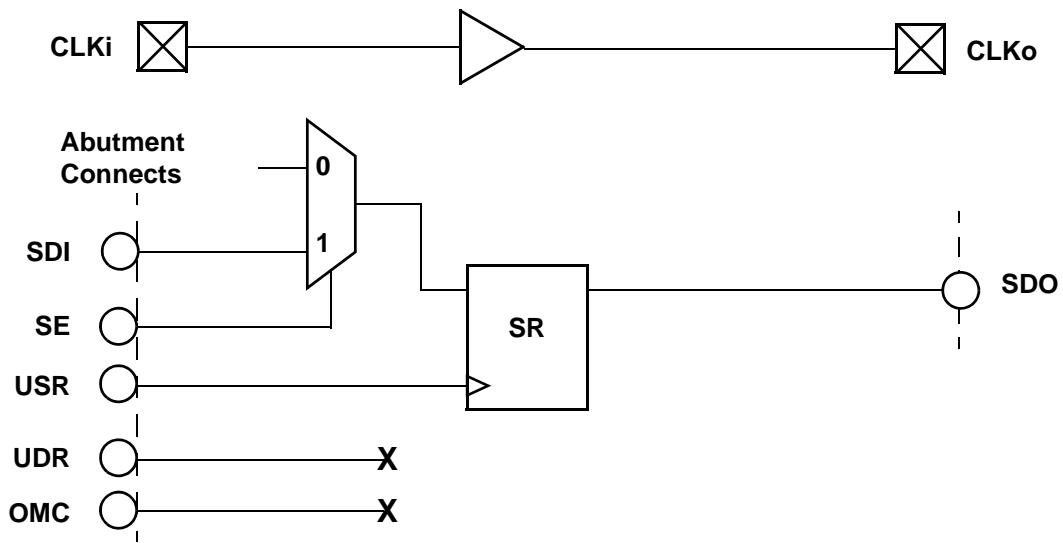


Figure B-13 Problem: Bidirectional Clock/Set/Reset

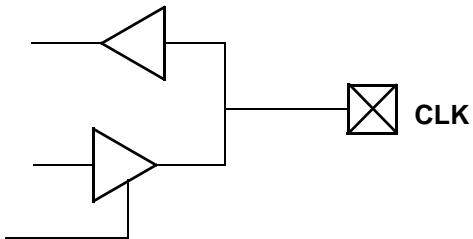
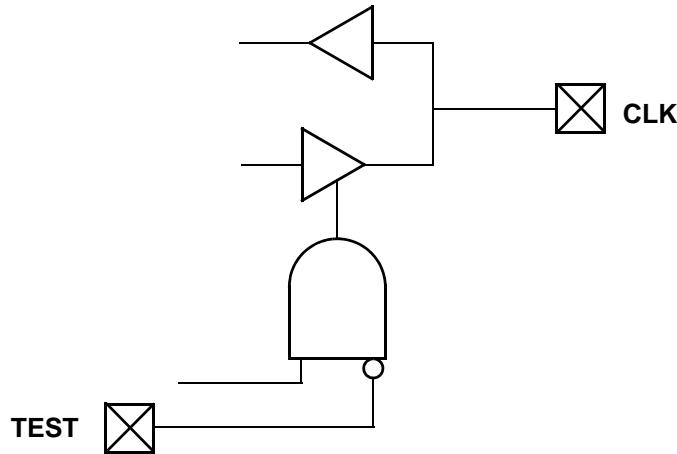


Figure B-14 Solution: Bidirectional Clock/Set/Reset



Pulsed Signals to Sequential Devices

Guideline 3

While TetraMAX is in ATPG test mode, do not allow an open path from a pulsed input signal (clock, asynchronous set/reset) to the data input of a sequential device.

- Do not allow a path from a pulsed input to both the data input and clock of the same flip-flop while TetraMAX is in ATPG test mode. As shown in [Figure B-15](#), the value of the data captured cannot be determined in the absence of timing analysis. If your design contains such a path, then while in ATPG test mode, shunt the path to either the data or clock pin with AND or OR logic, or with a MUX, as shown by Solution 1 and Solution 2 in [Figure B-16](#). In Solution 1, a controllable top-level input is used to replace the path of the clock/set/reset into the combinational cloud. In Solution 2, the TEST input blocks the path of the clock/set/reset into the combinational cloud so that it does not pass the clock pulse while in ATPG test mode.
- Do not allow a path from a pulsed input to both the data input and the asynchronous set or reset input of the same flip-flop while TetraMAX is in ATPG test mode. If your design contains such a path, while in ATPG test mode, shunt the path to either the data pin or the set/reset pin with AND logic, OR logic, or a MUX.

Figure B-15 Problem: Sequential Device Pulsed Data Inputs

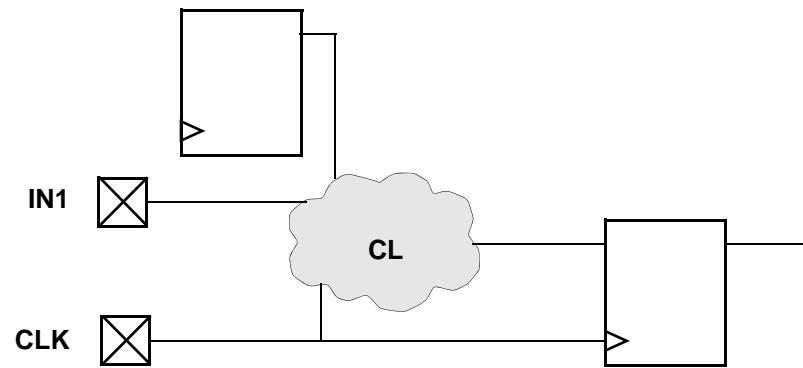
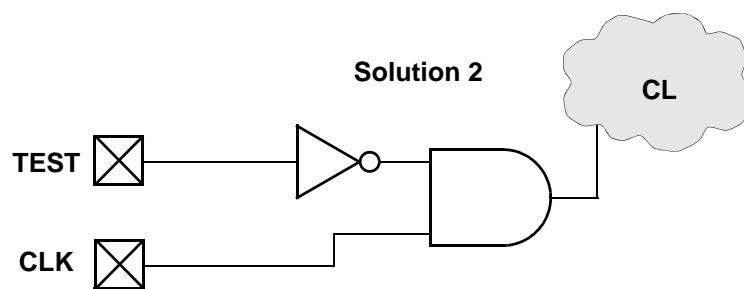
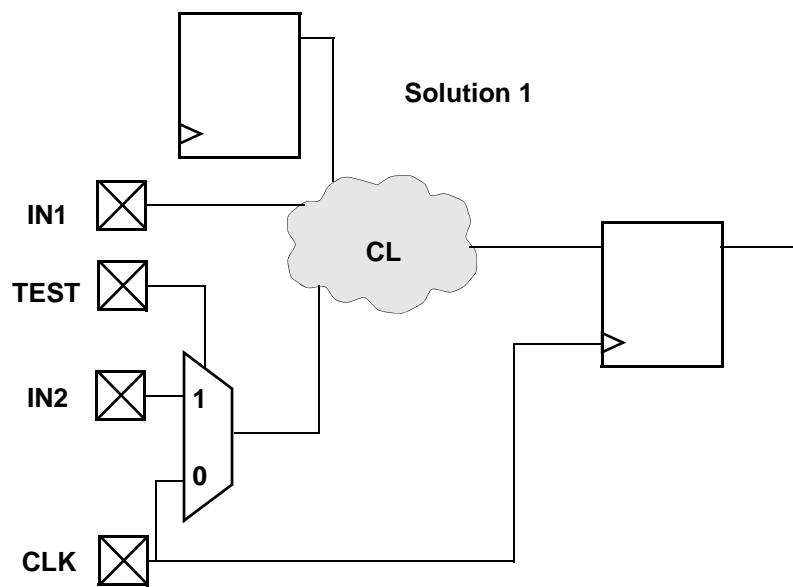


Figure B-16 Solution: Sequential Device Pulsed Data Inputs



Multidriver Nets

Guideline 4

For multidriver nets, ensure that exactly one driver is enabled during the shifting of scan chains in ATPG test mode. Plan this guideline into your design. For most designs with multidriver nets, there is danger of internal driver contention because shifting a scan chain has a random effect on the design state. See [Figure B-17](#).

Here are two methods for satisfying this design guideline:

- Have a primary input port that acts as a global override on internal driver enable signals in ATPG test mode, disabling all but one driver of the net and forcing that driver to an on state, as shown in [Figure B-18](#). This primary input port should be asserted during the scan chain load and unload operation. This design guideline is supported by DFT Compiler and is the default behavior of DFT Compiler.
- Use deterministic decoding on the driver enables. Use a 1-of- n logic to ensure that only one driver is enabled at all times and that at least one driver is enabled at all times, as shown in [Figure B-19](#). Deterministic decoding might not be appropriate for some designs. For example, for a design with hundreds of potential drivers, a 1-of- n decoder would be too large or would add too much delay to the circuit.

Figure B-17 Problem: Multidriver Nets

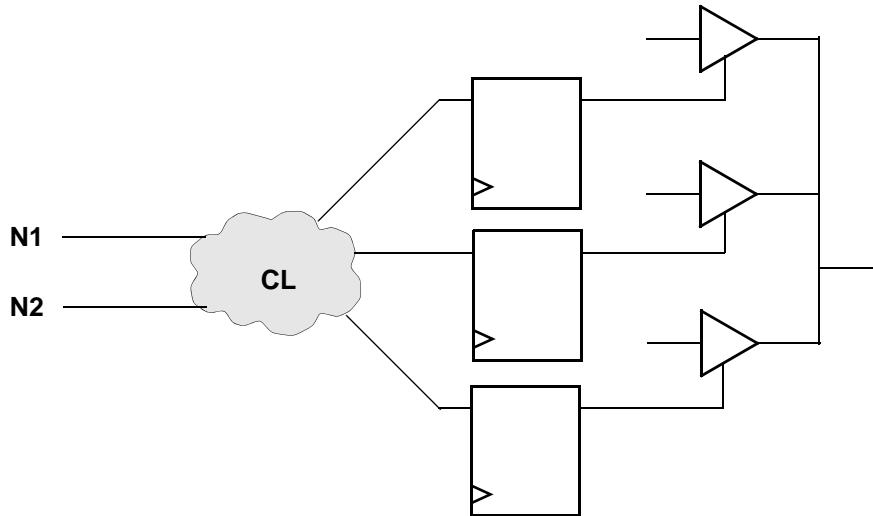


Figure B-18 Multidriver Nets: Global Override Input

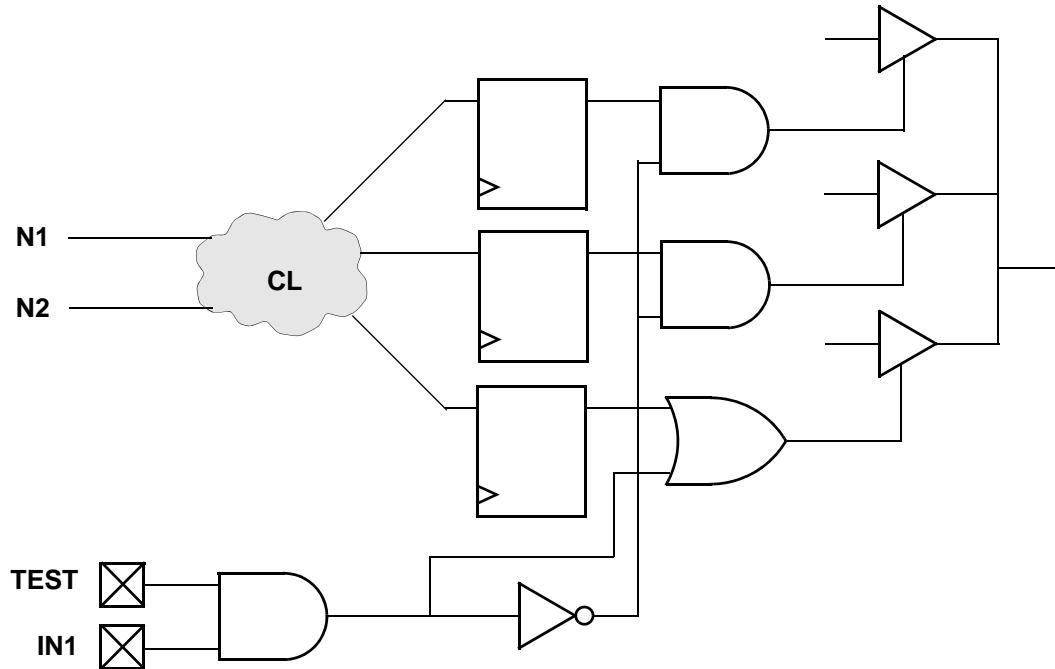
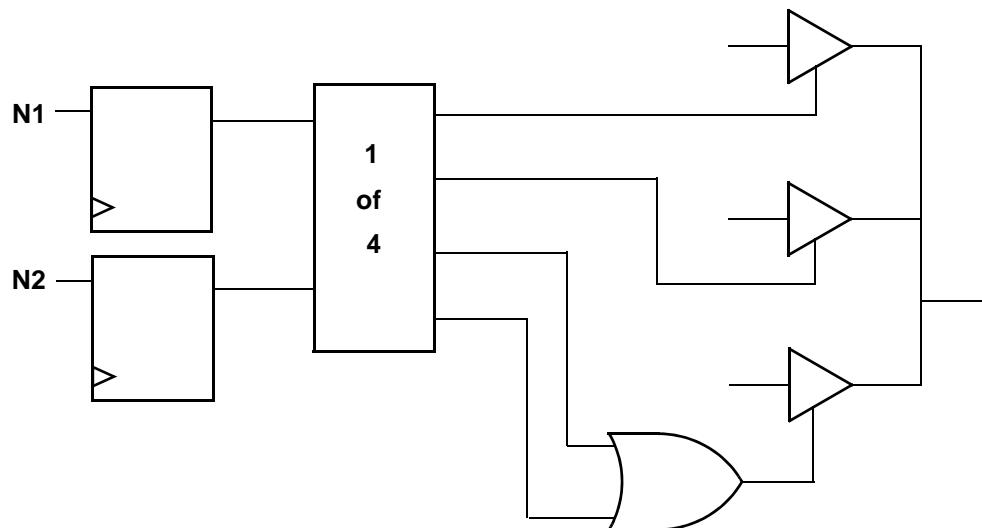


Figure B-19 Multidriver Nets: Deterministic Decoding



Bidirectional Port Controls

Guideline 5

Force all bidirectional ports to input mode while shifting scan chains in ATPG test mode, using a top-level port as control. See [Figure B-20](#) and [Figure B-21](#). In [Figure B-21](#), TEST controls the disabling logic and SCAN_EN ensures that the scan chain outputs are turned on.

The top-level port is often tied to a scan enable control port. However, there are advantages to performing this function on a different port, if extra ports are available, because keeping the control of the bidirectional ports separate from the scan enable gives the ATPG algorithm more flexibility in generating patterns.

Figure B-20 Problem: Bidirectional Port Controls

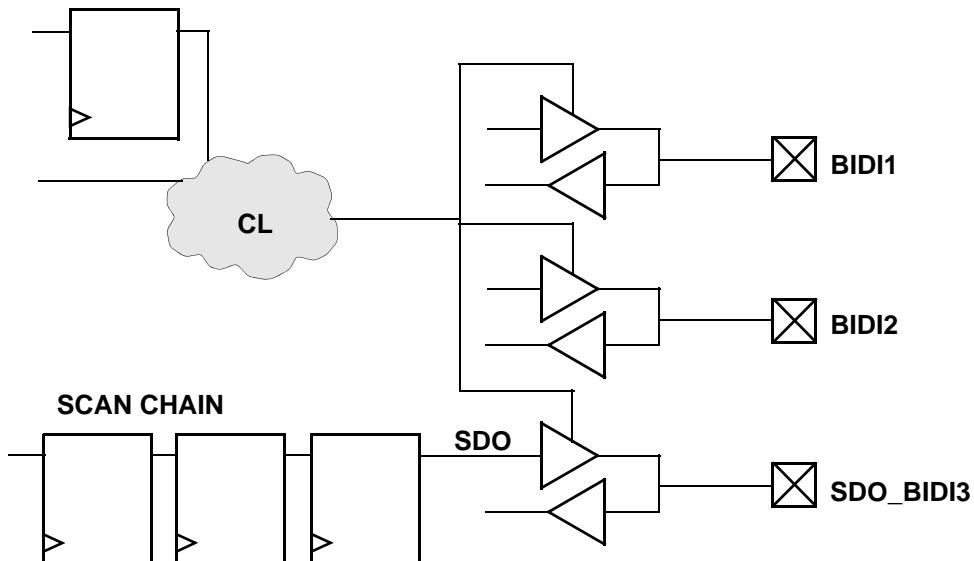
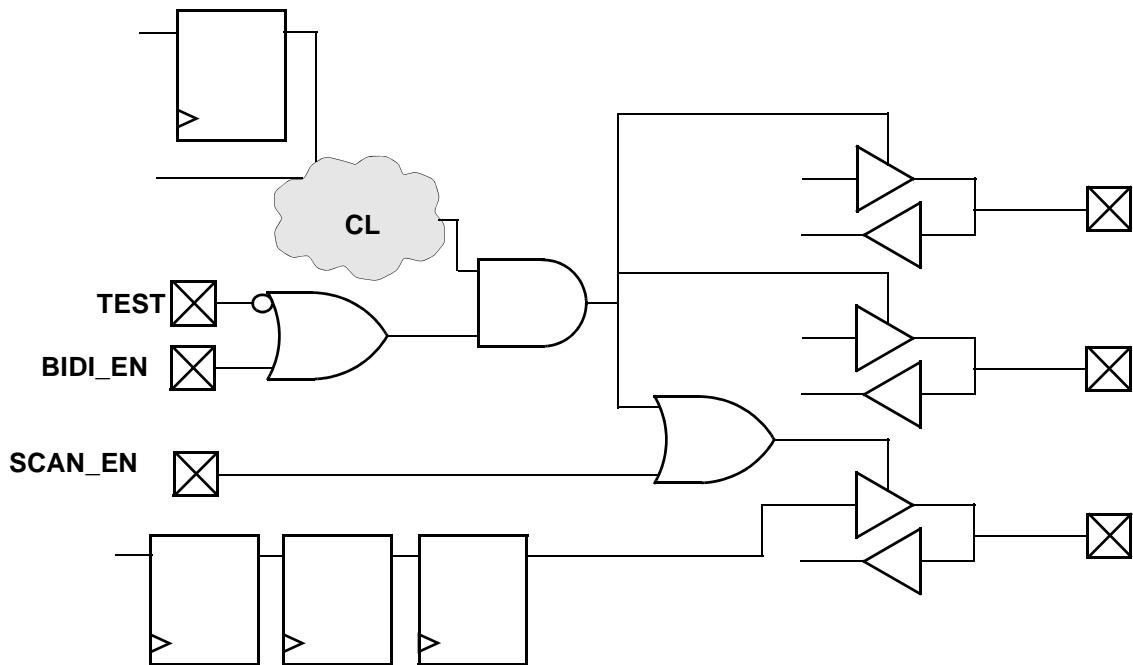


Figure B-21 Solution: Bidirectional Port Controls



If you follow this guideline along with Guideline 3, you can easily ensure that no internal or I/O contention can occur during scan chain load/unload operations.

This guideline is supported by DFT Compiler (using a single pin) and is the default behavior.

Guideline 6

Force scan chain outputs that use bidirectional or three-state ports into an output mode while shifting scan chains in ATPG test mode, using a top-level port (usually SCAN_ENABLE), as shown in [Figure B-21](#).

This guideline is the exception to Guideline 5 and is automatically supported by DFT Compiler if you specify a bidirectional port for use as a scan chain output.

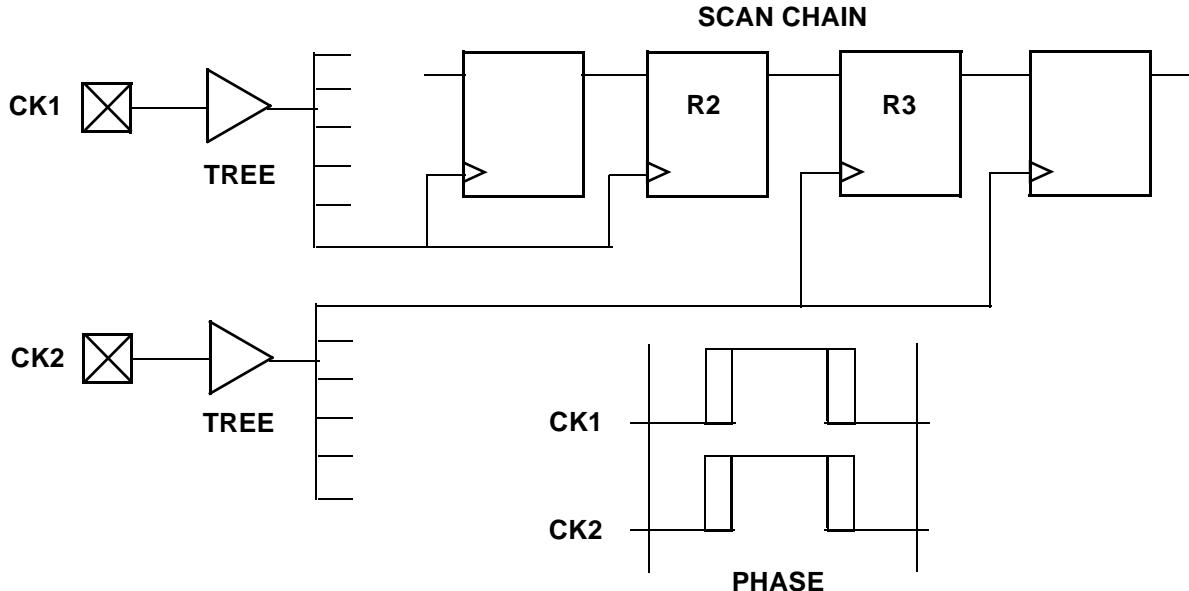
Clocking Scan Chains: Clock Sources, Trees, and Edges

Guideline 7

Use a single clock tree to clock all flip-flops in the same scan chain. If the design contains multiple clock trees, insert resynchronization latches in the scan data path between scan cell flip-flops that use different clock sources.

In Figure B-22, the two clock sources can cause a race condition. For example, if CK1 leads CK2 because of jitter or differences in clock tree delays, then R2 clocks before R3. Because R2's output is changing while R3 is clocking, a race condition results.

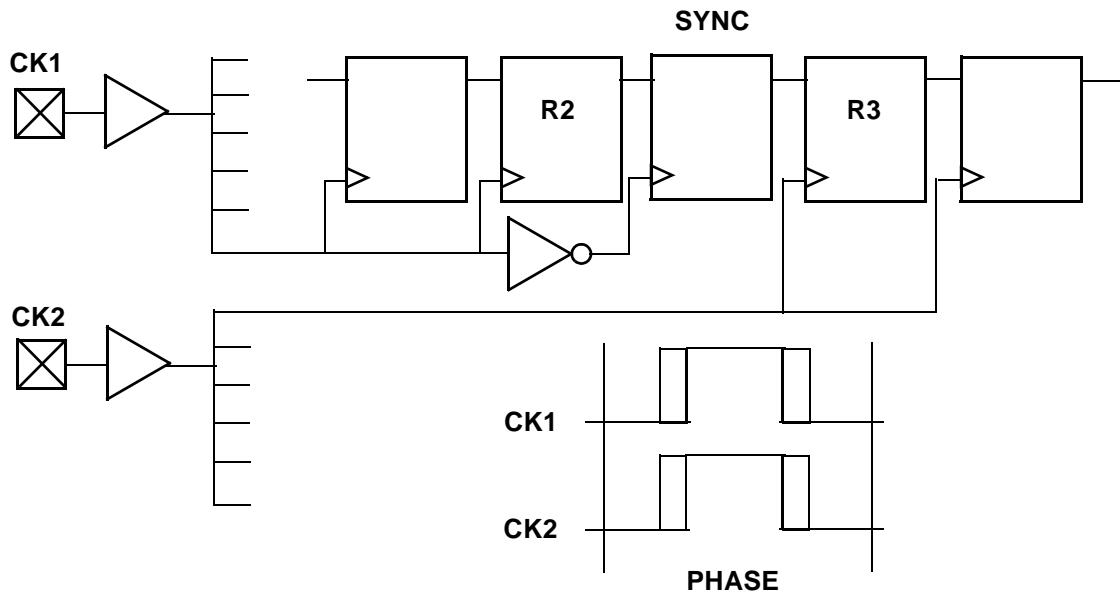
Figure B-22 Problem: Multiple Clock Trees



In Figure B-23, there is a resynchronization register or latch (SYNC) between R2 and R3, which is clocked by the opposite phase of the clock used for R2.

This design guideline is supported by DFT Compiler and is the default behavior of DFT Compiler.

Figure B-23 Solution: Multiple Clock Trees



Guideline 8

Treat each clock tree as a separate clock source in designs that have a single clock input port but multiple clock tree distributions.

Sometimes a design has a single clock input port but uses multiple clock tree distributions to produce “early” and “standard” clocks, as shown in [Figure B-24](#). Under these conditions, treat each clock tree as a separate clock source. Insert resynchronization latches between scan cells where the clock source switches from one clock tree to another, as shown in [Figure B-25](#).

This design style is supported by DFT Compiler but is not the default method of DFT Compiler.

Figure B-24 Problem: Single Clock With Multiple Clock Trees

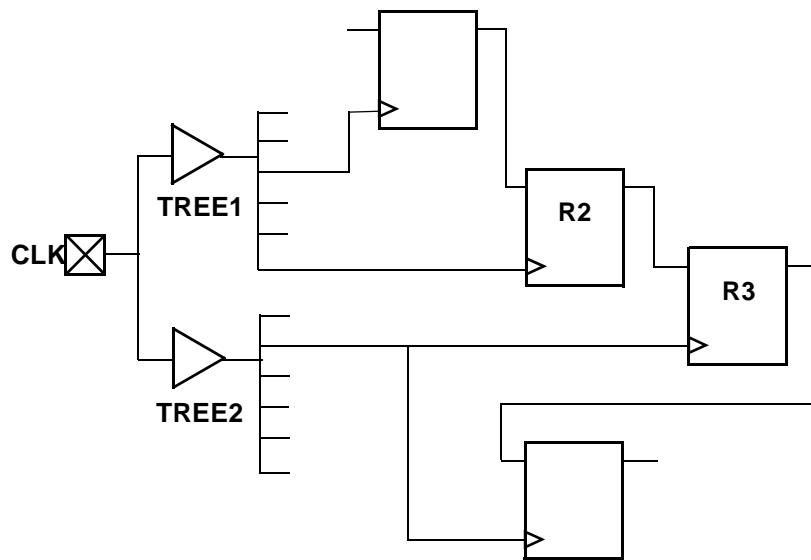
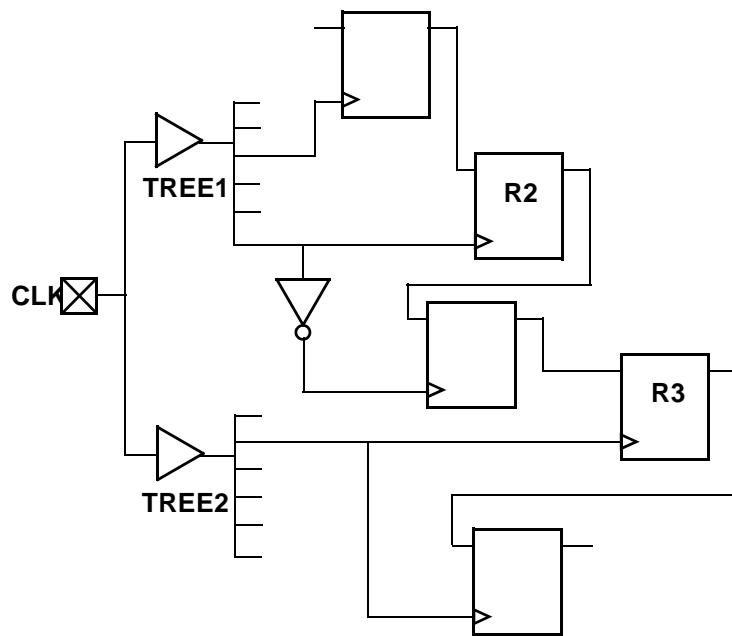


Figure B-25 Solution: Single Clock With Multiple Clock Trees



Guideline 9

If possible, clock all flip-flops on the same scan chain on the same clock edge. If this is not possible, then group together all flip-flops that are clocked on the trailing clock edge and place them at the front of the scan chain (closest to the scan chain input); and group together all flip-flops that are clocked on the leading clock edge and place them closest to the scan chain output.

In [Figure B-26](#), B1 and B2 are always loaded with the same data as A1 and A4, respectively, during scan chain loading, because they are clocked on the trailing edges. Thus, parts of the circuit that require A1 and B1 (or A4 and B2) to have opposite values are untestable.

In [Figure B-27](#), the scan chain registers are ordered so that all of the trailing-edge cells are grouped together at the front of the scan chain. B1 and B2 can be set independently of A1 and A4.

This design guideline is automatically implemented by DFT Compiler if you allow it to mix clock edges on a scan chain.

Figure B-26 Problem: Mixed Clock Edges on a Scan Chain

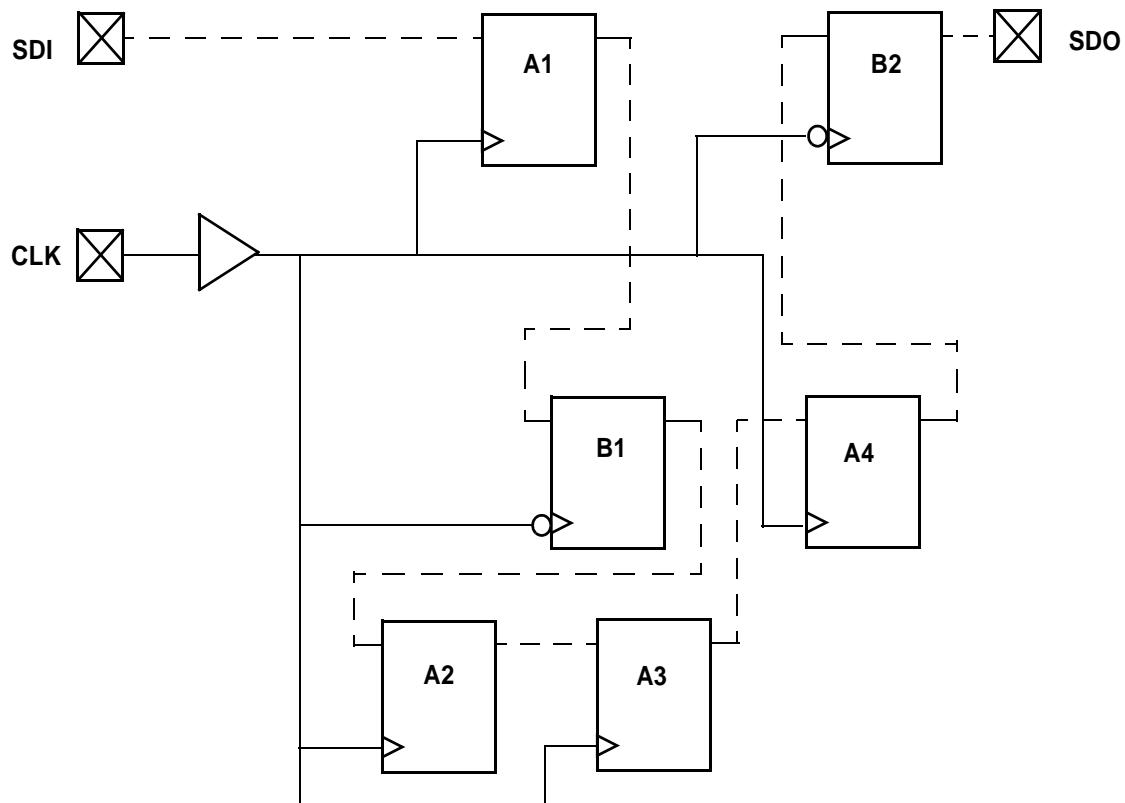
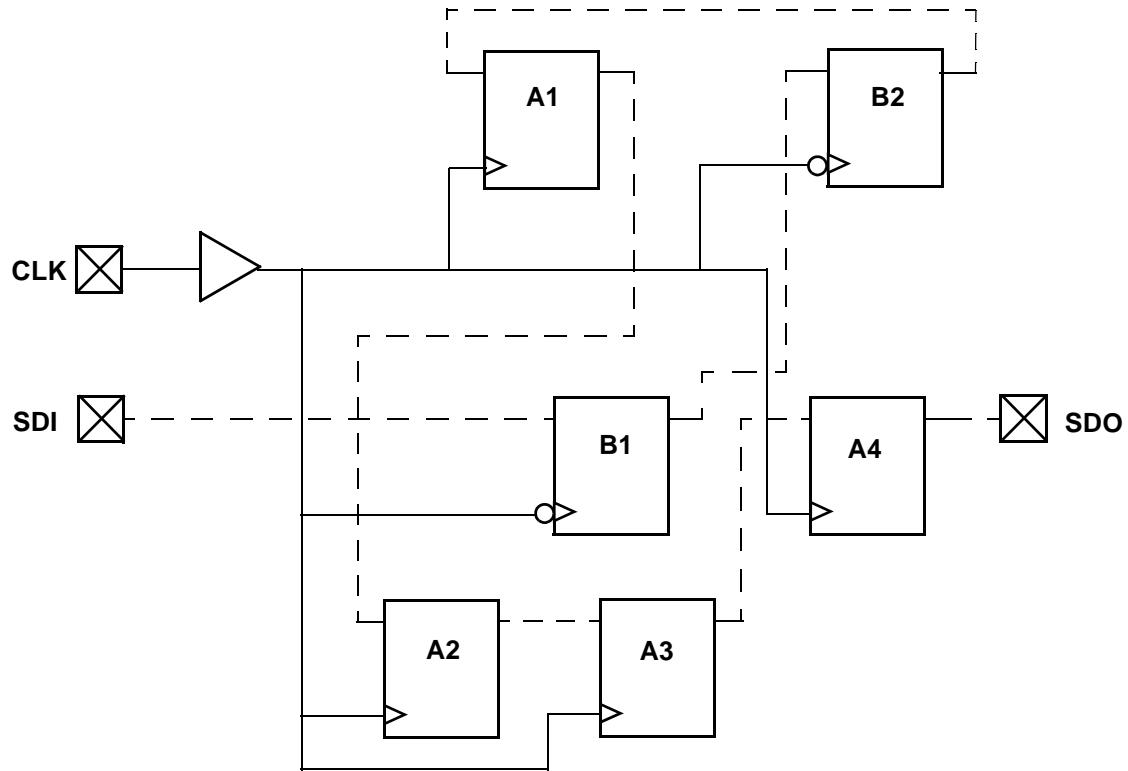


Figure B-27 Solution: Mixed Clock Edges on a Scan Chain



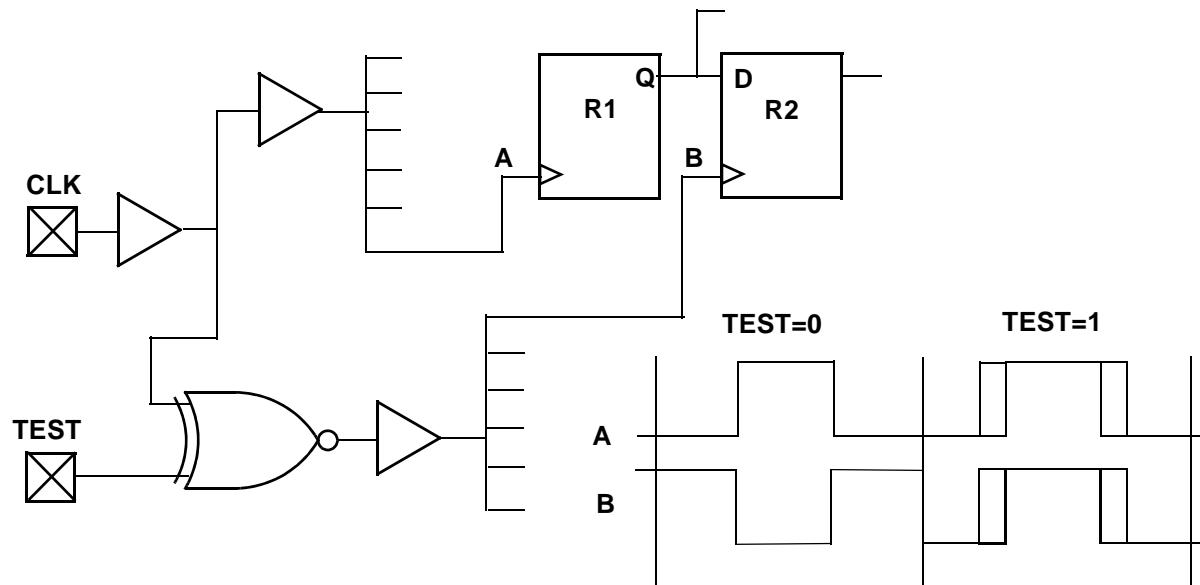
Guideline 10

Do not mix XNOR clock inversion techniques and clock trees.

A common design technique when both edges of the same clock are used for normal operation of scan chain flip-flops is to use an XNOR in place of an INV to form the opposite clock polarity. Then, in test mode, the XNOR can be switched from an inverter into a buffer.

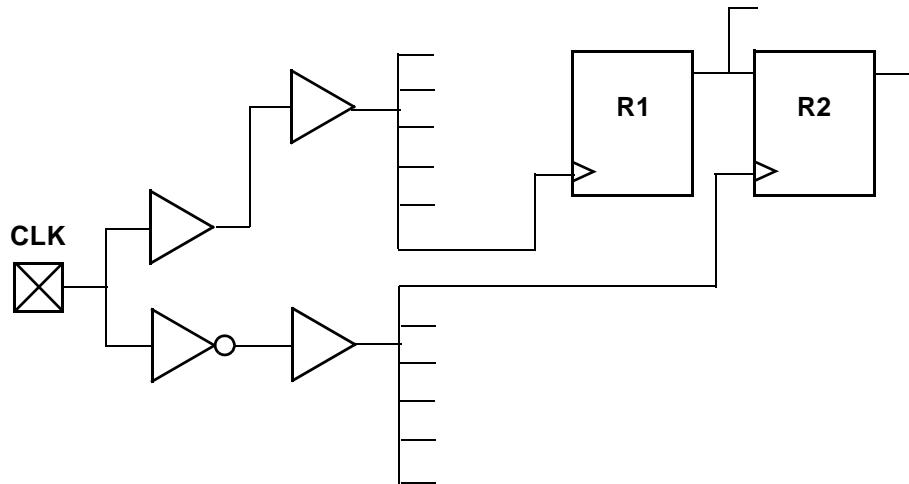
This technique is not advisable unless you can analyze the timing during test mode to ensure that no timing violations can occur during the application of any clocks. While in normal operation, there are essentially two clock zones of opposite phase. The phasing of the two clocks is such that reasonable timing is achieved between flip-flops that are on opposite phases of the clock. When one of the clocks is no longer inverted, two clock tree distributions are driven by the same-phase signal, resulting in timing-critical configurations in ATPG mode that do not exist in normal functional mode. See [Figure B-28](#).

Figure B-28 Problem: XNOR Clock Inversion and Clock Trees



To prevent this problem, replace the XNOR gate with an inverter, as shown in [Figure B-29](#). If you need the XNOR function, use it locally in the vicinity of the affected gates, rather than on the input side of a clock tree.

Figure B-29 Solution: XNOR Clock Inversion and Clock Trees



Protection of RAMs During Scan Shifting

Guideline 11

To protect RAMs from random write cycles, disable the RAM write clock or write enable lines while shifting scan chains in ATPG test mode.

In ATPG test mode, RAMs must remain undisturbed by random write cycles while the scan chains are being shifted. You can accomplish this by disabling the write clock or write enable line to each data write port during ATPG test mode. Often, the SCAN_ENABLE control is used for this function, coupled with an AND or OR gate, as appropriate.

However, to also achieve controllability over the write port, use a separate top-level input other than SCAN_ENABLE. The RAM write control is usually used as a pulsed port (RZ/RO), while the SCAN_ENABLE is a constant value (NRZ/NRO). Trying to achieve both at once usually presents problems that can be avoided by using separate ports.

RAM and ROM Controllability During ATPG

Guideline 12

If you want controllability of RAMs and ROMs for ATPG generation, connect their read and write control pins directly to a top-level input during ATPG test mode. This is most conveniently accomplished by using a MUX, which switches control from an internal to a top-level port. Multiple RAMs can share the same control port for the write port.

[Figure B-30](#), if the registers are in scan chains, random patterns that occur while loading and unloading scan chains are written to the RAMs. Thus, the RAM contents are unknown and treated as X.

[Figure B-31](#), MUX controls activated by TEST mode bring the write control signals up to the top-level input ports.

For achieving controllability and higher test coverage, direct control of write ports is more important than control of read ports.

Figure B-30 Problem: RAM/ROM Control

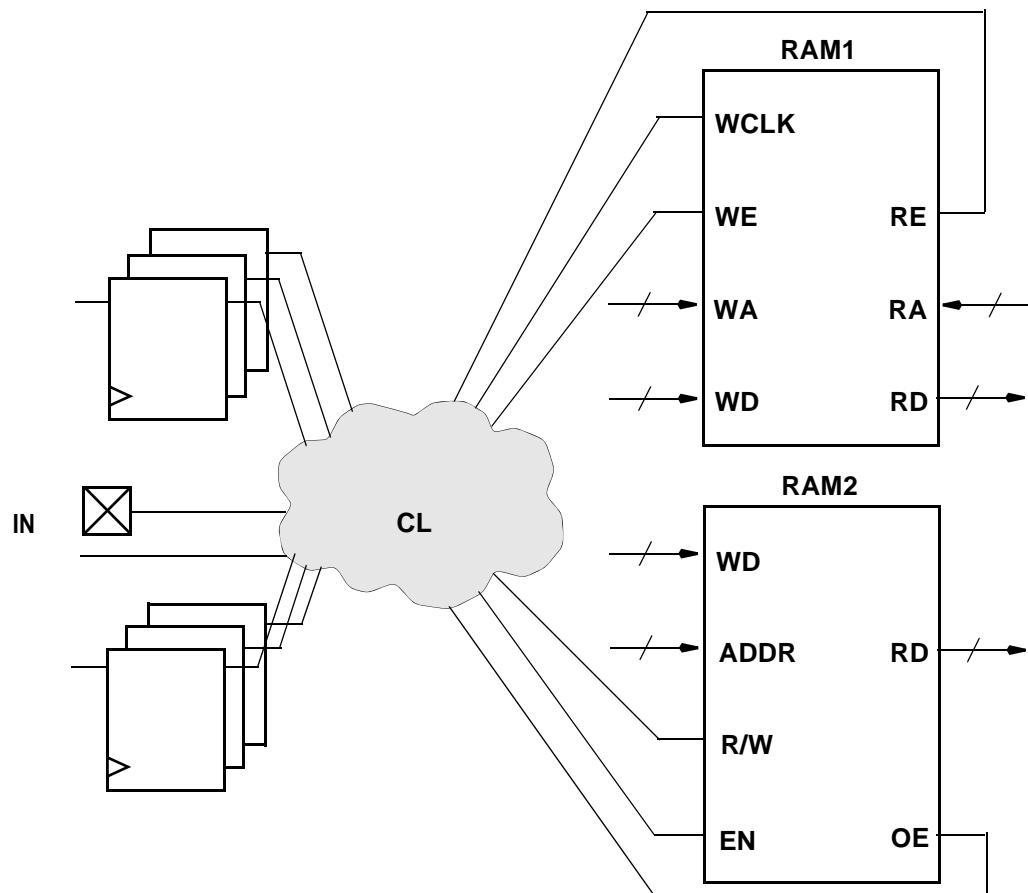
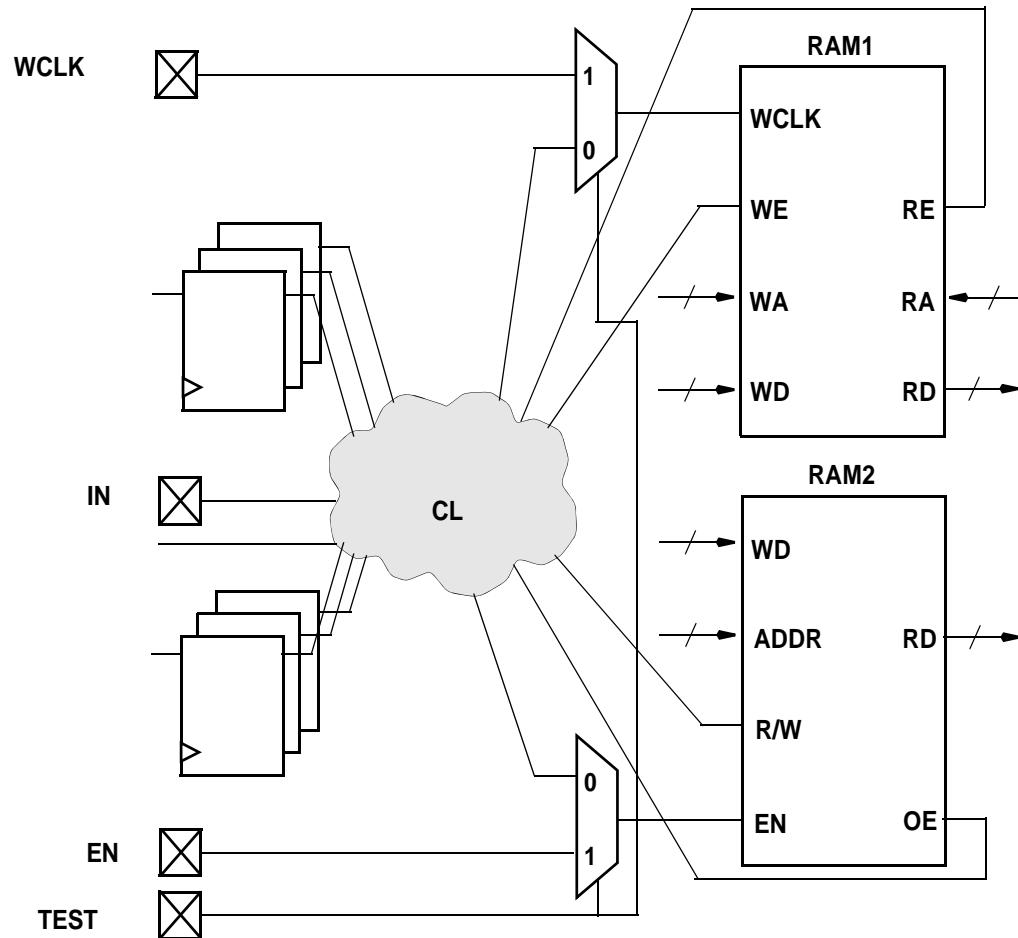


Figure B-31 Solution: RAM/ROM Control



Pulsed Signal to RAMs and ROMs

Guideline 13

Do not allow an open path from a pulsed signal to a data, address, or control input of a RAM or ROM (except read/write control) while in ATPG test mode.

If a combinational path exists from a defined clock or asynchronous set or reset port to a data, address, or control pin of a RAM or ROM, the ATPG algorithm treats the memory device as filled with X. The exceptions are the read clock and write clock signals, which are operated in a pulsed fashion but should not be mixed with a defined clock.

In Figure B-32, the address or data inputs are coupled with a clock/set/reset port, so their values are not constant while capture clocks are occurring elsewhere in the design. The result is that RAM read and write data cannot be determined; Xs are used instead.

In Figure B-33, the TEST input disables pulsed paths during ATPG test mode.

Figure B-32 Problem: RAMs/ROMs and Pulsed Signals

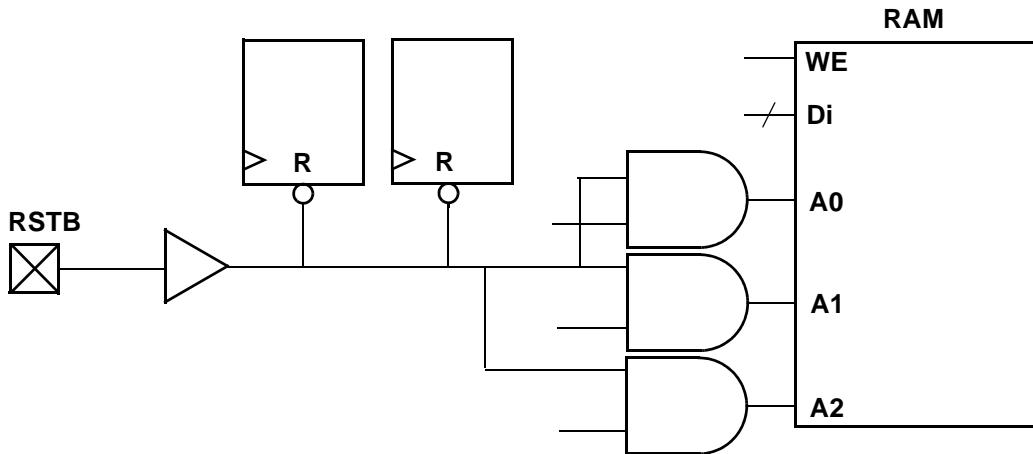
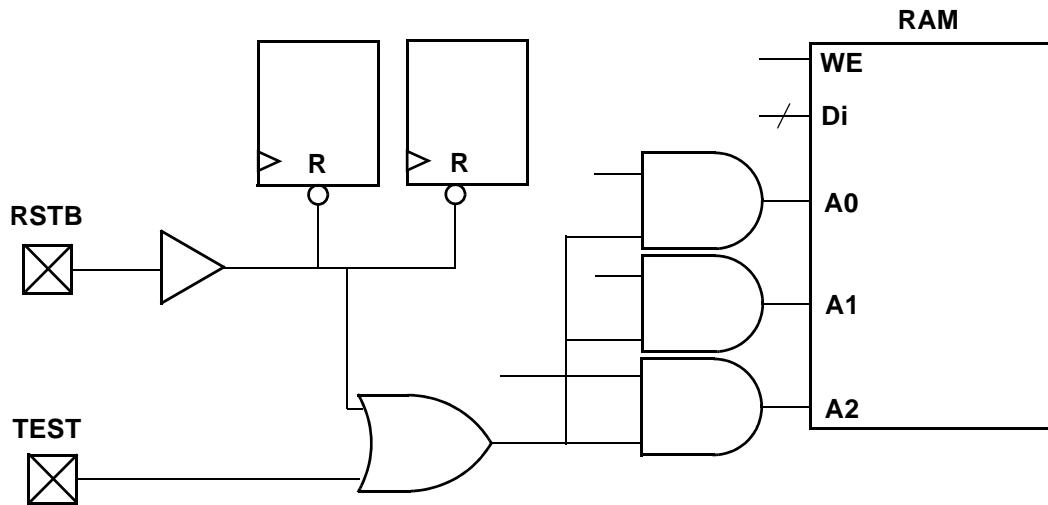


Figure B-33 Solution: RAMs/ROMs and Pulsed Signals



Bus Keepers

Guideline 14

While in ATPG test mode, do not allow a combinational gate path from any pulsed port to drive the enable controls of three-state drivers that contribute to a multidriver net, as illustrated in [Figure B-34](#). In [Figure B-35](#), the TEST input redirects the control to a top-level port, and the port is constrained to a value that does not affect the driver enables. Then, on the tester and during simulation, the port is driven with the same signal as that on the original CLK port.

A common practice is to gate all internal driver enables with some phase of a clock so that all drivers are off during the first half of each cycle and one driver is on during the second half. This practice solves some contention problems that occur during the transition of one driver off to another driver on, but it renders bus keeper usage impossible in ATPG test mode.

Figure B-34 Problem: Bus Keepers

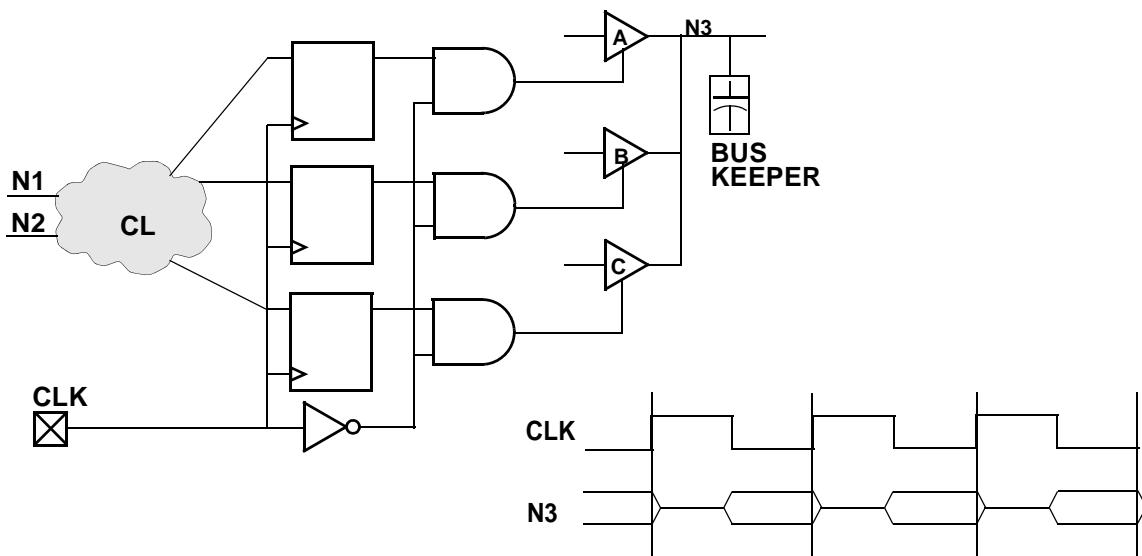
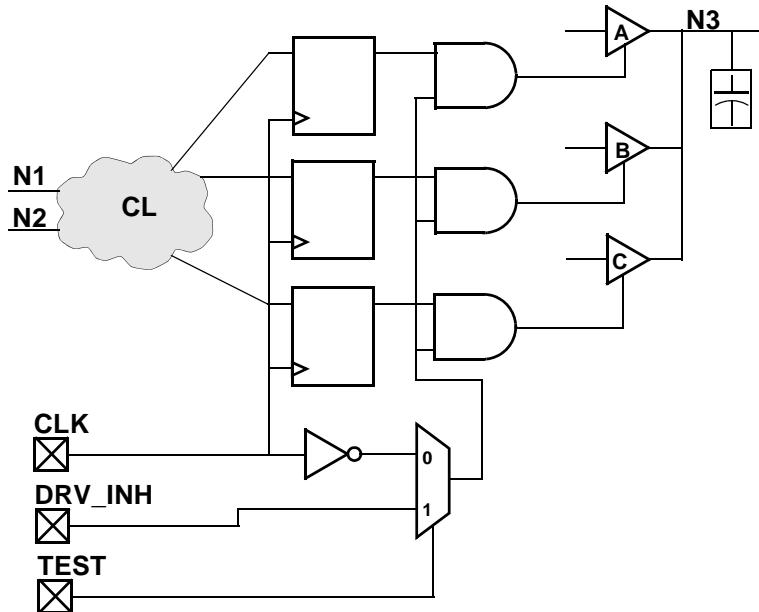


Figure B-35 Solution: Bus Keepers



Guideline 15

When using bus keepers, ensure a non-Z state on a multidriver net by the end of the `load_unload` procedure.

Guideline 16

When using bus keepers, do not allow the nonclocked events that occur before the system capture clock to disturb the multidriver net.

The system capture cycle should not disturb the multidriver net, at least not until after the clock/set/reset pulse, unless a change on a primary input enables one of the drivers and drives a known value on the net.

When you use a bus keeper, you expect it to retain the last value driven on the bus. Therefore, you do not need to design the driver enable controls so that one driver is always on. However, if the DRC analysis of the bus keeper finds violations, the beneficial effects possible with a bus keeper are ignored.

When no driver is enabled on the multidriver net, the bus assumes a Z or X state. When a Z passes through some other internal gate, it becomes an X; thus, an internal source generates and propagates a multitude of X states to observe points (for example, output ports and scan cells), which must be masked off in the ATPG patterns. There is a significant

increase in the number of pattern bits that the tester must mask off; thus, you can obtain patterns that are legal and generate high test coverage but are unusable on many testers because of the excessive number of compare masks required.

Ports for Test I/O

You might find that your design has no extra ports that can be dedicated to test I/O. This section offers some suggestions for identifying functional ports that can be redefined for test purposes during ATPG test mode.

At a minimum, only one extra port, TEST, is required. With TEST asserted, you can redefine the function of the other ports of the design for use as scan chain inputs or outputs, and as ATPG controls (scan_enable, bidi_disable, ram_write_disable). Of course, before you redefine normal functional ports, use any extra ports that are available.

Existing Ports for Scan Chain Input and Output

The best port to redefine as a scan chain input is one that already feeds into a flip-flop that will be placed in a scan chain.

The best port to redefine as a scan chain output is one that already comes directly from a flip-flop output that will be placed in a scan chain. By default, DFT Compiler chooses such ports as scan outputs.

Existing Ports for Any Test Usage

The following guidelines apply to the selection of existing ports to be used for any test purpose:

- You can redefine a three-state output or a standard output as a bidirectional port and use it in input mode with TEST asserted.
- One of the best ports to redefine as a shared port is an input port that feeds directly into a flip-flop that is part of a scan chain (but not the scan input). If you redefine this input port in test mode, the flip-flop remains fully controllable because it is in a scan chain.

Note:

A few faults might appear on this net that are undetectable in ATPG mode but are detectable using functional patterns.

- To gain another input port with negligible effect on total test coverage, redefine as a bidirectional port any output port that comes directly from a flip-flop in a scan chain, if the output is not already a scan chain output. Use this bidirectional port in input mode with TEST asserted.
 - Output ports that are derived and pass-through clocks are usually well tested in functional test mode but difficult to test in ATPG mode. You could redefine these as bidirectional ports and use them in input mode during ATPG testing.
 - The next-best choice of a port to be used as a test-related input is an input port that has a minimal amount of fanout before entering a flip-flop. If you choose to redefine the port so that it must be held either high or low all the time, you minimize the resulting circuitry with undetected faults. In contrast, a port that feeds a buffer tree and gets distributed to hundreds of other gates would be a poor choice to use for testing.
-

Checklists for Quick Reference

This section provides checklists of the design guidelines and port redefinition suggestions presented in this chapter. Use these checklists as a convenient reminder as you implement your design.

ATPG Design Guideline Checklist

Follow these guidelines during ATPG test mode:

1. Ensure that clocks and asynchronous set/reset signals come from a primary input.
 - a. Do not use clock dividers.
 - b. Do not use gated clocks.
 - c. Do not use phase-locked loops (PLLs) as clock sources.
 - d. Do not use pulse generators.
2. Provide complete control of clock paths to scan chain flip-flops.
 - a. If a clock passes through a MUX, constrain the select line of the MUX to a constant value.
 - b. If a clock passes through a combinational gate, constrain the other inputs of the gate to a constant value.
 - c. Pass clock signals directly through JTAG I/O cells without passing through a MUX, unless the MUX control can be constrained.
 - d. Avoid using bidirectional clocks and asynchronous set/reset ports.
3. Do not allow an open path from a pulsed input signal (clock or asynchronous set/reset) to a data-capture input of a sequential device.

- a. Do not allow a path from a pulsed input to both the data input and the clock of the same flip-flop.
 - b. Do not allow a path from a pulsed input to both the data input and the asynchronous set or reset input of the same flip-flop.
4. For multidriver nets, ensure that only one driver is enabled during the shifting of scan chains.
 5. Force all bidirectional ports to input mode while shifting scan chains, using a top-level port as control.
 6. Force scan chain outputs that use bidirectional or three-state ports into output mode while shifting scan chains, using a top-level port (usually SCAN_ENABLE).
 7. Use a single clock tree to clock all flip-flops in the same scan chain.
 8. Treat each clock tree as a separate clock source in designs that have a single clock input port but multiple clock tree distributions.
 9. Use the same clock edge for all flip-flops in the same scan chain.
 10. Do not mix XNOR clock inversion techniques and clock trees.
 11. To protect RAMs from random write cycles, disable RAM write clock or write enable lines while shifting scan chains.
 12. Connect RAM and ROM read and write control pins directly to a top-level input while in ATPG test mode.
 13. Do not allow an open path from a pulsed signal to a RAM's or ROM's data, address, or control inputs (except read/write control).
 14. Do not allow a combinational gate path from any pulsed port to drive the enable controls of three-state drivers that contribute to the multidriver net.
 15. When using bus keepers, ensure a non-Z state on multidriver nets by the end of the scan chain load/unload.
 16. When using bus keepers, do not allow the nonclocked events that occur before the system capture clock to disturb the multidriver net.

Ports for Test I/O Checklist

Follow these port usage guidelines:

1. A port that already feeds the input of a flip-flop in a scan chain is the best port to redefine as a scan chain input.
2. A port that already comes directly from the output of a flip-flop in a scan chain is the best port to redefine as a scan chain output.

3. A three-state output can be redefined as a bidirectional port and used in input mode while TEST is asserted.
4. A standard output can be redefined as a bidirectional port and used in input mode while TEST is asserted.
5. An input port that feeds directly into the input of a flip-flop in a scan chain can be redefined as a shared port.
6. An output port that comes directly from the output of a flip-flop in a scan chain can be redefined as a bidirectional port and used in input mode while TEST is asserted.
7. An output port that is a derived clock or pass-through clock can be redefined as a bidirectional port and used in input mode while TEST is asserted.
8. An input port that has a small amount of fanout before entering a flip-flop is a good choice to be redefined for use as a test-related input while TEST is asserted.

C

Importing Designs From DFT Compiler

This appendix provides a brief overview of how to take a design from DFT Compiler to TetraMAX to generate ATPG patterns.

Importing a Design From DFT Compiler

Before you begin, you should be aware of the differences between DFT Compiler and TetraMAX in the treatment of your design. These are the main differences:

- Bidirectional port timing
- Ordering of events within an ATE cycle
- Latch models
- Support for DFT Compiler commands in TetraMAX

These differences are explained in detail in the appendix “Exporting Designs to TetraMAX ATPG” in the *DFT Compiler Overview* manual (provided with the DFT Compiler tool).

Before you import a design from DFT Compiler to TetraMAX, you need to ensure that the design has valid scan chains and does not have design rule violations. These are the steps to import the design:

1. Before doing any work with DFT Compiler (including scan insertion), set the test timing variables to the values specified by your ASIC vendor.
2. Identify the netlist format that you are exporting to TetraMAX, using the `test_stil_netlist_format` environment variable.
3. To guide netlist formatting, set the environment variables that affect how designs are written out.
4. If you want to pass capture clock group information to TetraMAX, set the `test_stil_multiclock_capture_procedures` variable to true, and use the `check_test` (not `check_scan`) command in the next step.
5. Check for design rule violations and fix any violations.
6. Write out the netlist.
7. Write out the STIL protocol file.

After you perform these steps, you can read in the design with TetraMAX. For more information on performing these steps, see “Exporting Designs to TetraMAX ATPG” in the *DFT Compiler Overview* manual.

D

Utilities

This appendix describes the utility programs supplementing TetraMAX.

This appendix contains the following sections:

- [Ltran Translation Utility](#)
- [Translating PrimeTime Timing Exceptions](#)
- [Generating PrimeTime Constraints](#)

Ltran Translation Utility

This section provides basic information on starting Ltran in the shell mode, and specifying and modifying Ltran configuration files.

When you use the Write Patterns dialog box or `write patterns` command to write patterns in FTDL, TDL91, TSTL2, or WGL_FLAT format, TetraMAX invokes a separate translation process called "Ltran". This translation process runs independently in a new window. You can optionally launch Ltran in the shell mode or use an Ltran configuration file to control the output format.

Ltran in the Shell Mode

Ltran launches in the shell mode one of two ways:

- If you have not set the `DISPLAY` environment variable (which is common when you use a telnet session)
- If you have set the `LTRAN_SHELL` environment variable to 1

When using Ltran in the shell mode, the execution of TetraMAX stops until Ltran finishes. This is different than the xterm version that kept TetraMAX running and allowed parallel Ltran runs; for example, if you try writing files with the `-split` option of the `write patterns` command, which causes the intermediate file created and passed to the external translator to use the STIL pattern format (the default is to use WGL as the intermediate file). Now, each Ltran runs sequentially.

Since this is a third-party interface, any output from Ltran in GUI or shell mode may appear in the UNIX transcript in which TetraMAX was started, but that output will not be captured in the TetraMAX log file.

Linux platforms need the path to the xterm executables. These are located in the `/usr/X11R6/bin` directory. After you add this to your search path, you will be able to write out TDL91, TSTL2, and FTDL pattern formats from the TetraMAX Linux shell. This is especially true if you receive a "sh: xterm: command not found" message.

FTDL/TDL91/TSTL2 Configuration Files

If you select FTDL, TDL91, or TSTL2 as the format in the Write Patterns dialog box or in the `write patterns` command, executing the command invokes a separate translation process called Ltran, which begins as an independent operation in the new window. You can perform other tasks while the translation is being carried out.

The Write Patterns dialog box and `write patterns` command optionally let you specify an Ltran configuration file to be used for controlling the output format. If you do not specify a configuration file, a default file is used from the following directory:

```
$SYNOPSYS/auxx/syn/ltran
```

For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see [“Specifying the Location for TetraMAX Installation” on page 2-2](#).

The configuration files contained in this directory are:

- `wgl2ftdl` : WGL to FTDL
- `wgl2tdl91` : WGL to TDL91
- `wgl2tstl2` : WGL to TSTL2
- `stil2ftdl` : STIL to FTDL
- `stil2tdl91` : STIL to TDL91
- `stil2tstl2` : STIL to TSTL2

By default, when you use the `write patterns` command and specify FTDL, TDL91, and TSTL2 as the pattern format, the pattern generator first generates patterns in WGL format, and then Ltran translates the WGL patterns to the target format using the conversion parameters specified in the `wgl2ftdl`, `wgl2tdl91`, or `wgl2tstl2` configuration file.

Note:

The `stil2ftdl`, `stil2tdl91`, and `stil2tstl2` configuration files are provided for future expansion when STIL will be offered as an intermediate format option. In the current release, only WGL can be used as the intermediate format.

The default files are adequate for most translations, but you can modify a number of fields in the files to customize the output results. These user-editable fields are part of the simulator command and are identified with comments in the Ltran configuration files. All of these fields are optional and can be commented out with curly braces “{ }”. Most of these fields provide a way to specify header information in the output file, as summarized in the sections that follow.

To use a customized configuration file, copy one of the existing files to your own local directory, and then edit your copy to adjust the user-editable fields and controls. In the Write Patterns dialog box or `write patterns` command, specify the name of your modified configuration file.

Understanding the Configuration File

Each configuration file contains two mandatory command blocks (`OVF_BLOCK` and `TVF_BLOCK`) and one optional command block (`PROC_BLOCK`).

The commands in the mandatory command block `OVF_BLOCK` describe the format of data in the original vector file. The commands in the mandatory command block `TVF_BLOCK` provide instructions for formatting vectors in the target vector file. The commands in the optional command block `PROC_BLOCK` describe other processing required to translate the data in the original vector file into the target vector file.

The configuration file structure can be summarized as shown in [Example D-1](#).

Example D-1 Translation Configuration File Structure

```
OVF_BLOCK
  BEGIN
    OVF_BLOCK_COMMANDS
  END
PROC_BLOCK {Optional}
  BEGIN
    PROC_BLOCK_COMMANDS
  END
TVF_BLOCK
  BEGIN
    TVF_BLOCK_COMMANDS
  END
END
```

The configuration file is not case-sensitive. Pin names retain their case in the translation to the target vector file. Pin names can contain any printable ASCII characters (but not spaces), including any of the following characters:

, ; < > [] { } () = \ & | @

For the full syntax of the `OVF_BLOCK`, `PROC_BLOCK`, and `TBF_BLOCK` command blocks, see [“Configuration File Syntax” on page D-8](#).

Customizing the FTDL Configuration File

For FTDL output, the `write_patterns` command uses the `wgl2ftdl` configuration file. You can customize the configuration file by editing the following parameters:

- `-AUTO_GROUP`

This optional switch tells the `write_patterns` command to algorithmically identify similar signals and group them in the FTDL output file.

- Revision number

```

REVISION = "0001",           { edit "0001" as desired }

• Designer name

DESIGNER = "Designer",   { edit "Designer" as desired }

• Test vector function

TNAME = "FUNC",           { edit "FUNC" as desired }

• Test vector name

CNAME = "TEST",           { edit "TEST" as desired }

• Date of design file creation

DATE = "99/10/05" ;       { edit DATE as desired }

```

Customizing the TDL91 Configuration File

For TDL91 output, the write patterns command uses the `wgl2tdl91` configuration file. You can customize the configuration file by editing the following parameters:

- Library

```
LIBRARY_TYPE = "Library",      { edit "Library" as desired }
```

- Customer

```
CUSTOMER = "Customer",        { edit "Customer" as desired }
```

- Part number

```
TI_PART_NUMBER = "PartNum",    { edit "PartNum" as desired }
```

- Pattern set name

```
PATTERN_SET_NAME = "SetName", { edit "SetName" as desired }
```

- Pattern set type

```
PATTERN_SET_TYPE = "SetType", { edit "SetType" as desired }
```

- Revision number

```
REVISION = "1.00",            { edit REVISION as desired }
```

- Date of design file creation

```
DATE = "10/5/1999" ;         { edit DATE as desired }
```

Customizing the TSTL2 Configuration File

For TSTL2 output, the `write patterns` command uses the `wgl2tstl2` configuration file. You can customize the configuration file by editing the following parameters:

- Title

```
TITLE = "TITLE",      { edit "TITLE" as desired }
```

- Function test

```
FUNCTEST = "FC1"      { edit "FC1" as desired }
```

Additional Controls

In addition to the simulator adjustments just described, most of the configuration files have two Ltran controls that you can use to further customize the format of the pattern output files:

- `rename_bus_pins`
- `header nn`

If these controls are supported, they appear commented out by default but can be activated by removing the curly braces “{ }” surrounding them.

This is the syntax of the `rename_bus_pins` control:

```
rename_bus_pins $bus$vec;
```

The `rename_bus_pins` control flattens bused signal names. With this command, a bus signal name like `bus[5]` becomes `bus5`. The form of the mapped name can be controlled by changing the `busvec` string. For example:

```
rename_bus_pins $bus$_$vec_;
```

This example maps `bus[5]` into `bus_5_`.

The `header nn` control tells Ltran to place the names of signals in a vertical list as comments above their column position in the vectors. This control has the following syntax

```
header nn;
```

where `nn` is an integer that specifies how often to repeat the pin header listing, expressed as a number of lines.

Support for Other Formats

With the integrated Ltran program, TetraMAX provides vector interfaces to the FTDL, TDL91, and TSTL2 ASIC vendor formats. Each interface is implemented as a post-process (Ltran) spawned from TetraMAX, using WGL as the intermediate vector file format. When invoked, Ltran reads the WGL vector file and performs the desired format translation.

If you would like to translate vector files into formats other than those directly supported by TetraMAX, you can use a third-party program called vtran, in a manner similar to Ltran, to translate WGL or STIL output from TetraMAX into the desired format. In order to perform these translations, you need to do the following:

1. Obtain the vtran program from Source III, Inc.
2. Generate WGL or STIL vector files from TetraMAX.
3. Run vtran as a separate process to perform the translation from WGL or STIL to the desired format.

Currently, vtran supports translation to the following tester formats:

- SWAV (Credence)
- Teradyne
- PCF (HP 3070)

It also supports translation to the following simulator formats:

- QUEST & QVBF (Quickturn)
- IKOS
- LSIM
- Quicksim (Mentor Graphics)
- QSIM
- SILOS
- SPICE
- VTI
- WIF
- ZYCAD
- TDS (TSSI)
- Verilog testbench

- VHDL testbench
- WGL (scan or flat)
- EPIC (TimeMill, PowerMill)
- WAVES

In addition to these specific output formats, vtran can also be instructed to produce vector data in tabular format as defined by the user.

For further information on vtran, contact the supplier directly:

Source III, Inc.
 Web: www.sourcelll.com
 email: corp@sourcelll.com

Configuration File Syntax

The following sections describe the syntax of the statements in the OVF_BLOCK, PROC_BLOCK, and TVF_BLOCK command blocks.

OVF_BLOCK Statements

AUX_FILE [=] "filename";

Used to specify an auxiliary file for some canned readers.

BEGIN_LINE [=] n;

Used to define the line number in the OVF file at which VTRAN should begin processing vectors.

BEGIN_STRING [=] "string";

Used to define a unique text string in the OVF file after which VTRAN should begin processing vectors.

BIDIRECTS [=] pin_list;

Defines the names and order of pins in the OVF file that are bidirectional.

BUSFORMAT radix; or BUSFORMAT pin_list = radix;

Specifies the radix of buses in the OVF file.

CASE_SENSITIVE;

Allows there to be more than one signal with the same name spelling but differing only in case of letters in the name.

```
GROUP n [=] pin_list;
```

Together with the \$gstatesn keyword, it tells VTRAN how the pin states are organized.

```
INPUTS [=] pin_list;
```

Defines the names and order of input pins in OVF file.

```
MAX_UNMATCHED [=] n [verbose] :
```

Specifies the number of, and information contained in, warnings for lines in the OVF file that does not a format_string.

```
ORIG_FILE [=] "filename";
```

Used to specify the OVF file name to be translated.

```
OUTPUTS [=] pin_list;
```

Defines the names and order of output pins in the OVF file.

```
SCRIPT_FORMAT [=] "format#1" [, . . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
TABULAR_FORMAT [=] "format #1" [, . . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
TERMINATE TIME [=] n; or
```

```
TERMINATE LINE [=] m; or
```

```
TERMINATE STRING [=] "string";
```

Defines where in the OVF to stop processing, at a certain time, line number or when a string is reached.

```
WAVE_FORMAT [=] "format #1" [, . . ."format#n"] ;
```

Format descriptors for User-Programmed reader.

```
WHITESPACE [=] 'a', 'b', 'c', . . . , 'n' ;
```

Defines characters in the OVF file that are to be treated as though they are space (they are ignored).

PROC_BLOCK Statements

```
ADD_PIN pinname = state1 [WHEN expr=state2, OTHERWISE  
state3];
```

Tells VTRAN to add a new pin to the TVF file, and allows you to define the state of this pin.

```
ALIGN_TO_CYCLE [-warnings] cycle pin_list @ time, . . . ,  
pin_list  
@ time ;
```

Vectors can be mapped to a set of cycle data, the state of each pin in a given cycle is determined by its state at a specified strobe time in the OVF file.

ALIGN_TO_STEP [-warnings] step [offset];

Forces a minimum time resolution in the TVF file.

AUTO_ALIGN [-warnings] cycle;

Collapses print-on-change data in the OVF file to cycle data by computing strobe points from information given in the PINTYPE commands.

BIDIRECT_CONTROL pin_list = dir WHEN expr = state ;

Separates input data from output data on bidirectionals under control of a pin state or logical combination of pin states.

BIDIRECT_CONTROL pin_list = direction @ time ;

Separates input data from output data on bidirectionals based upon when the state transitions occur.

BIDIRECT_STATES INPUT state_list, OUTPUT state_list ;

Separates input data from output data on bidirectionals where unique state characters identify pin direction.

CYCLE [=] n;

Specifies the time step between vectors in the OVF when the format of the vectors does not include a time stamp.

DISABLE_VECTOR_FILTER;

Disables filtering of redundant vectors.

DONT_CARE 'X';

Defines the character state to which output pins should be set outside of their check windows.

EDGE_ALIGN pinlist @ rtime [,ftime] [xtime];

Modifies pin transition times by snapping them to predefined positions within each cycle.

EDGE_SHIFT pinlist @ rtime [,ftime] [,xtime];

Modifies pin transition times by shifting them by fixed amounts.

MASK_PINS [mask_character ='X'] [pin_list] @ t1, t2 [-CYCLE]
; or

MASK_PINS [mask_character ='X'] [pin_list] @ CONDITION expr =
state ;

Masks the state of specified pins to the mask_character within the time range between t1 and t2, or when a specified logic condition exists on other pins.

```
MERGE_BIDIRECTS state_list ; or  
MERGE_BIDIRECTS rules = n ;
```

Merges the input and output state information of a bidirectional pin to a single pin after it has been split and processed.

```
PINTYPE pintype pin_list @ start1 end1 [start2, end2] ;
```

Defines the behavior and timing to be applied to input and/or output pins during translation.

```
POIC;
```

Specifies that vectors in the OVF file should be translated to the TVF only when at least 1 input pin has changed in the vector.

```
SCALE [=] nn;
```

Linearly expands or reduces the time line of the OVF. Happens prior to any timing modifications.

```
STATE_TRANS [=] [dir] 'from1'-'>'to1', . . . ;
```

Tells VTRAN not to incorporate pin timing and behavior into the vectors themselves.

```
SEPARATE_TIMING;
```

Defines a mapping from pin states in the OVF file to states in the TVF file.

```
STATE_TRANS_GROUP pin_list = 'from1'-'>'to1', . . . ;
```

Supplements the STATE_TRANS command by providing state translations on an individual pin or group basis.

```
TIME_OFFSET [=] n ;
```

When reading the vectors from the OVF file, the time stamp can be offset by an arbitrary amount.

TVF_BLOCK Statements

```
ALIAS ovf_name = tvf_name, . . . ; or  
ALIAS "ovf_string"="tvf_string";
```

Provides a way to change the names of pins listed in the OVF file, for listing in the TVF file.

```
BIDIRECTS [=] pin_list;
```

Defines the names and order of pins to be listed in the TVF file which are bidirectional.

```
BUSFORMAT radix; or  
BUSFORMAT pin_list = radix;
```

Specifies the radix of buses in the TVF file.

COMMAND_FILE [=] "filename";

Specifies the name of a separate output command file for the target simulator, in addition to the vector data file.

DEFINE_HEADER [=] "text string";

Inhibits the automatic generation of headers and replaces it with a custom text string.

HEADER [=] n;

Causes a vertical list of the pin names to appear as comments in the TVF every *n* vector lines.

INPUTS [=] pin_list ;

Defines the names and order of pins to be listed in the TVF file which are inputs.

INPUTS_ONLY;

Causes only input and the input versions of bidirectional pins to be listed in the TVF.

LOWERCASE;

Forces all pin names in the TVF file to use lowercase letters.

OUTPUTS [=] pin_list ;

Defines the names and order of pins to be listed in the TVF file that are outputs.

OUTPUTS_ONLY;

Causes only output and the output versions of bidirectional pins to be listed in the TVF file.

RENAME_BUS_PINS format;

Provides a way of globally modifying all bus names in the TVF file.

RESOLUTION [=] n;

Specifies the resolution of time stamps in the output vector file (*n* = 1.0, 0.1, 0.01 or 0.001).

SCALE [=] nn ;

Linearly scales all times to the TVF file.

SIMULATOR [=] name [param_list];

Defines the target vector file format to be compatible with the simulator named.

STOBE_WIDTH [=] n;

Used with several of the simulator interfaces to define the width of an output strobe window.

SYSTEM_CALL ". . .text . . . ";

Upon completion of translating vectors from the OVF file to the TVF file, VTRAN sends this text string to the system just prior to termination.

```
TARGET_FILE [=] "filename";
Specifies the name of the output file.

TITLE [=] "title";
Specifies a special character string to be placed in the header of certain simulator vector files.

UPPERCASE;
Forces all pin names in the TVF to be listed with uppercase letters.
```

Translating PrimeTime Timing Exceptions

You can use the `write_timing_exceptions` procedure to translate PrimeTime timing exceptions to TetraMAX constraints. This procedure is part of the `pt2max.tcl` script, which is located in the `$SYNOPSYS/auxx/syn/tmax` directory. For a discussion about the use of the `SYNOPSYS_TMAX` environment variable, see “[Specifying the Location for TetraMAX Installation](#)” on page 2-2.

The `write_timing_exceptions` procedure translates timing exceptions into the format used for Full Sequential ATPG, including full sequential transition fault ATPG and all path delay fault ATPG. Note that the format used for Full Sequential ATPG is different than the format used for Basic Scan and Fast Sequential ATPG.

For Basic Scan and Fast Sequential ATPG translations, you will need to use the TetraMAX `read_sdc` command. For details on this process, see “[Specifying Setup Timing Exceptions From an SDC File](#)” on page 14-16. Note: the `write_timing_exceptions -precise` command formerly handled this process. Please make sure this command is removed from any TetraMAX-related scripts that use currently use.

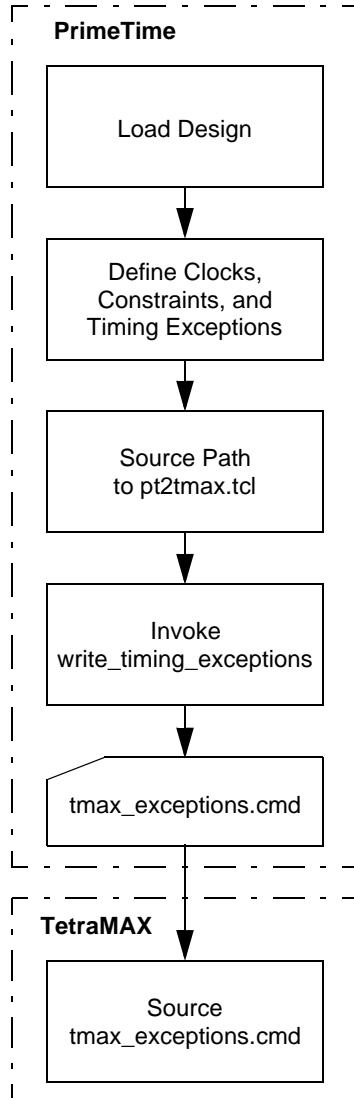
For Full Sequential ATPG translations, the `write_timing_exceptions` procedure performs the following tasks:

- Translates PrimeTime `set_false_path` and `set_multicycle_path timing` exceptions to TetraMAX constraints by converting `-from exceptions` to add `slow cell` commands and `-to exceptions` to add `capture mask` commands
- Locates the startpoints or endpoints for `-through exceptions` and applies constraints on the starting or ending points in TetraMAX
- Lets you select whether source or sink cells should be used when an exception has both `-to endpoints` and `-from startpoints`
- Ignores exceptions on PIs and POs (a TetraMAX `set delay -nopi_changes -nopo_measures` command is assumed). (Note that for an exception from a PI to a register, the register is masked, and for an exception from a register to a PO, the register is declared as a slow cell.)

Write Timing Exceptions Flow

[Figure D-1](#) illustrates the `write_timing_exceptions` flow.

Figure D-1 Write Timing Exceptions Flow



Syntax

The syntax of the `write_timing_exceptions` procedure is:

```
write_timing_exceptions
-precise
-priority [from | to | dominant]
-through [start | end | dominant]
-output <file_name>
-help
```

Argument	Description
<code>-precise</code>	This switch was formerly used to specify timing exceptions used with the TetraMAX Basic Scan and Fast Sequential test generation engines. However, this task is now performed by the TetraMAX <code>read_sdc</code> command. You should make sure you remove the <code>-precise</code> switch from all of your TetraMAX-related scripts.
<code>-priority [from to dominant]</code>	Specifies whether the startpoint or endpoint should be constrained when you have both <code>-from</code> or <code>-to</code> in the same exception. Specify <code>dominant</code> if you want the decision to be based on how many registers are affected by each exception. By default, <code>from</code> has higher priority.
<code>-through [start end dominant]</code>	Specifies <code>start</code> or <code>end</code> if you want <code>-through</code> exceptions traced to startpoints or endpoints. Specify <code>dominant</code> if you want <code>-through</code> exceptions propagated to either startpoint or endpoint based on the number of flip-flops affected. By default, <code>-through</code> exceptions are traced to startpoints.
<code>-output <file_name></code>	Specifies a file name to hold TetraMAX constraints. By default, <code>tmax_exceptions.cmd</code> is used.
<code>-help</code>	Displays usage.

Note:

The analysis is done on a per exception basis if `dominant` is used.

Example

Assume these exceptions are in a design for the results that follow:

```
set_multicycle_path 2 -to carry1/D  
set_false_path -from ff1/CK -to ff2/D  
set_false_path -from doodle/CK -through U571/Z
```

A `write_timing_exceptions -priority` specification creates these TetraMAX commands:

```
add slow cell doodle  
add capture mask carry1  
add capture mask ff2
```

Note: Please make sure you remove any `write_timing_exceptions -precise` specifications from your TetraMAX-related scripts. This switch has been replaced by the TetraMAX `read_sdc` command.

Summary of Translations

PrimeTime allows specifying any combination of `-from`, `-to`, and `-through` options for the `set_false_path` and `set_multicycle_path` timing exceptions. The following table shows how PrimeTime timing exceptions are translated into `add_slow_cell` and `add_capture_mask` commands.

Table D-1

Exception type in PrimeTime	Translated to TetraMAX command
Only <code>-from</code> port	Ignored
Only <code>-to</code> port	Ignored
Only <code>-from cell</code>	<code>add slow cell</code>
Only <code>-to cell</code>	<code>add capture mask</code>
Only <code>-through cell</code>	Depending on <code>-through</code> option, either find startpoints and apply <code>add slow cell</code> , or find endpoints and apply <code>add capture mask</code>
<code>-from and -through</code>	<code>add slow cell</code>
<code>-through and -to</code>	<code>add capture mask</code>

Table D-1

Exception type in PrimeTime	Translated to TetraMAX command
-from and -to	Depends on -priority option
-through -though	Not supported
-from -through -to	Depends on -prioriry options

Generating PrimeTime Constraints

You can use the `tmax2pt.tcl` script to generate PrimeTime constraints for performing static timing analysis of a design under test. This script extracts relevant data and creates a PrimeTime script that constrains the design in test mode.

This script is not compatible with pre-2007.12-SP1 versions of TetraMAX; it makes API calls that older TetraMAX versions do not support.

Although this flow simplifies the process of performing static timing analysis with PrimeTime, it is no substitute for the experienced user to validate timing analysis. See the *PrimeTime User Guide* for these details.

Input Requirements

You might need to supply certain information to TetraMAX and PrimeTime to complete this flow.

TetraMAX input data requirements are

- Netlists (not image files)
- Library
- STIL Protocol File (SPF)
- Tcl command script to build, run drc, and so on

PrimeTime input data requirements are

- Netlists
- Technology library (.db files)
- Tcl command scripts to read design, link, and so on

- Timing models
 - Layout data (for example, SDF)
-

Start Tcl Command Parser Mode

To use this flow, you must run the tool in Tcl command parser mode by invoking TetraMAX by using the `-tcl` command-line option; for example,

```
tmax -tcl ...
```

The command files must be in Tcl format and not in the native format. You can use the TetraMAX command translation script, `native2tcl.pl` to convert native-mode TetraMAX command scripts into Tcl command scripts. For instructions on how to download this script, see ["Converting TetraMAX Command Files to Tcl" on page 2](#).

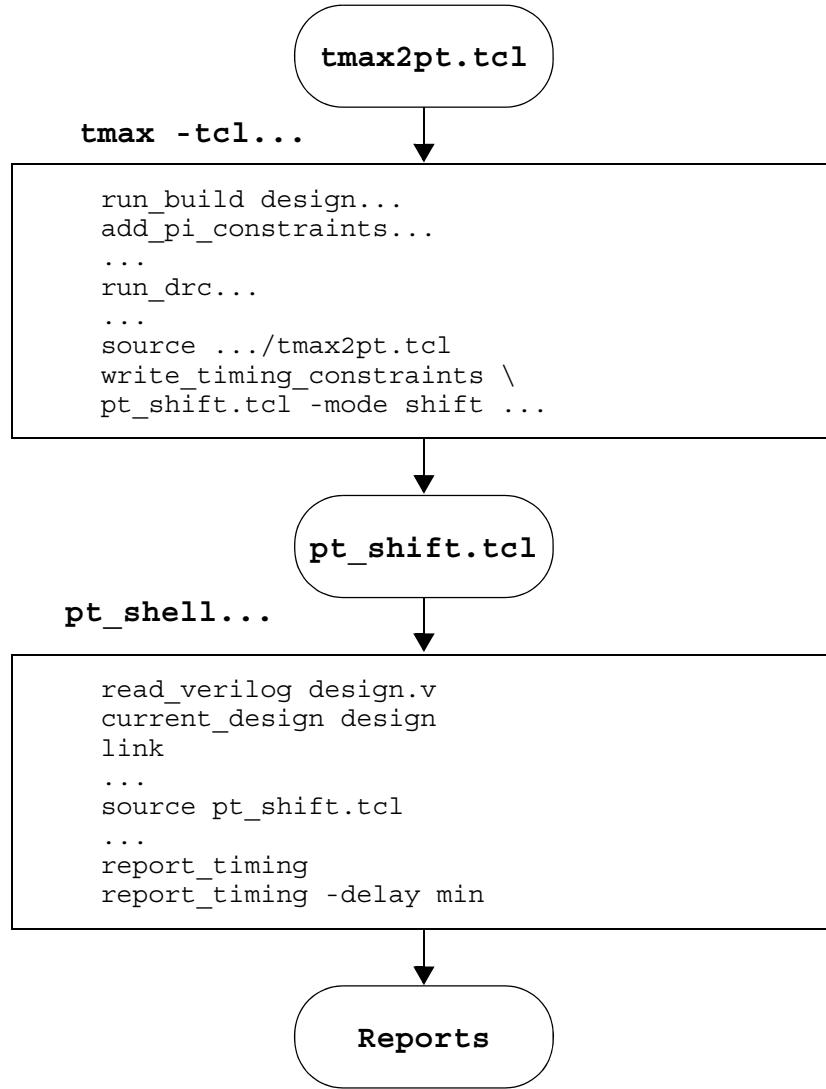
Set Up TetraMAX

In TetraMAX, the normal flow of configuring the design and TetraMAX for ATPG is required. However, ATPG does not have to be run. After running DRC and the configuration has been set, TetraMAX has enough data to support generating the PrimeTime script.

The `tmax2pt.tcl` script is located in the `$SYNOPSYS/auxx/syn/tmax` directory. This script must be sourced from TetraMAX (see [Figure D-2](#)); for example,

```
TEST-T> source $env(SYNOPSYS)/auxx/syn/tmax/tmax2pt.tcl
```

Figure D-2 Shift Mode Analysis Example



Use the `write_timing_constraints` Tcl procedure to create a PrimeTime Tcl script; for example:

```
TEST-T> write_timing_constraints pt_shift.tcl -mode shift
...
```

You should call `write_timing_constraints` for each mode. A separate script is created for each mode, and sourced in PrimeTime during separate sessions.

In PrimeTime, the flow of setting up the design does not change. The design, SDF, parasitics, and so forth are read. Next, the script generated by `write_timing_constraints` in TetraMAX is sourced in PrimeTime; for example:

```
pt_shell> source pt_shift.tcl
```

When the script written by `write_timing_constraints` is sourced inside the `pt_shell`, and an internal clock source (here named `<OCC_controller_clock_root>`, as an example) is included, the following reminder message is echoed:

```
TMAX2PT WARNING: Internal clock <OCC_controller_clock_root> timing is
defaulted.
Adjust this timing to correct values before checking.
```

You can set additional constraints such as clock latency and uncertainty if you want. You can also set up the environment and other conditions such as operating conditions and wire load models. Finally, the analysis can be performed.

The `tmax2pt.tcl` script includes the following option for `write_timing_constraints`:

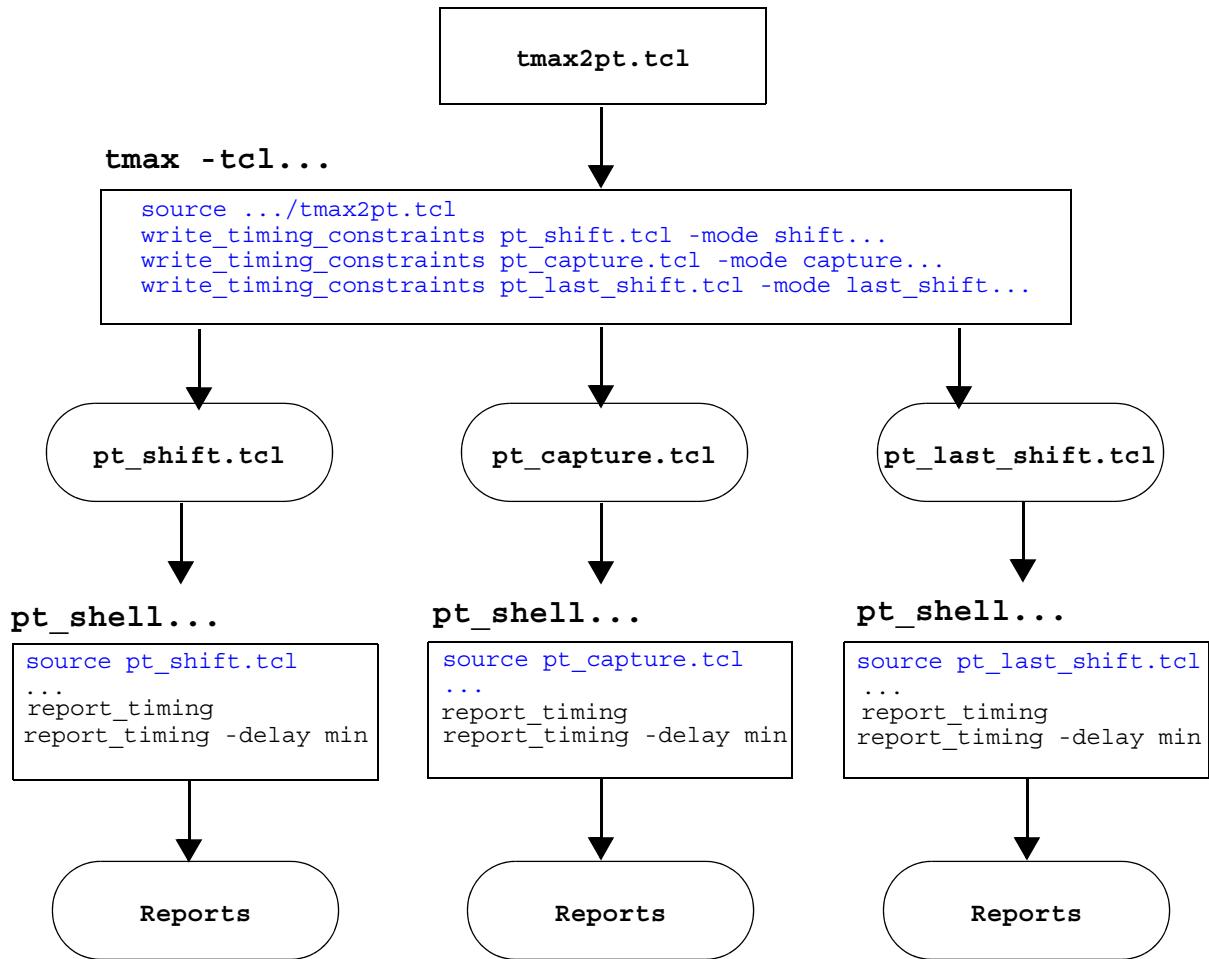
```
[-unit ns|ps]
```

The default is ns, which does the same as the previous default. If ps is specified, the picosecond timing in TMAX is converted to nanoseconds.

Perform an Analysis for Each Mode

As discussed previously, the flow involves performing a separate timing analysis for each mode. This is illustrated in [Figure D-3](#). Example usages for various common modes are given in the following paragraphs.

Figure D-3 Analysis of Three Modes Flow Example



To analyze timing for when the scan chain is *shifting*, the following is suggested:

```
TEST-T> write_timing_constraints pt_shift.tcl -mode shift \
-wft <wft_name>
```

You must select the WaveFormTable defined in the SPF to be used during the shift cycle and specify it by using the *-wft* option.

For *capture* cycles for stuck-at faults, the following usage is suggested:

```
TEST-T> write_timing_constraints pt_capture.tcl \
-mode capture -wft <wft_name>
```

You must select the WaveFormTable defined in the SPF to be used during the capture cycles and specify it by using the `-wft` option.

For analysis of transition fault patterns using *system clock launch*, the following usage is suggested:

```
TEST-T> write_timing_constraints pt_trans_sys_clk.tcl \
-mode capture -wft <launch_wft_name> -wft <capture_wft_name>
```

You must select the WaveFormTables defined in the SPF to be used for the launch and capture cycles and specify them by using the `-wft` option. The first WFT is used as the launch WFT and the second WFT is used as the capture WFT. Launch-capture can be done in the same way as the stuck-at capture analysis above, with the WFT being the `launch_capture`.

For analysis of transition fault timing for *last-shift launch*, the following usage is suggested:

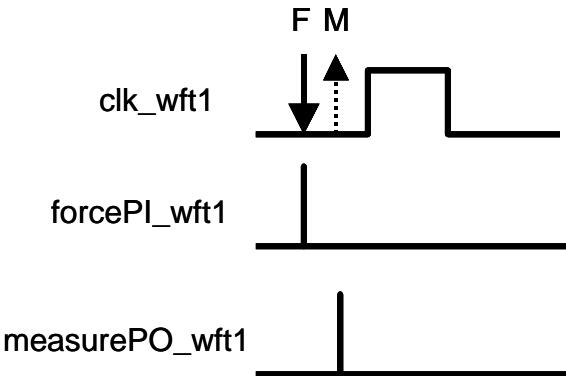
```
TEST-T> write_timing_constraints pt_last_shift.tcl \
-mode last_shift -wft <shift_wft_name> -wft
<capture_wft_name>
```

You must select the WaveFormTables defined in the SPF to be used for the shift and capture cycles and specify them by using `-wft` option. The first WFT is used in the launch cycle and the second WFT is used in the capture cycle. Constraints are specified only as `set_case_analysis` if both cycles have the same TetraMAX constraints. Exceptions, such as `false_path`, are specified only for the capture cycle. You should check that scan-enable transitioning in the second cycle meets the setup time for the capture clock in the second cycle. The same mode can time both the shift to capture transition, and the capture to shift transition.

Implementation

The timing waveforms for clocks and signals reflect what is used on the tester. Input and output timing are relative to virtual clocks with prefixes "forcePI" and "measurePO" (see [Figure D-4](#)). These clocks are impulse clocks with 0 percent duty cycles. The forcePI virtual clocks pulse at the beginning of the cycle. The measurePO clocks pulse at the earliest measure PO time. The timing data is taken from the SPF.

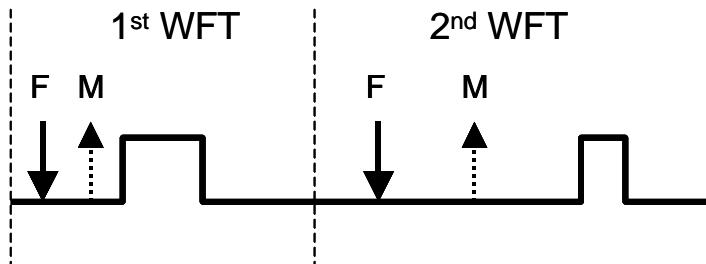
Figure D-4 Waveforms Used For Timing



Each PI, PO, and PIO is listed individually, because each can have a separate input or output delay. Also, each clock is individually listed.

For delay test timing analysis, a single clock net can have clock waveforms that vary due to different waveform tables. For example, the waveform may change between the last shift cycle and the capture cycle. PrimeTime has some facilities to handle this situation. This involves superimposing two clock cycles on top of each other, offset by the period of the first cycle. Each cycle will have its own set of forcePI and measurePO virtual clocks. This is shown in [Figure D-5](#). The WFTs used are based on the order specified.

Figure D-5 Superimposed Cycles For Two WFTs



Note:

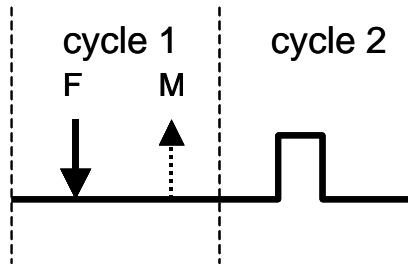
The `create_generated_clocks` command is used to allow clock reconvergence pessimism reduction to work on the two pulses. PrimeTime version Z-2006.12 or later should be used. (Earlier versions report spurious paths between master clocks in addition to the paths between the generated clocks.) The PTE-075 warnings reported by PrimeTime version Z-2006.12 are spurious and can be ignored.

Note that the analysis with superimposed clocks is specific to the two cycles specified. They do not cover other cycles, such as setup and propagation cycles around the launch and capture cycles of a system clock launch pattern.

The write_timing_constraints script attempts to apply a minimal set of timing exceptions to aid accurate timing analysis. For the last_shift mode, false and multicycle paths from the capture cycle are used. Case analysis exceptions are applied for multiple cycle modes only if both cycles have the same PI constraints during ATPG.

For capture cycles of end-of-cycle measures, the waveforms are expanded into a two-cycle period to adjust for the expansion of each capture vector into multiple vectors (see [Figure D-6](#)). Shift cycles remain single cycle.

Figure D-6 End of Cycle Pattern Expansion



E

STIL Language Support

This appendix provides a brief overview of the Standard Test Interface Language (STIL), and identifies how TetraMAX uses the STIL constructs.

This appendix contains the following sections:

- [STIL Overview](#)
- [TetraMAX ATPG and STIL](#)
- [STIL Conventions in TetraMAX](#)
- [IEEE Std. 1450.1 Extensions Used in TetraMAX](#)
- [Elements of STIL Not Used by TetraMAX](#)

STIL Overview

The STIL language is an emerging standard for simplifying the number of test vector formats that automated test equipment (ATE) vendors and computer-aided engineering (CAE) tool vendors must support.

As an emerging standard, STIL is evolving with additional standardization efforts. TetraMAX makes use of both the current STIL standard (IEEE Std. 1450-1999 Standard Test Interface Language (STIL) for Digital Test Vectors), and the IEEE Std. 1450.1 Design Extensions. Many of the extensions were developed in support of TetraMAX users and subsequently proposed to the IEEE Std. 1450.1 working group. Both of these efforts are detailed in the following sections.

IEEE Std. 1450-1999

The Standard Test Interface Language (STIL) provides an interface between digital test generation tools and test equipment. The following defines a test description language:

- Facilitates the transfer of digital test vector data from CAE to ATE environments
- Specifies pattern, format, and timing information sufficient to define the application of digital test vectors to a DUT
- Supports the volume of test vector data generated from structured tests

STIL is a representation of information needed to define digital test operations in manufacturing tests. STIL is not intended to define how the tester implements that information. While the purpose of STIL is to pass test data into the test environment, the overall STIL language is inherently more flexible than any particular tester. Constructs may be used in a STIL file that exceed the capability of a particular tester. In some circumstances, a translator for a particular type of test equipment may be capable of restructuring the data to support that capability on the tester; in other circumstances, separate tools may operate on that data to provide that restructuring.

The STIL language can be used for defining the test protocol input and the pattern input and output. STIL test protocol input is used for various design rules checking (and tester rules checking) and drives the test generation process. ATPG-generated STIL patterns may be structured such that intra-cycle timing, cyclization of test and raw data are separated into, respectively, Timing, Procedure,s and Pattern Blocks. This structure simplifies various rules checking, maintenance, and pattern mapping for system-on-chip testing.

To understand more about STIL, refer to the IEEE Std. 1450.0-1999 Standard Test Interface Language (STIL) for Digital Test Vectors. For general information about the STIL standard, click the Executive Overview link on the STIL home page at

<http://grouper.ieee.org/groups/1450/index.html>.

IEEE Std. 1450.1 Design Extensions to STIL

TetraMAX makes use of several IEEE Std. 1450.1 Design Extensions to support both test program definition and internal tool behaviors. Many of the extensions were developed in support of TetraMAX users and subsequently proposed to the 1450.1 working group. While these extensions are used by TetraMAX, they are not generated or present when stil99-compliant patterns are written, as described in the next section. The presence of 1450.1 extensions allows for a more flexible definition of STIL data. Without these constructs, STIL is more restrictive in its application, requiring complete regeneration when certain expected constructs are modified, which in turn can lead to a usage environment that is less flexible and is more likely to fail than an environment with the 1450.1 extensions present.

The documentation for these extensions is being developed by the IEEE working group and is expected to go to ballot during 2002. After ballot, the document will be available through the normal IEEE channels. Until the ballot is complete, copies may be obtained by contacting the working group. See this IEEE web site for information on this development effort:

<http://grouper.ieee.org/groups/1450/dot1/index.html>

TetraMAX ATPG and STIL

TetraMAX uses STIL in several different contexts. Design information may be provided to TetraMAX through the STIL procedure file (SPF). TetraMAX supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data as part of the SPF definitions. Complete test sets may be written out in STIL format. Also, Tester Rules Checking is provided through STIL-formatted files.

In constructing a SPF, you can define the minimum information needed by TetraMAX. However, any STIL files written as TetraMAX output contain an expanded form of the minimum information and might also contain pattern/response data that the ATPG algorithm produces. TetraMAX reads and writes in STIL, so once a STIL file is generated for a design, TetraMAX can read it again at a later time to recover the clock/constraint/chain data, the pattern/response data, or both.

TetraMAX is able to read some constructs that it does not generate. For example, an external pattern source may be read into TetraMAX for fault simulation but it might not be written out with the same constructs as read in.

The generation of the 1450.1 extensions is controlled by the `-stil` or `-stil99` option to the `write patterns` command. When the `-stil99` option is used, then only standard IEEE-1450 syntax is used without, of course, the benefit of the added functionality enabled by the extensions. Full flexibility and robust STIL definitions are supported via the `-stil` option.

STIL Conventions in TetraMAX

STIL supports a very flexible data representation. TetraMAX has defined conventions in the use of STIL to represent data in a uniform manner yet maintain this flexibility. These conventions are discussed here.

Use of STIL Procedures

When possible, TetraMAX generates calls to STIL procedures from the pattern body. STIL procedures are used in general by TetraMAX because a STIL procedure is self-contained; that is, the state of all signals used in a procedure is established and maintained only during that procedure execution. On return at the end of a procedure, the state of the signals is restored to the values they maintained before the call. This is in contrast to the STIL macro construct, where the final state of these signals must be returned and applied in the patterns before continuing to process STIL data.

By using STIL procedures, the sequence of pattern operations are insensitive to the procedure operations. If macros were used, the next set of pattern activity would be based on information returned from the last macro operation. Also, the execution/behavior of a STIL macro may be different depending on the value of the signals present at the start of the macro, whereas the behavior of the procedure is always the same. The effort both to start a macro with the current state at the call, and to return the right information to the calling context at the return of macro adds significant processing overhead of STIL data when macros are present.

Also, procedure constructs are defined by the STIL standard to be maintained through processing. Macro constructs are defined by the STIL standard to be expanded or “flattened” during processing, and are defined to not be present after processing. While specific processing environments may be able to maintain macro constructs, in general (and to follow the STIL specification) macros would be processed-out or “in-lined” by tools reading STIL data, while procedure constructs would remain in a processed stream.

Finally, because procedures are defined as stand-alone constructs, it is possible to manipulate the contents of a STIL procedure (within certain constraints such as not changing the functionality of that procedure), to manipulate the procedure without concern

of affecting the rest of the pattern operation. While this can be done with some macros as well, because macro behavior is not constrained to the execution of that macro, changes inside a macro can affect data in the rest of the pattern set.

Context of Partial Signal Sets in Procedure Definitions

Another consideration of TetraMAX's application of procedures is its ability to define values for only those signals used in the procedure. While the capture procedures reference all signals in the design (generally through application of the _pi and _po groups), the `load_unload` procedure can leverage a partial signal set context.

The `load_unload` procedure requires establishing values only on the signals necessary to support the scan-shift operation on the design. In addition, TetraMAX supports the definition of a `load_unload` procedure in the SPF that references signals later in the procedure (for example, during the shift block) that may not have been assigned a value by the first Vector of the procedure. These capabilities allow maximum flexibility in interpreting the `load_unload` operation during test generation. When STIL test patterns are generated by TetraMAX, the `load_unload` procedure will be “completed” to contain all signals used in the procedure, in the first Vector (or Condition) of that procedure, to create a standards-compliant definition. However, unspecified signals that do not affect the scan-shift operation will not be present in this procedure.

The STIL standard defines that unused signals in a procedure are assigned `DefaultState` values when the procedure is called. This is a valid state for these signals, because they cannot affect the procedure operation. This is not a requirement of the standard, however (requirements contain the word “shall”), and TetraMAX leverages the flexibility of an incomplete definition for generating other test formats, in particular WGL.

TetraMAX uses the flexibility of deferred and unspecified signal assignment in procedure definitions to maintain the last assigned state on these signals for WGL generation. This option of maintaining the last-assigned-state generates a WGL test program that minimizes transitions on signals at test, particularly transitions that don't affect the test behavior and may have other adverse effects.

In test contexts where STIL is being applied and procedure operations are being “expanded” or “in-lined” in the final test program, it may be valuable to consider the “default” handling of unused signals in the procedure to allow generation of a test that behaves similarly to the TetraMAX-generated WGL test.

Use of STIL SignalGroups

TetraMAX makes use of STIL SignalGroups to simplify creation of STIL protocol information. The STIL procedures file (SPF) may be a complete set of information for certain sections of the final STIL file, or it may be an incomplete file completed by TetraMAX when the final test file is generated.

SignalGroups are used in this context to simplify referencing to sets of signals, without needing to define these signal collections for a specific design, which simplifies SPF creation. In order to support this operation, however, TetraMAX must assume certain naming conventions. Also, the grouping conventions that TetraMAX supports relate directly to the operations that are performed by ATPG operations to generate test sequences.

By using STIL SignalGroups, the output pattern data generated by TetraMAX is more compact than the equivalent by-Signal constructs, and the output format is more general. For example, it can be easier to modify a signal name when that name occurs only inside the SignalGroups, than if that signal reference is used throughout the pattern data.

TetraMAX defines two primary groups, “_pi” and “_po”, which contain an overlapping set of InOut (bidirectional) Signal references. While this can be confusing in some situations, by maintaining these groups this way, the context of test generation as performed internally in TetraMAX is maintained in the output patterns. This supports direct correspondence of the test set with the internal operations performed by TetraMAX, which in turn reduces confusion between TetraMAX-based analysis of test behaviors and the actual information present in the test. However, this information can only be maintained completely through the use of P14501 extensions.

When only IEEE Std. 1450-1999 constructs are used, some loss of information can be expected in STIL programs, causing test programs to be written in a way that is dependent on the presence of constructs used. For example, bidirectional signal behavior, supported by complete representation of both the input behavior and the output behavior in the _pi and _po groups, respectively, allows for the potential modification of the capture procedures without changing the pattern data, in the P14501 context. This opportunity is much more limited under IEEE Std. 1450 constructs, as the pattern data may be written dependent on the capture procedure constructs, and the capture procedures may not be changed without changing the functionality of the pattern.

WaveformCharacter Interpretation

TetraMAX supports a fixed context for WaveFormCharacter (WFC) interpretation in STIL data. A minimum set of requirements are validated against the waveforms associated with these WFCs to avoid undue constraints and support test behaviors as necessary. See [Table E-1](#) for a description of the WFC interpretations supported by TetraMAX.

Table E-1 Supported WaveFormCharacter Interpretations

WFC	Interpretation
0	Drive-low during the waveform.
1	Drive-high during the waveform.
Z	Drive-inactive (typically implemented on ATE as a driver-off operation) during the waveform.
N	Drive-unknown during the waveform. This waveform, if used in the patterns, can be mapped to any of the drive operations above without affecting the test.
P	Drive an active pulse during the waveform. The pulse may be either a high-going pulse or a low-going pulse as appropriate for the type of clock. This waveform is supported only for signals identified as clocks in the design. In path-delay contexts, the timing of this pulse must match the second pulse of the D waveform, if the D waveform is defined.
D	Drive two pulses during the waveform. This definition is supported only for path-delay MUX operation.
E	Drive an active pulse during the waveform. This definition is supported only for path-delay MUX operation, and the timing of this pulse must match the timing of the first pulse of the D waveform.
H	Measure-high (STIL “CompareHigh”, or “CompareHighWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.
L	Measure-low (STIL “CompareLow”, or “CompareLowWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.
T	Measure-inactive (STIL “CompareOff”, or “CompareOffWindow” followed by “CompareUnknown”) during the waveform. In bidirectional contexts, this waveform must also define a drive-inactive (STIL Z) state before performing the measure.

Table E-1 Supported WaveFormCharacter Interpretations (Continued)

WFC	Interpretation
X	This waveform is used by TetraMAX to indicate a no-measure operation. From a STIL perspective, the contents of this waveform may be empty; the absence of activity may imply ATE operations to inhibit previous output measures. This waveform assumes the previous drive state continues to be asserted when used in bidirectional contexts; TetraMAX will define a drive state prior to specifying an X that continues to be applied here. Note this waveform should not contain a P state to provide the drive state because the P state does not maintain a drive-inactive value.

IEEE Std. 1450.1 Extensions Used in TetraMAX

The IEEE Std. 1450.1 extensions used by TetraMAX are identified and described in the paragraphs that follow.

Vector Data Mapping Using \m

The vector data mapping function allows for a new waveform definition to be selected for a given waveform character in a vector. This is most useful in the case of parameter passing to a macro or procedure; however, it can be used anywhere a waveform character string is formed.

In certain scan test styles (such as the LSI “LNI” protocol), it is necessary to measure the output of the design under test’s (DUT) bidirectional signals in one cycle and then drive the same logical values on the same bidirectional signals from the tester in the next cycle, while also turning the internal bidirectional signal drivers off. A test pattern thus has the following format:

1. Load scan chains.
2. Force values on primary inputs (all clocks are off, bidirectional signals are driven by design under test (DUT)).
3. Measure primary outputs and bidirectional signals (all clocks are off).
4. Force values on primary inputs (values are the same as in cycle 2, except the internal bidi drivers are turned off by asserting a special bidi_control input), force values on bidirectional signals (same logical values as measured in previous cycle).
5. Pulse capture clock.
6. Unload scan chains.

Turning off the internal bidirectional drivers in cycle 4 avoids possible contentions that can result in cycle 5 due to capturing new data into the state elements. The additional data to be applied on bidirectional signals in cycle 4 is redundant (can be computed from the data of cycle 3.) This test style needs to be supported without adding extra data to the STIL patterns and without changing the waveformcharacters in the patterns. Also, ATPG rules checking can verify the correctness of the patterns (for example, the internal bidirectional signals are turned off in cycle 4) before actually generating test data.

Note that ATPG-generated patterns are typically guaranteed to be contention-free on all bidirectional signals, both pre- and post-capture. Thus, the above protocol may not be required to avoid bidi contentions. However, ATPG tools may support this protocol for customers that have already designed their test flow with this protocol.

It is important that the `bidi_control` input turns off ALL internal bidi drivers in cycle 4 above. Otherwise, a contention-free pattern could be transformed into a pattern with contentions by the very protocol that attempts to avoid bidi contentions! For example, consider the following example, where `BIDI1` and `BIDI2` are bidirectional signals, and `BIDI_CTRL` is an input that, when 0, turns off the internal driver of `BIDI1`, but not of `BIDI2`:

ATPG-generated contention-free pattern:

1. Load scan chains.
2. Force values on primary inputs and bidirectional signals (`force BIDI_CTRL=1; BIDI1 = Z; BIDI2 = Z;`).
3. Measure primary outputs and bidirectional signals (`measure BIDI1=L; BIDI2=H;`).
4. Pulse capture clock (this has the effect of switching the internal drivers such as now both the `BIDI1` and `BIDI2` internal drivers are driving 0. There is no contention, because the tester continues to drive Z on both bidirectional signals, as in cycle 2.).
5. Unload scan chains.

The pattern above will be changed, after it has been generated, to match the LNI protocol:

1. Load scan chains.
2. Force values on primary inputs and bidirectional signals (`force BIDI_CTRL=1; BIDI1 = Z; BIDI2 = Z;`).
3. Measure primary outputs and bidirectional signals (`measure BIDI1=L; BIDI2=H;`).
4. Force values on primary inputs (`force BIDI_CTRL=0; BIDI1=0; BIDI2=1;`).
5. Pulse capture clock (this has the effect of switching the internal drivers such as now both the `BIDI1` and `BIDI2` internal drivers are driving 0. This causes a contention on `BIDI2`: its internal driver, not turned off, drives 0 while the tester drives 1, as in cycle 4).

6. Unload scan chains.

Syntax

The mapping operation is specified in either the Signals or the SignalGroups block as follows:

```
Signals {
    sig_name < In | Out | InOut | Pseudo > {
        ( WFCMap (from_wfc)* -> to_wfc; )*
    }
}
SignalGroups (domain_name) {
    groupname = sigref_expr {
        ( WFCMap (from_wfc)* -> to_wfc; )*
    }
}
```

WFCMap is an optional statement that, when used, indicates that any pattern data assigning from_wfc to the signal or signalgroup, should be interpreted as having been assigned from to_wfc instead.

To use the mapping of a signal or signalgroup, a new flag is added to the cyclized pattern data: \m Indicates that the defined mapping should be used.

If the vector mapping \m is used, but no WFCMap has been defined for the waveformcharacter to be mapped, the same waveformcharacter is used unchanged.

Example

In the following example, the vectors are labeled to correspond to the LNI-protocol cycles above. Cycle 3 uses the arguments passed in for _io first (HIL), then cycle 4 uses them again, this time mapped to (10), which remain in effect for cycle 5 as well.

```

Signals {
    a In; ck In; bidi_enable In; b Out; q1 InOut; q2 InOut;
}
Procedures procdomain {
    "capture_sysclk" {
        W myWFT; // where all waveformchars are defined
        "cycle 2": V { a=#; ck=0; bidi_enable=1; b=X; _io=ZZ ; }
        "cycle 3": V { b=#; _io=%%; }
        "cycle 4": V { bidi_enable=0; b=X; _io=\m ##; }
        "cycle 5": V { ck=P; }
    }
}
Pattern "__pattern__" {
    W myWFT;
    "cycle 1": Call "load_unload" { ... }
    Call "capture_sysclk" { a=0; b=H; _io=HL; }
    "cycle 6": Call "load_unload" { ... }
}

```

Vector Data Mapping Using \j

The “join” function allows you to have multiple waveform characters against the same signal in one vector. This enables structuring of STIL pattern output using procedures.

Syntax

Refer to [“Vector Data Mapping Using \m” on page E-8](#) for the syntax definition of the WFCMap statement.

General Example

The following is an example usage of the join function.

```

Signals {
    b InOut { WFCMap 0x -> k; }
}
Pattern p {
    V { b = 0; b = x; }
}

```

[Table E-2](#) shows examples of “two data” conditions on an InOut.

Table E-2 “Two Data” Conditions

Force	Measure
0, 1, Z, N	X
Z	L, H, T
0	L
1	H
0	H
1	H, T

The rules for handling multiple definitions of a signal are

- The normal behavior of a WFC-assignment to a signal is to replace the last-assigned WFC value.
- This behavior may be OVERRIDDEN on a per-vector bases, through the use of a “join” escape sequence. The “join” allows both WFCs to be evaluated, using the WFCMap, to resolve or specify a single new WFC that is the desired effect of these two WFCs.

For instance, take the case where two SignalGroups have a common element in them (signal 'b'):

```
_pi = '...+b';
(po = '...+b';
```

A procedure may “join” these two groups in a vector:

```
proc { cs { v { _pi=#; _po= \j #; }}}}
```

Signal 'b' needs to be resolved based on the combinations of WFCs that may be seen by these two groups. It might have a WFCMap declaration like

```
WFCMap 0x -> 0;
WFCMap 1x -> 1;
... etc. ...
```

This mechanism provides for the explicit resolution of “joined” data without creating new combinations of waveforms on-the-fly.

“Joining” requires the WFCMap to define two WFCs to equate to a single new resolved WFC. The WFCMap never requires more than two WFCs as [Figure E-1](#) presents:

Figure E-1 WFC Example

```

WFCMap 0A -> n;
WFCMap cn -> m;
v { b = 0; b = \jA; }
      |
      |
      |
      join 0 and A; get n
  
```

Usage Example

Consider a design with one input, one output and two bidirectional signals. STIL would declare them like this:

```
Signals { i:In; o:Out; b1:InOut; b2:InOut; }
```

STIL also defines signal groups:

```

Signalgroups {
    "_pi" = 'i + b1 + b2';
    "_po" = 'o + b1 + b2';
}
  
```

STIL patterns are written out using capture procedures. Unlike a “flat” vector-only STIL output, using capture procedures has many advantages:

- Pattern cyclization is encapsulated in the capture procedures; timing in the Timing block and data in the Pattern block. This allows easy understanding and maintenance of the three separately.
- Rules checking (DRC) is done only on the small Procedures block, independent of the huge Pattern block.
- Patterns are CTL (P1500) ready: only the procedures need to be modified, not the Patterns.

Capture procedures are defined like this:

```

Procedures {
    "capture" {
        V { "_pi"="### ; "_po"="###; }
    }
}

```

All of the above are fully STIL 1450-1999 compliant.

Now let's consider a STIL pattern that includes the following:

```

force_all_pis { i=0; b1=Z; b2=1; }
measure_all_pos ( o=H; b1=H; b2=X; )

```

The STIL output would translate the above into the following:

```
Call capture { "_pi"=0Z1; "_po"=HHX; }
```

Because of how STIL is interpreted by the consumer of the patterns, the actual arguments are substituted for the formal arguments # in the body of the procedure, and the signalgroups _pi and _po are expanded to their signals, resulting in:

```
V { i=0; b1=Z; b2=1; o=H; b1=H; b2=X; }
```

However, STIL also specifies that if multiple waveformcharacters are assigned to the same signal in a given vector, all but the last one are ignored. Thus, the above vector is equivalent to:

```
V { i=0; o=H; b1=H; b2=X; }
```

Now, how should the bidirectional assignments be interpreted? The first one, b1=H, means “measure High on b1”. This is consistent with the intention of the ATPG pattern, although it leaves the ambiguity of also implying that the tester driver should be tri-stated (b1 driven to Z) to take a measure. The STIL consumer software is supposed to take this into account.

The second bidirectional, b2=X, means “measure X (no measure) on b2”. Unfortunately, the drive part (b2 driven to 1) is lost. This is a real problem.

The 1450.1 solution is very simple and general: Provide a mapping to explicitly explain what to do with two waveformcharacter assignment. Thus, the 1450.1 procedure would be written as:

```

Procedures {
    "capture" {
        V { "_pi"="### ; "_po"= \j ##; }
    }
}

```

Notice the addition of the “join” modifier `\j`. The `\j` refers to the WFCMap mapping table that would be defined as:

```
Signalgroups {
    "_pi" = 'i + b1 + b2';
    "_po" = 'o + b1 + b2' {
        WFCMap 0X -> 0; WFCMap 1X -> 1; WFCMap ZX -> Z; WFCMap NX -> N;
    }
}
```

This provides an unambiguous interpretation of the above:

```
V { i=0; b1=Z; b2=1; o=H; b1=H; b2=X; }
```

to the desired:

```
force_all_pis { i=0; b1=Z; b2=1; }
measure_all_pos ( o=H; b1=H; b2=X; )
```

Signal Constraints Using Fixed and Equivalent

Structured test patterns often have signals constrained to have a certain value or waveform during a pattern sequence. This may be required, for example, for ATPG scan rules checking (such as a test mode signal always active) or for differential scan or clock inputs. The Fixed STIL construct defines signals that must have a constant waveform character and the Equivalent construct defines signals that are linked to other signals. These constructs help reduce pattern volume, because the value of a constraint signal does not need to be specified explicitly in the pattern data. Also, ATPG rules checking requires signal constraint information as input.

ScanStructures Block

Simulation of scan patterns outside the test-pattern generator is often performed to verify timing, design functionality, or the library used to generate the patterns. The speed of the simulator is limited by simulating the load/unload (Shift) operation of scan chains. Optimal simulation performance is achieved with no shifts, bypassing scan chain logic and asserting/verifying the scan data directly on the scan cells.

The ScanStructures block is extended to include additional information required for efficient simulation of scan patterns, that is, eliminating the need to simulate load/unload (shift) cycles. Additional constructs are defined on the ScanCell statement inside the ScanChain block. In addition, the capability is added to the ScanStructures block to support referencing previous ScanChain definitions from other ScanStructure blocks.

Elements of STIL Not Used by TetraMAX

The following sections list the STIL output and input constructs that are not used in this version of TetraMAX. This list is provided to you as a guide for tools that are designed to read the STIL output file generated from TetraMAX.

The only elements of 1450.1 used by TetraMAX are identified previously in 1450.1 Extensions Used in TetraMAX; all other elements of 1450.1 are not used by TetraMAX.

This information is provided specifically from the context of TetraMAX as a stand-alone tool. TetraMAX-generated STIL output is applied in several contexts and tool flows, for example as part of the CoreTest environment (CoreTest is in development). These contexts will use additional STIL constructs and behaviors not used by TetraMAX alone.

TetraMAX STIL Output

Here is a list of output constructs that TetraMAX does not currently support. To understand more about these constructs, refer to the numbered paragraph in IEEE Std. 1450-1999 as indicated in the list.

Note:

The TetraMAX internal pattern source will not write or produce STIL with the constructs described in this section. Future versions may make use of these constructs as necessary.

11. UserKeywords

TetraMAX does not generate any UserKeywords. All keywords used are those defined in the standard.

12. UserFunctions

TetraMAX does not generate any UserFunctions. All timing expressions use expressions that are defined in the standard.

17. PatternBurst > SignalGroups, MacroDefs, Procedures, ScanStructures (named domains)

TetraMAX does not generate any references to named domains from within a PatternBurst. All references are to the globally defined blocks only. Other contexts of STIL generation may provide named domain blocks.

17. PatternBurst > Start, Stop

TetraMAX does not generate any start/stop information within a PatternBurst. All patterns are expected to execute from the beginning to the end of the pattern.

17. PatternBurst > PatList > pat_name {...} (optional statements per pattern)

TetraMAX does not generate any pattern names in a PatList that contain block information. The default generation of STIL data will rely on definitions in a global SignalGroups, MacroDefs, Procedures, and ScanStructure blocks only. Named block reference statements may be specified from other STIL contexts

18. Timing > WaveformTable > Inherit...

TetraMAX does not use the Inherit statement within WaveformTables. All waveform tables are completely self-contained.

18. Timing > WaveformTable > SubWaveforms

TetraMAX does not use the SubWaveform block within the Timing definition. All waveforms are composed of single drive and compare events.

18. Timing > WaveformTable > event_label

TetraMAX does not generate any event labels. All timing information is composed of timing values that are relative to the beginning of the period.

18. Timing -> WaveformTable > [event_num]

TetraMAX does not utilize multiple events in a waveform. All data from a pattern is made up of single waveform character references.

18. Timing -> WaveformTable > @ label references in timing
expressions

TetraMAX does not generate any relative timing. All timing values are specified as absolute times from the start of the period.

18.2 Waveform event definitions > expect events

TetraMAX does not generate any expect events. Only drive and compare events are used.

19. Spec, Selector

TetraMAX does not generate either Spec or Selector blocks. All timing values are specified as absolute numbers.

21.1 Cyclized data > \d

TetraMAX does not generate data using the decimal notation, which is selected by use of the \d escape sequence.

21.2 Multiple-bit cyclized data

TetraMAX does not generate any multiple bit vector information. All vectors contain only one wfc per signal.

21.3 Non-cyclized data

TetraMAX does not generate any non-cyclized data. All vectors are made up of WFC data that refers to cyclized waveform data in a Timing block.

22.6 Loop statement

TetraMAX support of Loop operations is very restrictive to certain contexts. Generally, all pattern vectors are executed in a straight-line sequence.

22.7 MatchLoop statement

TetraMAX does not generate any MatchLoop conditions. All patterns vectors are executed in a straight-line sequence.

22.8 Goto statement

TetraMAX does not generate any Goto statements. All patterns vectors are executed in a straight-line sequence.

22.9 Breakpoint

TetraMAX does not generate any Breakpoint statements. It is assumed that all vectors will fit into available ATE memory.

22.11 Stop

TetraMAX does not generate any Stop statements within a pattern. All patterns are expected to execute from the beginning through to the last vector.

23.1 Pattern > TimeUnit

TetraMAX does not generate the TimeUnit statement. This statement is only used with uncyclized data, which is not generated by TetraMAX.

TetraMAX STIL Input

Here is a list of constructs that TetraMAX can read, but ignores. These will not be written out. To understand more about these constructs, refer to the numbered paragraph in IEEE Std. 1450-1999 as indicated in the list.

10. Include Statement

Supported (by the reader, not produced by the writer).

11. UserKeywords Statement

Ignored (by the reader, not produced by the writer).

12. UserFunctions Statement

Ignored (by the reader, not produced by the writer).

17. PatternBurst block syntax

References to named SignalGroups, MacroDefs, Procedures, and ScanStructures statements are supported (by the reader, not produced by the writer). Start, Stop and Termination statements are not supported by the reader.

F

STIL99 vs. STIL

This appendix provides an overview of the differences between the STIL99 and STIL pattern formats (please see the table starting on page F-2).

Argument	Description
<code>statement STIL 1.0;</code>	<code>statement STIL 1.0 { Design 2005; }</code>
<code>Header { Title " TetraMAX ..."; Date "Tue Feb ..."; Source "comment"; History { Ann (* ...previous Annotations in the History section *) Ann {* ...fault, pattern, drc reports, clocks and constrained pins *} } } }</code>	<code>Header { Title " TetraMAX ..."; Date "Tue Feb ..."; Source "comment"; History { Ann (* ...previous Annotations in the History section *) Ann {* ...fault, pattern, drc reports, clocks and constrained pins *} } } }</code>

Argument	Description
UserKeywords ScanChainGroups (conditionally) ActiveScanChains (conditionally) ;	UserKeywords BistStructures (conditionally) ClockStructures (conditionally) FreeRunning (conditionally) DontSimulate ScanChainGroups and ActiveScanChains are keywords in 1450.1. They are used as UserKeywords only in -stil99 format. Conditionally means with respect to the type of the design being parsed through TetraMAX.
Ann {* ANNOTATION *} Used only in the Header History section	Ann {* ANNOTATION *} Used only in the Header History section
Signals { "sig": 1. Always quoted 2. Does not use [] array notation In Out InOut Pseudo (Used for internal chain scan references on some BIST environments)	Signals { "sig": 1. Always quoted 2. Does not use [] array notation In Out InOut Pseudo (Used for internal chain scan references on some BIST environments)
; Instead of using semicolon, {} bracket format used if the following attributes are present: ScanIn; (no integer number) ScanOut; (no integer number)	; Instead of using semicolon, {} bracket format used if the following attributes are present: ScanIn; (no integer number) ScanOut; (no integer number) WFCMap { 0X->0; 1X->1; ZX->Z; NX->N; PX->P; P[0 1]->P; } * // end WFCMap 1 } See Appendix E for more details on WFCMap. The mapping operation is specified in either the Signals or the SignalGroups.

Argument	Description
<pre>SignalGroups { (No signalgroups domain name)</pre> <p>Supports user names and generates specific groups: <code>_pi</code> lists all inputs+bidirectionals, <code>_po</code> lists all outputs+bidirectionals.</p> <p>{ } format used if the following attributes are present:</p> <pre>ScanIn; (no integer number) ScanOut; (no integer number)</pre>	<pre>SignalGroups "user_name" { (No signalgroups domain name)</pre> <p>Supports user names and generates specific groups: <code>_pi</code> lists all inputs+bidirectionals, <code>_po</code> lists all outputs+bidirectionals.</p> <p>{ } format used if the following attributes are present:</p> <pre>ScanIn; (no integer number) ScanOut; (no integer number) WFCMap { 0X->0; 1X->1; ZX->Z; NX->N; (_pi _po, -still only) FX->P; } FreeRunning(Period time; LeadingEdge time; TrailingEdge time; OffState <D U>;)</pre>

TetraMAX will accept the following Predefined SignalGroups:

- o `_in` = input pins
- o `_out` = output pins
- o `_io` = bidirectional pins
- o `_pi` = inputs + bidirectional pins
- o `_po` = outputs + bidirectional pins
- o `_si` = scan chain inputs
- o `_so` = scan chain output

<pre>PatternExec { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</pre>	<pre>PatternExec "user_name" { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</pre>
---	---

Argument	Description
<pre> Patternburst "_burst_ { (SignalGroups "user_name" ;)* (user spec'ed) (MacroDefs "user_name" ;)* (user spec'ed) (Procedures "user_name" ;)* (user spec'ed) (ScanStructures "user_name" ;)* (user spec'ed) PatList { user specified_pattern_name -or- "_pattern_"; (Fixed pattern name) })* // end pat_name or _burst_name } + // end of PatList (ParallelPatList (SyncStart Independent LockStep) (PAT_NAME_OR_BURST_NAME { } Extend; })* // end ParallelPatList // end of PatternBurst </pre>	<p>Default generation if no input patternexecs will be a single unnamed patterexec. If named patternexecs are input, you must identify one patterexec to be used by TetraMAX, and only this patterexec will be written out.</p> <pre> Patternburst "_burst_ { (SignalGroups "user_name" ;)* (user spec'ed) (MacroDefs "user_name" ;)* (user spec'ed) (Procedures "user_name" ;)* (user spec'ed) (ScanStructures "user_name" ;)* (user spec'ed) PatList { "_pattern_"; (Fixed pattern name) user_specified_pattern_name -or- "_pattern_"; (fixed pattern name), *and* independent async. freerunning clock bursts Extend; (conditionally; async. freerunning clocks) } </pre>
<pre> Timing { (No name generated) WaveformTable user_name { (Default name: "_default_WFT_") Period integer_time_units; </pre>	<p>Default generation if no patternbursts are specified on input will use the name "_burst_".</p> <pre> Timing { (No name generated) WaveformTable user_name { (Default name: '_default_WFT_') Period integer_time_units; </pre> <p>Note current environment supports integer units of time only.</p>

Argument	Description
<pre>Waveforms { groups_or_signal_names { inputs: O I Z N outputs: H L T X clocks: P D E (Always single-WFC references, separated)</pre>	<pre>Waveforms { groups_or_signal_names { inputs: O I Z N outputs: H L T X clocks: P D E (Always single-WFC references, separated)</pre>
<pre>ScanStructures { (Unnamed) ScanChain name { ScanLength integer ; ScanCells name_list ; ScanIn signal_name ; ScanOut signal_name ; ScanMasterClock signals ; ScanSlaveClock signals ; ScanInversion 0,1 ;</pre>	<pre>ScanStructures "user_name" { (user spec'ed) -or- ScanStructures { (Unnamed) ScanChain name { ScanLength integer ; ScanCells name_list ; ScanIn signal_name ; ScanOut signal_name ; ScanMasterClock signals ; ScanSlaveClock signals ; ScanInversion 0,1 ;</pre>
<pre>ScanChainGroups { (Used for some BIST designs) Generates groups of chains AND groups of groups. Groups are used in UserKeyword blocks and ActiveScanChains statements. }</pre>	<pre>ScanChainGroups { (Used for some BIST designs) Generates groups of chains AND groups of groups. Groups are used in UserKeyword blocks and ActiveScanChains statements. }</pre>

Argument	Description
<pre>sigref_expr = vec_data; //STIL Cyclized Pattern data LIST OF WFCs — for example "_po" = HHHL In STIL, only assignment of WFC characters is allowed, except '\r' to repeat one WFC character. No '\h' for hex or other options used in the data. No Base statement in declarations; all assignments are by WFC. \r(integer) WFC — only one from the list of choices...</pre>	<pre>sigref_expr = vec_data; //STIL Cyclized Pattern data LIST OF WFCs — for example "_po" = HHHL In STIL, only assignment of WFC characters is allowed, except '\r' to repeat one WFC character. No '\h' for hex or other options used in the data. No Base statement in declarations; all assignments are by WFC. \r(integer) WFC — only one from the list of choices...</pre>

TetraMAX supports a fixed context for WaveFormCharacter (WFC) interpretation in STIL data. A minimum set of requirements are validated against the waveforms associated with these WFCs to avoid undue constraints and support test behaviors as necessary. For a listing of WFC Support, see [Table E-1 on page E-7](#).

	<pre>\m \j Usage change for dot-1 compliance See Appendix E for details on: Vector Data Mapping Using \m Vector Data Mapping Using \j</pre>
<code>sigref_expr = serial_data;</code>	<code>sigref_expr = serial_data;</code>
<pre>(LABEL :) "precondition all signals": on initial C in Pattern "pattern N": used in patterns.</pre>	<pre>(LABEL :) "precondition all signals": on initial C in Pattern "pattern N": used in patterns.</pre>
User labels allowed in procedures and macros	User labels allowed in procedures and macros
<pre>V(ector) { (cyclized data) V { cyclized_data_assignments_only }</pre>	<pre>V(ector) { (cyclized data) V { cyclized_data_assignments_only }</pre>
<pre>W(aveformTable) NAME ; W name ;</pre>	<pre>W(aveformTable) NAME ; W name ;</pre>

Argument	Description
C { cyclized_data_assignments_only }	C { cyclized_data_assignments_only }
Call name ; Call name { scan_and_cyclized_arguments }	Call name ; Call name { scan_and_cyclized_arguments }
Macro name ; Macro name { scan_and_cyclized_arguments }	Macro name ; Macro name { scan_and_cyclized_arguments }
Loop integer { ... } Allowed in setup procedures & some BIST procs	Loop integer { ... } Allowed in setup procedures & some BIST procs
Vector statements only with constant WFC assignments }	Vector statements only with constant WFC assignments }
ScanChain CHAINNAME ; ActiveScanChains group_or_chain_names ; Used around Shift blocks	ScanChain CHAINNAME ; ActiveScanChains group_or_chain_names ; Used around Shift blocks
	F(fixed) { (cyclized-data)* (non-cyclized-data)* } F { cyclized_data_assignments } Used in procedures
	E(quivalent) { (\m) sigref_expr)* ; E sig \m sig ; Used in procedures See Appendix E under "Signal Constraints Using Fixed and Equivalent"

Argument	Description
<pre> Pattern "_pattern_" { Standard pattern structure: "precondition all signals": C { _po - ... _pi = ... } Structure change to this for proper bidi representation: W default_WaveformTable_name ; Macro "test_setup"; "pattern 0": ... pattern sequences follow } </pre>	<pre> Pattern user_specified_pattern_name { -or- Pattern "_pattern_" { (fixed name by default) Standard pattern structure: "precondition all signals": C { _po - ... _pi = ... } Assignment change to this for proper bidi representation: W default_WaveformTable_name ; Macro "test_setup"; "pattern 0": ... pattern sequences follow } </pre>
<pre> Procedures { (Unnamed Procedures block) Procedures "diagnosis" { (In some BIST contexts) (PROCEDURE_NAME { TetraMAX flow uses fixed name to identify application. (pattern-statements)* support # and % assignment to specific types of groups: _po, _pi, and in some circumstances groups of bidi-only and clock-only signals. } } </pre>	<pre> Procedures "user_name" ; (user spec'ed) -or- Procedures { (Unnamed Procedures block) Procedures "diagnosis" { (In some BIST contexts) (PROCEDURE_NAME { TetraMAX flow uses fixed name to identify application. (pattern-statements)* support # and % assignment to specific types of groups: _po, _pi, and in some circumstances groups of bidi-only and clock-only signals. } } </pre>

Argument	Description
<pre>MacroDefs { (Unnamed MacroDefs block) { MACRO_NAME { TetraMAXflow uses fixed name to identify application test_setup, bist_setup macros do not use parameters W { }, C { }, V { } statements } }</pre>	<pre>MacroDefs "user_name" ; (user spec'ed) -or MacroDefs { (Unnamed MacroDefs block) { MACRO_NAME { TetraMAXflow uses fixed name to identify application test_setup, bist_setup macros do not use parameters W { }, C { }, V { } statements } }</pre>
<pre>PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters may be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters may be specified after the Shift. }</pre>	<pre>PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters may be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters may be specified after the Shift }</pre>

Argument	Description
----------	-------------

Parameters supported in specific contexts only.

In TetraMAX, the # sign is primarily used — not the % sign. You should only use the # sign in very specific situations within certain procedure types. With a fixed name, like load_unload, the # sign is associated with groups associated as scanin and scanoutputs. The # references the scanins and scanouts. You can parameterized the _pi group on last_shift_launch. The parameters are constrained to _so _si _po _pi.

Predefined Signal Groups in STIL

To minimize typing that you can have to perform to define a DRC file by hand, TetraMAX has a number of predefined signal groups it recognizes. A SignalGroup is a method in STIL for describing a list of pins using a symbolic label. Symbolic labels allow a large number of pins to be referenced without a large amount of typing.

TetraMAX will accept the following predefined SignalGroups that:

- o _in = input pins
- o _out = output pins
- o _io = bidirectional pins
- o _pi = inputs + bidirectional pins
- o _po = outputs + bidirectional pins
- o _si = scan chain inputs
- o _so = scan chain outputs

If your STIL DRC description defines a symbolic group with the same name as the predefined TetraMAX groups, then your definition supersedes the predefined definition.

NOTE: There is not a predefined signal group called _clks. Tmax does not create an _clks group the user needs to define the signals they want to be clocks in the flow, and put those signals into the _clks group. If the user is using the extended capture procedures with multiple cycles, then the user needs to create and define this group and reference that signal group in these procedures.

Index

Symbols

\ (continuation character) 3-4

Numerics

64-bit mode 2-7, 2-8

A

abbreviating commands 21-3
abort limit 4-27, 12-6
aborted bus analysis, completing 20-12
add atpg constraints command in TetraMAX
 16-18
add clocks command 9-5
add distributed processors command 18-2,
 18-8
add faults command 4-20, 10-4, 11-18
 for transition delay faults 14-10
Add Faults dialog box 11-18
add net connections command 4-43
add pi constraints command 4-37, 9-4
Add PI Constraints dialog box 9-4
add pi equivalences command 4-36
Add PI Equivalences dialog box 4-36
algorithms
 dependent pattern mapping 13-8

independent pattern mapping 13-7
pattern mapping 13-6
alias command 21-8
aliases, predefined 2-7
allowing unstable set/reset inputs 4-18
AN fault class 7-41, 10-8
analog blocks 16-20
analyze faults command 20-10
analyze path rule violations 15-14
analyzing untestable faults 15-25
ancillary waveform tables 15-16
AP fault class 10-7
appending command output 21-6
asynchronous set/reset 4-18
ATPG
 abort limit 4-27
 analyzing problems 7-41
 atpg mode sequence 12-3
 auto mode 12-10
 constraints 4-38
 controlling 12-2
 custom models 4-43
 design flow 4-2
 distributed processing 18-1, 18-6, 18-14
 effectiveness 10-11
 effort level 12-6
 environment setup 4-35

increasing effort over multiple passes 4-27
libraries, condensing 4-47
maximizing test coverage 4-28
merge effort 4-24
minimizing patterns 4-28
model building 4-8, 6-17
options 12-2
overview A-6
preparation steps 4-20
quick test coverage estimation 4-24
random decisions option 12-5
reviewing test coverage 4-29
run settings 4-23
running distributed processing 18-3
set atpg command 4-23, 4-24
setting a pattern limit 12-5
setting an abort limit 12-6

ATPG design guidelines B-2
bidirectional port controls B-13
bus keepers B-25
checklist B-28
clock control B-6
clock sources, clock trees, and clock edges B-14
internally generated pulsed signals B-2
multidriver nets B-11
protection of RAMs during scan shifting B-21
pulsed signal
 to RAMs and ROMs B-23
 to sequential devices B-9
RAM and ROM controllability during ATPG B-21

ATPG modes 12-2
algorithm sequence 12-3
for transition delay fault model 14-4
in run justification command 7-31
overview 1-3
running algorithms separately 12-4

at-speed clock, defined 15-2

AU fault category 10-8

auto mode ATPG run 12-10

B

background jobs, running 2-7
Basic-Scan ATPG 12-2
 algorithm sequence 12-3
 Basic-Scan-only ATPG run 12-4
batch mode command execution 3-5
bidirectional
 port controls (design guideline) B-13
 ports, forcing to Z 9-38
binary image files,image files
 secure 6-20
black box
 models 6-5
bookmark help topic 3-17
boundary scan A-12
bridging faults 16-5
budget (number of strobes) 16-6
buffer, simulation 12-22
BUILD> mode 3-4
building the ATPG model 4-8, 6-17
 processes 6-17
 Set Build dialog box 6-19, 6-20
built-in commands 21-5
burn-in testing 16-5
bus analysis, aborted 20-12
bus keepers
 design guideline B-25
buses 16-19

C

capital letters (in command syntax) 3-9
capture clock edge, defined 15-2
capture clock, defined 15-2
capture procedures 9-17
capture vector, defined 15-2
case sensitivity in netlists 4-6
case-sensitivity in netlists 6-3
checklist

for ATPG design B-28
for test port selection B-29

circuit design, general IDDQ principles 16-20

clock
control (design guideline) B-6
master-slave 9-33
sources, trees, and edges (design guideline)
B-14
timing specifications 9-5

clock grouping 4-49

clock matrix report 4-50

clock on 20-6

clock_launch (transition delay fault ATPG mode) 14-4

clocking scan chains (design guideline) B-14

ClockPO patterns 20-6

CMOS
characteristics 16-2
inverter 16-2
NAND gate 16-3

collapsed and uncollapsed faults 4-33

collapsed fault list 10-3

command
script 21-9

command abbreviation 21-3

command buttons 3-3

command continuation 3-4

command entry 3-3

command files 2-4, 3-5
example 4-34

command history 3-5

command logging 3-5

command mode indicator 3-4

command nesting 21-4

command toolbar 3-3

command-line
entry 3-4
window 3-4

commands
add clocks 9-5

add distributed processors 18-2, 18-8
add faults 4-20, 10-4, 11-18, 14-10
add net connections 4-43
add pi constraints 4-37, 9-4
add pi equivalences 4-36
alias 21-8
analyze faults 20-10
built-in 21-5
command entry 3-3
interrupting 21-9
log file 3-5
man 3-9, 4-11
online help 3-8
read faults 4-20, 10-3
read memory_file 6-17
read netlist 4-6, 6-2
redirect 21-6
remove distributed processors 18-2, 18-8
remove faults 4-20, 10-4
remsh 18-5
report distributed 18-3
report distributed processors 18-8
report faults 4-30, 20-9
report feedback paths 7-29
report licenses 2-3
report rules 4-11, 4-15, 7-35
report summaries 4-30
report violations 4-12, 7-35
rsh 18-5
run atpg 4-23, 12-4, 18-3
run build_model 4-8, 6-17
run drc 4-10
run fault_sim 11-20
run fault_simulation 18-10
run justification 7-31, 20-11
run simulation 11-19
run_atpg 18-14
set atpg 4-23, 4-24, 12-2
set build 6-19, 6-20
set contention 4-22
set delay 14-11
set distributed 18-3

set distributed -worki_dir 18-2
 set drc 4-18
 set faults 10-3, 10-9
 set learning 6-19
 set patterns 4-21, 11-15, 12-2
 set pindata 7-18
 set random_patterns 12-2
 set scan ability 4-42
 set simulation 11-19
 set work directory 18-7
 set workspace sizes 2-12
 source 21-9
 TetraMAX IDDQ 16-17
 tmax 2-4, 18-4
 write faults 4-33
 write patterns 4-32
 compressing patterns 12-10
 conditions for quiescence in TetraMAX 16-18
 configuration files

- FTDL D-2
- TDL91 D-2
- TSTL2 D-2
- WGL_FLAT D-2

 constraints

- ATPG 4-38
- DRC 16-19
- primary input, in STIL 9-3

 contention

- bus contention status 7-37
- choosing settings for checking 4-22
- Contention Ability Checking report 7-38
- possible causes 7-40
- set contention command 4-22

 continuation character (\) 3-4
 controllability

- definition A-3
- determining with run justification command 20-11

 controllability/observability checking 7-30
 critical delay path

- defining 15-10
- critical path, defined 15-2
- custom ATPG models 4-43

D

- defining
 - critical delay path 15-10
- delay path, defined 15-3
- delay paths
 - viewing 15-14
- delay test
 - report untested paths 15-24
 - untested paths 15-23
- delay test waveform tables 15-15
- dependent pattern mapping 13-8
- design flows
 - diagrams 16-6
 - TetraMAX IDDQ pattern generation
 - IDDQ
 - pattern generation 16-6
- design guidelines for ATPG
 - checklist B-28
- design rule checking 4-9, 16-9
 - allowing unstable sets/resets 4-18
 - constraints 16-19
 - design rule categories 4-14
 - feedback paths 4-19
 - for transition faults 14-8
 - procedure simulation 4-18
 - process (steps carried out) 4-16
 - rerunning DRC 20-4
 - reviewing results 4-11
 - rule severity levels 4-15
 - rule violation effects 4-16
 - Run DRC dialog box 4-9
 - scan chain tracing 4-17
 - Set Rules dialog box 4-15
 - shadow register analysis 4-17
 - summary report 4-11
 - transparent latches 4-18
 - unstable cells 4-18

using GSV 4-12
 violations 7-32
 designing for IDDQ testability 16-19
 analog blocks 16-20
 buses 16-19
 circuit design, general 16-20
 connections 16-22
 I/O pads 16-19
 oscillators, free-running 16-20
 power and ground 16-21
 RAMs 16-20
 SOC (system-on-a-chip) rules 16-23
 summary 16-22
 switch/FET primitives 16-21
 detected (DT) fault category 10-6
 detected by implication (DI) fault class 10-6
 detected by simulation (DS) fault class 10-6
 detection
 non-robust, defined 15-3
 robust, defined 15-3
 deterministic pattern generation A-7
 DFT Compiler
 design flow 1-5
 large design 1-7
 exporting a design to TetraMAX C-1
 DI fault class 10-6
 diagnosing manufacturing test failures 19-1
 diagnosis report 19-11
 dialog box
 Add PI Equivalences 4-36
 Report Faults 4-32
 Run Build Model 4-8
 distributed ATPG 18-3, 18-6, 18-14, 18-16
 distributed fault simulation 18-6, 18-10, 18-16
 distributed processing
 adding machines 18-2
 ATPG 18-6, 18-14
 ATPG considerations 18-6
 controlling timeouts 18-3
 .cshrc file considerations 18-5
 events 18-11
 fault simulation 18-6
 flow 18-3
 how to use 18-6
 identifying a work directory 18-2
 interpreting fault sim results 18-12
 licensing schemes 18-16
 load sharing facility (LSF) 18-9
 load sharing software setup 18-6
 remote shell considerations 18-5
 removing a machine 18-2
 reporting current slave machines 18-3
 saving results 18-16
 setting up the environment 18-7
 slave log files 18-16
 starting ATPG 18-3
 starting fault simulation 18-10
 verifying environment 18-4
 DR fault class 10-6
 DRC dialog box 9-6
 DRC. See design rule checking 16-9
 DRC violations 7-32
 bidirectional contention 7-35
 bus and wire gate contention 7-37
 during
 load_unload procedure 20-5
 Shift procedure 20-6
 during test_setup macro 20-7
 scan chain blockage 7-32
 See also design rule checking
 DRC> mode 3-4
 DS fault class 10-6
 DT fault category 10-6
 dumpports system task for VCDE 11-9
 duplicate module definition 6-13
 dynamic
 pattern compression 12-10

E

EDIF netlist
 case-sensitivity 6-3

requirements 4-4
 Edit Clocks dialog box 9-5
 empty box models 6-5
 environment, distributed processing 18-7
 equivalent ports 4-36
 error (design rule severity level) 4-15

F

failure data format (for diagnosis) 19-2
 failures, diagnosing 19-1
 false path, defined 15-3
 false paths 15-22
 Fast-Sequential ATPG 12-2
 algorithm sequence 12-3
 capture cycles 12-3
 effort level 12-3
 Fast-Sequential-only ATPG run 12-4
 fatal (design rule severity level) 4-15
 fault categories 10-5
 AU 10-8
 DT 10-6
 ND 10-8
 PT 10-7
 UD 10-7
 fault classes 10-5
 AN 10-8
 AP 10-7
 DI 10-6
 DR 10-6
 DS 10-6
 NC 10-8
 NO 10-8
 NP 10-7
 UB 10-8
 UR 10-8
 UT 10-7
 UU 10-7
 fault coverage 10-11
 ATPG effectiveness calculation 10-11

calculation 10-11
 test coverage calculation 10-10
 fault dictionary 10-5
 fault grading 11-1
 fault lists 10-2
 adding faults from a file 10-3
 adding/reading faults 4-20
 collapsed and uncollapsed 10-3
 files 10-2
 initializing 11-18
 random sampling 10-4
 writing 4-32
 fault models
 IDDDQ 16-8, 16-11
 in TetraMAX
 overview 16-8
 specifying 16-17
 overview 1-4
 pseudo-stuck-at model 16-8
 stuck-at 1-4
 toggle model 16-8
 fault simulation 11-1, 11-20
 ATE requirements 11-4
 combining ATPG and functional test patterns 11-21
 declaring clocks for DRC 11-12
 design
 flow summary 11-2
 preparation 11-11
 distributed processing 18-6, 18-10
 external pattern formats 11-6
 functional test patterns
 preparing 11-3
 reading 11-14
 good machine simulation 11-19
 initializing the fault list 11-18
 nonscan pattern requirements 11-7
 overview A-6
 pattern file examples 11-7
 report summaries command 11-21
 running 11-20
 DRC 11-12

scan and nonscan patterns 11-6
 scan pattern requirements 11-6
 timing insensitivity 11-5
 writing the fault list 11-21
 fault summary report 4-30, 10-9
faults 10-1
 ATPG possibly detected (AP) 10-7
 ATPG untestable (AU) 10-8
 ATPG-not detected (AN) 10-8
 categories/classes 10-5
 collapsed and uncollapsed 4-33
 detected (DT) 10-6
 detected by implication (DI) 10-6
 detected by simulation (DS) 10-6
 detected robustly (DR) 10-6
 not analyzed-possibly detected (NP) 10-7
 not controlled (NC) 10-8
 not detected (ND) 10-8
 not observed (NO) 10-8
 possibly detected (PT) 10-7
 transition 14-1
 undetectable (UD) 10-7
 undetectable blocked (UB) 10-8
 undetectable redundant (UR) 10-8
 undetectable tied (UT) 10-7
 undetectable unused (UU) 10-7
 undetectable versus untestable 10-6
 untestable faults, analyzing 20-10
 feedback paths 4-19
 files
 script 21-9
FTDL
 configuration file D-2
Full-Sequential ATPG 12-2, 12-3
 algorithm sequence 12-3
 Full-Sequential-only ATPG run 12-4
 limitations 12-4
 sequential capture procedure 9-26
function test patterns
 preparing 11-3
functional test patterns
 combining with ATPG patterns 11-21
 reading 11-14
functional versus IDQ testing 1-2

G

generating
 path delay tests 15-12
Graphical Schematic Viewer (GSV)
 Block ID window 7-8
 combinational feedback loops 7-29
 DRC violations 4-12
 expanding the display 7-8
 gate primitive display 7-13
 hiding buffers and inverters 7-10
 hiding objects 7-7
 instance path names 7-17
 inversion bubbles 7-15
 navigation 7-6
 net expansion symbols 7-8
 node display 7-8
 pin data
 display 7-17
 types 7-19
 pin name labels 7-15
 Primitive and Design views 7-17
 primitives displayed 7-11
 RAM and ROM primitives 7-15
 route traversal 7-9
 selecting objects 7-6
 sequential simulation data 12-23
 Show Block dialog box 7-2
 single-point failure display 12-24
 starting 7-2
 from a report 7-3
 toolbar 3-3
 graphical user interface 3-2
GUI 2-8
GZIP netlists 4-6

H

help

- help window 3-9
- on commands 3-8
- online 3-8

I

I/O

- pads 16-19

IDDQ

- Add ATPG Constraints command 16-18
- commands 16-17
- design principles 16-19
- design-for-test summary 16-22
- DRC rule violations 16-9
- fault model 16-8
- Set Faults command 16-17
- Set IDDQ command 16-17
- strokes needed 16-6
- test pattern generation 16-11
- testing overview 16-2
- types of defects detected 16-5

IDDQ commands

- in TetraMAX 16-17

ignore (design rule severity level) 4-15

image files

- binary 6-20

import

- path lists 15-9

independent pattern mapping 13-7

index (online) 3-15

internal scan A-7

interrupting a long process 2-11

interrupting commands 21-9

invoking TetraMAX 2-4, 2-6

J

JTAG A-13

JTAG/TAP controller in STIL 9-38

justification 7-30

K

kernel settings 2-12

L

last_shift_launch (transition delay fault ATPG mode) 14-4

launch clock edge, defined 15-3

launch clock, defined 15-3

launch vector, defined 15-3

learning following ATPG build 6-19

library models, reading 4-7

licenses 2-3, 17-2

- distributed ATPG and fault sim 18-16
- reporting 2-3

load sharing facility 18-9

load_unload procedure 9-8

- DRC violations during 20-5

log file 3-5

logic simulation 12-21

- comparing simulated and expected values 12-21

sequential data 12-23

simulation buffer 12-22

single-point failure simulation 12-23

low test coverage, causes 20-8

LSF network management tool 18-9

Ltran configuration files D-2

Ltran shell mode D-2

M

macros 3-5

main window of TetraMAX 3-2

man command 3-9, 4-11

manufacturing test failures, diagnosing 19-1

m
 masking
 outputs 4-40
 patterns 12-25
 ports 4-38
 scan cells 4-40
 master_observe procedure 9-33
 master-slave clock 9-33
 maximum gates, decisions, lines 2-12
 measures, masking out 12-25
 memory
 default initialization 6-15
 initialization file 6-15
 initializing contents 6-15
 instance-specific initialization 6-16
 model 6-13
 Read Memory dialog box 6-16
 Verilog template 6-14
 menu bar 3-3
 merge effort 4-24
 message logging 3-5
 methods 2-6
 missing module error 6-3
 models, reading 4-7
 module, internal representation report 20-2
MUXClock 15-19
 ATPG requirements 15-22
MUXClock path delay patterns
 limitations 15-22
MUXClock transition patterns
 limitations 14-16

N
 NC fault class 10-8
 ND fault category 10-8
 n-detect 4-29
 nested commands 21-4
 net, multidriver
 design guideline B-11
 netlist

case sensitivity 4-6, 6-3
 duplicate names 4-6
 EDIF, requirements for ATPG 4-4
 GZIP format 4-6
 missing module error 6-3
 reading 4-5
 requirements for ATPG 4-4
 troubleshooting 20-4
 using multiple files 4-7
 Verilog, requirements for ATPG 4-5
 VHDL, requirements for ATPG 4-5
 wildcard characters 6-2
 netlists
 clearing memory 4-7
 master module 4-7
 NO (not observed) fault, analyzing 7-45
 NO fault class 10-8
 nonscan cells
 previewing effects of change to scan cells
 4-41
 not analyzed-possibly detected (NP) fault class
 10-7
 not controlled (NC) fault class 10-8
 not detected (ND) fault category 10-8
 not observed (NO) fault class 10-8
 NP fault class 10-7

O
 observability
 definition A-3
 off-path input, defined 15-3
 online help 3-8
 bookmark topic 3-17
 detailed contents list 3-14
 index 3-15
 major contents list 3-12
 search/find text 3-16
 text-only help 3-8
 window-based help 3-9
 on-path input, defined 15-3

operators
 redirection 21-8
 options
 path delay tests 15-14
 oscillators, free-running 16-20
 outputs
 masking 4-40

P

path definition syntax 15-10
 path delay
 ATPG options 15-14
 tests, generating 15-12
 path delay fault, defined 15-3
 path delay patterns
 MUXClock 15-19
 path delay testing 15-7
 path lists
 importing 15-9
 reading 15-13
 reporting 15-13
 path rule
 violations, analyzing 15-14
 path, defined 15-3
 path, false 15-22
 pattern
 input 12-13
 limit 12-5
 masking, per-cycle 12-25
 output 12-13
 source selection
 (formats) 4-21
 pattern compression 12-10
 balancing compression and CPU runtime
 12-10
 dynamic 12-10
 compression report 12-11
 pattern mapping 13-6
 algorithms 13-6
 dependent 13-8

independent 13-7
 using 13-9
 PatternMap
 embedded module requirements 13-2
 functional test pattern requirements 13-3
 overview 13-2
 patterns
 compressing 12-10
 extracting a pattern subrange 12-7
 masking 12-25
 merging multiple pattern files 12-8
 reading 12-13
 splitting a pattern into smaller patterns 12-7
 STIL functional pattern input 12-14
 using patterns generated separately 12-8
 Verilog input example 12-15
 WGL input example 12-19
 writing 4-31, 12-13
 per-cycle pattern masking 12-25
 PI constraints 9-3
 PI equivalences 4-36
 pin data types 7-19
 clock off example 7-21
 constrain values example 7-22
 load data example 7-23
 multiple ATPG pattern example 7-27
 shift data example 7-25
 single ATPG pattern example 7-26
 test setup example 7-25
 tie data example 7-28
 ports
 bidirectional, controls (design guideline)
 B-13
 bidirectional, forcing to Z 9-38
 choosing for test I/O B-27
 deleting nonlogic ports (power, ground) 4-43
 equivalent 4-36
 for test I/O (selection checklist) B-29
 masking 4-38
 reporting port names 20-2
 scan chain input and output, choosing B-27

possibly detected (PT) fault category 10-7
power and ground nets 16-21
power/ground ports, deleting 4-43
primary input (PI) equivalences 4-36
primitive display (in GSV) 7-11
pseudo-stuck-at model 16-8, 16-11
PT fault category 10-7
pull-down menus 3-3
pulsed ports in STIL 9-13
pulsed signal
 internally generated (design guideline) B-2
 to RAMs and ROMs (design guideline) B-23
 to sequential devices (design guideline) B-9

Q

quiescence
 conditions defined 16-18
 conditions in TetraMAX 16-17
 defined 16-2

R

RAM
 controllability during ATPG (design guideline) B-21
 model 6-13
 protection during scan shifting (design guideline) B-21
 pulsed signal to (design guideline) B-23
RAMs 16-20
random decisions in ATPG 12-5
random pattern generation A-7
random sampling (for generating fault lists) 10-4
read faults command 4-20, 10-3
 for transition delay faults 14-10
Read Memory File dialog box 6-16
read memory_file command 6-17
read netlist command 4-6, 6-2

Read Netlist/Image dialog box 4-6
reading path lists 15-13
reading patterns 12-13
redirect command 21-6
redirecting command output 21-6
redirection operators 21-8
remove distributed processors command 18-2, 18-8
remove faults command 4-20, 10-4
Remove Faults dialog box 10-5
remsh command 18-5
report
 untestable paths 15-24
report distributed command 18-3
report distributed processors command 18-8
report faults command 4-30, 20-9
Report Faults dialog box 4-32
report feedback paths command 7-29
report licenses command 2-3
report modules command
 to list module port names 20-2
report primitives command
 to list ports names 20-2
report rules command 4-11, 4-15, 7-35
report summaries command 4-30
report violations command 4-12, 7-35
reporting path lists 15-13
reports
 clock matrix 4-50
resistive faults 16-5
ROM
 controllability during ATPG (design guideline) B-21
 initialization file 6-15
 model 6-13
 pulsed signal to (design guideline) B-23
rsh command 18-5
rule categories 4-14
rule severity levels 4-15

run atpg command 4-23, 18-3
 ATPG mode sequence 12-3
 for transition delay faults 14-12
 run-only options 12-4

Run ATPG dialog box 4-23, 4-24

Run ATPG options 4-23

Run Build Model dialog box 4-8

run build_model command 4-8, 6-17

run diagnosis command
 diagnosis report 19-11

run drc command 4-10

Run DRC dialog box 4-9

Run Fault Simulation dialog box 11-20

run fault_sim command 11-20

run fault_simulation command 18-10

run justification command 7-31, 20-11

Run Justification dialog box 7-31

run simulation command 11-19

Run Simulation dialog box 11-19

run_atpg command 18-14

S

save/restore 4-19

scan cells
 masking 4-40

scan chain input and output ports, choosing
 B-27

scan clock, defined 15-3

scripts 21-9
 comment lines 21-10
 continue or stop on error 21-10

search for text (online help) 3-16

secure image files 6-20

sequential capture procedure 9-26
 example 9-28
 sequential data display 12-23
 syntax 9-28

set atpg command 4-23, 4-24, 12-2
 for transition delay faults 14-11

set build command 6-19, 6-20
 to delete unused gates 4-47

Set Build dialog box 6-19, 6-20

set contention command 4-22

Set Contention dialog box 4-22

set delay command 14-11

set distributed command 18-3

set distributed -work_dir command 18-2

set drc command 4-18

set faults command 10-3, 10-9
 for transition delay faults 14-10

set faults command in TetraMAX 16-17

set IDDQ command in TetraMAX 16-17

set learning command 6-19

set patterns command 4-21, 11-15, 12-2

Set Patterns dialog box 11-14
 to use external patterns 4-21

VCDE strobe specification 11-15

set pindata command 7-18

set random_patterns command 12-2

Set Rules dialog box 4-15

set scan ability command 4-42

Set Scan Ability dialog box 4-42

set simulation command 11-19

set work directory command 18-7

set workspace sizes command 2-12

settling time 16-6

Setup dialog box
 hiding buffers and inverters 7-10
 pin data display format 7-18
 selecting Primitive or Design view 7-17

setup file 2-8

severity levels of design rules 4-15

sh_command_abbrev_mode variable 21-3

shadow_observe procedure 9-34

shell, running in 2-5

Shift procedure 9-8
 DRC violations during 20-6

shortening commands 21-3

Show Block dialog box 7-2
 signal timing waveform tables 9-16
 signal, pulsed
 internally generated (design guideline) B-2
 to sequential devices (design guideline) B-9
 simulation, logic 12-21
 single-point failure simulation 12-23
 slave machines 18-3
 slow-to-rise/fall (transition delay) faults 14-2
 SOC (system-on-a-chip) rules 16-23
 Solaris platform, 64-bit mode 2-7
 source command 21-9
 SPF. See STIL procedure file
 startup file 2-8
 disabling 2-5
STIL
 functional pattern input 12-14
 input E-18
 output E-16
 overview E-2
STIL procedure file (SPF) 9-1
 asynchronous set and reset ports 9-5
 basic signal timing 9-14
 bidirectional port control 9-11
 capture procedures 9-17
 clock definition 9-4
 constrained primary inputs 9-30
 creating a new SPF 9-2
 end-of-cycle measure 9-29
 equivalent primary inputs 9-31
 guidelines 9-37
 IEEE Std. 1450.-1999 9-2
 JTAG/TAP controller 9-38
 load_unload procedure 9-8
 master_observe procedure 9-33
 multiple scan groups 9-39
 overview 9-2
 primary input (PI) constraints 9-3
 pulsed ports 9-13
 quotation marks 9-8
 reflective I/O capture 9-32
 scan chain definition 9-6
 sequential capture procedure 9-26
 shadow_observe procedure 9-34
 Shift procedure 9-8
 test_setup macro 9-9
 testing the SPF 9-37
 troubleshooting 20-5
 writing a template 9-7
 stopping a long process 2-11
 strobe specification for VCDE input 11-15
 strobes
 number of strobes needed 16-6
 Submit/Stop button 2-11, 3-5
 switch/FET primitives 16-21
 system-on-a-chip rules 16-23

T

Tcl
 extensions 21-5
 restrictions 21-5
 special characters 21-4
 syntax and native commands 21-2
TDL91
 configuration file D-2
test
 non-robust, defined 15-3
 robust, defined 15-4
test coverage 10-10
 ATPG effectiveness calculation 10-11
 causes of low test coverage 20-8
 fault coverage calculation 10-11
 reviewing 4-29
 target test coverage value 12-5
 test coverage calculation 10-10
test failures, diagnosing 19-1
test I/O ports
 choosing B-27
 selection checklist B-29
test_setup macro
 DRC violations during 20-7

testability, designing for 16-19
 Test-Accelerate-Max license key 18-16
 tester failure data format 19-2
 TestGen
 identifying untestable paths 15-23
 testing
 characterization models 15-5
 I/O paths 15-6
 manufacturing models 15-4
 path delay 15-7
 tests
 path delay, generating 15-12
 TetraMAX
 add atpg constraints command 16-18
 design rule checking (DRC) 16-9
 fault models 16-8
 generating IDDQ test patterns 16-11
 IDDQ commands 16-17
 set faults command 16-17
 set IDDQ command 16-17
 tmax command 2-4, 18-4
 tmax_cmd.perl 21-2
 tmax.rc file 2-8
 TMAXRC variable 2-8
 toggle
 fault model 16-8
 model 16-11
 toolbars 3-3
 transcript window 3-6
 clearing text 3-8
 copying text 3-7
 finding text 3-7
 saving or printing text 3-7
 selecting text 3-7
 transition delay fault ATPG 14-1
 adding faults to the fault list 14-10
 ATPG modes (`last_shift_launch`,
 `clock_launch`, `any`) 14-4
 Basic-Scan, Fast-Sequential, and Full-
 Sequential modes 14-5
 cycle-time switching 14-14
 DRC 14-8
 fault model 14-2
 launch and capture 14-2
 limitations 14-9
 pattern compression 14-13
 pattern formatting 14-14
 pattern generation 14-11
 primary input control 14-11
 reading a fault list file 14-10
 report faults command 14-13
 run atpg command 14-12
 selecting the fault model 14-10
 set atpg command 14-11
 set delay command 14-11
 slow-to-rise and slow-to-fall faults 14-2
 STIL protocol 14-6
 test flow (summary diagram) 14-3
 write faults command 14-13
 transition fault waveform tables 14-6
 translating
 native-mode scripts 21-2
 transparent latches 4-18
 troubleshooting
 netlists 20-4
 STIL procedures 20-5
 TSTL2
 configuration file D-2

U

UB (undetectable, blocked) fault, analyzing
 7-43
 UB fault class 10-8
 UD fault category 10-7
 uncollapsed fault list 10-3
 undetectable
 (UD) fault category 10-7
 blocked (UB) fault class 10-8
 fault, definition 10-6
 redundant (UR) fault class 10-8
 tied (UT) fault class 10-7

unused (UU) fault class 10-7
unstable cells 4-18
untestable fault
 analyzing 7-41, 15-25, 20-10
 definition 10-6
untestable paths
 reporting 15-24
unused logic, removing 4-44
uppercase letters (in command syntax) 3-9
UR fault class 10-8
using
 pattern mapping 13-9
UT fault class 10-7
UU fault class 10-7

V

variables
 sh_command_abbrev_mode 21-3
variables, user-defined 2-4
variables, using 2-9
VCDE
 input patterns 11-9
 pattern input 11-15
 strobe specification 11-15
Verilog
 case-sensitivity 6-3
 input pattern example 12-15
 netlist requirements 4-5

VHDL
 case-sensitivity 6-3
 netlist requirements 4-5
viewing
 delay paths 15-14
violations 16-9
 path rule 15-14
vtran D-7

W

warning (design rule severity level) 4-15
waveform tables
 ancillary timing 15-16
 delay test 15-15
 signal timing 9-16
 transition fault 14-6
WGL
 input pattern example 12-19
WGL_FLAT
 configuration file D-2
Why do IDDQ testing? 16-2
wildcards, using to read netlists 6-2
window, TetraMAX main 3-2
workspace size 2-12
write faults command 4-33
write patterns command 4-32
Write Patterns dialog box 4-31
writing patterns 12-13