

TestMAX ATPG and TestMAX Diagnosis User Guide

Version S-2021.06, June 2021

SYNOPSYS[®]

Copyright and Proprietary Information Notice

© 2021 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

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.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>.

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

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Contents

1. TestMAX ATPG and TestMAX Diagnosis Overview	53
Launching TestMAX ATPG and TestMAX Diagnosis	54
Setting the Thread Count in TestMAX ATPG	55
Multithreading in TestMAX ATPG and TestMAX Diagnosis	56
Multithreading Limitations	57
TestMAX ATPG Multithreading Command Option Support	58
run_atpg	59
run_fault_sim	59
run_simulation	60
set_atpg	60
set_delay	61
set_drc	61
ATPG Capabilities	61
TestMAX ATPG Modes	62
Features and Benefits	63
Operation Modes	65
2. Getting Started	66
Basic TestMAX ATPG Processes	66
Installing TestMAX ATPG	66
Specifying the Location for TestMAX ATPG Installation	67
Setting the Environment	67
Launching TestMAX ATPG	68
Executable Commands	68
Setup Command Files	69
Using Command Files	70
Batch Files	71
Launching TestMAX ATPG Using Command Files	71
Using Variables	72
Tcl Mode	73
Native Mode	74
Running the TestMAX ATPG GUI	74

Starting and Stopping the TestMAX ATPG GUI	74
Interrupting a Long Process	74
Setting Preferences	76
Saving GUI Preferences	77
Basic ATPG Flow	78
Reference Methodology	80
Getting Started for TestMAX DFT Users	80
Design Flow Using TestMAX DFT and TestMAX ATPG	81
<hr/>	
3. ATPG Design Flow	85
ATPG Design Flow Overview	85
Basic ATPG Run Script	87
Running the Basic ATPG Design Flow	87
Preparing a Netlist	88
Configuring to Read a Netlist	89
Reading a Netlist	90
Reading Library Models	91
Preparing to Build the ATPG Model	92
Building the ATPG Model	92
Performing Design Rule Checking (DRC)	94
Specifying STIL Procedures	94
Specifying DRC Settings	95
Options for Specifying DRC Settings	96
Starting DRC	97
Reviewing the DRC Results	99
Understanding Rule Violations	100
Viewing DRC Violations in the GSV	101
Preparing for ATPG	103
Specifying General ATPG Settings	103
Options for Specifying ATPG Settings	104
Specifying Fault Lists	105
Selecting an Existing Fault List File	106
Generating a Fault List Containing All Fault Sites	106
Including Specific Faults in a Fault List	106
Writing Faults to a File	107
Example Fault Lists	108

Specifying Fault Models	108
Selecting a Fault Model	110
Specifying the Pattern Source	111
Scan and Nonscan Functional Patterns	112
STIL Functional Pattern Input	112
Verilog Functional Pattern Input	115
WGL Functional Pattern Input	121
VCDE Functional Pattern Input	124
Options for Selecting the Pattern Source	125
Specifying the ATPG Mode	127
Basic Scan Mode Settings	128
Fast-Sequential Mode Settings	129
Setting Full-Sequential Mode	130
Running ATPG	131
Running ATPG in Basic Scan or Fast-Sequential Mode	132
Using Automatic Mode to Generate Optimized Patterns	133
Setting Automatic Mode	134
Quickly Estimating Test Coverage	134
Specifying a Test Coverage Target Value	139
Increasing ATPG Effort Over Multiple Passes	139
Multiple Session Test Pattern Generation	140
Splitting Patterns	140
Extracting a Pattern Sub-Range	141
Merging Multiple Pattern Files	141
Using Pattern Files Generated Separately	142
Compressing Patterns	143
Balancing Pattern Compaction and CPU Runtime	143
Compression Reports	144
Analyzing ATPG Output	146
Standard Format	146
Expert Format	148
Verbose Format with Merge (without -auto_compression)	149
Verbose Format with Merge and -auto_compression	151
Reviewing Test Coverage	153
Writing ATPG Patterns	156
<hr/>	
4. ATPG Modeling	157
Modeling Topics	157
ATPG Modeling Primitive Summary	157

Contents

TestMAX ATPG Memory Modeling	159
Basic Template	159
Defining Write Ports	160
Defining Read Ports	162
Read Off Behavior	164
Complete Example	165
Memory Address Range	165
Multiple Read or Write Ports	166
Rules and Limitations	167
Controlling Contention Behavior	167
Memory Modeling Syntax in Backus-Naur Form (BNF)	174
RAM and ROM Modeling Examples	176
Memory Data File Examples	192
Interpreting UDP Messages	193
Variant #1	193
Variant #2	196
Variant #3	199
Variant #4	200
Debugging UDP-based Models	202
Modeling Examples	203
Optimistic MUX	203
MUX, 4-to-1	204
Latch	205
Latch With Active Low Asynchronous Set/Reset	205
Latch With Asynchronous Set/Reset	206
Latch With Asynchronous Set/Reset Dominant Over EN	207
Latch With Asynchronous Set/Reset Dominant Over EN, Reset Dominant	208
Latch With Asynchronous Set/Reset Dominant Over EN, Set Dominant ..	208
Dual Port Latch	209
Positive-edge Clocked DFF With Notify	210
DFF With Active Low Asynchronous Set/Reset and Notify	211
DFF With Active Low Asynchronous Reset and Notify	212
DFF With Active High Asynchronous Set/Reset	213
DFF With Synchronous Reset and Notify	213
Negative-Edge Clocked DFF With Active Low Asynchronous Clear and Notify	214
DFF and Latch	215
JK Flip-Flop With Active Low Asynchronous Set/Reset and Notify	216
Bus Keeper Examples	217
Scan Cell Models	218
Scan Cell Models - MUX Flop Scan	218
Scan Cell Models - Master Slave Latch	220

Contents

Scan Cell Models - MUX Latch Scan	221
Scan Cell Models - Clocked Scan Flip-Flop	222
Scan Cell Models - Clocked Scan Latch	223
Scan Cell Models - Single Latch LSSD	224
Scan Cell Models - Double Latch LSSD	225
Scan Cell Models - Clocked LSSD	226
Scan Cell Models - Auxiliary Clocked LSSD	228
Scan Cell Models - Retention Cell	229
ATPG Simulation Primitives	229
AND Primitive	231
Simulation Behavior	231
Verilog Netlist Usage	231
ADRBUS Primitive (Address Bus)	232
Simulation Behavior	232
Verilog Netlist Usage	233
BUF Primitive (Buffer)	233
Simulation Behavior	233
Verilog Netlist Usage	233
BUS Primitive	234
Simulation Behavior	234
Verilog Netlist Usage	235
BUSK Primitive (Bus Keeper)	235
Simulation Behavior	236
Verilog Netlist Usage	236
CMUX Primitive (Conservative Multiplexer)	236
Simulation Behavior	237
Verilog Netlist Usage	237
DATABUS Primitive (Data Bus)	238
Simulation Behavior	238
Verilog Netlist Usage	238
DFF Primitive	239
Approximate Simulation Behavior	239
Textual Simulation Behavior	240
Verilog Netlist Usage	241
DLAT Primitive	241
Simulation Behavior	242
Verilog Netlist Usage	243
EQUIV Primitive (Equivalence)	243
Simulation Behavior	243
Verilog Netlist Usage	244
INV Primitive (Inverter)	244

Contents

Simulation Behavior	244
Verilog Netlist Usage	245
MEMORY Primitive (RAM/ROM Memory)	245
Simulation Behavior	246
Verilog Netlist Usage	247
MOUT Primitive (Macro Output)	247
Simulation Behavior	247
Verilog Netlist Usage	248
MUX Primitive (Multiplexer)	248
Simulation Behavior	248
Verilog Netlist Usage	249
NAND Primitive	249
Simulation Behavior	249
Verilog Netlist Usage	250
NOR Primitive	250
Simulation Behavior	250
Verilog Netlist Usage	251
OR Primitive	251
Simulation Behavior	252
Verilog Netlist Usage	252
PI Primitive (Primary Input)	253
Simulation Behavior	253
Verilog Netlist Usage	253
PIO Primitive (Primary Input/Output)	253
Simulation Behavior	254
Verilog Netlist Usage	254
PO Primitive (Primary Output)	254
Simulation Behavior	254
Verilog Netlist Usage	254
RPORT Primitive (Read Port)	255
Simulation Behavior	255
Verilog Netlist Usage	256
SEL01 Primitive	256
Simulation Behavior	256
Verilog Netlist Usage	257
SEL1 Primitive	257
Simulation Behavior	257
Verilog Netlist Usage	258
SW Primitive (Switch)	258
Simulation Behavior	258
Verilog Netlist Usage	259

Contents

TIE0 Primitive	259
Simulation Behavior	259
Verilog Netlist Usage	260
TIE1 Primitive	260
Simulation Behavior	260
Verilog Netlist Usage	260
TIEX Primitive	261
Simulation Behavior	261
Verilog Netlist Usage	261
TIEZ Primitive	262
Simulation Behavior	262
Verilog Netlist Usage	262
TSD Primitive (tristate Device)	262
Simulation Behavior	263
Verilog Netlist Usage	263
WIRE Primitive	264
Simulation Behavior	264
Verilog Netlist Usage	264
XNOR Primitive (Exclusive NOR)	265
Simulation Behavior	265
Verilog Netlist Usage	266
XOR Primitive (Exclusive OR)	266
Simulation Behavior	266
Verilog Netlist Usage	267
<hr/>	
5. Command Interface	268
TestMAX ATPG GUI	268
Command Entry	270
Menu Bar	270
Command Toolbar and GSV Toolbar	270
Command-Line Window	271
Command Mode Indicator	271
Command-Line Entry Field	272
Command Continuation	272
Command History	273
Stop Button	273
Commands From a Command File	273
Command Logging	274
Transcript Window	274
Setting the Keyboard Focus	275

Using the Transcript Text	275
Selecting Text in the Transcript	276
Copying Text From the Transcript	276
Finding Commands and Messages in the Transcript	276
Saving or Printing the Transcript	277
Clearing the Transcript Window	277
Interacting with the TestMAX ATPG GUI	278
Using Keys in the Command Line	278
Using the Graphical Schematic Viewer	278
Using the Transcript Window	279
Saving Preferences	280
Using Online Help	280
Browser-Based Online Help	280
Setting Up Online Help in Linux	281
Launching Online Help	281
Installing and Running Stand-Alone Online Help in Windows	284
How to Browse, View, and Copy Scripts	286
Text-Only Help	289
<hr/>	
6. Using the Graphical Schematic Viewer	291
Getting Started With the GSV	291
Using the SHOW Button to Start the GSV	292
Starting the GSV From a DRC Violation or Specific Fault	293
Navigating, Selecting, Hiding, and Finding Data	296
Navigating Within the GSV	296
Selecting Objects in the GSV Schematic	296
Hiding Objects in the GSV Schematic	297
Using the Block ID Window	297
Expanding the Display From Net Connections	298
Hiding Buffers and Inverters in the GSV Schematic	300
ATPG Model Primitives	301
Tied Pins	302
Primary Inputs and Outputs	302
Basic Gate Primitives	303
Additional Visual Characteristics	304
RAM and ROM Primitives	305
Displaying Symbols in Primitive or Design View	306
Displaying Instance Path Names	307
Displaying Pin Data	307

Using the Setup Dialog Box to Display Pin Data	308
Pin Data Types	309
Displaying Clock Cone Data	310
Displaying Clock Off Data	311
Displaying Constrain Values	312
Displaying Load Data	313
Displaying Shift Data	314
Displaying Test Setup Data	315
Displaying Pattern Data	315
Displaying Tie Data	317
Analyzing a Feedback Path	317
Checking Controllability and Observability	318
Using the Run Justification Dialog Box	319
Using the run_justification Command	319
Analyzing DRC Violations in the GSV	320
Troubleshooting a Scan Chain Blockage	320
Troubleshooting a Bidirectional Contention Problem	322
Analyzing Buses	324
BUS Contention Status	325
Understanding the Contention Checking Report	325
Reducing Aborted Bus and Wire Gates	326
Using the Analyze Buses Dialog Box	327
Using the set_atpg and analyze_buses Commands	327
Causes of Bus Contention	327
Analyzing ATPG Problems	328
Analyzing an AN Fault	329
Analyzing a UB Fault	330
Analyzing a NO Fault	332
Printing a Schematic to a File	332
<hr/>	
7. Using the Hierarchy Browser	334
Launching the Hierarchy Browser	334
Basic Components of the Hierarchy Browser	336
Using the Hierarchy Pane	336
Viewing Data in the Instance Pane	340
Copying an Instance Name	342
Viewing Data in the Lib Cells/Tree Map Pane	343

Performing Fault Coverage Analysis	345
Understanding the Types of Coverage Data	345
Expanding the Design Hierarchy	347
Viewing Library Cell Data	350
Adjusting the Threshold Slider Bar	351
Identifying Fault Causes	352
Displaying Instance Information in the GSV	355
Exiting the Hierarchy Browser	356
<hr/>	
8. Using the Simulation Waveform Viewer	358
Getting Started With the SWV	358
Understanding the SWV Color Codes	359
Supported Pin Data Types and Definitions	360
Invoking the SWV	362
Using the SWV Interface	363
Understanding the SWV Layout	364
Refreshing the View	364
Manipulating Signals	365
Using the Signal List Pane	365
Adding Signals	365
Deleting Signals	366
Inserting Signals	366
Identifying Signal Types in the Graphical Pane	367
Using the Time Scales	368
Using the Marker Header Area	368
Adding and Deleting Pointers	369
Moving a Marker Pointer	369
Measuring Between Two Pointers	370
Using the SWV With the GSV	370
Using the SWV Without the GSV	372
Example Flow	372
Example 2	372
Example 3	373
SWV Inputs and Outputs	373
Analyzing Violations	373
<hr/>	
9. Using Tcl With TestMAX ATPG	374

Converting TestMAX ATPG Command Files to Tcl Mode	374
Converting a Collection to a List in Tcl Mode	375
Tcl Syntax and TestMAX ATPG Commands	375
Specifying Lists in Tcl Mode	376
Tcl Mode and Backslashes	377
Using Positional Arguments	377
Abbreviating Commands and Options in Tcl Mode	377
Using Tcl Special Characters	378
Using the Result of a Tcl Command	379
Using Built-In Tcl Commands	379
TestMAX ATPG Extensions and Restrictions in Tcl Mode	380
Redirecting Output in Tcl Mode	380
Using the redirect Command in Tcl Mode	381
Getting the Result of Redirected Tcl Commands	382
Using Redirection Operators in Tcl Mode	382
Using Command Aliases in Tcl Mode	382
Interrupting Tcl Commands	383
Using Command Files in Tcl Mode	383
Adding Comments	384
Controlling Command Processing When Errors Occur	384
Using a Setup Command File	385
An Introduction to the TestMAX ATPG Tcl API	385
Retrieving Information	385
Using the -filter Option	386
Using the -regexp Option	386
<hr/>	
10. Design Netlists and Library Models	387
Netlist Format Requirements	388
EDIF Netlist Requirements	388
Logic 1/0 Using Global Nets	388
Logic 1/0 by Special Library Cell	388
Verilog Netlist Requirements	389
VHDL Netlist Requirements	390
About Reading a Netlist	390
Using Wildcards to Read Netlists	391
About Reading Library Models	392

Contents

Controlling Case-Sensitivity	392
Setting Parameters for Learning	393
Learned Behavior Types	393
Controlling the ATPG Learning Algorithm	394
About Building the ATPG Model	395
Processes That Occur When Building the ATPG Model	396
Flattening Optimization for Hierarchical Designs	397
Identifying Missing Modules	404
Removing Unused Logic	406
Using Black Box and Empty Box Models	409
Declaring Black Boxes and Empty Boxes	410
Behavior of RAM Black Boxes	412
Case 1	412
Case 2	412
Case 3	412
Case 4	413
Case 5	414
Case 6	415
Troubleshooting Unexplained Behavior	416
Handling Duplicate Module Definitions	417
Creating Custom ATPG Models	417
Condensing ATPG Libraries	419
Assertions	420
Implementing Assertions	420
Using Assertions with PLLs and Memories	422
Assertion Descriptions	425
Limitations	427
Memory Modeling	427
Memory Model Functions	427
Basic Memory Modeling Template	428
Initializing RAM and ROM Contents	429
The Memory Initialization File	429
Default Initialization	430
Instance-Specific Initialization	430
Improving Test Coverage for RAMs	431
<hr/>	
11. STIL Procedures	433

Contents

STIL Procedure File Guidelines	433
Creating a New STIL Procedure File	435
Declaring Primary Input Constraints	436
Using the Add PI Constraints Dialog Box	436
Using the add_pi_constraints Command	437
Declaring Clocks	437
Using the Edit Clocks Dialog Box	437
Using the add_clocks Command	438
Asynchronous Set and Reset Ports	438
Declaring Scan Chains and Scan Enables	438
Using the DRC Dialog Box	439
Declaring Scan Chains at the Command Line	439
Writing the SPF Template	440
Example SPF Template File	440
Defining STIL Procedures	442
Defining Scan Chains	443
Defining the load_unload Procedure	444
Controlling Bidirectional Ports	445
Defining the Shift Procedure	446
Defining the test_setup Procedure	447
Using Loop Statements	449
Predefined Signal Groups in STIL	450
Defining Basic Signal Timing	450
Defining Pulsed Ports	452
Selecting Strobed or Windowed Measures in STIL	454
Supporting Clock ON Patterns in STIL	455
Defining the End-of-Cycle Measure	457
Defining Capture Procedures in STIL	458
Limiting Clock Usage	459
Defining Constrained Primary Inputs	460
Defining Equivalent Primary Inputs	461
Defining PO Masks	461
Defining System Capture Procedures	462
Creating Generic Capture Procedures	464
Generating Generic Capture Procedures	465
Controlling Multiple Clock Capture	468
Using Allclock Procedures	470
Using load_unload for Last Shift-Launch Transition	471
Example Post-Scan Protocol	472
Generic Capture Procedures Limitations	473
Defining Sequential Capture Procedures	474

Using Default Capture Procedures	474
Using a Sequential Capture Procedure	475
Sequential Capture Procedure Syntax	475
Defining Reflective I/O Capture Procedures	476
Using the master_observe Procedure	477
Using the shadow_observe Procedure	478
Using the delay_capture_start Procedure	479
Using the delay_capture_end Procedure	481
Using the test_end Procedure	482
Scan Padding Behavior	483
Using the Condition Statement in STIL	485
Excluding Vectors From Simulation	486
Using the DontSimulate Statement for Loops and Reference Clocks	486
Syntax and Example for Excluding Vectors	487
Defining Internal Clocks for PLL Support	488
Specifying an On-Chip Clock Controller Inserted by DFT Compiler	490
Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller	492
ClockTiming Block Syntax	492
Timing and Clock Pulse Overlapping	493
Controlling Latency for the PLLStructures Block	495
ClockTiming Block Selection	495
ClockTiming Block Example	496
Specifying Internal Clocking Procedures	497
ClockConstraints and ClockTiming Block Syntax	498
Specifying the Clock Instruction Register	500
Specifying External Clocks	500
Example 1	501
Example 2	502
JTAG/TAP Controller Variations for the load_unload Procedure	503
Multiple Scan Groups	504
DFTMAX Compression with Serializer	511
12. Design Rule Checking	512
Understanding the DRC Process	513
Contention Analysis	513
BUS Contention Ability Checking	514
BUS Z State Ability Checking	515

Contents

Contention Prevention Checking	515
Simulation Contention Detection	515
ATPG Contention Prevention	515
Post-Capture Contention Checking	516
Scan Chain Tracing	516
Clock Grouping	516
Reducing the Pattern Count Through Clock Grouping	517
Clock Grouping Analysis	518
Generating a Clock Group Report	520
Clock Grouping Limitations	521
Declaring Equivalent and Differential Input Ports	521
Using the Add PI Equivalences Dialog Box	522
Using the add_pi_equivalences Command	522
Cells With Asynchronous Set/Reset Inputs	523
Masking Input and Output Ports	523
Masking Scan Cell Inputs and Outputs	524
Specifying Cell Constraints Locations and Scan Cell Controls	524
Using the Add Cell Constraints Dialog Box	525
Using the add_cell_constraints Command	525
Previewing Potential Scan Cells	525
Scan Cell Types	526
Identifying Scan Cells	527
Reporting Scan Cells	528
Scan Cell Inversion Data	528
Using the set_scan_ability Command	528
Using the Set Scan Ability Dialog Box	529
Transparent Latches	529
Shadow Register Analysis	529
Feedback Paths Analysis	530
Procedure Simulation	530
Changing the Design Rule Severity	530
Using the Set Rules Dialog Box	531
Using the set_rules Command	531
Understanding the DRC Summary Report	532
Binary Image Files	536
Creating and Reading Image Files	536

Creating a Non-Secure Image File	538
Creating a Secure Image File	538
Save/Restore in TEST Mode	540

13. Optimizing ATPG	541
Optimizing Basic Scan Patterns	542
Using ATPG Constraints	543
Adding ATPG Constraints to Block a Timing-Sensitive Path	543
Defining, Reporting, and Removing No Detection Credit Cells	544
Using ATPG Constraints to Control ATPG Assertions	544
Using the Random Decision Option	546
Obtaining Target Test Coverage Using Fewer Patterns	546
Maximizing Test Coverage Using Fewer Patterns	547
Improving Test Coverage With Test Points	547
Test Points Analysis Options	548
Running the Test Points Analysis Flow	548
Limitation	549
Limiting the Number of Patterns	549
Limiting the Number of Aborted Decisions	549
Using ATPG Checkpoint Files	550
Creating Test Patterns for Diagnosing Scan Chain Failures	551
Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns	552
Generating Pattern Sets	553
Performing Scan Chain Diagnosis	554
Running Scan Chain Diagnosis	554
Understanding the Scan Chain Diagnosis Report	555
Diagnosing Defects Related to Power Issues	555
Creating End-of-Cycle Measures in ATPG Patterns	556
Drawbacks of Using End-of-Cycle Measures	557
Requirements Needed to Produce End-of-Cycle Measures	557
Deleting Top-Level Ports From Output Patterns	558
Detecting Faults Multiple Times Using N-Detect	558
WGL Pattern Generation Options	559
Creating LSI-Compatible WGL Patterns	560
Creating NEC-Compatible WGL Patterns	562

WGL Scan Chain Padding	563
WGL Scan Chain Definitions	564
Macro Usage in WGL	564
Grouping Bidirectional Port Data in WGL	566
Controlling Port Data Order in WGL	567
Specifying Windowed Measures in WGL	568
Delayed Input Force Timing and Force Prior in WGL	568
Balancing Vector and Scan Statements in WGL	569
Mapping Bidirectional Ports Within Vector Statements in WGL	570
Mapping Bidirectional Ports Within Scan Statements in WGL	574
Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL	575
Selecting Scan Chain Inversion Reference in WGL	576
Effect of CELLDEFINE in WGL	578
Ambiguity of the Master Cell in WGL	579
Running Multicore ATPG	580
Comparing Multicore ATPG and Distributed ATPG	580
Invoking Multicore ATPG	581
Typical Multicore ATPG Run	582
Multicore Interrupt Handling	582
Understanding the Processes Summary Report	583
Multicore Limitations	584
Running Logic Simulation	584
Comparing Simulated and Expected Values	585
Patterns in the Simulation Buffer	586
Sequential Simulation Data	586
Single-Point Failure Simulation	587
GSV Display of a Single-Point Failure	588
Data Volume and Test Application Time Reduction Calculations	588
Test Data Volume Calculations	589
Test Application Time Calculations	590
Pattern Porting	590
Pattern Porting Flow	591
Core-Level DFTMAX Insertion	592
Core-Level TestMAX ATPG Generation	592
Top-Level Requirements	593
Pattern Generation Requirements at Core Level	593
Top-Level Pattern Simulation	593

14. Fault Lists and Faults	594
Working with Fault Lists	594
Using Fault List Files	595
Collapsed and Uncollapsed Fault Lists	596
Random Fault Sampling	598
Fault Dictionary	599
Fault Categories and Classes	599
Fault Class Hierarchy	599
DT (Detected) = DR + DS + DI + D2 + TP	600
PT (Possibly Detected) = AP + NP + P0 + P1	601
UD (Undetectable) = UU + UO + UT + UB + UR	602
AU (ATPG Untestable) = AN	603
AE (ATPG Untestable) = AE	603
ND (Not Detected) = NC + NO	604
Fault Summary Reports	605
Fault Summary Report Examples	605
Test Coverage	607
Fault Coverage	608
ATPG Effectiveness	609
Using Clock Domain-Based Faults	610
Using Signals That Conflict With Reserved Keywords	613
Finding Particular Untested Faults Per Clock Domain	614

15. Fault Simulation	615
Supported Fault Models	615
Fault Simulation Design Flow	616
Preparing Functional Test Patterns for Fault Simulation	618
Pattern Compliance with ATE	618
Checking Patterns for Timing Insensitivity	619
Timing Sensitivity	619
Preparing Your Design for Fault Simulation	620
Preprocessing the Netlist	620
Reading the Design and Libraries	620
Building the ATPG Design Model	620
Declaring Clocks	621
Running DRC	621

DRC for Nonscan Operation	622
DRC for Scan Operation	624
Reading Functional Test Patterns	624
Using the Set Patterns Dialog Box	624
Using the set_patterns Command	625
Specifying Strobes for VCDE Pattern Input	625
Initializing the Fault List	627
Using the Add Faults Dialog Box	628
Using the add_faults Command	628
Performing Good Machine Simulation	628
Using the Run Simulation Dialog Box	629
Using the set_simulation and run_simulation Commands	629
Performing Fault Simulation	629
Using the Run Fault Simulation Dialog Box	630
Using the run_fault_sim Command	630
Writing the Fault List	631
Combining ATPG and Functional Test Patterns	631
Creating Independent Functional and ATPG Patterns	631
Creating ATPG Patterns After Functional Patterns	632
Creating Functional Patterns After ATPG Patterns	633
Using TestMAX ATPG with Z01X	635
Transition Fault Flow	636
Running Multicore Simulation	637
Invoking Multicore Simulation	638
Interrupt Handling	638
Processes Summary Report	638
Resimulating ATPG Patterns	640
Limitations	640
Per-Cycle Pattern Masking	641
Flow Options	641
Masks File	642
Running the Flow	642
Limitations	645
16. On-Chip Clocking Support	646
OCC Background	646

OCC Definitions, Supported Flows, Supported Patterns	647
OCC Limitations	648
TestMAX DFT to TestMAX ATPG Flow	650
OCC Support in TestMAX ATPG	653
Design Set Up	653
OCC Scan ATPG Flow	654
Waveform and Capture Cycle Example	654
Using Synchronized Multi Frequency Internal Clocks	655
Enabling Internal Clock Synchronization	655
Clock Chain Reordering	655
Clock Chain Resequencing	656
Finding Clock Chain Bit Requirements	658
Reporting Clocks	658
Reporting Patterns	660
Using Internal Clocking Procedures	661
Enabling Internal Clocking Procedures	661
Performing DRC with Internal Clocking Procedures	661
Reporting Clocks	662
Performing ATPG with Internal Clocking Procedures	663
Grouping Patterns By ClockingProcedure Blocks	663
Writing Patterns Grouped by Clocking Procedure	665
Reporting Patterns	665
Limitations	666
OCC-Specific DRC Rules	666
<hr/>	
17. TestMAX Diagnosis	668
Understanding Diagnosis	669
Diagnosis Reporting	669
Running Diagnosis	670
Using the Run Diagnosis Dialog Box	671
Using the run_diagnosis Command	671
Running the TestMAX Diagnosis Flow	671
Script Example	674
Writing and Reading Binary Image Files	674
Reading Pattern Files	675
Reading Patterns	675
Reading Multiple Pattern Files	676
Translating DFTMAX Compressed Patterns Into Normal Scan Patterns	676

Example Flow	677
Translation Limitations	678
Failure Data Files	678
Pattern-Based Failure Data File	679
Pattern-Based Failure Data File for DFTMAX Serialized Adaptive Scan	681
Cycle-Based Failure Data File	681
Cycle-Based Failure Data File Format	682
Failure Data File Extensions	683
Adding Header Information to a Failure Data File	685
Creating a Header Section	685
Creating a Header Schema File	686
Examples	687
Failure Data File Limitations	690
Class-Based Diagnosis Reporting	690
Filtering Candidates	691
Filtering Bridge Candidates	691
Resetting User-Specified Filters	692
Reporting Detailed Candidate Information	692
Example Flow	693
Understanding the Class-Based Diagnosis Report	694
Class-Based Cell-Aware Diagnosis	696
Fault-Based Diagnosis Reporting	697
Verbose Format	700
Physical Diagnosis Format	701
Scan Chain Diagnosis Format	703
Using a Dictionary for Diagnosis	705
Example Flow	706
Diagnosis Dictionary Commands	707
Limitations	708
Failure Mapping Report for DFTMAX Patterns	708
Composite Fault Model Data Report	709
Parallel Diagnosis	712
Specifying Parallel Diagnosis	712
Converting Serial Scripts to Parallel Scripts	713
Using Split Datalogs to Perform Parallel Diagnosis for Split Patterns	714
Diagnosis Log Files	715
Parallel Diagnosis Limitations	717

18. Using Physical Data for Diagnosis	718
Physical Diagnosis Flow Overview	718
Creating and Validating a PHDS Database	720
Reading a PHDS Database into TestMAX ATPG	722
Starting and Stopping the DAP Server Process	723
Setting Up a Connection to the PHDS Database	724
Name Matching Using a PHDS Database	725
Name Matching Overview	725
Understanding the Name Matching Coverage Report	726
Reporting the Name Matching Coverage	727
Using Name Matching Results for Diagnosis	728
Setting Up and Running Physical Diagnosis	729
Running Physical Diagnosis	730
Static Subnet Extraction Using a PHDS Database	731
Reporting Physical Subnet ID Data	732
Understanding Physical Subnet ID Data	733
Writing Physical Data for Yield Explorer	734

19. Power Aware ATPG	736
Input Data Requirements	737
Setting a Power Budget	737
Preparing Your Design	738
Reporting Clock-Gating Cells	738
Constraining Clock-Gating Cells for Power Aware ATPG	739
Setting a Strict Power Budget	740
Running Power Aware ATPG	740
Applying Quiet Chain Test Patterns	742
Testing with Asynchronous Primary Inputs	742
Power Reporting By Clock Domain	743
Setting a Capture Budget for Individual Clocks	749
Testing for Partitions	751
Specifying a Test Coverage Target for Partitions	751
Specifying Capture Power for Partitions	751
Specifying Shift Power for Partitions	751

Reporting Power Per Partition	751
Example	752
Limitations	753
Retention Cell Testing	753
Typical Retention Cell Used for Testing by TestMAX ATPG	754
Creating the chain_capture Procedure	755
Identifying Retention Cells for Testing	756
Pattern Generation for Retention Cells	756
Pattern Formatting for Retention Cells	757
Pattern Formatting by Masking Non-Retention Cells	758
Retention Cell Testing Limitations	760
Power Aware ATPG Limitations	760
<hr/>	
20. Bridging Fault ATPG	762
Bridging Fault ATPG Flow Overview	762
Running the Bridging Fault ATPG Flow	763
Setup	763
Input Faults	764
Manipulating the Fault List	764
Examining the Fault List	764
Fault Simulation	765
Running ATPG	765
Analysis	765
Example Script	766
Detecting Bridging Faults	766
Defining Bridging Faults	767
Bridge Locations	767
Strength-Based Patterns	768
Bridging Fault Model Limitations	769
Running the Dynamic Bridging Fault ATPG Flow	769
Dynamic Bridging Fault Model Introduction	770
Preparing to Run Dynamic Bridging Fault ATPG	770
Specifying a List of Input Faults	771
Manipulating the Fault List	771
Examining the Fault List	772
Fault Simulation	772
Running ATPG	773

	Analyzing Fault Detection	773
	Example Script	774
	Limitations	775
<hr/>		
21.	Cell-Aware Test	776
	Cell-Aware Test Flow	776
	Targeting Internal Cell Defects	778
	Cell Test Models	780
	Generating Cell Test Models	781
	Running Cell-Aware ATPG	782
	Example Script	784
	Running Cell-Aware Simulation	785
	Cell-Aware Diagnosis	786
	Identifying a Defect Within a Cell	787
	Running Cell-Aware Physical Diagnosis	789
<hr/>		
22.	Transition Delay Fault ATPG	791
	Using the Transition Delay Fault Model	791
	Transition Delay Fault ATPG Flow	792
	Typical Transition Delay Fault ATPG Run	794
	Transition Delay Fault ATPG Timing Modes	795
	Launch-On Shift Mode Versus System Clock Launch Mode	796
	Launch-On Extra Shift Timing	797
	STIL Protocol for Transition Faults	799
	Creating Transition Fault Waveform Tables	800
	DRC for Transition Faults	802
	Limitations of Transition Delay Fault ATPG	803
	Specifying Transition Delay Faults	803
	Selecting the Fault Model	804
	Adding Faults to the Fault List	804
	Reading a Fault List File	804
	Pattern Generation for Transition Delay Faults	805
	Using the set_atpg Command	805
	Using the set_delay Command	806
	Using the run_atpg Command	806
	Pattern Compression for Transition Faults	807

Using the report_faults Command	807
Using the write_faults Command	807
Pattern Formatting for Transition-Delay Faults	808
MUXClock Support for Transition Patterns	809
Specifying Timing Exceptions From an SDC File	810
Reading an SDC File	810
Interpreting an SDC File	811
How TestMAX ATPG Interprets SDC File Commands	811
Controlling Clock Timing, ATPG, and Timing Exceptions for SDC	812
Reporting SDC Results	813
Slack-Based Transition Fault Testing	813
Basic Usage Flow	814
Extracting Slack Data from PrimeTime	814
Utilizing Slack Data in the TestMAX ATPG Flow	814
Command Support	815
Special Elements of Slack-Based Transition Fault Testing	817
Allowing Variation From the Minimum-Slack Path	817
Defining Faults of Interest	818
Reporting Faults	818
Limitations	819
Engine and Flow Limitations	819
ATPG Limitations	819
Limitations in Support for Bus Drivers	819
23. Path Delay Fault and Hold Time Testing	820
Path Delay Fault Theory	820
Path Delay Fault Term Definitions	821
Models for Manufacturing Tests	823
Models for Characterization Tests	824
Testing I/O Paths	825
Path Delay Test Patterns	825
Path Delay Testing Flow	826
Obtaining Delay Paths	829
Hold Time ATPG Test Flow	829
Generating Path Delay Tests	831
Flow for Generating Path Delay Tests	832
Using set_delay Options	833

Reading and Reporting Path Lists	833
Analyzing Path Rule Violations	833
Viewing Delay Paths	833
Path Delay ATPG Options	834
Internal Loopback and False/Multicycle Paths	834
Creating At-Speed Waveform Tables	835
Maintaining At-Speed Waveform Table Information	838
MUXClock Support for Path Delay Patterns	838
Enabling MUXClock Functionality	839
Delay Test Vector Format	839
Limitations of MUXClock Support for Path Delay Patterns	841
ATPG Requirements to Support MUXClock	841
Handling Untested Paths	841
Understanding False Paths	842
Understanding Untestable Paths	842
Reporting Untestable Paths	843
Analyzing Untestable Faults	844
TestMAX ATPG Commands for Path Delay Fault Testing Example	844
24. Quiescence Test Pattern Generation	846
Why Do IDDQ Testing?	846
CMOS Circuit Characteristics	847
IDDQ Testing Methodology	848
Types of Defects Detected	849
Number of IDDQ Strokes	849
About IDDQ Pattern Generation	850
Fault Models	851
DRC Rule Violations	852
Generating IDDQ Test Patterns	853
IDDQ Test Pattern Generation Flow	854
Using the <code>iddq_capture</code> Procedure	854
Off-Chip IDDQ Monitor Support	855
Specifying Additional Signals in the Netlist	855
Defining the <code>iddq_capture</code> Procedure to Support Additional Signals	856
Using IDDQ Commands	861
Using the <code>set_faults</code> Command	861
Using the <code>set_iddq</code> Command	861

Using the add_atpg_constraints Command	862
IDDQ Bridging	863
Design Principles for IDDQ Testability	864
I/O Pads	864
Buses	865
RAMs and Analog Blocks	865
Free-Running Oscillators	866
Circuit Design	866
Power and Ground	866
Models With Switch/FET Primitives	866
Connections	867
IDDQ Design-for-Test Rule Summary	867
Additional System-on-a-Chip Rules	868
<hr/>	
25. Running Distributed ATPG	869
Debugging Name Matching Errors	870
Debugging Missing Instances	871
Debugging Hierarchical Mismatches	871
Checking Your Environment for Distributed Processing	873
Machine Access and Setup for Distributed ATPG	873
Preparing to Run Distributed Processing	874
Setting Up the Distributed Environment	875
Setting Up the Distributed Environment With Load Sharing	877
Verifying Your Environment	879
Remote Shell Considerations	879
Tuning Your .cshrc File	880
Checking the Load Sharing Setup	880
Starting Distributed ATPG	880
Saving Results	883
Distributed Processor Log Files	883
Starting Distributed Fault Simulation	884
Events After Starting A Distributed Run	885
Interpreting Distributed Fault Simulation Results	885
Debugging Distributed ATPG Issues	886
Distributed ATPG Limitations	888

26.	Persistent Fault Model Support	889
	Persistent Fault Model Overview	889
	Persistent Fault Model Operations	890
	Switching Fault Models	891
	Working With Internal Pattern Sets	891
	Manipulating Fault Lists	891
	Automatically Saving Fault Lists	892
	Automatically Restoring Fault Lists	892
	Removing Fault Lists	893
	Adding Faults	893
	Reporting Persistent Fault Models	894
	Direct Fault Crediting	895
	Example Commands Used in Persistent Fault Model Flow	898

27.	Using TestMAX ATPG and DFTMAX Ultra Compression	900
	Generating Patterns for DFTMAX Ultra Designs	900
	Pattern Types Required by DFTMAX Ultra	901
	Script Example for Generating Patterns for DFTMAX Ultra	901
	Manipulating Patterns for DFTMAX Ultra	902
	Controlling the Peak and Average Power During Shifting	902
	Increasing the Maximum Shift Length of Patterns	903
	Optimizing Padding Patterns	903
	Removing and Reordering Patterns	904
	High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnosis	905
	Identifying Defective Chains	906
	Generating High Resolution Patterns	906
	Rerunning Diagnosis	906
	Flow Example	906
	Test Validation and VCS Simulation for DFTMAX Ultra Designs	907
	Limitations for Using DFTMAX Ultra	907

28.	Troubleshooting	909
	Reporting Port Names	909
	Reviewing a Module Representation	910
	Rerunning Design Rule Checking	911
	Troubleshooting Netlists	912

Troubleshooting STIL Procedures	913
Opening the STL Procedure File	913
STIL load_unload Procedure	913
STIL Shift Procedure	914
STIL test_setup Macro	915
Correcting DRC Violations by Changing the Design	916
Analyzing the Cause of Low Test Coverage	916
Where Are the Faults Located?	917
Why Are the Faults Untestable or Difficult to Test?	918
Using Justification	919
Completing an Aborted Bus Analysis	920
Using Pipeline Guidance	920
Specifying the Head Pipeline Structures in the SPF	921
Using set_drc -pipeline_structures	922
<hr/>	
29. ATPG FAQ	923
What is the Difference Between Multicore Processing and Multithreading?	924
How Can I Avoid Generating Patterns With Floating BIDI Ports?	926
How Do I Abbreviate Commands?	926
Tcl Mode	926
Example	927
Native Mode	927
What Special Characters Are Used in Tcl Mode?	927
What Are Limited Regular Expressions?	928
Regular Expression Meta-Characters	928
Usage Notes:	929
Examples	929
Using Escape Characters With Wildcards and Regular Expressions	930
What are the Compressor Connections in report_scan_chains Output?	930
What are Some Examples of Pin Data?	940
Bidi Control Value	940
Clock Cone	940
Clock On	941
Clock Off	941
Constraint Data	942
Debug Sim Data	943

Contents

Delay Data	943
Error Data	943
Fault Data	943
Fault Sim Result	944
FULL_SEQ_Scoap_data	945
Full Sequential TG Data	945
Good Sim Results	946
Load	946
Master Observe	947
None	947
Pattern	947
SCOAP Data	948
SDC Case Analysis	949
Seq Sim Data	949
Shadow Observe	951
Shift	951
Stability Patterns	952
Test Setup	952
Tie Data	953
How Do I Use the write_testbench Command to Customize MAX Testbench Output?	953
Example	954
Validating Simulation Libraries Used For ATPG	956
How Do I Customize Ltran Output for FTDL, TSTL2, or TDL91?	959
Customizing Ltran Configuration Files	959
Customizing Simulator Format-Specific Controls	960
Common Ltran Controls	960
Character Padding	960
How TestMAX ATPG Processes Setup and Hold Violations	961
Example of How TestMAX ATPG Handles an Ambiguous Case	961
Interpreting UDP Messages	962
Variant #1	962
Variant #2	965
Variant #3	968
Variant #4	970
Debugging UDP-based Models	971
What is the Difference between the add_capture_masks vs add_cell_constraints Commands?	972

Contents

Masking a Scan Cell by Instance Name	972
Masking a Nonscan Cell by Instance Name	973
JTAG Support	973
Common Tasks for Supporting JTAG	973
Initializing TAP Using test_setup	974
Keeping the TAP Controller from Changing State	975
When to Constrain TMS	976
Controlling TAP using load_unload	976
Accessing Internal Scan Chains Through the TAP	977
Limiting Clocks during ATPG	977
TAP controllers with no reset pin	978
Node File Format for Bridging Faults	978
Star-RCXT Format	978
Node File Format	979
Optimizing Basic Scan Patterns	981
Design and ATPG Usage Tips for Designs with Phase Lock Loops (PLLs)	983
Design Considerations:	983
ATPG Tool Considerations:	983
Shared Scan-In Designs	984
Creating End-of-Cycle Measures in ATPG Patterns	985
Drawbacks of Using End-of-Cycle Measures	986
Requirements Needed to Produce End-of-Cycle Measures	986
Troubleshooting Pattern Simulation Failures	987
Your ATPG Patterns are Failing: What Next?	988
Interpreting the Simulation Failure Messages	991
Isolating a Failing Pattern to Assist in Troubleshooting	994
Eliminating a Few Failing Patterns from a Larger Set	995
Locating the Target Fault Site for the Failing Pattern	995
Isolating a Fault List to Assist in Troubleshooting	997
Interpreting the report_patterns Command	998
Viewing Pattern Data in the Graphical Schematic Viewer	999
Using the analyze_simulation_data Command	1000
WGL Pattern Generation Options	1001
Creating LSI-Compatible WGL Patterns	1002
Creating NEC-Compatible WGL Patterns	1004
WGL Scan Chain Padding	1005
WGL Scan Chain Definitions	1006

Contents

Macro Usage in WGL	1007
Grouping Bidirectional Port Data in WGL	1009
Controlling Port Data Order in WGL	1010
Specifying Windowed Measures in WGL	1011
Delayed Input Force Timing and Force Prior in WGL	1011
Balancing Vector and Scan Statements in WGL	1012
Mapping Bidirectional Ports Within Vector Statements in WGL	1014
Mapping Bidirectional Ports Within Scan Statements in WGL	1017
Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL	1018
Selecting Scan Chain Inversion Reference in WGL	1019
Effect of CELLDEFINE in WGL	1021
Ambiguity of the Master Cell in WGL	1022
Subnet Formats for Diagnosis	1023
Handling Escape Characters in Tcl Mode	1026
Passing Complex Options to LSF/GRID	1027
<hr/>	
30. Scripts	1028
Basic ATPG Run Script	1029
Basic TestMAX ATPG Run	1030
ATPG Run No SPF	1032
Bridging Fault ATPG	1033
Cell-Aware ATPG	1034
Dynamic Bridging Fault ATPG	1036
Low Power ATPG	1038
Multicore ATPG	1039
Scan-Through-TAP ATPG Flow	1041
Transition Delay Fault ATPG	1042
Transition Delay Fault ATPG Using LOES Timing	1043
Basic TestMAX ATPG Diagnosis Run	1045
Distributed Processing Fault Simulation Flow	1047
DFTMAX What-If Analysis	1048
DFTMAX Ultra High Resolution Pattern Flow	1048
Fault Coverage of Combined ATPG and JTAG Test Vectors	1049
Generating Patterns for DFTMAX Ultra	1050

IDDQ Bridging Flow	1051
Slack-Based Testing	1053

31. Validating Test Patterns	1055
TestMAX ATPG Pattern Format Overview	1055
Writing STIL Patterns	1056
Design to Test Validation Flow	1058

32. Using MAX Testbench	1060
Overview	1060
Installation	1061
Obtaining Help	1061
Running MAX Testbench	1061
write_testbench Command Syntax	1063
MAX Testbench Command-Line Parameters Used With the write_testbench Command	1065
stil2Verilog Command Syntax	1071
Setting the Run Mode	1077
Configuring MAX Testbench	1078
Understanding the Failures File	1086
MAX Testbench and Legacy Scan Failures	1086
MAX Testbench and DFTMAX Compression Failures	1087
MAX Testbench and Serializer Scan Failures	1089
Using the Failures File	1090
Using Split STIL Pattern Files	1094
Execution Flow for -split_in Option	1095
Splitting Large STIL Files	1095
Why Split Large STIL Files?	1096
Executing the Partition Process	1096
Example Test	1097
Controlling the Timing of a Parallel Check/Assert Event	1098
Using MAX Testbench to Report Failing Scan Cells	1102
Flow Overview	1103
Flow Example	1105
MAX Testbench Runtime Programmability	1108
Basic Runtime Programmability Simulation Flow	1108
Runtime Programmability for Patterns	1109

Contents

Example: Using Runtime Predefined VCS Options	1112
MAX Testbench Runtime Programmability Limitations	1113
MAX Testbench Support for IDDQ Testing	1114
Compile-Time Options for IDDQ	1114
IDDQ Configuration File Settings	1115
Generating a VCS Simulation Script	1116
How MAX Testbench Works	1117
Predefined Verilog Options	1118
MAX Testbench Limitations	1121
Example of the Configuration Template	1121
MAX Testbench Error Messages and Warnings	1124
Error Message Descriptions	1125
Warning Message Descriptions	1131
Informational Message Descriptions	1148
Troubleshooting MAX Testbench	1150
Introduction	1150
Troubleshooting Compilation Errors	1151
FILELENGTH Parameter	1151
NAMELENGTH Parameter	1152
Memory Allocation	1152
MDEPTH Parameter	1152
Troubleshooting Miscompares	1153
Handling Miscompare Messages	1153
Understanding MAX Testbench Parallel Miscompares	1156
Localizing a Failure Location	1157
Adding More Fingerprints	1159
Debugging Simulation Mismatches Using the write_simtrace Command	1160
Overview	1160
Debugging Flow	1161
Input Requirements	1162
Using the write_simtrace Command	1163
Understanding the Simtrace File	1163
Error Conditions and Messages	1164
Example Debug Flow	1164
Restrictions and Limitations	1167
Debugging Parallel Simulation Failures Using Combined Pattern Validation	1167
Overview	1168
Understanding the PSD File	1169
Creating a PSD File	1171
Using the run_atpg Command to Create a PSD File	1172
Using the run_simulation Command to Create a PSD File	1173

Flow Configuration Options	1175
Example Simulation Miscompare Messages	1175
Displaying the Instance Names of Failing Cells	1179
Debug Modes for Simulation Miscompare Messages	1180
Pattern Splitting	1182
Splitting Patterns Using TestMAX ATPG	1182
Splitting Patterns Using MAX Testbench	1186
Specifying a Range of Split Patterns Using MAX Testbench	1188
MAX Testbench and Consistency Checking	1188
Using the PSD File with DFTMAX Ultra Compression	1189
Script Example	1190
Limitations for Debugging Simulation Failures Using CPV	1190
<hr/>	
33. Using Loadable Nonscan Cells in TestMAX ATPG	1192
Simulation Support	1192
ATPG Support	1193
Multithreading ATPG	1193
Fault Simulation Support	1193
Reporting Loadable Nonscan Cells	1193
Analyzing	1195
Limitations	1196
<hr/>	
34. PowerFault	1198
PowerFault Simulation	1198
PowerFault Simulation Technology	1198
IDDQ Testing Flows	1200
IDDQ Test Pattern Generation	1201
IDDQ Strobe Selection From an Existing Pattern Set	1202
Licensing	1202
Verilog Simulation with PowerFault	1203
Preparing Simulators for PowerFault IDDQ	1203
Using PowerFault IDDQ With Synopsys VCS	1204
Using PowerFault IDDQ With Cadence NC-Verilog	1205
Using PowerFault IDDQ With Cadence Verilog-XL	1209
Using PowerFault IDDQ With Model Technology ModelSim	1211
PowerFault PLI Tasks	1213
Getting Started	1213

Contents

PLI Task Command Summary Table	1214
PLI Task Command Reference	1216
Faults and Fault Seeding	1240
Fault Models	1241
Fault Models in TestMAX ATPG	1241
Fault Models in PowerFault	1241
Fault Seeding	1242
Seeding From a TestMAX ATPG Fault List	1242
Seeding From an External Fault List	1244
PowerFault-Generated Seeding	1244
Options for PowerFault-Generated Seeding	1245
Stuck-At Fault Model Options	1245
Bridging Faults	1253
PowerFault Strobe Selection	1257
Overview of IDDQPro	1258
Invoking IDDQPro	1259
ipro Command Syntax	1259
Strobe Selection Options	1260
Report Configuration Options	1262
Log File and Interactive Options	1265
Interactive Strobe Selection	1265
cd	1267
desel	1267
exec	1268
help	1268
ls	1268
prc	1268
prf	1268
prs	1269
quit	1269
reset	1269
sela	1269
selm	1269
selall	1270
Strobe Selection Tutorial	1270
Simulation and Strobe Selection	1271
Interactive Strobe Selection	1274
Understanding the Strobe Report	1279
Example Strobe Report	1279
Fault Coverage Calculation	1280
Adding More Strobes	1281
Deleting Low-Coverage Strobes	1281

Contents

Fault Report Formats	1282
TestMAX ATPG Fault Report Format	1282
Verifault Fault Report Format	1283
Listing Seeded Faults	1283
Verifault Interface	1283
Iterative Simulation	1285
Using PowerFault Technology	1286
PowerFault Verification and Strobe Selection	1287
Verifying TestMAX ATPG IDDQ Patterns for Quiescence	1287
Selecting Strobes in TestMAX ATPG Stuck-At Patterns	1288
Selecting Strobe Points in Externally Generated Patterns	1289
Testbenches for IDDQ Testability	1290
Separate the Testbench From the Device Under Test	1290
Drive All Input Pins to 0 or 1	1290
Try Strobes After Scan Chain Loading	1291
Include a CMOS Gate in the Testbench for Bidirectional Pins	1291
Model the Load Board	1291
Mark the I/O Pins	1291
Minimize High-Current States	1291
Maximize Circuit Activity	1292
Combining Multiple Verilog Simulations	1292
Improving Fault Coverage	1294
Determine Why the Chip Is Leaky	1294
Evaluate Solutions	1295
Floating Nodes and Drive Contention	1300
Floating Node Recognition	1300
Drive Contention Recognition	1303
Status Command Output	1304
Status Command Overview	1304
Leaky Reasons	1305
Nonleaky Reasons	1307
Driver Information	1308
Behavioral and External Models	1309
Disallowing Specific States	1309
Disallowing Global States	1309
Multiple Power Rails	1310
Testing I/O and Core Logic Separately	1314
35. Types of Reports	1316
Output From the report_scan_ability Command	1317

Contents

Standard Format	1317
Output From the report_scan_cells Command	1318
Standard Format	1318
-Pin Format	1319
Verbose Format	1320
Output From the report_scan_chains Command	1322
Standard Format	1322
Verbose Format	1322
Output From the report_scan_path Command	1323
Standard Format	1323
Verbose Format	1324
Output From the report_settings Command	1325
Standard Format	1326
Output From the report_summaries Command	1328
Standard Format	1328
Verbose Format	1328
Primitives Report	1329
Library Cells Report	1330
Optimizations Report	1330
Sequential Depths Report	1331
Output From the report_version Command	1332
Standard Format	1332
Full Format	1332
Short Format	1332
Address Format	1332
Banner Format	1333
Verbose Format	1333
Output From the report_violations Command	1333
Standard Format	1333
Output From the report_wires Command	1334
Summary Format	1334
Standard Format	1334
Verbose Format	1335
Output From the analyze_buses Command	1336
Standard Format - defaults	1336
Standard Format - zstate	1336

Contents

Standard Format - exclusive	1336
Standard Format - prevention	1337
Output From the analyze_faults Command	1337
Standard Format for Blocked pin_pathname	1338
Standard Format for Successful pin_pathname	1338
Standard Format for Class	1339
Blockage and Constraint Value Source Points Format	1341
Verbose Format	1342
Output From the report_atpg_constraints Command	1343
Summary Format	1343
Standard Format	1344
Output From the report_atpg_primitives Command	1344
Summary Format	1345
Standard Format	1345
Verbose Format	1345
Output From the report_buses Command	1346
Summary Format	1346
Standard Format	1347
Verbose Format	1348
Output From the report_cell_constraints Command	1348
Standard Format	1348
Output From the report_clocks Command	1349
Standard Format	1349
Matrix Format	1350
Internal Clocks Format	1352
Verbose Format	1353
Output From the report_commands Command	1354
Summary Format	1354
Standard Format	1354
Usage Format	1354
Output From the report_memory Command	1355
SUMMARY FORMAT	1355
Standard Format	1356
Verbose Format	1356
Standard Format With Constants	1357
Output From the report_modules Command	1358

Contents

Summary Format	1358
Standard Format	1359
Verbose Format	1359
Output From the report_net_connections Command	1360
Standard Format	1360
Output From the report_nets Command	1361
Standard Format	1361
Output From the report_nofaults Command	1362
Standard Format	1362
Output From the report_nonscan_cells Command	1362
Summary Format	1363
Standard Format	1363
Output From the report_patterns Command	1364
Summary Format	1364
Standard Format	1365
Pattern Type Format	1366
Understanding the Cycle Count for Designs with OCCs	1368
Output From the report_pi_constraints Command	1369
Standard Format	1369
Output From the report_pi_equivalences Command	1369
Standard Format	1369
Output From the report_po_masks Command	1370
Standard Format	1370
Output From the report_primitives Command	1370
Summary Format	1371
Standard Format	1372
Verbose Format	1373
Pin/PI/PO/PIO Format	1373
Output From the report_rules Command	1374
Standard Format	1374
Output From the run_build_model Command	1375
Standard Format	1375
Verbose Format	1376
Output From the run_fault_sim Command	1379
Standard Format	1379

Contents

Output From the run_justification Command	1381
Standard Format (Examples)	1381
Output From the run_simulation Command	1382
Standard Format	1382
With Simulation Mismatches	1383
Using the -max_fails Option	1384
Using the -progress_message Option of the set_simulation Command	1385

36. Glossary	1387
At-speed Clock	1389
ATPG Primitive ID	1389
ATPG Primitive Name	1390
Black Box	1391
Bus Keeper	1391
Capture Clock	1391
Capture Clock Edge (Capture Edge)	1392
Capture Vector	1392
Circuit Path	1392
Clock	1393
Clock Cone	1393
Comment Lines	1393
Repeating Commands	1393
Continuation Character	1394
Delay Path	1394
Effect Cone	1395
Empty Box	1395
False Path	1395
Fanin Number	1395
backward	1396
Fanout Number	1397
forward	1398
report_primitives	1399
Primitive ID	1404

Contents

Gray Box	1405
Head of the Path	1405
How to Copy and Paste	1405
Cut/Paste between X11 window and TestMAX ATPG GUI window	1405
Instance Name	1406
Launch Clock	1406
Launch Clock Edge (Launch Edge)	1406
Feedback Path ID	1407
Majority Gate	1407
Measure Scan Chain Output	1407
Modifying Timing Data in an Existing STL Procedures File	1407
Module Name	1408
Module Pin Name	1408
Net Name	1409
Non-robust Detection of a Path Delay Fault	1409
Non-robust Test (For a Path Delay Fault)	1410
Nonscan Behavior: C0	1410
Nonscan Behavior: C1	1410
Nonscan Behavior: CU	1410
Nonscan Behavior: L0	1411
Nonscan Behavior: L1	1411
Nonscan Behavior: LE	1411
Nonscan Behavior: LS	1411
Nonscan Behavior: RAM_out	1411
Nonscan Behavior: TE	1411
Nonscan Behavior: TLA	1412
Null Module	1412
Off-path Input	1412
Off State	1412
On-path Input	1413
Output Redirection	1413
Path Delay Fault	1414

Contents

Pin Pathname	1414
Port Name	1414
Primitive ID	1415
Sequential Model Port Priorities	1416
Reconverging Path	1416
Robust Detection of a Path Delay Fault	1416
Robust Test (For a Path Delay Fault)	1416
Scan Clock	1417
SCOAP	1417
Setup Vector (Launch Vector)	1417
Shift Position	1418
Simulation Events	1418
Tail of the Path	1418
Test For A Path Delay Fault	1419
Unstable Set / Resets	1419
WFCMap	1419
Ungated Circuitry	1419

37. Limitations

A. Test Concepts	1425
Why Perform Manufacturing Testing?	1425
Understanding Fault Models	1426
Stuck-At Fault Models	1426
Detecting Stuck-At Faults	1427
Transition Delay Fault Models	1429
Detecting Transition Delay Faults	1429
Using Fault Models to Determine Test Coverage	1430
IDDQ Fault Model	1430
Fault Simulation	1431
Automatic Test Pattern Generation	1431
Translation for the Manufacturing Test Environment	1432
Coverage Calculations	1432
Test Coverage	1433

Fault Coverage	1433
ATPG Effectiveness	1433
Internal Scan	1433
Example	1434
Applying Test Patterns	1435
Scan Design Requirements	1436
Controllability of Sequential Cells	1436
Observability of Sequential Cells	1437
Full-Scan Design	1437
Partial-Scan ATPG Design	1438
What Is Boundary Scan?	1438

B. ATPG Design Guidelines	1441
ATPG Design Guidelines	1441
Internally Generated Pulsed Signals	1442
Clock Control	1445
Pulsed Signals to Sequential Devices	1447
Multidriver Nets	1448
Bidirectional Port Controls	1450
Exception	1451
Clocking Scan Chains: Clock Sources, Trees, and Edges	1451
Clock Trees	1452
Clock Flip-Flops	1453
XNOR Clock Inversion and Clock Trees	1455
Protection of RAMs During Scan Shifting	1456
RAM and ROM Controllability During ATPG	1456
Pulsed Signal to RAMs and ROMs	1458
Bus Keepers	1458
Non-Z State on a Multidriver Net	1459
Non-Clocked Events	1460
Bus Keepers	1461
Non-Z State on a Multidriver Net	1462
Non-Clocked Events	1462
Checklists for Quick Reference	1463
ATPG Design Guideline Checklist	1463
Ports for Test I/O Checklist	1464

C. Importing Designs From TestMAX DFT	1466
--	-------------

D. Utilities	1468
Ltran Translation Utility	1468
Ltran in the Shell Mode	1469
FTDL, TDL91, and TSTL2 Configuration Files	1470
Understanding the Configuration File	1471
Customizing the FTDL Configuration File	1472
Customizing the TDL91 Configuration File	1472
Customizing the TSTL2 Configuration File	1474
Additional Controls	1474
Configuration File Syntax	1475
OVF_BLOCK Statements	1475
PROC_BLOCK Statements	1476
TVF_BLOCK Statements	1479
Generating PrimeTime Constraints	1480
Input Requirements	1481
Starting the Tcl Command Parser Mode	1481
Setting Up TestMAX ATPG	1482
Making Adjustments for OCC Controllers	1485
Performing an Analysis for Each Mode	1486
Implementation	1488
Converting Timing Violations Into Timing Exceptions	1490
Importing PrimeTime Path Lists	1492
Path Definition Syntax	1496
stilgen Utility and Configuration Files	1498
Using stilgen for Pattern Porting	1498
stilgen Configuration File Syntax for Pattern Porting	1499
Port-Mapping File Syntax	1500
Using stilgen for Protocol Generation	1500
stilgen Configuration File Syntax for Protocol Generation	1501
Pattern Porting Example	1504
Protocol Generation Notes	1505
Supported Configurations	1506
Limitations	1509

E. STIL Language Support	1510
STIL Overview	1510
IEEE Std. 1450-1999	1511

IEEE Std. 1450.1 Design Extensions to STIL	1511
TestMAX ATPG and STIL	1512
STIL Conventions in TestMAX ATPG	1512
Use of STIL Procedures	1513
Context of Partial Signal Sets in Procedure Definitions	1513
Use of STIL SignalGroups	1514
WaveFormCharacter Interpretation	1515
IEEE Std. 1450.1 Extensions Used in TestMAX ATPG	1516
Vector Data Mapping Using \m	1516
Syntax	1518
Example	1519
Vector Data Mapping Using \j	1519
Syntax	1520
General Example	1520
Usage Example	1521
Signal Constraints Using Fixed and Equivalent	1523
ScanStructures Block	1524
Elements of STIL Not Used by TestMAX ATPG	1524
TestMAX ATPG STIL Output	1525
TestMAX ATPG STIL Input	1527
Testing the STIL Procedure File	1527
<hr/>	
F. STIL99 Versus STIL	1529
<hr/>	
G. Defective Chain Masking for DFTMAX	1540
Introduction	1540
Running the Flow	1541
Placing Constraints on the Defective Chain	1541
Generating Patterns	1542
Regenerating Patterns	1542
Examples	1542
Limitation	1544
<hr/>	
H. Simulation Debug Using MAX Testbench and Verdi	1545
Setting the Environment	1545

Contents

Preparing MAX Testbench	1546
Linking Novas Object Files to the Simulation Executable	1547
Running VCS and Dumping an FSDB File	1547
Running Verdi	1547
Debugging MAX Testbench and VCS	1548
Changing Radix to ASCII	1549
Displaying the Current Pattern Number	1549
Displaying the Vector Count	1550
Using Search in the Signal List	1551

About This User Guide

The *TestMAX ATPG and TestMAX Diagnosis User Guide* describes the usage and methodology for TestMAX ATPG and TestMAX Diagnosis. Both products are used to check testability design rules and to automatically generate manufacturing test vectors for a logic design.

This manual provides background material on design-for-test (DFT) concepts, especially test terminology and scan design techniques. You can obtain more information on TestMAX ATPG and TestMAX ATPG Diagnosis features and commands by accessing TestMAX ATPG and Diagnosis Online Help.

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 DFT engineers who incorporate the test vectors produced by TestMAX ATPG and TestMAX Diagnosis into test programs for a particular tester or who work with DFT netlists. Engineers involved in the testing and diagnostics of manufactured parts also find this manual useful.

This preface includes the following sections:

New in This Release

Information about new features, enhancements, and changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the TestMAX ATPG Release Notes on the SolvNetPlus site.

Related Products, Publications, and Trademarks

For additional information about the ATPG tool, see the documentation on the Synopsys SolvNetPlus support site at the following address:

<https://solvnetplus.synopsys.com>

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

- <list_of_related_products_with_trademarks>

1

TestMAX ATPG and TestMAX Diagnosis Overview

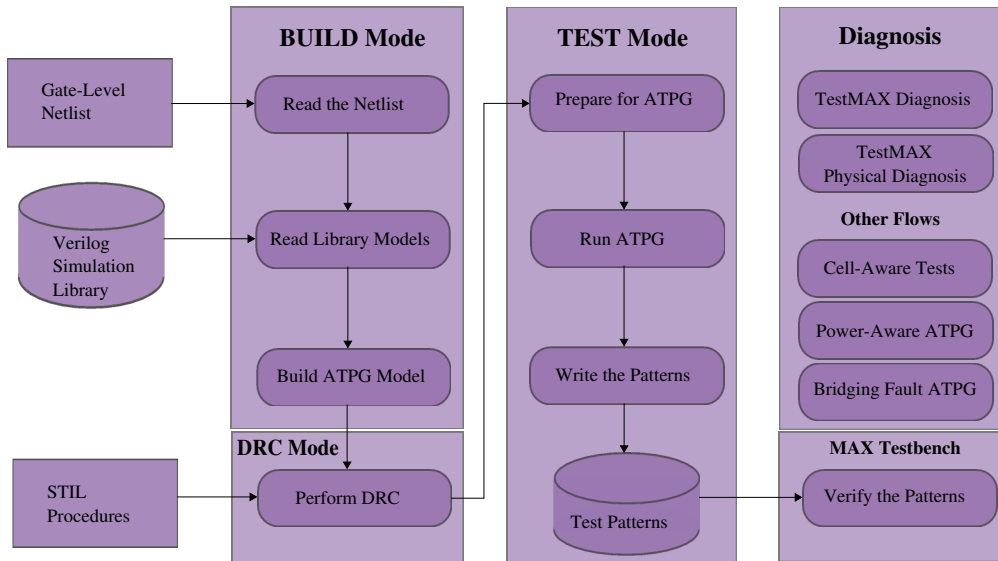
TestMAX ATPG and TestMAX Diagnosis is part of the Synopsys TestMAX suite of tools. Key components include advanced compression, diagnosis, and fault simulation engines that are fast, memory-efficient, and optimized for fine-grained multithreading of ATPG and diagnosis processes across multiple cores. TestMAX ATPG and TestMAX Diagnosis achieve higher diagnostics throughput by transparently running multiple simulations.

TestMAX ATPG and TestMAX Diagnosis use multithreading for most processes. Multithreading provides virtually unlimited memory usage for its core engines. This enables the full use of multiple cores and results in extremely fast runtime. For more information on multithreading, see [Multithreading in TestMAX ATPG and TestMAX Diagnosis](#).

By default, TestMAX ATPG and TestMAX Diagnosis use eight threads. For information on how to set the thread count for multithreading, see [Setting the Thread Count in TestMAX ATPG](#).

If a particular command option or process does not support multithreading, non-threaded processing is used instead. For a description of all command options that have limited multithreading support, see [TestMAX ATPG Multithreading Command Option Support](#). For a description of processes that have limited multithreading support, see [Multithreading Limitations](#).

Figure 1



Launching TestMAX ATPG and TestMAX Diagnosis

To start TestMAX ATPG and TestMAX Diagnosis, specify the `tmax2` executable located in the `$SYNOPTSYS/bin/` directory. You can set up the same environment and include the same options and input used by the `tmax` executable, as shown in the following steps:

1. Set your environment.

```
setenv SYNOPTSYS /synopsys/m_branch/ set path =($SYNOPTSYS/bin $path)
setenv SNPSLMD_LICENSE_FILE synopsys_licenses/my_license.lic
```

2. Do one of the following:

- Launch TestMAX ATPG and TestMAX Diagnosis in shell mode using the `-shell` option with the `tmax2` command.
`tmax2 my_command_file -shell`
- Launch the TestMAX GUI by specifying the `tmax2` command without the `-shell` option.
`tmax2 my_command_file`

The following table summarizes the various methods you can use to invoke TestMAX ATPG.

Invoke With ¹	You Get	Process List
<code>tmax2</code>	64-bit kernel plus GUI	<code>tmax2</code> , <code>tmax64</code> , <code>tmaxgui</code> , sometimes MAX Testbench (using the <code>write_testbench</code> command)
<code>tmax2 -shell</code>	64-bit kernel	<code>tmax2</code> , <code>tmax64</code> , sometimes MAX Testbench (using the <code>write_testbench</code> command)
<code>tmax2 -man</code>	Online help	<code>tmax2</code> , HTML browser
<code>tmax2 -help</code>	List of options	

¹ - The order of switches is not important.

² - The `tmax2` process invokes the shell. The CPU is idle after the kernel launches, but remains open so that the kernel has a transcript window in which to display output.

³ - The `tmax2` and `tmax2 -shell` commands sometimes invoke the `stil2verilog` command.

For a complete list of options associated with TestMAX ATPG, see "tmax and tmax2 Command Syntax."

Setting the Thread Count in TestMAX ATPG

When TestMAX ATPG starts, it uses eight threads by default.

To change the default values, use the `-num_threads` option with the `set_atpg` command and the `set_simulation` command. Make sure to specify a positive integer, and always use the same number of threads for both commands. Also, make sure to specify this option before the `run_drc` or `read_image` commands.

```
set_atpg -num_threads 12
set_simulation -num_threads 12
```

There is no limit to the number of cores you can use. However, since performance is not optimized for more than 20 cores, you can use this as a limit.

You can display the number of threads currently in use for ATPG or simulation using the `report_settings` command. If you specify the `report_settings atpg` command or the `report_settings simulation` command, the following message is displayed:

```
num_threads=8
```

Look for messages similar to the following example after the first time you specify the `run_atpg`, `run_diagnosis`,

`run_fault_sim`, or the `run_simulation` commands:

```
Parallel simulation data created for 16 threads  
#TLA_input_gates=2312 #levels=1 CPU_time=0.00 sec.  
Parallel simulation data created for 16 threads  
#internal_gates=1876366 #levels=6 CPU_time=1.02 sec.
```

The numbers will differ depending on run-specific details. The essential information is that threads are being used, as shown in bold in the previous example (for 16 threads).

Image files written after changing the `-num_threads` settings cannot be read into TestMAX ATPG.

Multicore settings for TestMAX ATPG are not used for commands that use threads. They are used for commands that are not supported with threads. Multicore settings are saved in image files and can be disabled after they are read by setting the `-num_processes` option to 0 in the `set_atpg` and `set_simulation` commands, as shown in the following example:

```
set_atpg -num_processes 0  
set_simulation -num_processes 0
```

Multithreading in TestMAX ATPG and TestMAX Diagnosis

TestMAX ATPG and TestMAX Diagnosis is built upon the use of multithreading. This technique concurrently executes small, multiple tasks or threads based upon instructions from a single central controlling scheduler.

A *thread* is the flow of execution through a single process. Each thread is comprised of its own program counter to track instructions, system registers to store its current working variables, and a stack which contains the execution history.

Some of the key benefits of multithreading include:

- *Improved Responsiveness*

Each activity is defined as a thread, which can be replicated as many times as required.

- *Efficient Use of Multiprocessors*

Because performance improves transparently with additional processors, there is no need to account for the number of available processors. Numerical algorithms and the use of parallelism run much faster when implemented with threads on a multiprocessor.

- *Efficient and Adaptive Program Structure*

Multithreaded architecture is more efficiently structured as a series of multiple independent execution units rather than a single, monolithic thread.

- *Minimizes the Use of System Resources*

The use of two or more processes that access common data through shared memory usually requires more than one thread of control. But in multithreaded architecture, each process has a full address space and operating environment state, which minimizes overall system resources.

For more information on multithreading, see the "What is the Difference Between Multicore Processing and Multithreading?" topic in TestMAX ATPG and TestMAX Diagnosis Online Help.

Note: TestMAX ATPG and TestMAX Diagnosis do not use multithreading in all processes. For more information, see [TestMAX ATPG Multithreading Command Option Support](#) and [Multithreading Limitations](#).

Multithreading Limitations

The following multithreading limitations apply to features of the design or protocol, but not limitations of command options. These limitations apply to the P-2019.03 release and are subject to change in future releases. For command option limitations, see [TestMAX Command Option Support](#). The following multithreading limitations apply to features of the design or protocol, but not limitations of command options. These limitations apply to the -P-2019.03- release and are subject to change in future releases. For command option limitations, see [TestMAX Command Option Support](#).

- *DFTMAX LogicBist*

When using TestMAX DFTMAX LogicBIST, an M729 message is printed and non-threaded processing is run instead of multithreading.

- *Launch on Last Shift*

TestMAX ATPG multithreading does not generate patterns if you specify the `set_delay -launch_cycle last_shift` command. If you specify this command, an M729 message is printed and a non-threaded process is run instead. TestMAX ATPG supports Launch on Extra Shift (LOES) if you specify the `set_delay -launch_cycle extra_shift` command.

- *Internal and Synchronized Multi-Frequency Clocking Procedures*

There is partial multithreading support for both internal and synchronized multi-frequency clocking procedures:

- Internal clocking procedures are supported for two-clock transition ATPG. Fast-sequential ATPG that requires more than two frames and pattern grouping is not supported.
- Synchronized multi-frequency clocking (or synchronized OCC) is supported for two-clock transition ATPG with integer clock period ratios. Fast-sequential ATPG that requires more than two clock cycles is not supported. Non-integer clock period ratios and multi-frame paths are not supported.
- *Fault Models that Switch to Non-Threaded Processing:*

The following fault models are not supported in multithreading:

 - Hold Time
- *Full-sequential ATPG and Full-Sequential Simulation*

Full-sequential ATPG and full-sequential simulation cannot be run using multithreading. If you attempt to do so, an M729 message is printed and non-threaded processing is run instead.
- *TestMAX ATPG Homogeneous Product Usage*

TestMAX ATPG is available as an 8-core product (supports up to 8 threads per seat) and a 20-core product. You cannot use both 8-core and 20-core TestMAX ATPG products in the same pattern generation session.

TestMAX ATPG Multithreading Command Option Support

Some options in TestMAX are limited in their support of multithreading (described in the [Multithreading in TestMAX ATPG and TestMAX Diagnosis](#) topic). In these cases, TestMAX ATPG uses either non-threaded processing or ignores the option.

This topic describes how TestMAX ATPG handles various command options impacted by the limitations of multithreading. These limitations apply to the P-2019.03 release and related service pack releases and are subject to change in future releases.

For a list of general feature limitations specific to TestMAX ATPG multithreading, see the [TestMAX ATPG Multithreading Limitations](#) topic.

TestMAX ATPG multithreading command are:

- [run_atpg](#)
- [run_fault_sim](#)

- [run_simulation](#)
- [set_atpg](#)
- [set_delay](#)
- [set_drc](#)

run_atpg

The following `run_atpg` command options switch to non-threaded processing:

- `full_sequential_only`
- `-distributed`
- `-random`
- `-resolve_differences`

These options are ignored by TestMAX ATPG:

- `-auto_compression` (the only effect of this option is to add all faults if the fault list is empty)
- `-optimize_patterns`
- `-rerun_for_timing_exceptions`

run_fault_sim

The following `run_fault_sim` command options switch to non-threaded processing:

- `-checkpoint`
- `-detected_pattern_storage`
- `-distributed`
- `-nodrop_faults`
- `-sequential`
- `-store`

This option is ignored by TestMAX ATPG fault simulation:

- `-strong_bridge`

run_simulation

The following `run_simulation` command options switch to non-threaded processing:

- `-fast`
- `-sequential`
- `-sequential_update`
- `-update`

set_atpg

The following `set_atpg` command options are silently ignored by multithreaded TestMAX ATPG:

- `-capture_cycles` with a setting of 0 (two-cycle patterns can be generated)
- `-shared_io_analysis`

The following option has a different effect when running multithreaded TestMAX ATPG than non-threaded processing. In this case, ATPG uses multithreading, but resimulation uses non-threaded processing simulation:

- `-resim_atpg_patterns`

These options switch to non-threaded processing:

- `-allow_clockon_measures`
- `-fast_path_delay`

These options are ignored by multithreaded TestMAX ATPG:

- `-abort_limit` (values up to 100 are used, but values greater than 100 are ignored)
- `-calculate_power`
- `-decision`
- `-fast_min_detects_per_pattern` (use the `-basic_min_detects_per_pattern` option instead)
- `-full_*` (all full sequential settings are unsupported)
- `-lete_fastseq`
- `-merge`
- `-new_capture`
- `-nosingle_load_per_pattern` (the default)

- `-num_processes`
- `-parallel_strobe_data_file` (the PSD file is created but it is incorrect; use the `run_simulation` PSD generation flow instead)
- `-save_patterns`
- `-power_aware_asyncs`

set_delay

The following `set_delay` command options are silently ignored by the multithreaded TestMAX ATPG:

- `-nodisturb_clock_grouping` (use the `run_atpg -nodisturb_clock_grouping` command to get this behavior)
- `-extra_force` | `-noextra_force`

The following option affects both multithreading and non-threaded processing. The only difference is that in multithreading, it is used during stuck-at ATPG, to constrain two-cycle patterns and in transition ATPG:

- `-common_launch_capture_clock`

The following option switches to non-threaded processing, except for the `system_clock` and `extra_shift` parameters, which are supported by multithreading:

- `-launch_cycle last_shift` | `any`

set_drc

The following `set_drc` command option is silently ignored by multithreading, except when the `-dynamic`, `-one_hot`, or `-any` options are specified, which are supported by the TestMAX ATPG:

- `-clock`

ATPG Capabilities

TestMAX ATPG supports a wide variety of ATPG functionality, and has the following capabilities:

- Reads design netlists in Verilog, VHDL, and EDIF formats; and test protocol information in STIL format
- Writes test pattern files in a variety of standard and proprietary formats: WGL, STIL, Fujitsu TDL, TI TDL91, and Toshiba TSTL2

- Offers a choice of the following ATPG modes:
 - Basic-Scan ATPG, an efficient combinational-only mode for full-scan designs
 - Fast-Sequential ATPG for limited support of 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, non-decoded 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

See Also

- [ATPG Modes](#)
- [Supported Fault Models](#)

TestMAX ATPG Modes

TestMAX ATPG offers three types of ATPG modes:

- *Basic-Scan ATPG*

In basic-scan mode, TestMAX ATPG 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, nonscan 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 is available only in non-threaded mode. Like fast-sequential ATPG, full-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_only` option of the `run_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 creates its own sequence of clocks and other signals to target the as-yet-undetected faults in the design.

See Also

- [ATPG Capabilities](#)
- [Supported Fault Models](#)

Features and Benefits

TestMAX ATPG provides the following key features and benefits:

- Increases product quality with power aware test patterns for high defect detection.
For more information, see "[Power Aware ATPG](#)."
- Reduces testing costs through the use of advanced pattern compaction techniques.
For more information, see "[Compressing Patterns](#)."
- Increases designer productivity by leveraging integration with Synopsys DFTMAX compression.

For more information, see the *TestMAX DFT, DFTMAX, and DFTMAX User Guide*.

- Multicore support for faster runtime.

For more information, see "[Running Multicore ATPG](#)" and "[Running Multicore Simulation](#)."

"

- Integrated graphical user interface and simulation waveform viewer.

For more information, see "[Using the Graphical Schematic Viewer](#)," "[Using the Hierarchy Browser](#)," and "[Using the Simulation Waveform Viewer](#)."

- Comprehensive scan design rule checking.

For more information, see "[Performing Test Design Rule Checking \(DRC\)](#)."

- Generates patterns targeting specific defect mechanisms.

For more information, see "[Supported Fault Models](#)."

- Supports on-chip clocking using phase-lock loops (PLLs).

For more information, see "[On-Chip Clocking Support](#)."

- Supports quiescent test validation.

For more information, see "[Quiescence Test Pattern Generation](#)."

- Integrated fault simulator for functional vectors.

For more information, see "[Fault Simulation](#)."

- Yield Diagnosis with automatic defect isolation.

For more information, see "[Diagnosing Manufacturing Test Failures](#)."

See Also

- [ATPG Capabilities](#)
- [ATPG Modes](#)
- [Supported Fault Models](#)

Operation Modes

For each TestMAX ATPG session, you progress through a series of operation modes. Each mode reflects the types of processes you can perform:

- *BUILD Mode*

This is the initial mode in which you read in your design netlists and libraries, and create and read ATPG simulation models in preparation for design rule checking.

- *DRC Mode*

In this mode, you perform design rule checking (DRC), which analyzes your design against a set of predefined rules, and reports any anomalies.

- *TEST Mode*

In this mode, you perform ATPG, fault simulation, and fault diagnosis, and write simulation testbenches.

See Also

- [ATPG Capabilities](#)
- [Supported Fault Models](#)

2

Getting Started

The following sections describe how to get started with TestMAX ATPG:

- [Basic TestMAX ATPG Processes](#)
- [Basic ATPG Flow](#)
- [Reference Methodology](#)
- [Getting Started for TestMAX DFT Users](#)

Basic TestMAX ATPG Processes

The following sections describe the basic TestMAX ATPG processes:

- [Installing TestMAX ATPG](#)
- [Setting the Environment](#)
- [Launching TestMAX ATPG](#)
- [Executable Commands](#)
- [Setup Command Files](#)
- [Using Command Files](#)
- [Using Variables](#)
- [Running the TestMAX ATPG GUI](#)

Installing TestMAX ATPG

To obtain the TestMAX ATPG installation files, download them from Synopsys using electronic software transfer (EST) or File Transfer Protocol (FTP).

TestMAX ATPG can be installed as a standalone 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 standalone 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 TestMAX ATPG installation. You need to explicitly set this environment variable.

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

Specifying the Location for TestMAX ATPG Installation

TestMAX ATPG 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, TestMAX ATPG 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 as follows:

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

Setting the Environment

Before invoking TestMAX ATPG , you need to set your environment. Make sure your `PATH` environment variable includes the path to the Synopsys tools, which is typically `SYNOPSYS/bin`. Also set the license file using the `SNPSLMD_LICENSE_FILE` setting. For example:

```
setenv SYNOPSYS /synopsys/m_branch/
set path =($SYNOPSYS/bin $path)
setenv SNPSLMD_LICENSE_FILE synopsys_licenses/my_license.lic
```

You can optionally specify the `TMAX_SHELL` variable to avoid the need to constantly specify the `-shell` switch with the `tmax2` command.

```
% setenv TMAX_SHELL
```

If the `TMAX_SHELL` environment variable is defined, you can override it by invoking:

```
% tmax2 -gui // overrides TMAX_SHELL=1
```

Launching TestMAX ATPG

You can launch TestMAX ATPG in either shell mode or using the TestMAX ATPG GUI:

- To launch TestMAX in shell mode, specify the `-shell` option with the `tmax2` command.

```
tmax2 my_command_file -shell
```

- To launch the TestMAX ATPG GUI specify the `tmax` or `tmax2` command without using the `-shell` option.

```
tmax2 my_command_file
```

The following table summarizes the various methods you can use to invoke TestMAX ATPG.

Table TestMAX ATPG Invocation Methods

Invoke With ¹	You Get	Process List
<code>tmax2</code>	64-bit kernel plus GUI	<code>tmax2</code> , <code>tmaxgui</code>
<code>tmax2 -shell</code>	64-bit kernel	<code>tmax2</code>
<code>tmax2 -man</code>	Online help	<code>tmax2</code> , HTML browser
<code>tmax2 -help</code>	List of options	

¹ - The order of switches is not important.

² - The `tmax2` process invokes the shell. The CPU is idle after the kernel launches, but remains open so that the kernel has a transcript window in which to display output.

Executable Commands

The following sections describe the various executable commands associated with TestMAX ATPG and TestMAX Diagnosis:

- [stil2Verilog](#) – Starts MAX Testbench as a standalone executable
- [write_testbench](#) – Starts MAX Testbench within the TestMAX environment

See Also

- [Setup Command Files](#)
- [Variables](#)

Setup Command Files

A setup command file is similar to a [command file](#), except that it is executed automatically when TestMAX ATPG starts. You can include any commands in a setup command file that are used in a command file.

TestMAX ATPG includes a `tmaxtcl.rc` setup file (for Tcl mode) or a `tmax.rc` setup file (for legacy mode).

Upon startup, TestMAX ATPG automatically executes command files from multiple locations based upon the following order:

1. `$$SYNOPTSYS/admin/setup/tmax.rc`
2. `$TMAXRC`, if defined (intended for use by ASIC vendors)
3. `$HOME/.tmaxrc` or `$HOME/.tmaxtclrc`
4. `tmaxrc`, `tmax.rc`, `.tmaxtclrc`, or `tmaxtcl.rc` in the current working directory

Setup command files are executed before any command files specified in the TestMAX ATPG invocation line. You can specify a command file using any of the following techniques:

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

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

Within a script as a "here" document:

```
#!/bin/sh

tmax [other_args] -shell <<!

source command_file
exit -force

!

% tmax [other_args]

BUILD> source command_file
```

By default, commands in a command setup file are not echoed to the transcript. To see the commands as they are executed, place a `set_messages -display` command at the beginning of the command setup file.

To invoke TestMAX ATPG without using a setup command file, use the `-nostartup` switch:

```
% tmax -nostartup
```

Using Command Files

A command file is a simple ASCII text file containing any command accepted by TestMAX ATPG. You can place multiple commands into a file and execute them sequentially by running the name of the command file using the `source command_file_name` command.

There are several ways you can specify a command file:

- `% tmax command_file [other_args]...`
- `% tmax [other_args]... command_file`
- Within a script as a “here” document:

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

- `% tmax [other_args]`

```
BUILD-T> source command_file
```

You can use the `abort`, `noabort`, or `exit` options of the `set_commands` command to specify the command file execution to stop or continue when TestMAX ATPG encounters an error.

You can specify whether comments and command output will appear in the transcript or log files using the `set_messages` command.

You can reference one command file from within another by nesting `source command_file_name` commands.

You can specify a command file to be executed when you invoke TestMAX ATPG on a UNIX or NT machine running a command shell. The syntax is:

```
% tmax command_file  
% tmax command_file -shell
```

Use of the optional `-shell` argument runs the non-GUI form of TestMAX ATPG. This might be more convenient if no user interaction or interactive debugging is expected, or when you are running TestMAX ATPG from a remote telnet session or other environment where a graphic display is not available.

Note: You can condition TestMAX ATPG to pause by invoking it with the `-shell` argument followed by a command file specification; for example:

```
% tmax -shell  
% source command_file
```

Batch Files

When you operate TestMAX ATPG in batch mode using command files, you should use the `set_commands noabort` command at the beginning and the `exit -force` command at the end of each command file. Then you can safely use commands such as the following at the shell prompt:

```
% tmax batch_command_file -shell &
```

Without these commands at the beginning and end of a command file, the situation could arise where the tool encounters an error, but there is no way to make it exit.

Launching TestMAX ATPG Using Command Files

The following example launches TestMAX ATPG using a command file:

```
% tmax -shell spec_command_file.cmd
```

The following example shows a typical command file, which reads in a design that has been debugged to eliminate DRC problems. The commands in this file create and store ATPG patterns and fault lists while saving the execution log.

Example 1: Typical Command File

```
# --- basic ATPG command sequence  
#  
set_messages log last_run.log -replace  
#  
# --- read design and libraries  
#  
read_netlist spec_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 spec_top_level_name  
report_rules -fail  
#  
# --- define clocks and pin constraints  
#  
add_clocks 1 CLK MCLK SCLK  
add_clocks 0 resetn ioscl4m  
add_pi_constraints 1 testmode  
#
```

```
# --- define scan chains & STIL procedures, perform DRC checks
#
run_drc spec_design.spf
report_rules -fail
report_nonscan_cells -summary
report_buses -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
#
```

See Also

- [Command Files](#)
- [Using Command Files in Tcl Mode](#)
- [Command Entry](#)
- [Invoking TestMAX ATPG](#)

Using Variables

TestMAX ATPG 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.

A variable is recognized by the leading dollar sign (\$), followed by the variable name, as shown in the following examples:

```
set_messages log $specLOG -replace
read_netlist $LIBDIR/cmos/verilog/*.v
write_patterns $tmp/testbench.v -format verilog -replace
```


TestMAX ATPG supports two types of variables:

- UNIX environment variables

These variables are typically defined using the `setenv` command, or the `set` and `export` commands.

- User-defined environment variables

These variables are defined in the TestMAX ATPG invocation line as shown in the following example:

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

You can define multiple variables by repeating the `-env` argument of the `tmax` command. To view the current setting of any user-defined environment variable, specify the `report_settings` command. A variable defined with `-env` will override any existing environment variable with the same name.

TestMAX ATPG recognizes UNIX environment variables specified within a command. You can also set variables in a script using the `set` or `setenv` commands. The `set` command can be used for most commands; the `setenv` command makes the variable available for programs called from the TestMAX ATPG shell.

For example:

```
setenv LTRAN_SHELL 1
setenv SNPSLMD_QUEUE
```

There are several differences in behavior between Tcl mode and native mode when using variables.

Tcl Mode

In Tcl mode, you can use the `getenv` or `get_unix_variable` commands to return the value of the variable. You can also use the `$env(VAR)` syntax.

Some usage examples are as follows:

```
set_messages -log [get_env LOG_DIR]/tmax.log
report_rules -fail > [getenv RPTS]/violations.rpt
set_atpg -num_processes [get_env cpu]
source $env(SYNOPSIS)/auxx/syn/tmax/tmax2pt.tcl
```

For more information on Tcl mode, see [Using Tcl With TestMAX ATPG](#).

Native Mode

In native mode, you can use variables only at the beginning of the path file names, as shown in the following examples:.

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

Running the TestMAX ATPG GUI

The following sections describe how to set up and run the TestMAX ATPG GUI:

- [Starting and Stopping the TestMAX ATPG GUI](#)
- [Interrupting a Long Process](#)
- [Setting Preferences](#)
- [Saving GUI Preferences](#)

Starting and Stopping the TestMAX ATPG GUI

If you are in shell mode, you can display the TestMAX ATPG GUI in its current state by entering the `gui_start` command:

```
BUILD-T> gui_start
```

This command switches the context to listen-only in the GUI console.

After you start the TestMAX ATPG GUI (using the `gui_start` command), enter the `gui_stop` command to exit the GUI:

```
BUILD-T> gui_stop
```

This command stops the TestMAX ATPG GUI session and reverts to the TestMAX ATPG shell command prompt. If you did not use the `gui_start` command to start the GUI, the `gui_stop` command exits the TestMAX ATPG 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 TestMAX ATPG using the `gui_start` command to start the GUI, the `gui_stop` command exits the TestMAX ATPG application.

Interrupting a Long Process

While TestMAX ATPG is processing commands, the Submit button in the TestMAX ATPG 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 may also perform the Stop function.)

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 the following 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, TestMAX ATPG normally stops execution of the entire file, unless the file contains the following line:

```
set_commands noabort
```

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

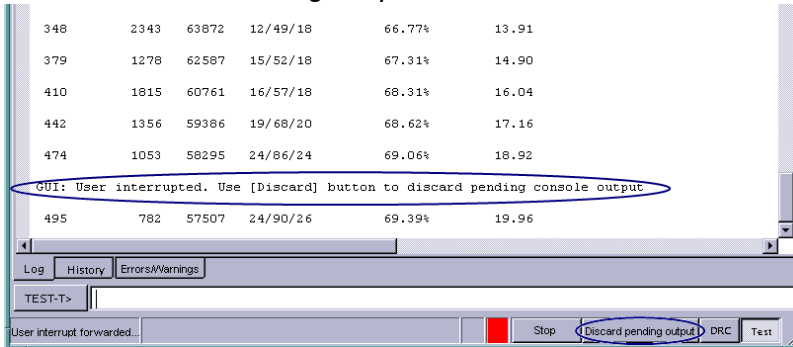
Discarding Pending Output

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

```
report_atpg_constraints -all
```

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

Figure 2 Discard Pending Output Button

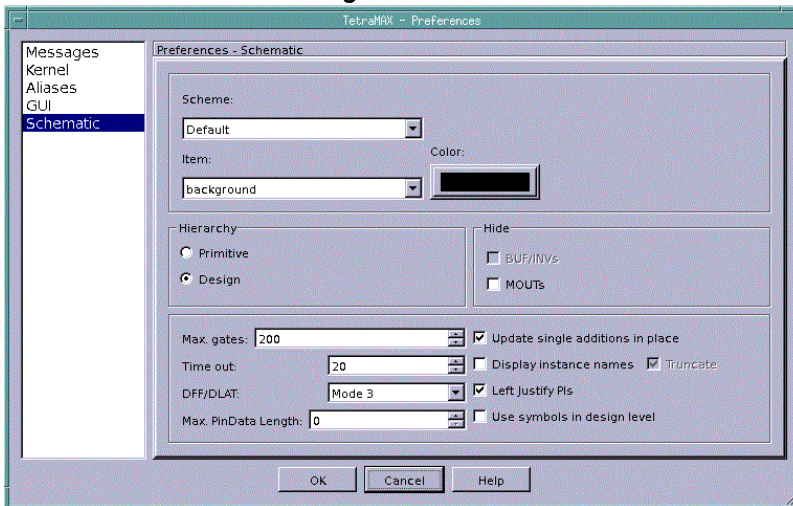


Note that the Stop button will always be enabled and operational if the kernel is still processing the current command.

Setting Preferences

You can adjust settings such message formatting, workspace size, aliases, GUI font and color display, and schematic display options. To access the Preferences dialog box, select Edit > Preferences, as shown in the following figure.

Figure 3 Preferences Dialog Box



The following table describes the various Preferences settings:

Table 1 TestMAX ATPG GUI Preferences

Category	Description
Messages	Directs output messages to a log file, formats messages with a prefix, displays comments, sets message level to standard or expert.
Kernel	Sets maximum workspace sizes for ATPG gates, decisions, file line length, string line length, connectors.
Aliases	Adds, removes, and modifies alias names and text.
GUI	Sets the display of the toolbar, the default font size and type, and the default font color for commands and error messages.
Schematic	Sets the default color scheme, the hierarchy display, the display of gates and instance names, and the pin data length.

To save any Preferences specifications you made, select Edit -> Save Preferences or Edit -> Autosave Preferences.

As an alternative to the Preferences dialog box, you can use the `set_workspace_sizes` command to change the workspace size settings.

See Also

- [Displaying Symbols in Primitive or Design View](#)

Saving GUI Preferences

You can save GUI preferences so that the settings are used the next time you invoke TestMAX ATPG.

When you invoke the TestMAX ATPG GUI, it reads some of the default graphical schematic viewer (GSV) preferences from the `tmax.rc` file. The TestMAX ATPG 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, TestMAX

ATPG saves your changes in the `$(HOME)/.config/Synopsys/tmaxgui.conf` file before it exits. The next time you invoke TestMAX ATPG, it does the following:

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

See Also

- [TestMAX ATPG GUI Main Window](#)
- [Using the Graphical Schematic Viewer](#)

Basic ATPG Flow

To run the basic ATPG flow:

1. Set your environment and launch TestMAX ATPG.

To launch TestMAX, specify the `tmax2` command.

```
setenv SYNOPSYS /synopsys/m_branch/ set path =($SYNOPSYS/bin $path)
setenv SNPSLMD_LICENSE_FILE \
synopsys_licenses/my_license.lic tmax2 my_command_file -shell
```

2. Prepare your netlist.

TestMAX ATPG can read netlists in Electronic Design Interchange Format (EDIF), Verilog, and VHDL formats. You might need to make some minor edits to make the netlists compatible with TestMAX ATPG. For more information, see [Preparing a Netlist](#).

3. Read your netlist into TestMAX ATPG using either the `read_netlist` command or the Read Netlist dialog box in the TestMAX ATPG GUI. The following example specifies the `read_netlist` command to read in all Verilog netlists in the `/tech` directory:

```
BUILD-T> read_netlist /tech/*.v
```

For more information, see [Reading a Netlist](#).

4. Read the Verilog library models using either the `read_netlist` command or the Read Netlist dialog box in the TestMAX ATPG GUI. The following example reads in all Verilog library model files in the `/proj1234/shared_verilog` directory:

```
BUILD-T> read_netlist /proj1234/shared_verilog/*.v -noabort
```

For more information on reading the Verilog library models, see [Reading Library Models](#).

5. Create an in-memory design model using either the `run_build_model` command or the Run Build Model dialog box in the TestMAX ATPG GUI. The following example builds a design model based on the last unreferenced module read by the `read_netlist` command or the Read Netlist dialog box:

```
BUILD-T> run_build_model
```

For more information, see [Preparing to Build the ATPG Model](#) and [Building the ATPG Model](#).

6. Create the STIL procedures file. For more information, see [STIL Procedures](#).
7. Define the clocks and asynchronous set and reset ports using either the STIL procedures file (SPF), the `add_clocks` command or the Add Clocks dialog box in the TestMAX ATPG GUI. The following example use the `add_clocks` command to specify a set of clocking parameters:

```
DRC-T> add_clocks 0 CLK1 -timing 200 50 80 40 -unit ns -shift
```

For more information on specifying timing and clocks, see [Defining Basic Signal Timing](#) and [Declaring Clocks](#).

8. Set up and perform design rule checking (DRC) using the `set_drc` and `run_drc` commands, or the Run DRC dialog box in the TestMAX ATPG GUI. The following example, prepares and runs DRC using the `set_drc` and `run_drc` commands:

```
DRC-T> set_drc -oscillation 200 -clock -any DRC-T> run_drc  
spec_stil_file.spf
```

For more information, see [Specifying DRC Settings](#), [Starting DRC](#), and [Performing Design Rule Checking](#).

9. Set up and run ATPG using the `set_atpg` and `run_atpg` commands or the Run ATPG dialog box in the TestMAX ATPG GUI. The following example uses the `set_atpg` and `run_atpg` command to prepare for and run ATPG:

```
TEST-T> set_atpg -patterns 500 -coverage 98 TEST-T> run_atpg
```

For more information, see [Preparing for ATPG](#) and [Running ATPG](#).

10. Using the `write_patterns` command or the Write ATPG dialog box in the TestMAX ATPG GUI to save the ATPG patterns. The following example uses the `write_patterns` command to write a set of serial STIL patterns.

```
TEST-T> write_patterns patterns.stil -serial -format stil
```

Reference Methodology

TestMAX ATPG Reference Methodology Scripts are also available for download at: <https://solvnet.synopsys.com/rmgen/>

Getting Started for TestMAX DFT Users

The following steps show how to generate ATPG patterns when you start from a netlist in which TestMAX DFT has performed scan insertion:

1. Before performing scan insertion, configure TestMAX DFT for optimal results in TestMAX ATPG.

```
test_default_delay 0 test_default_bidir_delay 0 test_default_strobe 40
test_default_period \
100 test_stil_multiclock_capture_procedures true
```

If your ASIC vendor has specific requirements, you might need to change these settings.

2. Within `dc_shell` or `design_analyzer`, write the netlist and the test protocol file.

```
test_stil_netlist_format verilog write -hierarchy -format verilog
-output \
name.v write_test_protocol -format stil -out name.spf
```

3. Read the netlist and models into TestMAX ATPG.

Use the `read_netlist` command or the Read Netlist dialog box in the TestMAX ATPG GUI. For more information, see the [Reading the Netlist](#) and [Reading Library Models](#) topics.

4. Create the in-memory design model.

Use the `run_build_model` command or the Run Build Model dialog box in the TestMAX ATPG GUI.

For details, see [Building the ATPG Model](#).

5. Define clocks and scan chains.

For details, see [Declaring Clocks in STIL](#). If your vendor requires LSI WGL protocols, additional edits to the DRC procedure file might be required. See [LSI Compatible WGL](#).

```
run_drc name.spf
```

6. Perform circuit initialization and Design Rule Checking (DRC) using the STIL procedure file.

For details, see [Performing Test Design Rule Checking](#).

7. Initialize the fault list and set ATPG options, effort, and dynamic compression.

For details, see [Preparing for ATPG](#).

8. Run ATPG to develop patterns.

For details, see [Running ATPG](#).

9. Write the fault lists using the `write_faults` command.

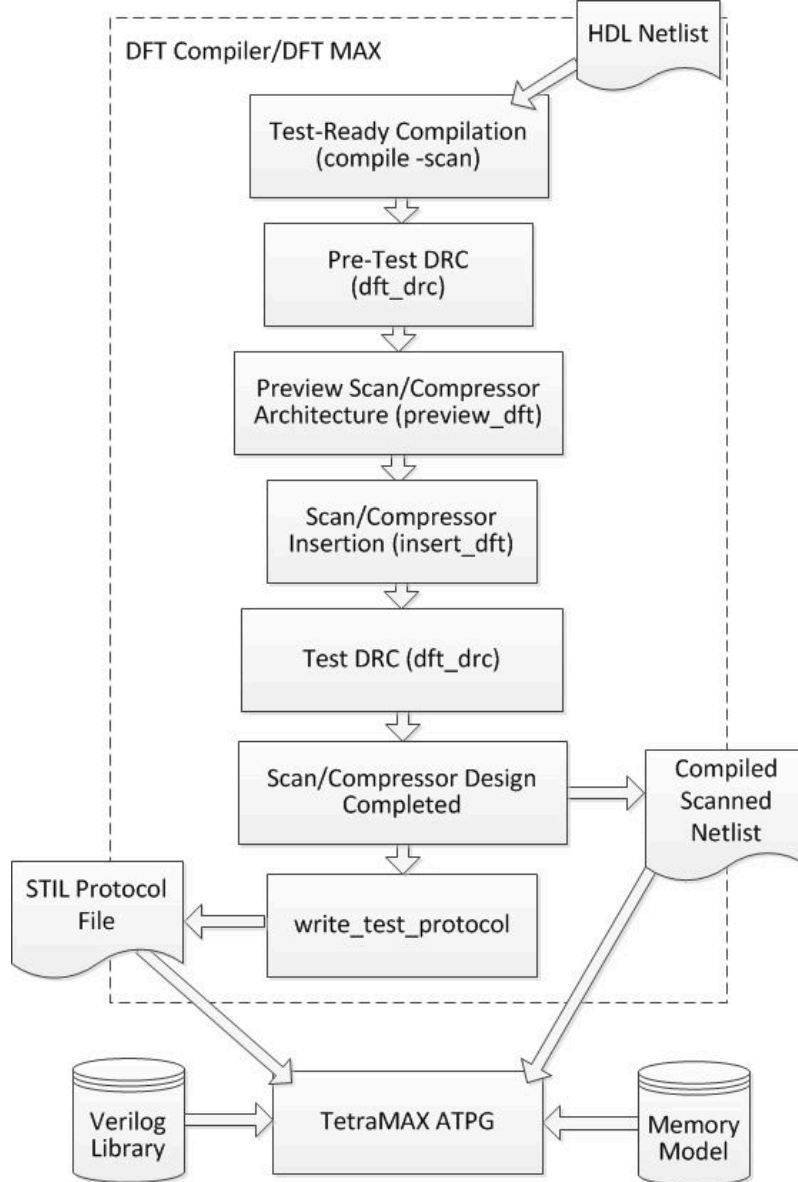
For details, see [Writing Fault Lists](#).

Design Flow Using TestMAX DFT and TestMAX ATPG

TestMAX ATPG is compatible with a wide range of design-for-test tools, such as TestMAX DFT.

The following figure shows how TestMAX ATPG fits into the TestMAX DFT design-for-test flow for a module or a medium-sized design of less than 750K gates.

Figure 4 Design Flow for a Module or Medium-Sized Design



The design flow shown in the preceding figure is as follows:

1. Starting with an HDL netlist at the register transfer level (RTL) within TestMAX DFT, run a test-ready compilation, which integrates logic optimization and scan replacement.

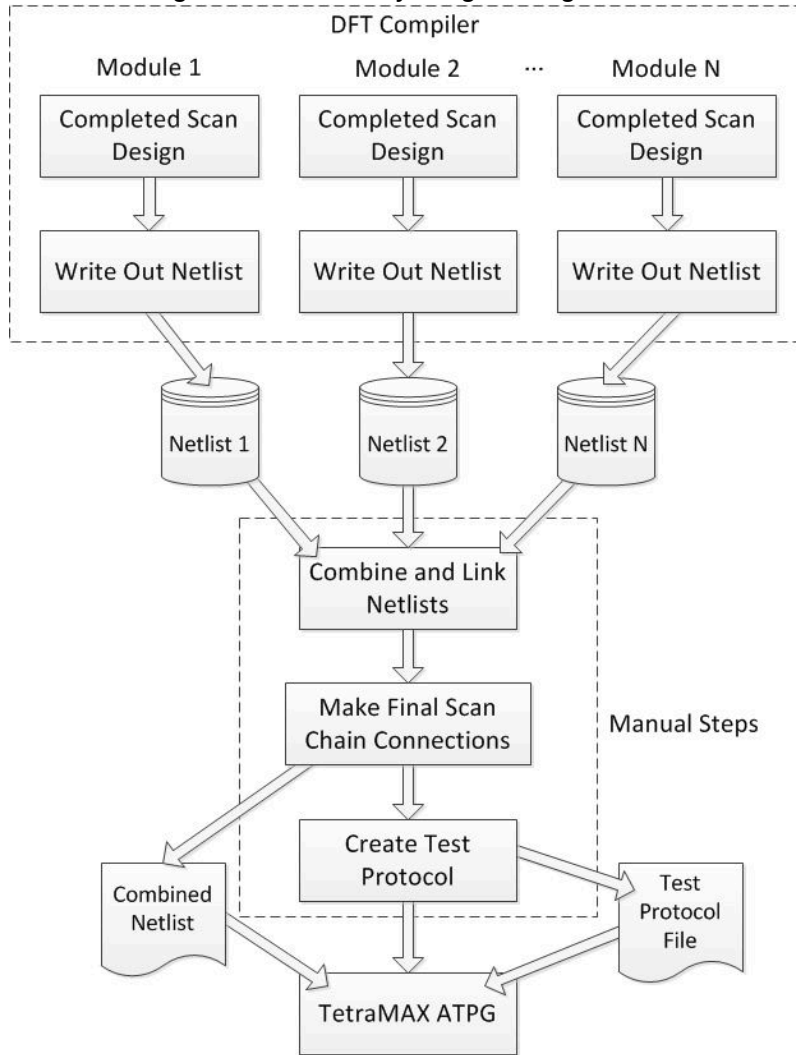
The `compile -scan` command maps all sequential cells directly to their scan equivalents. At this point, you still don't know whether the sequential cells meet the test design rules.

2. Perform test design rule checking; the `check_scan` command reports any sequential cells that violate test design rules.
3. After you resolve the DRC violations, run the `preview_scan` command to examine the scan architecture that is synthesized by the `insert_scan` command. Repeat this procedure until you are satisfied with the scan architecture, then run the `insert_scan` command, which implements the scan architecture.
4. Rerun the `check_scan` command to identify any remaining DRC violations and to infer a test protocol. For details about the TestMAX DFT design flow through completion of the scan design, see the *TestMAX DFT User Guide*.
5. When your netlist is free of DRC violations, it is ready for ATPG. For medium-sized and smaller designs, TestMAX DFT provides the `write_test_protocol` command, which allows you to write out a STL procedure file. TestMAX ATPG reads the STL procedure file and design netlist.

For details of the TestMAX ATPG portion of the design flow, see [ATPG Design Flow](#).

The following figure 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).

Figure 5 Design Flow for a Very Large Design



For large designs, you initially follow the design flow shown in [Figure 4](#) 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 5](#), 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 procedure file, see [STIL Procedure Files](#). You use the combined netlist and the manually generated test protocol file as inputs to TestMAX ATPG. For more information, see [ATPG Design Flow](#).

3

ATPG Design Flow

The ATPG process creates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. The generated patterns are used to test devices and to determine the cause of failure. ATPG effectiveness is measured by the amount of modeled defects, or fault models, that are detected and the number of generated patterns.

The following sections describe the basic ATPG design flow:

- [ATPG Design Flow Overview](#)
- [Running the Basic ATPG Design Flow](#)
- [Preparing a Netlist](#)
- [Configuring to Read a Netlist](#)
- [Reading a Netlist](#)
- [Reading Library Models](#)
- [Preparing to Build the ATPG Model](#)
- [Building the ATPG Model](#)
- [Performing Design Rule Checking \(DRC\)](#)
- [Preparing for ATPG](#)
- [Running ATPG](#)
- [Analyzing ATPG Output](#)
- [Reviewing Test Coverage](#)
- [Writing ATPG Patterns](#)

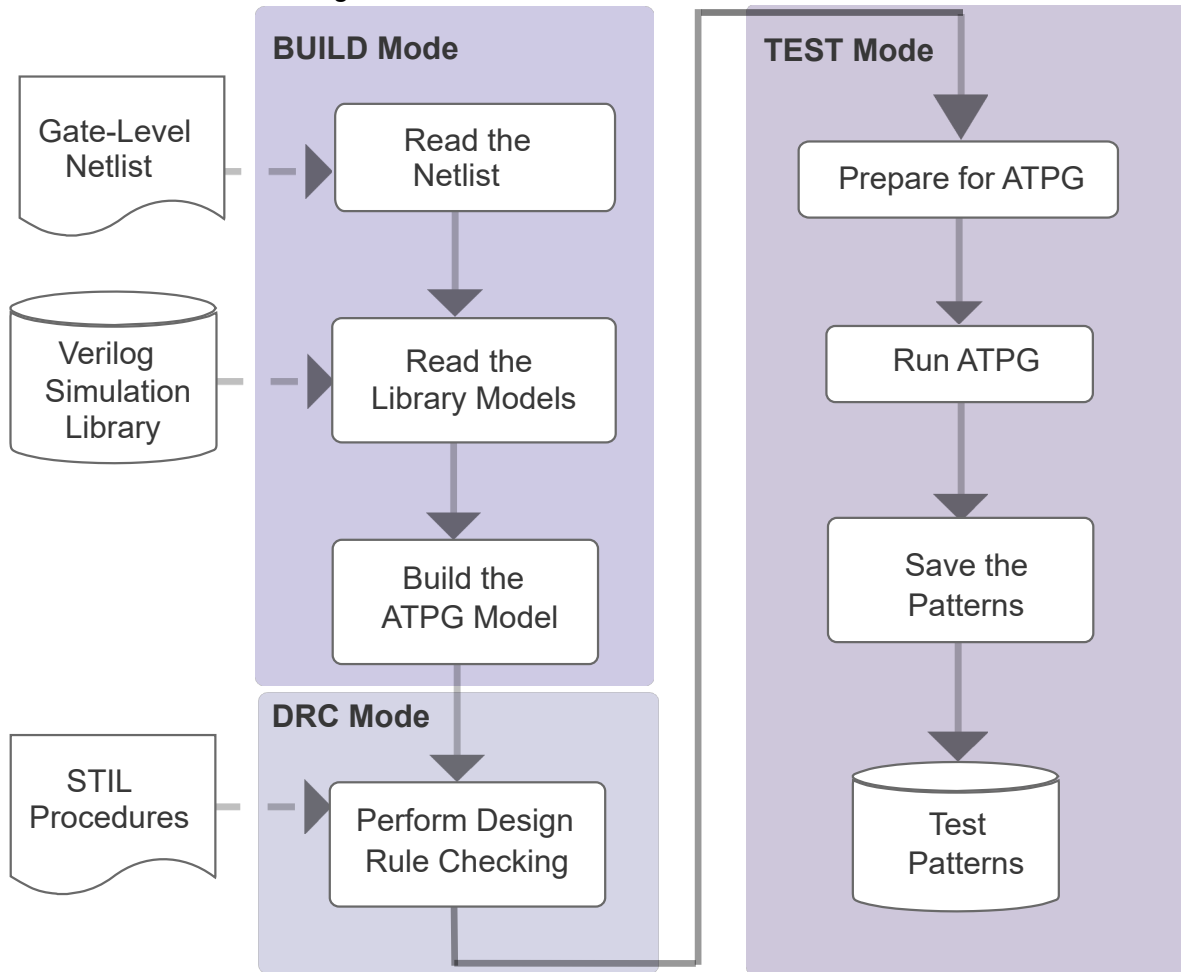
ATPG Design Flow Overview

The basic ATPG flow applies to most designs. To get started running ATPG, you must provide a supported netlist, a model library, and a set of STIL procedures used for design rule checking.

STIL procedures are usually provided via a STIL procedures file generated from the DFT Compiler tool or an equivalent tool. You can also provide many of the parameters via TestMAX ATPG commands. For complete information on STIL procedures, see [STIL Procedures](#).

The following figure shows the basic ATPG design flow. For a step-by-step overview of the ATPG design flow, see [Running the Basic ATPG Design Flow](#).

Figure 6 Basic ATPG Design Flow



If you encounter problems with your design, see [Using the GSV for Review and Analysis](#), which provides information on graphical analysis and troubleshooting.

Basic ATPG Run Script

```
set_messages -log mylog -replace

# Read in the netlist library and
# Verilog library
read_netlist -library library/*.v

read_netlist design.v

# Build the ATPG Model
run_build_model DESIGN_TOP

# Set up and run DRC
set_drc stil_procedures.spf
run_drc

# Add faults and run ATPG
add_faults -all
run_atpg -auto

# Write the patterns
write_patterns DESIGN.stil -format STIL \
-replace

write_patterns DESIGN.bin -replace
```

Running the Basic ATPG Design Flow

The basic ATPG design flow consists of the following steps:

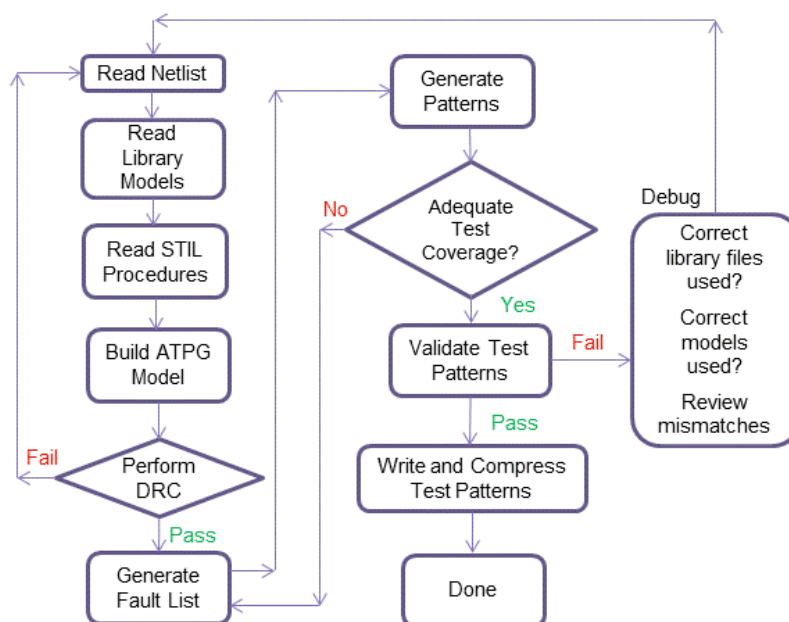
1. Prepare your netlist or netlists (see [Preparing a Netlist](#)).
2. Read the netlist (see [Reading a Netlist](#)).
3. Read the library models (see [Reading Library Modules](#)).
4. Build the ATPG design model (see [Setting Up and Building the ATPG Model](#)).
5. Perform test DRC and make any necessary corrections (see [Performing Test Design Rule Checking](#)).
6. Prepare the design for ATPG, set up the fault list, analyze buses for contention, and set the ATPG options (see [Preparing for ATPG](#)).
7. Run automatic test pattern generation (see [Running ATPG](#)).
8. Analyze the ATPG pattern generation output (see [Analyze ATPG Output](#)).

9. Review the test coverage (see [Reviewing Test Coverage](#)).
10. Rerun ATPG, as needed.
11. Write and save the test patterns (see [Writing ATPG Patterns](#)).

For an example of a typical command file used for running a basic ATPG design flow in TestMAX ATPG, see [Using Command Files](#).

The following figure shows a typical ATPG design flowchart.

Figure 7 ATPG Design Flowchart



See Also

- [Design Flow Using TestMAX DFT and TestMAX ATPG](#)
- [ATPG Design Guidelines](#)

Preparing a Netlist

TestMAX ATPG accepts netlists in Verilog, EDIF, and VHDL formats. For more information on these formats, see [Netlist Format Requirements](#).

Netlists can be flat or hierarchical and can be in standard ASCII format or GZIP format. TestMAX ATPG automatically detects compressed files and decompresses them during the read operation.

Before reading in a netlist or library models, you should compare the names of the modules in your netlist to the names of the Verilog library models you are using. If there are duplicate module definitions, TestMAX ATPG uses the last definition it encounters. If you read in your netlist and then read in library models, the modules in your netlist are overwritten by any library models using the same names.

You can specify the following options in preparation for reading a netlist:

- Set the maximum number of parsing errors allowed before terminating the parsing of the current netlist file
- Use the last module read if your design has duplicate modules. This allows TestMAX ATPG to reread a file if you edit a module or read multiple files if there is a duplication of module definitions.
- Accept or ignore the 'celldefine, 'enable_portfaults, and 'supress_faults Verilog compiler directives
- Set check and warning behavior for reading netlists and designs
- Specify if conservative or combinational MUX gates are extracted from conservative UDP models of a MUX
- Set parameters for handling dominance behavior between set, reset, and clock pins
- Specify behavior for escape characters, redefined modules, scalar nets, and X modeling

For complete descriptions of these options, see the description of the `set_netlist` command in TestMAX ATPG Help.

See Also

- [Configuring to Read a Netlist](#)
- [Reading a Netlist](#)

Configuring to Read a Netlist

You can use either the `set_netlist` command or the Set Netlist dialog box or Read Netlist dialog box to specify options for reading a netlist into TestMAX ATPG.

The following example shows how to use the `set_netlist` command to allow a maximum of 15 parsing errors, extract combinational MUX Gates from conservative MUX UDP models, and use the remaining default parameters for reading a netlist:

```
BUILD> set_netlist -max_errors 15 -conservative_mux combo_udp
```

To use the TestMAX ATPG GUI to set the parameters specified in the previous example:

1. Do one of the following:
 - From the menu bar, select Netlist > Set Netlist Options.
The Set Netlist dialog box appears.
 - From the command toolbar, click the Netlist button.
The Read Netlist dialog box appears.
2. In either the Set Netlist dialog box or the Read Netlist dialog box, enter 15 in the Maximum Errors text field, and select Combinational UPD in the Conservative MUX drop-down menu.
3. Click OK.

For a complete description of the requirements and contents of netlists used for TestMAX ATPG, see [Design Netlists and Libraries](#).

Reading a Netlist

You can read one or more netlists associated with your design using the `read_netlist` command or the Read Netlist dialog box in the TestMAX ATPG GUI.

The following example specifies the `read_netlist` command to read in all Verilog netlists in the `/tech` directory:

```
BUILD-T> read_netlist /tech/*.v
```

You can specify as many `read_netlist` commands as necessary to read in all portions of a design. You can read multiple files from the same directory using wildcards (for example, `*.v`). For more information, see [Using Wildcards to Read Netlists](#).

To read a netlist using the Read Netlist dialog box:

1. Do one of the following:
 - From the menu bar, select Netlist > Read Netlist.
 - From the command toolbar, click the Netlist button.

In both cases, the Read Netlist dialog box appears with the selected default values.

2. Change or select any values to meet your requirements. The options in the Read Netlist dialog box are equivalent to the options for the `read_netlist` command (see the description in TestMAX ATPG Help).
3. Click OK.

See Also

- [Working with Design Netlists and Models](#)
- [About Reading a Netlist](#)
- [Netlist Requirements](#)
- [Reading the Library Models](#)

Reading Library Models

To read library models, use the `read_netlist` command or the Read Netlist dialog box.

The following example uses the `read_netlist` command to read in all Verilog library model files in the `/proj1234/shared_verilog` directory and to not terminate the process if there is an error reported for a model:

```
BUILD-T> read_netlist /proj1234/shared_verilog/*.v -noabort
```

You can specify as many `read_netlist` commands as necessary to read in all portions of a design. You can read multiple files from the same directory using wildcards (for example, `*.v`). For more information, see [Using Wildcards to Read Netlists](#).

To read library models using the Read Netlist dialog box:

1. Do one of the following:
 - From the menu bar, select Netlist > Read Netlist.
 - From the command toolbar, click the Netlist button.

In both cases, the Read Netlist dialog box appears with the selected default values.

2. Change or select any values to meet your requirements. To duplicate the previous command line example, make sure the Abort on Error check box is not selected. The options in the Read Netlist dialog box are equivalent to the options for the `read_netlist` command (see the description in TestMAX ATPG Help).
3. Click OK.

See Also

- [About Reading a Library Model](#)
- [Reading a Netlist](#)
- [Building the ATPG Model](#)

Preparing to Build the ATPG Model

You can use either the `set_build` command or the Set Build dialog box to set parameters for building the ATPG model.

The following example uses the `set_build` command to set the parameters to build an ATPG model. In this case, it specifies TestMAX ATPG to use a period (.) as a hierarchical delimiter and to not to delete any unused gates:

```
DRC-T> set_build -hierarchical_delimiter . -nodelete_unused_gates
```

You can make the same settings from the previous example using the Set Build dialog box, as shown in the following steps:

1. Do one of the following:
 - From the menu bar, select Netlist > Set Build Options.
 - From the command toolbar, click the Build button. When the Build Model dialog box appears, click the Set Build Options button.

In both cases, the Set Build dialog box appears with the selected default values.

2. Change or select the values in the Set Build dialog box to meet your requirements.

The options in this dialog box are equivalent to the options for the `set_build` command (see the description in TestMAX ATPG Help). To match the options specified in the previous example, enter a period (.) in the Hierarchical text field and unselect the Delete Unused Gates checkbox.

3. Click OK.

See Also

- [Building the ATPG Model](#)

Building the ATPG Model

You can use the `run_build_model` command or the Run Build dialog box to build the ATPG model.

TestMAX ATPG builds a model based on the last unreferenced module read by the `read_netlist` command or the Read Netlist dialog box. This means you do not need to specify any options with the `run_build_model` command, as shown in the following example:

```
BUILD-T> run_build_model
```

You can also specify a particular module, as shown in the following example, which includes the command transcript:

```
BUILD-T> run_build_model spec_asic
-----
Begin build model for topcut = spec_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
-----
```

To build the ATPG model using the Run Build Model dialog box:

1. Do one of the following:

- From the menu bar, select Netlist > Run Build Model.
- From the command toolbar, click the Build button.

In both cases, the Run Build dialog box appears with the selected default values.

2. Change or select the additional values in the Run Build dialog box to meet your requirements. The options in this dialog box are equivalent to the options for the `run_build_model` command and the `set_learning` command (see the descriptions for both commands in TestMAX ATPG Help). To match the option specified in the previous example, enter `spec_asic` in the Top Module name text field.

3. Click OK.

The build model process begins.

See Also

- [About Building the ATPG Model](#)
- [Processes That Occur When Building the ATPG Model](#)

Performing Design Rule Checking (DRC)

During DRC, TestMAX ATPG performs a set of checks to ensure that the scan structure is correct and to determine how to use the scan structure for test generation and fault simulation. These checks include ensuring that the scan chains operate properly, identifying scan cells, identifying nonscan cell behavior, and ensuring that clocks obey the required rules.

The following sections describe how to prepare for and perform DRC:

- [Specifying STIL Procedures](#)
- [Specifying DRC Settings](#)
- [Starting DRC](#)
- [Reviewing the DRC Results](#)
- [Understanding Rule Violations](#)
- [Viewing DRC Violations in the GSV](#)

Specifying STIL Procedures

The STIL language describes scan-shifting protocol, test procedures, and ATPG signal, timing, and data information. STIL procedures provide information TestMAX ATPG uses as a basis to perform design rule checking (DRC).

TestMAX ATPG supports a subset of STIL syntax that describe:

- Scan chain inputs and outputs
- Pin constraints for test modes
- Clock ports and waveform definitions
- Shifting and capturing protocols
- Initialization sequences

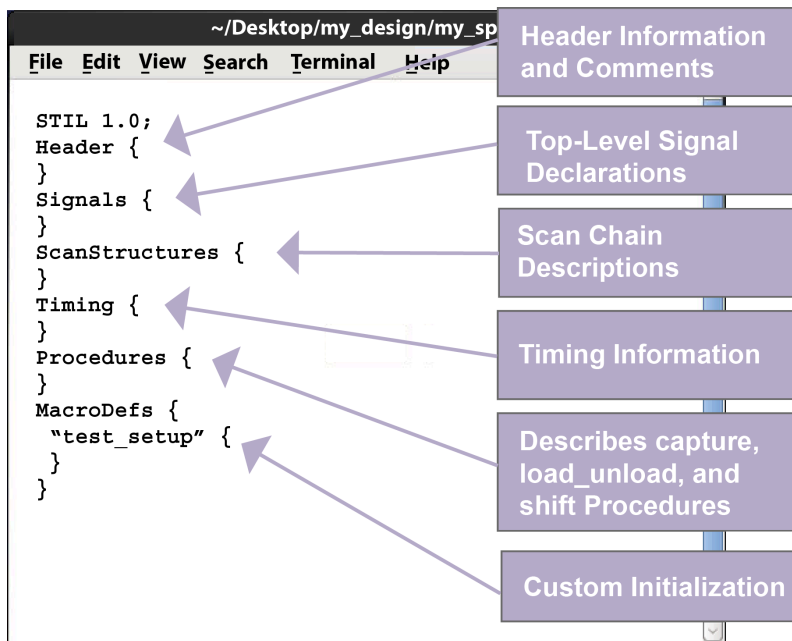
There are several ways you can provide STIL procedures to TestMAX ATPG for DRC:

- Create an SPF using Synopsys' TestMAX DFT tool.
- Create an SPF template file using the `write_drc_file` command. For details, see [Creating a New SPF](#).
- Use the QuickSTIL tab in the DRC dialog box of the TestMAX ATPG GUI.

- Use TestMAX ATPG commands, such as `add_clocks`, `add_scan_chains`, and `add_pi_constraints`."

The following figure provides a brief description of the major sections of a STIL procedures file.

Figure 8 STIL Procedures File



Specifying DRC Settings

Prior to performing DRC, make sure you specified a set of STIL procedures as described in [Specifying STIL Procedures](#). These procedures provide key information that TestMAX ATPG needs to perform DRC.

You can set a variety of parameters that control the DRC process, including:

- Specify clock grouping and skew values
- Set the number of simulation passes before oscillation
- Define restrictions on clock usage for pattern generation
- Specify DLAT clock checks and DRC violation parameters for DFF and DLAT devices with unstable sets or resets
- Display primitives in scan chains that were sensitized during scan chain tracing

- Generate patterns with capture cycles that always have clock pulses from a controller clock
- Limit the reporting of shadows
- Specify the top-level port that globally enables or disables bidirectional pins
- Define the number of PLL clock pulses supported per load and the number of pulses to extend the simulation of the load procedure
- Allow patterns to have more than one capture clock procedure per load
- Store simulated time periods of the test_setup procedure, stability patterns, and unload mode data

Options for Specifying DRC Settings

You can use the `set_drc` command or the DRC dialog box to specify DRC parameters.

The following example shows how to specify the `set_drc` command:

```
DRC-T> set_drc -oscillation 200 -clock -any
```

This example uses the `-oscillation` option to specify that 200 simulation passes are allowed during DRC simulation before oscillation is declared. It also uses the

`-clock -any` setting to allow pattern generation using any single clock, including patterns that don't use clocks.

To use the Run DRC dialog box to set DRC parameters:

1. Do one of the following:
 - From the menu bar, select Rules > Run DRC
 - From the command toolbar, click the DRC button

In both cases, the DRC dialog box appears with the Run tab active.
2. In the Set field of the DRC dialog box, specify the options you want to apply to the DRC process. To duplicate the settings of the `set_drc` command in the previous example:
 - a. Enter 200 in the Oscillation Passes text field
 - b. Select -Any from the Capture Clock drop-down menu.
3. Click the Set button to save your settings.

For more information on specifying DRC options, see [Design Rule Checking](#).

See Also

- [Starting Test DRC](#)
- [Reviewing the DRC Results](#)

Starting DRC

Before starting DRC, make sure you specified the appropriate STIL procedures. See [Specifying STIL Procedures](#) and [Specifying DRC Settings](#).

To start DRC, use the `run_drc` command or the Run DRC dialog box in the TestMAX ATPG GUI.

The following example uses the `run_drc` command to start DRC:

```
DRC-T> run_drc spec_stil_file.spf
```

The argument in the example, `spec_stil_file.spf`, is the name of the STIL procedure file.

To use the Run DRC dialog box to perform DRC:

1. Do one of the following:
 - From the menu bar, select Rules > Run DRC.
 - From the command toolbar, click the DRC button

In both cases, the DRC dialog box appears with Run tab active.

2. In the Test Protocol File Name field, enter the path name of the STIL procedure file previously created, or use the Browse button to navigate and select the file.
3. Click Run.

As TestMAX ATPG performs the DRC checks, it produces a status report and lists the DRC violations, as shown in the following example.

See "DRC Rules" in TestMAX ATPG Help for a list of the rule categories, including links to each category.

Typical DRC Run

```
BUILD-T> run_drc spec_stil_file.spf
-----
Begin scan design rule 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.
-----
```

Chapter 3: ATPG Design Flow

Performing Design Rule Checking (DRC)

```

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

After you run DRC (see [Starting Test DRC](#)), TestMAX ATPG generates a summary report that provides a starting point for reviewing the DRC results. To view a description of the summary report, see [Understanding the DRC Summary Report](#).

You should inspect and correct all DRC violations that are classified as errors. If you ignore or overlook these violations, the ATPG patterns might fail in simulation or on the real device. For more information about DRC rule violations and how to fix them, see [Understanding DRC Rule Violations](#).

To view descriptions of specific violations and how to fix them, see "DRC Rules by Category" in TestMAX ATPG Help).

If you want to view a summary of failing rule messages, enter the following command:

```
DRC-T> report_rules -fail
```

The DRC summary report in the following example shows one class of clock rule warnings (C17) and two classes of bus rule warnings (Z4 and Z9).

The following example shows an example of the `report_rules -fail` output.

Example 2 Reporting Rules That Fail

```
TEST-T> 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-T> report_violations c17-2
DRC-T> report_violations c17
DRC-T> report_violations c
DRC-T> report_violations -all
```

See Also

- [Understanding run_drc Output](#)
- [Starting Test DRC](#)
- [Viewing Violations in the GSV](#)

Understanding Rule Violations

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 nine major categories:

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

Links to descriptions of individual rules are provided in the "Rules Violation Messages" topic in TestMAX ATPG Help.

When a rule is violated, each violation is assigned a unique violation ID, which is the rule ID followed by a dash and then a sequence number. For example, the rule violation ID for the 24th violation of a Z4 rule is Z4-24. You can use this number to identify a specific violation for reporting or analysis.

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.

Each rule violation also includes a brief description of what is checked by the rule. For example, a B8 rule violation explains that the circuit contains an unconnected module input pin.

The effects of a rule violation 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.

TestMAX ATPG Help provides a complete description of each rule violation. The "What Next" section in the description for a rule suggests an action you can take to analyze the cause of the rule violation and determine whether the violation is fixable by changing a procedure or setup, or whether the design may have to be changed.

For rule violations with an error severity, the occurrence message is displayed when the rule violation occurs. For rule violations with a warning severity, the summary message is displayed at the end of the process. You can selectively display the occurrence messages for a warning using the `report_violations` command.

Viewing DRC 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. For more information on using the GSV, see [Using the Graphical Schematic Viewer](#).

To analyze a warning message in the GSV:

1. Click the Analyze button in the command toolbar at the top of the [TestMAX ATPG GUI main window](#).

The Analyze dialog box appears.

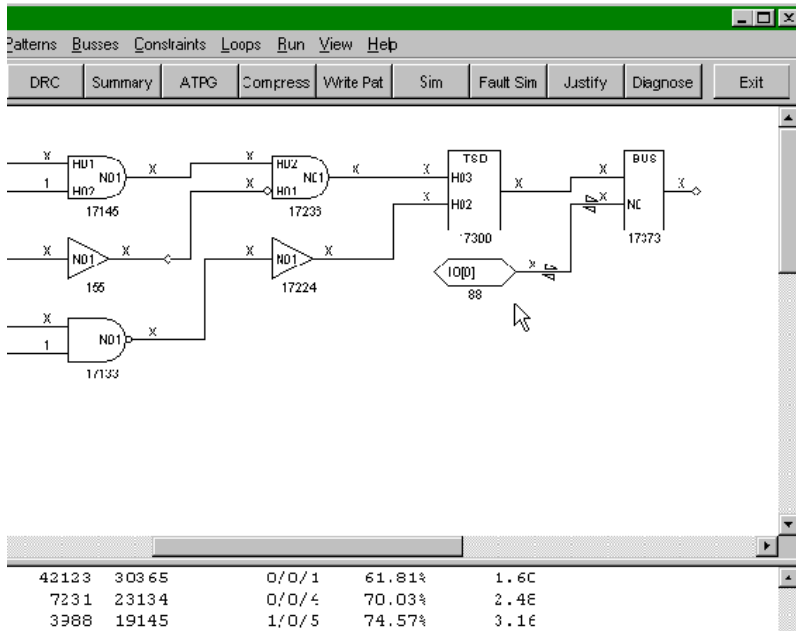
2. Click the Rules tab if it is not already active.

A dialog 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.

3. Select a violation from the list or type a specific violation occurrence number in the Rule Violation field.
4. Click OK.

The GSV opens and displays the violation. The transcript window also displays the error message.

Figure 9 Schematic Display of DRC Violation



An example Z4 violation message is shown in the following example:

```
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.

The preceding figure 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](#).

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 is lowered because the ATPG algorithm will not generate any pattern that would cause contention.

Some messages can be safely ignored. Others can be resolved through adjustment of a procedure definition; and others require a change to the design.

See Also

- [Using the Graphical Schematic Viewer](#)

Preparing for ATPG

ATPG creates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. The generated patterns are used to test devices and to determine the cause of failure.

To prepare for ATPG, you can specify general ATPG settings, set up the fault list, select the fault model (stuck-at, IDDQ, path delay, hold time, transition, or bridging), select the pattern source (internal, external, or random patterns), and select the ATPG mode (basic-scan, fast-sequential, or full-sequential).

The following tasks show you how to prepare for ATPG:

- [Specifying General ATPG Settings](#)
- [Specifying Fault Lists](#)
- [Specifying Fault Models](#)
- [Specifying the Pattern Source](#)
- [Specifying the ATPG Mode](#)

See Also

- [Running ATPG](#)

Specifying General ATPG Settings

There are a variety of general parameters you can use to control pattern generation by TestMAX ATPG. For example, you can:

- Specify the maximum number of patterns to generate before terminating ATPG
- Set the maximum CPU time allowed per fault before terminating fault detection
- Limit the maximum coverage for ATPG to attain before terminating
- Set the minimum number of system cycles for each pattern
- Use fill options for running internal scan and compressed scan patterns
- Establish checkpoints to save patterns and fault lists to files

For complete details on these settings and all other settings that control ATPG, see [ATPG Settings](#).

Options for Specifying ATPG Settings

You can specify several types of general settings for ATPG using the `set_atpg` command or the TestMAX ATPG GUI, as shown in the following examples:

- The following command specifies TestMAX ATPG to generate a maximum of 500 patterns and to terminate ATPG when the coverage reaches 98 percent:

```
set_atpg -patterns 500 -coverage 98
```

- The following command specifies that each pattern must have minimum of 5 system cycles and to report extras messages during the pattern merge operation:

```
set_atpg -min_ateclock_cycles 5 -verbose
```

- The following command specifies TestMAX ATPG to use the random decision method when compressing patterns and to save patterns to the `chkp_patt` file every 360 CPU seconds:

```
set_atpg -checkpoint {360 chkp_patt}
```

The following steps make the same settings specified in the previous examples using the Run ATPG dialog box:

1. Do one of the following:

- Select Run > Run ATPG
- Click the ATPG button in command bar

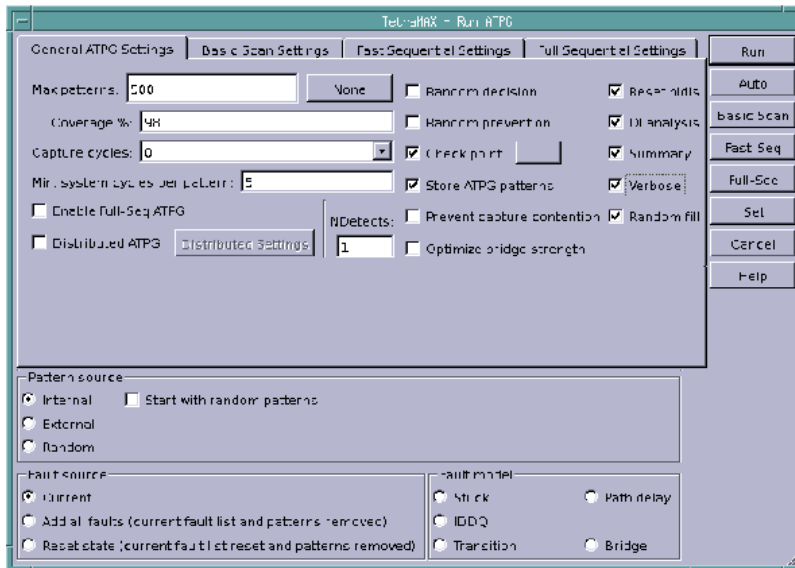
In both cases, the Run ATPG dialog box appears.

2. Click the General ATPG Settings tab, then do the following:

- a. Enter 500 in the Max patterns text field.
- b. Enter 98 in the Coverage % text field.
- c. Enter 5 in the Min. system cycles per pattern text field.
- d. Click the Verbose check box.
- e. Click the Random fill check box.
- f. Click the Check point check box. In the Set Check Point dialog box, enter `chkp_patt` in the Pattern file name text field and 360 in the Time interval text field.
- g. Click OK.

The following figure shows the appearance of the Run ATPG dialog box after entering the specifications from the previous steps (note that the default settings are also selected).

Figure 10 General Pattern Generation Options in the Run ATPG Dialog Box



Specifying Fault Lists

TestMAX ATPG maintains a list of potential faults for a design. You can specify TestMAX ATPG to use an existing fault list provided in a formatted ASCII file, create a fault list, or use only particular faults.

For complete information on using faults and fault lists, see [Working with Faults and Fault Lists](#).

The following sections show several methods for specifying and creating fault lists:

- [Selecting an Existing Fault List File](#)
- [Generating a Fault List Containing All Fault Sites](#)
- [Including Specific Faults in a Fault List](#)
- [Writing Faults to a File](#)
- [Example Fault Lists](#)

Selecting an Existing Fault List File

To specify TestMAX ATPG to use an existing fault list file, do one of the following:

- Use the `read_faults` command, as shown in the following example:

```
TEST-T> read_faults spec_faults.all
```

- Use the Add Faults dialog box by doing the following:

1. Select Faults > Add Faults

The Add Faults dialog box appears.

2. Click the Read File button, and enter or browse and select the name of the fault file.
3. Click OK

Generating a Fault List Containing All Fault Sites

To generate a fault list that includes all possible fault sites in the ATPG design model, do one of the following:

- Specify the `add_faults` command, as shown in the following example:

```
TEST-T> add_faults -all
```

- Use the Add Faults dialog box by doing the following:

1. Select Faults > Add Faults

The Add Faults dialog box appears.

2. Click the All button.
3. Click OK

Including Specific Faults in a Fault List

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

- Specify objects to be excluded using the `add_nofaults` command and then execute the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_nofaults /sub_block_A/adder  
TEST-T> add_nofaults /io/demux/alu TEST-T> add_faults -all
```

- Remove faults based on fault locations in a fault list file specified by the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_faults -all TEST-T> read_faults fault_list_file -delete
```

- Remove faults using the `remove_faults` command after executing the `add_faults -all` command, as shown in the following example:

```
TEST-T> add_faults -all TEST-T> remove_faults /sub_block_A/adder
TEST-T> remove_faults /io/demux/alu
```

- If you have a small number of faults, you can add them explicitly using the `add_faults` command:

```
TEST-T> remove_faults -all TEST-T> add_faults /proc/io TEST-T>
add_faults /demux TEST-T> add_faults /reg_bank/bank2/reg5/Q
```

Note: You can perform these same tasks in the TestMAX ATPG GUI using the Add Faults, Add No Faults, Remove Faults dialog boxes.

Writing Faults to a File

You can use the `write_faults` command or the Report Faults dialog box to write a fault list to a file for analysis or to read back in for future ATPG sessions:

- Write a fault list containing only AU class faults, as shown in the following example:

```
TEST-T> write_faults faults.AU -class au -replace
```

- Write a fault list for all faults:

```
TEST-T> write_faults filename -all -replace
```

- Write only the undetectable blocked (UB) and undetectable redundant (UR) fault classes:

```
TEST-T> write_faults filename -class UB -class UR -replace
```

- Write only the faults down one hierarchical path:

```
TEST-T> 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_faults -report` command. The following command overrides the default by using the `-collapsed` option:

```
TEST-T> write_faults filename -all -replace -collapsed
```

- Generate a fault list using the Report Faults dialog box:

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 might appear to the right of the Report Type list box, depending on your selection.

3. Select the options you want.
4. Click OK.

Example Fault Lists

The following example 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.

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, the following example shows the same fault list written with the `-collapsed` option specified.

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
```

See Also

- [Fault Lists and Faults](#)

Specifying Fault Models

Effective testing requires an accurate behavioral description of a design containing defects. Fault models represent how a manufacturing defect affects a design, and are

crucial in identifying target faults and performing fault analysis. You can run TestMAX ATPG using any of the following fault models:

- *Stuck-At* — This is the default model used by TestMAX ATPG, and is the industry standard model used for generating test patterns. 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. For more information on the stuck-at fault model and fault models in general, see "[Understanding Fault Models](#)"
- *Transition Delay* — Generates test patterns to detect single-node slow-to-rise and slow-to-fall faults. Using this model, TestMAX ATPG launches a logical transition upon completion of a scan load operation and uses a capture clock procedure to observe the transition results. For more information, see "[Transition-Delay Fault ATPG.](#)"
- *Path Delay* — 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 "[Path Delay Fault and Hold Time Testing.](#)"
- *Hold Time* — This model is similar to the transition delay and path delay models, except that it detects a fault through the shortest possible path to increase the probability of finding small delay defects or process variations. For more information, see "[Hold Time ATPG Test Flow.](#)"
- *IDDQ* — Assumes that a circuit defect causes excessive current drain due to an internal short circuit from a node to ground or to a power supply. For this model, TestMAX ATPG 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. For more information, see "[Quiescence Test Pattern Generation.](#)"
- *Bridging* — Detects shorts that cause a connection between two normally unconnected signals. These 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. For more information, see "[Bridging Fault ATPG.](#)"
- *IDDQ Bridging* — Uses the IDDQ model to generate additional patterns and increase the IDDQ coverage. The IDDQ bridging model uses only the toggle version of the standard IDDQ model, which means that the fault site at a gate input does not require propagation to an output of the same gate to be identified as a fault. For more information, see "[IDDQ Bridging.](#)"

- *Dynamic Bridging* — Combines components of the static bridging fault model and the transition fault model to analyze transition effects in the presence of a specified value on a bridge aggressor node. For more information, see "[Running the Dynamic Bridging Fault ATPG Flow.](#)"

Selecting a Fault Model

TestMAX ATPG uses the stuck-at fault model by default. You can select any supported fault model using the `-model` option of the `set_faults` command or the Set Faults or Run ATPG dialog boxes.

The following example shows how to use the `set_faults` command to specify the transition-delay fault model:

```
TEST-T > set_faults -model transition
```

The following table shows the keywords used with the `-model` option to specify the various fault models:

Keyword	Fault Model
<code>stuck</code>	Stuck-At
<code>iddq</code>	IDDQ
<code>iddq_bridging</code>	IDDQ Bridging
<code>transition</code>	Transition-Delay
<code>path_delay</code>	Path Delay
<code>hold_time</code>	Hold Time
<code>bridging</code>	Bridging
<code>dynamic_bridging</code>	Dynamic Bridging

The following sets of steps show you how to specify a fault model using the Set Faults dialog box:

1. Select **Faults > Set Fault Options** from the menu bar.
The Set Faults dialog box appears.
2. In the **Model** section, click the button associated with the fault model you want to use.
3. Click **OK**.

To use the Run ATPG dialog box to specify a fault model:

1. Do one of the following:

- Select Run > Run ATPG from the menu bar
- Click the ATPG button in the command toolbar

In both cases, the Run ATPG dialog box appears.

2. In the Fault model section, click the button associated with the fault model you want to use.

3. Click the Set button to save your specification.

Specifying the Pattern Source

You can configure TestMAX ATPG to use the following pattern sources:

- *Internal patterns* - These patterns are stored in memory and generated internally by TestMAX ATPG. You can identify internal patterns by running the `report_patterns` command.
- *External patterns* - These patterns are stored in a file. TestMAX ATPG can read external pattern files in several formats, including Verilog, VHDL, STIL, and WGL. These patterns must use the same syntax TestMAX ATPG uses when it writes patterns.
- *Random patterns* - These patterns are defined by parameters set by the `set_random_patterns` command)

Patterns should be stored in a binary file, if possible. The WGL and STIL formats are unable to accurately store all the pattern data required by TestMAX ATPG. When you read back a STIL or WGL pattern file, the fast-sequential patterns might be interpreted as a full-sequential patterns, and errors are reported. Do not assume that TestMAX ATPG can correctly read STIL or WGL patterns created by tools other than TestMAX ATPG.

The following sections describe the pattern types and formats accepted by TestMAX ATPG, including how to select the pattern source:

- [Scan and Nonscan Functional Patterns](#)
- [STIL Functional Pattern Input](#)
- [Verilog Functional Pattern Input](#)
- [WGL Functional Pattern Input](#)

- [VCDE Functional Pattern Input](#)
- [Options for Selecting the Pattern Source](#)

Scan and Nonscan Functional Patterns

TestMAX ATPG accepts two primary types of functional pattern files:

- *Scan functional patterns* - These patterns contain scan chain load and unload sequences and define structures and procedures that can be recognized as scan-chain related.

They must use the same style and format that TestMAX ATPG uses to write ATPG patterns. All scan chains, clocks, and primary input constraints must match the usage in the patterns. The load_unload, Shift, and other test procedures must be consistent with the patterns.

- *Nonscan functional patterns* - These patterns have no recognizable structure and do not contain procedures of a standard scan pattern. Nonscan patterns can exercise scan chains, can be completely functional, or can perform a combination of scan chain and functional testing. They must use a simple, sequential application of input stimulus and output measures and they must not define scan chains or any ATPG-related procedures (for example, load_unload or Shift).

If the functional nonscan patterns do not contain timing information, you can use a STIL procedure file to define pin timing, and reference the STL procedure file using the `run_drc` command or the Run DRC dialog box. The following steps describe this process:

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 might be necessary to remove `add_clocks` commands.

2. Switch to TEST mode without the use of an STL procedure file. Typically, you must change the severity of many of the rules from their defaults to either warning or ignore.
3. After 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.

STIL Functional Pattern Input

TestMAX ATPG accepts pattern input in STIL format using some limited variations of the example shown in the following example.

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.

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;

    vcoct1 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 + full1 + pl + map + vect + sdo1 + sdo2 + tout
                  + vcoct1';
}

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;
    vcoct1=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; full1=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

TestMAX ATPG accepts pattern input in Verilog format using some limited variations of the example shown in the following example.

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 TestMAX ATPG to track the pattern boundaries.

Example 2 Functional Pattern Input in Verilog

```
`timescale 1 ns / 100 ps
module amd2910_test;
    reg [0:8*9] POnames [19:0];
    integer nofails, bit, pattern;
    wire [11:0] d;
    wire [3:0] i;
    wire oe, tsel, ci, rld, ccen, cc, sdil, 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) );

// define pulse on "i_cp"
```

```
event pulse_i_cp;
always @ pulse_i_cp begin
    #500 cp = 1;
    #100 cp = 0;
end

// 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 ---
--", pattern);
        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
```

```
        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;
        // --- initalize 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] = "p1"; POnames[14] = "map";
        POnames[15] = "vect"; POnames[16] = "tout"; POnames[17] =
"vcoct1";
        POnames[18] = "sdo1"; POnames[19] = "sdo2";

        #0; pattern= 0;
        se=0; sdi2=0; sdi1=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'b00000000000000000000;
        ->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'b00001111111111111111;
->capture_i_cp;

#1000; pattern= 4;
d=12'b0000000000010; XPCT=20'bXXXX10110000000000010;
MASK=20'b00001111111111111111;
->capture_i_cp;

#1000; pattern= 5;
d=12'b000000000100; XPCT=20'bXXXX1011000000000100;
MASK=20'b00001111111111111111;
->capture_i_cp;

#1000;
$display("Simulation of %0d cycles completed with %0d errors",
        pattern, nofails );
$finish;
end
endmodule
```


WGL Functional Pattern Input

TestMAX ATPG accepts pattern input in WGL format using some limited variations of the example shown in the following example.

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 3 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];
```

Chapter 3: ATPG Design Flow

Preparing for ATPG

```
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 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 X X X ] (500nS);

vector(2, 1uS, 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 ] (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 1 0 0 0 1 X 0 Z 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 1 0 0 0 1 X 0 Z Z 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 1 0 0 0 1 X 0 Z 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 1 0 0 0 1 X 0 Z Z 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 1 0 0 0 1 X 0 Z 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 1 0 0 0 1 X 0 Z Z 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 1 0 0 0 1 X 0 Z Z
Z Z Z Z Z Z Z Z Z Z ] (5uS);

end

end
    
```

VCDE Functional Pattern Input

TestMAX ATPG can read 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.

Creating a VCDE data file is fairly simple for most Verilog simulators. You need to 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`.

If you want to generate a VCDE file from a TestMAX ATPG Verilog testbench, you can use the `+define+tmx_vcde` variable to help generate that file. Do this by adding the `+define+tmx_vcde` variable to your VCS command line when you simulate the TestMAX ATPG-generated Verilog testbench. A VCDE file called `sim_vcde.out` is automatically created.

Do not 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 the following 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. TestMAX ATPG 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 TestMAX ATPG, when you read in VCDE patterns, you specify the cycle period and measure points within each cycle. TestMAX ATPG uses this information to construct internal measure points and expected data. For more information, see ["Specifying Strokes for VCDE Pattern Input."](#)

Options for Selecting the Pattern Source

You can select the pattern source using the `set_patterns` command, the Set Patterns dialog box, or the Run ATPG dialog box. In addition, you can specify various options that affect how TestMAX ATPG uses the patterns.

The following examples show how to use the `set_patterns` command to specify the pattern source:

- To use internal patterns, specify the `-internal` option, as shown in the following example:

```
TEST-T> set_patterns -internal
```

- To use external patterns, specify the `-external` option and the name of the file containing the patterns (in the following example, the file name is `b010.vi1`). You can also use the `-append` option to append the external patterns to any existing internal patterns, and use the `-load_summary` option to enable the `report_summaries` command to display the total number of scan loads used by the basic-scan and fast-sequential patterns, as shown in the following example:

```
TEST-T> set_patterns -external b010.vi1 -append -load_summary
```

- To use random patterns, do the following:
 1. Specify the parameters for defining the random patterns using the `set_random_patterns` command, as shown in the following example:

```
TEST-T> set_random_patterns -length 3000 -observe_master
```
 2. Specify the `-random` option of the `set_patterns` command, as shown in the following example:

```
TEST-T> set_patterns -random
```

The Set Patterns dialog box generally uses the same options specified by the `set_patterns` command. To select the pattern source using the Set Patterns dialog box:

1. From the menu bar, choose Patterns > Set Pattern Options.

The Set Patterns dialog box appears.
2. Do one of the following:
 - To select internal patterns as the pattern source, click the Internal button in Pattern source section.
 - To select external patterns, do the following:
 - a. Click the External button in the Pattern source section.
 - b. Enter the pattern file name you want to use as the pattern source, or use the Browse button to navigate and select the file.
 - To select random patterns, click the Random button.
3. Select or enter any applicable options in the Set Patterns dialog box.
4. Click OK.

To select the pattern source using the Run ATPG dialog box:

1. Click the ATPG button in the command toolbar.

The Run ATPG dialog box appears.

2. Do one of the following:
 - To select internal patterns as the pattern source, click the Internal button in Pattern source section.
 - To select external patterns, do the following:
 - a. Click the External button in the Pattern source section.
 - b. Enter the pattern file name you want to use as the pattern source, or use the Browse button to navigate and select the file.
 - To select random patterns, click the Random button.
 3. Select or enter any applicable options in the Set Patterns dialog box.
 4. Click OK.
-

Specifying the ATPG Mode

TestMAX ATPG can use three different modes when performing pattern generation. Each mode provides different types and levels of optimization. Since ATPG normally requires multiple runs, the mode you select depends on your particular pattern generation goals and where you are in the ATPG process.

TestMAX ATPG supports the following ATPG modes:

- *Basic Scan Mode* - This is the default mode for TestMAX ATPG, and is usually the first mode you run. It enables TestMAX ATPG to operate as a full-scan, combinational-only ATPG tool. To get high test coverage, the sequential elements must be scan elements.
- *Fast-Sequential Mode* - This mode provides limited support for partial-scan designs, and accommodates multiple capture procedures between scan load and scan unload. Fast-sequential mode allows data to be propagated through nonscan sequential elements in the design, such as functional latches, nonscan flops, and RAMs and ROMs. However, all clock and reset signals to these nonscan elements must be directly controllable at the primary inputs of the device.
- *Full-Sequential Mode* - This mode supports multiple capture cycles between scan load and unload, which increases 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.

Each mode is described in more detail in the following sections:

- [Basic Scan Mode Settings](#)
- [Fast-Sequential Mode Settings](#)
- [Setting Full-Sequential Mode](#)

Basic Scan Mode Settings

The basic scan mode is the default mode for running ATPG. This mode uses the combinational ATPG method, which tests the individual nodes (or flip-flops) of a logic circuit without concern to the overall operation of the circuit. During test, basic-scan mode forces a simplified connection of flip-flops that effectively bypasses their normal interconnections. This allows TestMAX ATPG to use a relatively simple vector matrix to quickly test all the comprising flip-flops and to trace failures to specific flip-flops.

You can use the `set_atpg` command or the Run ATPG dialog box to set several options specific to basic-scan mode. For example, you can do the following:

- Specify the `-abort_limit` option to set the maximum number of remade decisions before terminating a basic-scan test generation effort.
- Specify the `-resim_atpg_patterns` option to enable and disable the resimulation of patterns generated by basic-scan ATPG to increase the robustness of patterns.

The following example shows how to specify both options:

```
TEST-T> set_atpg -abort_limit 8 -resim_atpg_patterns nofault_sim
```

To perform these same tasks using the Run ATPG dialog box:

1. Do one of the following:
 - Select Run > Run ATPG from the command menu
 - Click the ATPG button in the command toolbar.

In both cases, the Run ATPG dialog box appears.

2. Click the Basic Scan Settings tab.
3. Enter 8 in the Abort limit field, and select Mask in the drop-down menu of the Resim basic scan patterns field.
4. Click the Set button.

After making the appropriate settings, you can run ATPG in basic scan mode. For details on this process, see [Running TestMAX ATPG in Basic Scan, Fast-Sequential, or Full-Sequential Mode](#).

Fast-Sequential Mode Settings

Fast-sequential ATPG provides limited support for partial-scan designs (designs containing some nonscan sequential elements). This mode is particularly useful when there are AU (ATPG Undetectable) faults remaining after you run ATPG in basic-scan mode.

You can use the `-capture_cycles` option of the `set_atpg` command to specify an integer between 2 and 10 . This specification sets the level of effort used by the ATPG algorithm based on the number of capture procedures allowed between scan load and unload.

You should not set the `-capture_cycles` option value too high since it can cause excessive runtimes. In most cases, you should use a starting value of 4 (the default), and generate an initial set of patterns. You can then incrementally increase the value following each pattern generation until you achieve your required coverage.

You can use the `sequential_depths` options of the `report_summaries` command to identify the maximum depth for controlling, observing , and detecting faults, as shown in the following example:

```
TEST> report_summaries sequential_depths
type           depth    gate_id
-----
Control       1         21569
Observe       2         6866
Detect        3         6859
```

Based on this report, to obtain optimal runtime you should set the `-capture_cycles` option to 3 as shown in the following example:

```
TEST> set_atpg -capture_cycles 3
```

For optimal coverage, set the `-capture_cycles` option to 10:

```
TEST> set_atpg -capture_cycles 10
```

To perform these same tasks using the TestMAX ATPG GUI:

1. Do one of the following:
 - Select Report > Report Summaries from the command menu
 - Click the Summary button in the command toolbar.In both cases, the Report Summaries dialog box appears.
2. Select the Sequential depths button.
3. In the Output to section, select either the Report window, Transcript, or File.
4. Click OK.

The Report Summaries dialog prints a report that shows the sequential depths.

5. Do one of the following:
 - Select Run > Run ATPG from the command menu.
 - Click the ATPG button in the command toolbar.

In both cases, the Run ATPG dialog box appears.

6. Click the Fast Sequential Settings tab.
7. Enter a value in the Capture cycle field (for example, enter 4).
8. Click the Set button.

Setting Full-Sequential Mode

Full-sequential ATPG supports multiple capture cycles between scan load and unload, and supports RAM and ROM models, which increases the test coverage in partial-scan designs (similar to fast-sequential ATPG). However, in full-sequential mode, 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.

To enable TestMAX ATPG to use the full-sequential mode, specify the `-full_seq_atpg` option of the `set_atpg` command, as shown in the following example:

```
TEST-T> set_atpg -full_seq_atpg
```

Full-sequential mode supports a feature called *sequential capture*. If you define a sequential capture procedure in the STIL procedure file, you can 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, in which CLKP1 is followed by CLKP2. Otherwise, the tool creates its own sequence of clocks and other signals to target the as-yet-undetected faults in the design. For more information, see [Defining a Sequential Capture Procedure](#).

The following limitations apply to full-sequential ATPG:

- It supports stuck-at faults, transition faults, and path delay faults, but not IDDQ 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 are not implemented for Full-Sequential simulation:

```
-bidi_fill | -strong_bidi_fill -measure <sim|pat> -oscillation
```

Running ATPG

ATPG generates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. You use these patterns to test devices and to determine the cause of failure. Before running ATPG, make sure you have completed the recommended processes described in [Preparing for ATPG](#).

Basic scan mode (the default) is usually the first mode first you run in the ATPG process, followed by fast-sequential mode. For detailed descriptions of these modes, see [ATPG Modes](#).

After running ATPG, you can review a set of output reports that provide coverage information on primitives, faults, patterns, library cells, memories, and other data relevant to ATPG. Based on these reports, you can make incremental adjustments to meet your ATPG goals, such as obtaining a good balance between pattern compaction and execution speed.

The following sections describe how to run ATPG:

- [Running ATPG in Basic Scan or Fast-Sequential Mode](#)
- [Using Automatic Mode to Generate Optimized Patterns](#)
- [Quickly Estimating Test Coverage](#)
- [Specifying a Test Coverage Target Value](#)
- [Increasing ATPG Effort Over Multiple Passes](#)
- [Multiple Session Test Pattern Generation](#)
- [Compressing Patterns](#)

You can also set a variety of optimization parameters for running ATPG. For details on these settings, see [Optimizing ATPG](#).

Running ATPG in Basic Scan or Fast-Sequential Mode

You can run ATPG using either the `run_atpg` command or the Run ATPG dialog box. This topic explains how to run ATPG in the basic scan or fast-sequential modes. You can also run ATPG in automatic mode, which automatically selects the best settings and algorithms to provide reasonably good results. For details on automatic mode, see [Using Automatic Mode to Generate Optimized Patterns](#).

To run ATPG using the basic scan mode (the default), specify the `run_atpg` command without any options. This mode uses default two-clock transition ATPG when running distributed ATPG for system clock transition, and is usually the first mode you use during the ATPG process. The following example runs ATPG in basic scan mode:

```
TEST-T> run_atpg
```

For information on specifying settings for basic scan mode, see [Basic Scan Mode Settings](#).

You can run ATPG in fast-sequential mode using the `fast_sequential_only` option of the `run_atpg` command. This mode provides limited support for partial-scan designs, and accommodates multiple capture procedures between scan load and scan unload. The following example runs ATPG in fast-sequential mode:

```
TEST-T> run_atpg fast_sequential_only
```

You can also use the `-capture_cycles` option of the `set_atpg` command to set a level of effort used by fast-sequential mode. For information on specifying settings for fast-sequential mode, see [Fast-Sequential Mode Settings](#).

To use the Run ATPG dialog box to specify the ATPG mode and start the ATPG process, do the following:

1. Do one of the following:

- Select ATPG > Run ATPG
- Click the ATPG button

In both cases, the Run ATPG dialog box appears.

2. On the right side of the Run ATPG dialog box, click the button associated with the ATPG mode you want to run:

- To run basic scan mode, click the Basic Scan button.
- To run fast-sequential mode, click the Fast-Seq button.

Using Automatic Mode to Generate Optimized Patterns

You can specify TestMAX ATPG to use an automatic mode that optimally generates compact sets of ATPG patterns. This mode automatically selects the best settings and algorithms to provide reasonably good results. Automatic mode is a good starting point for most ATPG flows. Although this mode uses a set of default parameters, you can still make manual adjustments as necessary.

Automatic pattern compression uses a combination of algorithms to achieve optimal results: a fast test generation algorithm that results in a lower pattern count and a secondary algorithm that produces excellent fault detection results in a slower runtime.

TestMAX ATPG performs the following tasks in automatic mode:

- *Fault Population*

If there is no existing fault population, TestMAX ATPG automatically populates a fault list (the same as running the `add_faults -all` command). If a fault population exists, the faults are left undisturbed and used for the remainder of the automatic mode process. For more information, on setting the fault population, see [Specifying Fault Lists](#).

- *Pattern Source*

Internal patterns are used as the pattern source (the default). For more information on internal patterns, see [Setting the Pattern Source](#).

- *Pattern Generation*

TestMAX ATPG automatically uses basic-scan mode (the default) to generate an initial set of patterns. All patterns are stored and dynamic merge is enabled. The merge effort is automatically set to high, unless you have set the merge parameter to some other value. TestMAX ATPG adheres to any other `set_atpg` command settings you specified (see [Specifying General ATPG Settings](#) for details). If you set the `-capture_cycles` option of the `set_atpg` command to a value greater than 1, fast-sequential ATPG is performed after basic-scan ATPG. Also, if you set the `-full_seq_atpg` option of the `set_atpg` command, full-sequential ATPG is performed after basic-scan ATPG or fast-sequential ATPG.

- *Reports*

After TestMAX ATPG generates the patterns, it produces a fault summaries report, a test coverage report, and a pattern count report. In addition, the total CPU time is reported.

- *Restoration*

After completing the automatic mode process, TestMAX ATPG restores all settings to their original values.

Setting Automatic Mode

To run ATPG in automatic mode:

1. Use the `set_atpg` command or the Run ATPG Dialog box to select the ATPG abort limit, ATPG verbose mode, and the ATPG merge effort, if necessary. You can also create a non-default fault population, or you can use defaults for any or all of these settings. For more information, see [Specifying General ATPG Settings](#).
2. Do one of the following to initiate automatic mode:
 - Specify the `-auto_compression` option of the `run_atpg` command, as shown in the following example:

```
run_atpg -auto_compression
```
 - Use the Run ATPG dialog box, as shown in the following steps:
 - a. Do one of the following:
 - Select ATPG > Run ATPG
 - Click the ATPG buttonIn both cases, the Run ATPG dialog box appears.
 - b. Click the Auto button.

Note the following:

- Multiple fault sensitization is only available if you use the `-auto_compression` option.
- You can use the `-optimize_patterns` option of the `run_atpg` command to produce a very compact set of patterns with high test coverage. The trade-off is a longer runtime. For details, see [Optimizing Patterns during the run_atpg Process](#).

Quickly Estimating Test Coverage

You can quickly estimate the final test coverage by setting a low abort limit and low merge effort before running ATPG.

To quickly estimate coverage, use the `-abort_limit` and `-merge` option of the

`set_atpg` command, as shown in the following example:

```
TEST-T> set_atpg -abort_limit 5 -merge off
TEST-T> run_atpg
```

To estimate coverage using the Run ATPG dialog box:

1. Select ATPG > Run ATPG or click the ATPG button in the command toolbar.
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 the description of the `set_atpg` and `run_atpg` commands in TestMAX ATPG Help.
6. Click Run.

Examples

The following example 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.

Run ATPG Transcript, Merge Effort Turned Off

```
TEST-T> set_atpg -abort 5 -merge off
TEST-T> run_atpg
ATPG performed for 71800 faults using internal
pattern source.
```

```
-----
```

#patterns	#faults		
stored	detect/active	coverage	process CPU time
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			

```
-----
```

Chapter 3: ATPG Design Flow

Running ATPG

```

    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-T>

```

For comparison, the following example shows a transcript from an ATPG run on the same design with the merge effort set to high.

Run ATPG Transcript, Merge Effort Set to High

```

TEST-T> set_atpg -abort 5 -merge high
TEST-T> 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...

```


Chapter 3: ATPG Design Flow
Running ATPG

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	

Chapter 3: ATPG Design Flow

Running ATPG

```

    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-T>

```

The columns in the Run ATPG transcript are described as follows:

- `#patterns stored` – The total cumulative number of stored patterns (patterns that TestMAX ATPG 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 of 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.

Specifying a Test Coverage Target Value

By default, TestMAX ATPG 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 example:

```
TEST-T> set_atpg -coverage 88.5
```

To specify the test coverage target for specific partitions, use the `-coverage` option of the `set_atpg` command with the `-partition` option. For example:

```
TEST-T> set_atpg -coverage 90 -partition {p1 p2}
```

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.

Increasing ATPG 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 required results.

The following example sequence shows how to specify these settings:

```
TEST-T> set_atpg -abort_limit 10 -merge off
TEST-T> run_atpg
TEST-T> set_atpg -abort 50
TEST-T> run_atpg
TEST-T> set_atpg -abort 250
TEST-T> run_atpg
```

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

Multiple Session Test Pattern 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](#).

The following examples describe 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.

These examples are explained in more detail in the following sections:

- [Splitting Patterns](#)
- [Extracting a Pattern Sub-Range](#)
- [Merging Multiple Pattern Files](#)
- [Using Pattern Files Generated Separately](#)

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 example:

```
TEST-T> set_patterns -external session_1_patterns
TEST-T> write_patterns pat_file1 -last 499 -external
```

```
TEST-T> write_patterns pat_file2 -first 500 -external
```

Extracting a Pattern Sub-Range

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 example:

```
TEST-T> set_patterns -external session_1_patterns
```

```
TEST-T> write_patterns 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 example:

```
TEST-T> set_patterns -external patterns_1
```

```
TEST-T> run_atpg
```

```
TEST-T> set_patterns -external patterns_2
```

```
TEST-T> run_atpg
```

```
TEST-T> set_patterns -external patterns_3
```

```
TEST-T> run_atpg
```

```
TEST-T> 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:

```
TEST-T> set_patterns -delete
```

```
TEST-T> set_patterns -external patterns_1
```

```
TEST-T> run_simulation -store
```

```
TEST-T> set_patterns -external patterns_2
```

```
TEST-T> run_simulation -store
```

```
TEST-T> set_patterns -external patterns_3  
TEST-T> run_simulation -store  
TEST-T> 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 in the following manner:

- **Session 1:** You create an STL procedure file 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-T> write_faults sess1_faults.gz -all -uncollapsed -compress gzip
```

- **Session 2:** You create an STL procedure file that defines all scan chains. You read in the fault list saved in Session 1, as in the following command:

```
TEST-T> 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*250) + (300*400) = 495,000 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.

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.

The following sections describe the process for compressing patterns:

- [Balancing Pattern Compaction and CPU Runtime](#)
- [Compression Reports](#)

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 of 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](#)). 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. Specify the `run_atpg -auto_compression` command.
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

The following example shows a dynamic compression report generated using the `-verbose` option of the `set_atpg` command. The `-verbose` option produces the following 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.

Verbose Dynamic Compression Report

```
TEST-T> set_atpg -patterns 150 -merge medium -verbose
```

```
TEST-T> 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
```

Analyzing ATPG Output

You can analyze ATPG pattern generation output from the `run_atpg` command. This output includes the following formats:

- [Standard Format](#)
- [Expert Format](#)
- [Verbose Format with Merge Without -auto_compression](#)
- [Verbose Format with Merge Without -auto_compression](#)

Standard Format

```
TEST> run_atpg
ATPG performed for 72436 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          49465    22971   0/0/1          70.05%    6.50
```

64	6808	16163	0/0/3	77.82%	10.52
96	3779	12380	1/1/4	82.13%	13.48
128	2220	10156	2/2/6	84.66%	16.02
160	1264	8890	4/2/7	86.11%	18.54
192	1415	7474	4/3/11	87.73%	20.87
224	1021	6450	6/4/13	88.89%	23.04
256	835	5610	9/6/17	89.85%	25.17
288	722	4881	13/8/19	90.68%	27.20
320	653	4223	15/11/21	91.43%	29.16
352	572	3648	16/13/26	92.08%	31.15
:	:	:	:	:	:
831	78	378	176/105/132	95.69%	62.35
862	73	295	184/107/142	95.78%	64.08
889	49	212	205/113/143	95.87%	65.35

#patterns stored

This indicates the current number of patterns which are stored in the internal pattern set. These patterns were created during the ATPG process and are selected only if they are required for fault detection.

#faults detect/active

The first field indicates the number of faults that were detected in the current simulation pass. The second field indicates the number of faults that still remain active in the fault list. Depending on the fault-report setting, the fault counts are either uncollapsed (default) or collapsed.

#ATPG faults red/au/abort

The first field indicates the cumulative number of faults identified as redundant in the current ATPG process. The second field indicates the cumulative number of faults identified as ATPG untestable in the current ATPG process. The third field indicates the cumulative number of faults that were aborted in the current ATPG process. All of these fault counts are collapsed fault counts.

test coverage

This indicates the current value of the test coverage considering the current fault list and patterns previously evaluated. There is a user selectable credit given for possible-detected faults (default 50%) and ATPG untestable faults (default 0%). Depending on the fault-report setting, the test coverage is calculated using fault counts which are either uncollapsed (default) or collapsed.

process CPU time

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

Expert Format

```

TEST> run_atpg
ATPG performed for stuck fault model using internal pattern    source.

Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns    #faults    #ATPG faults test    process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=72346,
abort_limit=10...
32      32      32      49273 23072      1/0/1      69.89%      7.40
64      32      64      6890 16182      1/0/2      77.75%      11.59
96      32      96      3233 12948      2/0/6      81.45%      15.06
128     32 128      2295 10651      3/1/7      84.07%      18.04
160     32 160      1986 8662       4/2/7      86.33%      20.80
192     32      192     1256 7403       6/3/7      87.77%      23.37
224     32      224     971   6429      8/4/8      88.88%
25.76
256     32 256      842 5583      10/6/10     89.84%
28.12
288     32 288      702 4875      14/7/12     90.65%
30.44
320     32      320     639 4235      14/8/13     91.38%      32.73
352     32      352     514 3718      16/9/16     91.97%      35.02
:       :       :       :       :       :       :
:
832     32      830     80   294      143/92/58   95.78%      68.51
864     32      862     59   212      163/93/65   95.87%      70.85
896     32      894     58   133      179/94/70   95.96%      72.91
909     13      907     29    83      197/96/70   96.01%      73.72

Begin fast-seq ATPG: #uncollapsed_faults=179, abort_limit=10,
depth=4...
910     1      908     5    174      0/0/9      96.02%
73.87
911     1      909     1    172      0/1/38     96.02%
74.43

```

```

    912      1    910      1    171    0/1/47    96.02%
74.61
    913      1    911      2    169    0/1/48    96.02%
74.67

```

This form is generated when `set messages-level expert` is in effect.

#patterns eff/total

This report is identical to the Standard Form with the exception that an additional information appears as the 2nd and 3rd columns. The 2nd column is the number of patterns in the current working group of 32 which were effective and kept. The 3rd column is the cumulative total number of patterns kept.

Verbose Format with Merge (without -auto_compression)

```

run_atpg
ATPG performed for stuck fault model using internal pattern
source.
Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns #faults #ATPG faults test process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=266012,
abort_limit=10...
Patn 1: #merges=537 #failed_merges=20 #faults=154875 #det=8693
CPU=1.14 sec
Patn 2: #merges=316 #failed_merges=20 #faults=124914 #det=53161
CPU=1.71sec
Patn 3: #merges=256 #failed_merges=20 #faults=107012 #det=29741
CPU=2.19sec
Patn 4: #merges=83 #failed_merges=20 #faults=95837
#det=17827
CPU=2.48sec
Patn 5: #merges=235 #failed_merges=20 #faults=85826
#det=15586
CPU=2.92sec
.....
Patn 28: #merges=40 #failed_merges=20 #faults=34518
#det=1181
CPU=9.10 sec
Patn 29: #merges=44 #failed_merges=20 #faults=33872
#det=959
CPU=9.28 sec
Patn 30: #merges=56 #failed_merges=20 #faults=33223
#det=1009
CPU=9.47 sec
Patn 31: #merges=32 #failed_merges=20 #faults=32730 #det=829
CPU=9.63 sec
32 32

```

```

32      210799      55209
      2/0/0      80.42%      10.23
Patn 0: #merges=43 #failed_merges=20 #faults=32127
      #det=1179 CPU=10.39sec
Patn 1: #merges=28 #failed_merges=20 #faults=31515 #det=1059
CPU=10.54 sec
.....
Patn 31: #merges=33 #failed_merges=20 #faults=21284
      #det=353
      CPU=15.18 sec
64      32
      64 18839 36366
      4/0/0      86.43%      15.43
Patn 0: #merges=18 #failed_merges=20 #faults=21145
      #det=232
      CPU=15.55 sec
.....
Patn 31: #merges=32 #failed_merges=20 #faults=15576
      #det=225
      CPU=19.81 sec
96      32
      96 9508 26846
      7/0/2      89.47%      20.02

```

This form is generated when `set_atpg -verbose -merge` is in effect.

#merges

This indicates the number of additional patterns merged with the original pattern. Each pattern successfully detects a fault on at least one target fault (primary fault). The combined pattern can also detect additional faults (secondary faults). A merge count of 10 means the single pattern is doing the work of 11 patterns and it will detect at least the 11 target faults (primary faults) and might also detect many more faults that were not the original targets (secondary faults).

#failed_merges

This indicates the number of faults for which a pattern was generated but that new pattern could not be merged with the existing pattern for the primary fault site. When the count is less than the merge limit (low=20, medium=100, high=500) then TestMAX ATPG ran out of active faults/patterns to merge into the primary pattern before it reached the iteration limit. When the count is equal to the merge limit, then the limit was reached and there were still faults that could have attempted to be merged.

When you see the merge limit being reached over and over again, there can be value in increasing the merge effort using the `-merge` option of the `set_atpg` command. However, this increase in merge effort will come at a cost of additional CPU time.

When the failed merge count is consistently less than the limit on each pattern attempt, then the optimal setting for the merge effort is just higher than the maximum failure count.

If you assume in the previous example that the merge effort was 300, then the majority of patterns showed a `#failed_merges` count less than 300 and this value is reasonably good. Increasing the merge effort to 400 or 500 can improve the number of patterns merged for patterns 0, 1, and 4 in the first group of 32, but at a cost of increased runtime. The optimal value for merge effort often requires repeated ATPG runs and seeking the optimal value can often take more time is efficient. One shortcut approach is to set the merge effort high, say 3000, and then watch the progress for the first 32 patterns and then stop the run. Using the information learned in the first 32 patterns to set the merge effort for a more complete run. When `-auto_compression` is not used, only the first parameter of `set_atpg -merge` is in effect.

#faults

If single-pattern fault simulation is performed, this number represents the number of faults detected by the pattern. If single-pattern fault simulation is not performed, this number represents the number of faults targeted by the test generator (that is, primary, secondary and side-path detection faults). In either case, fault simulation at the end of the interval ultimately decides which faults are truly detected and which are not.

A heuristic algorithm is used to decide whether or not to perform single-pattern fault simulation. This algorithm attempts to simultaneously maximize coverage and minimize pattern count and CPU time.

In some cases, single-pattern fault simulation is performed on some patterns in an interval (thus the `#faults` can be high). But simulation may not be performed on other patterns (thus the `#faults` is low). This does not indicate incorrect behavior and is not a cause for concern.

CPU=

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

Verbose Format with Merge and -auto_compression

```
run_atpg -auto_compression
  ATPG performed for stuck fault model using internal pattern
source.
  Fast-seq simulation is used to verify Basic-Scan patterns.
-----
#patterns #patterns      #faults      #ATPG faults test      process
```

```
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=3199364,
abort_limit=10...
Patn 1: #merges=0/922(0%) #failed_merges=0/34 #faults=1783833
#det=14023 CPU=32.58 sec clocks=

Patn 2: #merges=0/86808(3%) #failed_merges=0/3484 #faults=1435411
#det=642019 CPU=58.32 sec clocks= cclk pclk

Patn 3: #merges=0/46986(0%) #failed_merges=0/125 #faults=1366319
#det=101465 CPU=81.48 sec clocks= cclk crst_ pclk
.....
Patn 31: #merges=0/2110(0%) #failed_merges=0/272 #faults=411938
#det=12251 CPU=324.24 sec clocks= pclk
Warning: 3 (4) basic-scan patterns failed current pass simulation
check and is treated as ignored measures. (M212)
32 32 32 2492119 707234 2/4/6 77.30% 362.76
Local redundancy analysis results: #redundant_faults=4398,
CPU_time=3.00 sec

Patn 0: #merges=0/1761(0%) #failed_merges=0/242 #faults=400847
#det=9765 CPU=370.68 sec clocks= cclk pclk
.....
Patn 31: #merges=0/869(0%) #failed_merges=0/62 #faults=276129
#det=3269 CPU=484.81 sec clocks= ZXIN ZADCK
Warning: 1 (1) basic-scan patterns failed current pass simulation
check and is treated as ignored measures. (M212)
64 32 64 238165 462910 3/6/10 83.51% 499.63

Patn 0: #merges=0/1437(0%) #failed_merges=0/231 #faults=272195
#det=6860 CPU=502.74 sec clocks= inclk zxin ad[0]

Patn 1: #merges=0/1253(0%) #failed_merges=0/155 #faults=268677
#det=6429 CPU=505.99 sec clocks= clk inclk crst_ pclk
.....
Patn 31: #merges=0/568(0%) #failed_merges=0/39 #faults=221584
#det=2037 CPU=587.17 sec clocks= inclk
```

This form is generated when `set_atpg -verbose -merge` is in effect.

`#merges=d1/d2(d3%)`

This indicates the number of additional patterns merged with the original pattern. Each pattern successfully detects a fault on at least one target fault (primary fault). The combined pattern can also detect additional faults (secondary faults). A merge count of 10 means the single pattern is doing the work of 11 patterns and it will detect at least the 11 target faults (primary faults) and may also detect many more faults that were not the original targets.

o d1 is number of secondary faults detected and merged into the pattern.

o d2 is number of faults detected and merged through multiple fault sensitization.

o d3 is the percentage of of multiple fault sensitization merges that did not detect any faults (d2 and d3 are printed only when using `-auto_compression` and verbose mode is turned on)

The first and second values passed to the `set_atpg -merge` command control the secondary fault merge effort and multiple fault sensitization merge effort, respectively.

`#failed_merges=d4/d5`

Where d4 is the number of failed merges of secondary faults and d5 is the number of failed merges of multiple fault sensitization (d5 is printed only when `-auto_compression` is used and verbose mode is turned on)

`#faults`

This indicates the calculated number of collapsed faults that are still active in the fault list.

`#detects`

This indicates the number detected faults.

`CPU=`

This indicates the cumulative number of CPU seconds that have been used up to this point in the current ATPG process.

`clock=s`

This is a list of clocks pulsed during the capture cycle. With dynamic clock grouping, you can have multiple clocks pulsing together in the same capture cycle, which results in a considerable reduction in the pattern count. This field is printed only when `-auto_compression` is used.

Note: For d3, `#faults` and `#detects`, you might sometimes see "---". When the design is large and only multiple fault sensitization is in progress, it is more efficient and productive to run fault simulation at the end of an interval (that is, 32 patterns). For these conditions, because each pattern is not fault simulated as soon as it is generated, some information required in verbose messages is not available.

Reviewing Test Coverage

You can view the results of the test coverage and the number of patterns generated using the `report_summaries` command or the Report Summaries dialog box.

The following example shows how to generate a fault summary report using the `report_summaries` command:

```
TEST-T> report_summaries
```

For the complete syntax and option descriptions, see the description of the `report_summaries` command in TestMAX ATPG Help.

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

1. From the command toolbar, click the Summary button. The Report Summaries dialog box appears.
2. Select the appropriate summary settings.

For details about available settings, see the description of the `report_summaries` command in TestMAX ATPG Help.

3. Click OK.

An example output report showing the fault counts and the test coverage obtained by using the uncollapsed fault list is shown in the following example. A detailed description of each fault class is shown in [Fault Lists and Faults](#).

Example 3: Uncollapsed Fault Summary Report

```
TEST-T> 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
```

The following example shows the same report with collapsed fault reporting. Notice that there are fewer total faults, and fewer individual fault categories.

Collapsed Fault Summary Report

```
TEST-T> set_faults -report collapsed
TEST-T> 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 “[Analyzing the Cause of Low Test Coverage](#)”

Writing ATPG Patterns

TestMAX ATPG can write pattern files in binary, STIL, and WGL. By default, TestMAX ATPG generates new internal patterns. To save the test patterns, you can use the `write_patterns` command or the Write Patterns dialog box.

For information on translating adaptive scan patterns into normal scan-mode patterns, see [Reading Pattern Files](#).

By default, TestMAX ATPG writes parallel patterns in the unified STIL flow format when the `-format` option of the `write_patterns` command is specified with the `stil` or `stil99` arguments.

The following examples show how to use the `write_patterns` command to write serial STIL patterns:

```
write_patterns patterns.stil -serial -format stil
```

The following example writes patterns in a proprietary binary format that can be read by TestMAX ATPG:

```
write_patterns patterns.bin -format binary -replace
```

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

1. From the command toolbar, click the Write Pat button.
The Write Patterns dialog box appears.
2. In the Pattern File Name field, enter the name of the pattern file to be written or 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.
4. Click OK.

For descriptions of all the options for writing patterns, see the description of the

`write_patterns` command in TestMAX ATPG Help.

For information on generating patterns for DFTMAX Ultra, see [Pattern Types Accepted by DFTMAX Ultra](#).

4

ATPG Modeling

This section contains the following topics:

- [Modeling Topics](#)
- [Scan Cell Models](#)
- [ATPG Simulation Primitives](#)

Modeling Topics

[ATPG Modeling Primitive Summary](#)

[TestMAX ATPG Memory Modeling](#)

[Memory Modeling Specification](#)

[RAM/ROM Modeling](#)

[Memory Data File Example](#)

[Interpreting UDP Messages](#)

[UDP Modeling Examples](#)

ATPG Modeling Primitive Summary

You can build ATPG models using Verilog, EDIF, or VHDL netlists that reference ATPG modeling primitives within TestMAX ATPG. For Verilog, you can describe an ATPG model using either Verilog primitives or TestMAX ATPG primitives, or a mixture of both. After reading the netlist and building a simulation model, TestMAX ATPG converts all design modules into internal simulation primitives.

In general, there is one-to-one correspondence between ATPG modeling primitives and ATPG simulation primitives. However, some of the ATPG modeling primitives, such as WBUF and WIRE, are converted to other simulation primitives during the model build process. Also, there are some simulation primitives that do not have a corresponding ATPG modeling primitive, such as a BUS.

To view a list of simulation primitives currently used by your simulation model, use the `report_primitives -summary` command. To find out how a particular design module was converted to TestMAX ATPG simulation primitives, use the `report_modules -verbose` command.

When used in a Verilog module, all ATPG modeling primitives begin with an underscore, such as "`_AND`". When used in an EDIF netlist, the primitives are referenced starting with "&_", as in "&_AND". When used in VHDL netlists, the modeling primitives have no prefix; a component with the same name has priority over the primitive.

Most ATPG modeling primitives can accept up to 32 inputs, and have a single output, as shown in the following example:

```

AND ( in1, ..., inN, out )
BUF ( in, out )
BUSK0 inout ; # bus keeper, keeps only 0
BUSK1 inout ; # bus keeper, keeps only 1
BUSK01 inout ; # bus keeper, keeps 0 and 1
CMUX ( sel, d0, d1, out )
DFF ( set, rst, clk1, d1, [clkN, dN,...] , out )
DLAT ( set, rst, clk1, d1, [clkN, dN,...] , out )
INV ( in, out )
MUX ( sel, d0, d1, out )
NAND ( in1, ..., inN, out )
NOR ( in1, ..., inN, out )
OR ( in1, ..., inN, out )
SW ( ena, in, out )
TIE0 ( [in1, ..., inN,] out )
TIE1 ( [in1, ..., inN,] out )
TIEX ( [in1, ..., inN,] out )
TIEZ ( [in1, ..., inN,] out )
TSD ( ena, in, out )
WBUF ( in, out ) # weak buffer, drops strength
WIRE ( in1, ..., inN, out )
XNOR ( in1, ..., inN, out )
XOR ( in1, ..., inN, out )

---- RAM/ROM related (cannot be directly used) ----

ADRBUS ( An, An-1, An-2, ..., A0, addr_out_vector )
DATABUS ( Dn, Dn-1, Dn-2, ..., D0, data_out_vector )
MEMORY ( set, rst, [wclk1, wen1, addr_vec1, data_vec1,...] ,
out_vector )
RPORT ( rclk, addr_in_vec, data_in_vec, out_vector1 [,out_vectorN]... )
MOUT ( rport_out_vec, out_vector )

#
# Example Verilog module using TestMAX ATPG models
#
module FLOP (se,d,sdi,clk,rb,q,qb);
input se, d, sdi, clk, rb;
output q, qb;

```

```

_TIE0 u1 (tz);
_INV u2 (rb, rst);
_MUX u3 (se, d, sdi, n1);
_DFF u4 (tz, rst, clk, n1, q);
_INV u5 (q, qb);
endmodule
module rtranif1 (pad1, pad2, ctrl);
inout pad1, pad2; input ctrl;
rnmos n1 (pad1, pad2, ctrl);
rnmos n2 (pad2, pad1, ctrl);
endmodule

```

TestMAX ATPG Memory Modeling

TestMAX ATPG uses a limited Verilog behavioral syntax to define RAM and ROM models for ATPG. This is equivalent to defining simple RAM and ROM functional models. For information on the supported subset of the Verilog language syntax, see Memory Modeling Language Specification and RAM/ROM Examples.

To use TestMAX ATPG memory modeling, you should be familiar with the BNF (Backus-Naur Form) memory syntax description and understand its usage concepts.

TestMAX ATPG attempts to map the behavioral description of a memory with a fixed set of ATPG primitives with fixed behavior. This modeling language is a simplified way to describe a netlist of connected ATPG primitives. TestMAX ATPG does not support behavioral simulation, and many language constructs supported by Verilog are not allowed for memory modeling.

Basic Template

The following template is for a 16-word by 8-bit RAM. It consists of a Verilog module definition with the inputs and outputs, the output holding register `data_out`, and the memory storage array `memory`.

```

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-through

    ...memory port definitions...

endmodule

```

The basic structure of this template applies to most RAMs. However, the port list changes as you define more complicated RAMs or ROMs with multiple ports. Also, you must use bussed ports for ATPG modeling of RAMs.

If you use an `event` declaration, make sure you define it after declaring the port input/output and registers and memory.

The supported Verilog syntax is limited. You cannot use Verilog syntax to define all the read and write ports. In particular, the use of `begin` and `.. end` statements is limited. Make sure your intended usage of these statements matches the examples exactly, or do not use them.

Defining Write Ports

There are two types of memory port controls: level sensitive and edge sensitive. A level-sensitive read or write port is continuously active when the control input is asserted. The edge-sensitive read or write port is only active on an edge transition.

Edge-Sensitive Write Port

An edge-sensitive write port has a single write control input that can be active on the rising or falling edge of the input. It also has a level-sensitive, single qualifier input. This write qualifier can be either active high or low, and is an optional input control.

Here are some common examples of edge-sensitive write ports:

```
# example #1: edge sensitive, no qualifiers

always @(posedge write) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end

# example #2: rising edge sensitive, qualifier

always @(posedge write) if (CS) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end

# example #3: falling edge sensitive, qualifier

always @(negedge write) if (!CSB) begin
  memory[write_addr] = data_in;
  #0; ->WRITE_OP;
end
```

Example #2 is an edge sensitive write port with the optional write qualifier. A write occurs on a rising edge of the `write` control net as long as the `cs` input is high.

Example #3 is a falling edge sensitive write port with an active low control `CSB` as a write qualifier. Notice the use of the exclamation mark `!` to indicate that `CSB` must be zero to enable the write operation.

If you have a more complex set of write qualifier controls, add some glue logic using Verilog primitives to produce a single qualifier control which is passed to the write port definition. For example, if you want a rising edge sensitive write control with three enables, `CS`, `en2`, and `en3b`, with the first two active high and the third one active low you would do something like example #4:

```
# example #4: rising edge sensitive, multiple qualifiers

and U1 (wen, CS, en2, !en3b);

always @(posedge write) if (wen) begin
    memory[write_addr] = data_in;
    #0; ->WRITE_OP;
end
```

Level-Sensitive Write Port

A level-sensitive write port has a single write control input that can be either active high or active low. It also has the optional level sensitive write qualifier input. In addition it must also be sensitive to changes of write address or write data while the write control is asserted.

Here are some common examples of level sensitive write ports:

```
# example #5: level sensitive, no qualifiers

always @(write or write_addr or data_in) if (write) begin
    memory[write_addr] = data_in;
    #0; ->WRITE_OP;
end

# example #6: level sensitive, CS qualifier

and u1 (WEN, write, CS);

always @(WEN or write_addr or data_in) if (WEN) begin
    memory[write_addr] = data_in;
    #0; ->WRITE_OP;
end

# example #7: active low control, active low qualifier

and u1 (WEN, !write, !CSB);

always @(WEN or write_addr or data_in) if (WEN) begin
    memory[write_addr] = data_in;
    #0; ->WRITE_OP;
end
```

Note: The first net in the `always` sensitivity list must match the net used in the following `if` clause.

Example #5 is level sensitive write port without the optional write qualifier control. A write occurs while `write` is high. If `write` is high and any change to either `write_addr` or `data_in` occurs, this will update the write operation.

Example #6 shows the use of the optional write qualifier. In this case the control net `CS` must be high for the write operation to be active.

Example #7 shows the use of the exclamation mark "!" to define an active low `write` control as well as an active low `CSB` qualifier.

As in the edge sensitive write port, if we have a more complex set of write qualifiers we support this by adding some glue logic to form a single write qualifier net. Example #8 shows a 3-term write qualifier in which `CS` and `en2` must be high, and `en3b` must be low in order for a write operation to occur while `write` is high.

```
# example #8: level sensitive, multiple qualifiers
and U1 (WEN, write, CS, en2, !en3b);

always @(WEN or write_addr or data_in) if (WEN) begin
    memory[write_addr] = data_in;
    #0; ->WRITE_OP;
end
```

Defining Read Ports

Read ports can be either edge sensitive or level sensitive, with a user-selected polarity. Like the write ports, there is support for a single read clock qualifier.

A read port which is edge sensitive is created with DFF primitives on the data outputs of the ATPG Memory primitive. This implementation means that for a simultaneous write/read the data on the read port is the OLD data from the RAM by default. This behavior can be changed using the read/write contention selection.

A read port which is level sensitive is created with LATCH primitives on the data outputs of the ATPG Memory primitive. This implementation means that for a simultaneous write/read the data on the read port is the NEW data being written to the RAM.

Edge-Sensitive Read Port

An edge-sensitive read port has a single read control input that might be active on the rising or falling edge of the input. A single (optional) read clock qualifier is supported.

Here are some common examples of edge-sensitive read ports:

```
# example #9: rising edge sensitive

always @(posedge read) data_out = memory[read_addr];
```

```
# example #10: falling edge sensitive

always @(negedge read) begin
  data_out = memory[read_addr];
end
```

Example #9 is an edge sensitive read port for which the read occurs on the rising edge of the `read` control net. Example #10 shows a falling edge read control.

What if a read qualifier is needed? To support this we add some external glue logic to form a single read enable control. For example, if we desire a rising edge sensitive read CLK with two enables, `CS`, and `en2b`, with the first active high and the second active low we would do something similar to example #11:

```
# example #11: rising edge sensitive, multiple qualifiers

and U1 (REN, CS, !en2b);

always @(posedge CLK) if (REN) data_out = memory[read_addr];
```

Level-Sensitive Read Port

A level-sensitive read port has a single read control input that can be either active high or active low. In addition it must also be sensitive to changes of read address as well as any write or set/reset operation that can change the data being read.

Here are some common examples of level-sensitive read ports:

```
# example #12: active high level sensitive

always @(read or read_addr or WRITE_OP) if (read)
  data_out = memory[read_addr];

# example #13: active low level sensitive

always @(read or read_addr or WRITE_OP) if (!read)
  data_out = memory[read_addr];
```

Note: The first net in the `always` sensitivity list must match the net used in the following `if` clause.

Example #12 describes a read port which is active when `read` is high, and which will update the value supplied to `data_out` if the `read_addr` changes or the `WRITE_OP` occurs while a read is active. Example #13 shows an active low read control.

Read Off Behavior

You can also model a RAM or ROM with a tristate output or a return-to-zero or return-to-one. For a level sensitive read port, you need to make a slight modification to the read port syntax to add an `else` clause as follows:

```
# example #14: level sensitive read with tristate outputs

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'bzzzzzzzz;

# example #15: level sensitive read, read off = zero

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'b0;

# example #16: level sensitive read, read off = ones

always @(read or read_addr or WRITE_OP)
  if ( read ) data_out = memory[read_addr]
  else data_out = 8'b11111111;
```

You can use an independent output enable for the tristate outputs by adding an additional register to hold the RAM data outputs, and passing this through a tristate function using an additional `always` clause. This `always` clause needs to be sensitive to the output enable `OEB` control as well as any changes on the data output register.

```
# example #17 : level sensitive with independent output enable

output [7:0] data_out;
reg [7:0] data_out, data_reg; /* add data_reg */

always @(read or read_addr or WRITE_OP)
  if (read) data_reg = memory[read_addr];

always @(OEB or data_reg)   if (!OEB) data_out = data_reg;   else
data_out = 8'bZZZZZZZZ;
```

In example #17 the first `always` clause defines a level sensitive read port and the second `always` defines the tristate output behavior.

An edge sensitive read port does not easily support read-off behavior or zeros or ones. However, it does support an independent output enable. An example is shown in #18.

```
# example #18 : edge sensitive with independent output enable

output [7:0] data_out;
reg [7:0] data_out, data_reg; /* add data_reg */

and u1 (RCLK, read, CS);
```

```

always @(posedge RCLK)
  data_reg = memory[read_addr];

always @(OEB or data_reg)
  if (!OEB) data_out = data_reg;
  else data_out = 8'bzzzzzzzz;

```

Complete Example

This complete example shows a simple RAM with 256x8 words, a chip select, and a single read and write port. The write port is rising edge control. The read port is level sensitive active low and has an independent tristate output control. This example uses a common address bus for both the read and write ports.

```

# example #19 : a completed RAM

module ATPG_RAM (CS, OE, read, write, data_in, data_out, addr);
  input CS, OE; # chip select, output enable
  input read, write; # read and write controls
  input [7:0] data_in; # 8 bit data width
  input [3:0] addr; # 16 words
  output [7:0] data_out; # module outputs
  reg [7:0] data_reg; # RAM outputs
  reg [7:0] data_out; # output holding register
  reg [7:0] memory [0:15]; # memory storage
  event WRITE_OP;

  and u1 (REN, !read, CS); # form read enable
  and u2 (TSO, OE, CS); # form tristate out control

  always @(posedge write) if (CS) begin
    memory[addr] = data_in;
    #0; ->WRITE_OP;
  end

  always @(REN or addr or WRITE_OP)
    if (REN) data_reg = memory[addr];

  always @(TSO or data_reg)
    if (TSO) data_out = data_reg;
    else data_out = 8'bzzzzzzzz;

endmodule

```

Memory Address Range

You do not need to use the entire $2^{**}N$ range of words. Instead, change the range definition on the `memory [0:15]` definition to correspond to the valid memory range required. For example, to use only the lower 12 words define: `reg [7:0] memory[0:11];`

Limiting the range on the memory address does not prevent the test generator from accessing the higher addresses in the ATPG vectors.

The range limitation:

1. Causes any write operation that has an address in the prohibited range to leave the memory contents unmodified
2. Causes any read operation that has an address in the prohibited range to place an X on all of the memory outputs. TestMAX ATPG reads X from the prohibited address range and ignores any write operations to the prohibited address range.

Multiple Read or Write Ports

More complex RAMS involving multiple read and write ports are supported by expanding the module port list and input/output definitions and by adding additional read and write port defining statements. The following example shows a RAM with four write ports and two read ports, all level sensitive, and with independent address, bus, and read/write controls:

```
# example #20 : a multi port RAM

module multi_port_ram (w1,a1,d1, w2,a2,d2, w3,a3,d3, w4,a4,d4,
    r5,a5,d5, r6,a6,d6);
    input w1,w2,w3,w4,r5,r6;
    input [3:0] a1,a2,a3,a4,a5,a6;
    input [7:0] d1,d2,d3,d4;
    output [7:0] d5,d6;
    reg [7:0] d5,d6;
    reg [7:0] MMY [0:15] ;
    event WRITE_OP;

    always @(w1 or a1 or d1) if (w1) begin
        MMY[a1] = d1;
        #0; ->WRITE_OP;
    end

    always @(w2 or a2 or d2) if (w2) begin
        MMY[a2] = d2;
        #0; ->WRITE_OP;
    end

    always @(w3 or a3 or d3) if (w3) begin
        MMY[a3] = d3;
        #0; ->WRITE_OP;
    end

    always @(w4 or a4 or d4) if (w4) begin
        MMY[a4] = d4;
        #0; ->WRITE_OP;
    end
end
```

```
always @(r5 or a5 or WRITE_OP) if (!r5) d5 = MMY[a5];  
always @(r6 or a6 or WRITE_OP) if (!r6) d6 = MMY[a6];  
  
endmodule
```

Rules and Limitations

The following rules and limitations apply to RAM/ROM modeling within TestMAX ATPG:

- You cannot simultaneously use both level-sensitive and edge -sensitive write ports. All write ports must be the same type. However, the read ports have no restrictions and can be mixed edge- and level-sensitive as well as different from the write ports.
- You should use bussed nets for defining data and address buses. If your RAM or ROM uses bit-blasted nets, then create a two-level hierarchical module and make sure the top level has bit-blasted nets and the lower level has bussed nets. See example #18 under RAM Modeling Examples.
- Most designs containing RAMs require that you turn on the Fast-Sequential ATPG algorithm using the `-capture_cycles` option of the `set_atpg` command to develop patterns which use the RAM.
- The supported Verilog syntax is very limited. Do not assume that you can use any legal Verilog syntax for the definition of the read and write ports. In particular, the allowed use and placement of `begin.. end` is limited to just a few areas. If you have not seen a `begin/end` usage in the examples to match what you wish to use, then it is probably not going to work. Another area of difference is that the allowed syntax is expecting `nets` or the `write_op event`, and not arbitrary expressions. Keep it simple. If you need to develop an expression for say a read or write enable term, do so using discrete Verilog `and/nand/or/nor` primitives to create a control net.
- Only a single `WRITE_OP` event is supported. If two events are defined, an N2 error is reported.

Controlling Contention Behavior

This section explains the default contention behavior on a memory device with independent read and write ports and multiple read and write ports. The following example illustrates how to explicitly define, by use of the ``define` statement, the desired contention behavior for simultaneously active read ports (`read_read`), simultaneously active read and write ports (`read_write`), and simultaneously active write ports (`write_write`). The example also illustrates the default settings.

```
# example #21 : Default contention behavior  
  
`define read_read normal  
`define read_write mixed  
`define write_write xbit
```

```
module multi_port_ram (w1,a1,d1, w2,a2,d2, r5,a5,d5, r6,a6,d6);
  input w1,w2,r5,r6;
  input [3:0] a1,a2,a5,a6;
  input [7:0] d1,d2;
  output [7:0] d5,d6;
  reg [7:0] d5,d6;
  reg [7:0] MMY [0:15] ;
  event WRITE_OP;

  always @(w1 or a1 or d1) if (w1) begin
    MMY[a1] = d1;
    #0; ->WRITE_OP;
  end

  always @(w2 or a2 or d2) if (w2) begin
    MMY[a2] = d2;
    #0; ->WRITE_OP;
  end

  always @(r5 or a5 or WRITE_OP) if (!r5) d5 = MMY[a5];

  always @(r6 or a6 or WRITE_OP) if (!r6) d6 = MMY[a6];

endmodule
`undef read_read
`undef read_write
`undef write_write
```

Read-Read Contention

The following behaviors can be specified with the `read_read` directive:

- `normal` - Simultaneously active read ports accessing the same address will return the data word at that address. This is the default behavior.
- `readx` - Simultaneously active read ports accessing the same address will return all X's for data.

Read-Write Contention

The following behaviors are specified with the `read_write` directive:

- `mixed` - For simultaneous active read and write ports to the same address, the write operation succeeds and the read operation returns the new value for level-sensitive read ports and the old value for the edge-sensitive read ports. This is the default behavior. This is the same behavior as `new` for level-sensitive read ports and `old` for edge-sensitive read ports.
- `new` - For simultaneous active read and write ports to the same address, the write operation succeeds and the read operation returns the new value. This behavior is fully supported for level-sensitive read ports, but applies only to ATPG pattern generation

of edge-sensitive read ports. Do not attempt to use a model with this behavior for fault simulation of functional patterns.

- `readx` - For simultaneous active read and write operation to the same address, the write operation succeeds, but the read operation returns all Xs for data.

`new_but_readx_across_ports` - This behavior is a combination of two previously defined behaviors: `new` and `readx`. When writing to a particular port, TestMAX ATPG considers the data on that port as newly written data. If another port simultaneously reads the same address, the data read on that particular port contains all Xs.

- `xword` - For simultaneous active read and write operation to the same address, the write operation writes Xs and the read operation returns all Xs for data.
- `xfill` - For simultaneous active read and write operation to the same address, the entire contents of the memory is set to X and the read operation returns Xs for data.

Write-Write Contention

The following behaviors can be specified with the `write_write` directive:

- `xbit` - Simultaneously active write ports to the same address will write Xs on bits of the data word which differ, and non-Xs on bits which are the same. For example, if data values of `4'b1010` and `4'b1100` were being written to the same address then the resulting stored value would be `4'b1xx0`. This is the default behavior.
- `xword` - Simultaneously active write ports to the same address will cause the entire data word addressed to be set to X.
- `xfill` - Simultaneously active write ports to the same address will cause the entire MEMORY contents to be set to X.
- `dominance` - Simultaneously active write ports to the same address will use port order to determine dominance. Port dominance is established by the order the ports are defined in the module. The ports defined last have priority over ports defined earlier. When more than one write port is active, the data value which is written is taken from the highest priority active write port. So, for example, if the first and third write port are active the value being written by the third write port is stored.
- `forbidden` - Simultaneously active write ports to the same address are forbidden and ATPG methods is employed to ensure no patterns are created where this condition exists.

The placement of ``define` within design or library source files should be carefully considered. The selected behavior specified by the use of the `'define` is applied to all memory models defined after the occurrence of these controls. To avoid unwanted or unexpected effects due to the persistence of these statements it is strongly suggested that

the `\undef` command be used immediately after each memory model to restore contention behavior to defaults. For example:

```
# example #22 : removing affect of \define

\define read_read readx
\define read_write readx
\define write_write xword
module multi_port_ram ( ...port_list...);
    :
    :
    :
endmodule
\undef read_read
\undef read_write
\undef write_write
```

Set Contention Controls

In addition to the various R/R, R/W, and W/W contention behaviors that can be defined within the RAM model, there is also an additional control the end user can select during ATPG pattern generation. This is the `RAM` option of the `set_contention` command. This option can be used to declare that having a single RAM with multiple write ports active is a type of "contention" and patterns which have this contention are to be avoided. This is a global control, and affects all RAMS in the design simultaneously.

Memory Image Initialization

Can we support initializing a RAM to specific contents for use by ATPG? Yes, this is done with either the `$readmemb()` or `$readmemh()` call placed within an `initial` block, depending upon whether the memory image file is in binary or hexadecimal format. This is a standard Verilog memory format file and is described on the page for [Memory Initialization Files](#).

The following is an example RAM with an initialization file.

```
# example #23 : Initialization file

module ATPG_RAM ( read, write, data_in, data_out, addr);,
    input read, write;
    input [7:0] data_in;
    input [3:0] addr;
    output [7:0] data_out;
    reg [7:0] data_out;
    reg [7:0] memory [0:15] ;

    always @(posedge write) memory[addr] = data_in;
    always @(posedge read) data_out = memory[addr]

    initial $readmemh("/net/tga/myproj/ram1.dat", memory);

endmodule
```

Note: Just because you define a memory initialization file does not mean the ATPG algorithm can make use of this data. Your RAM must pass DRC checks for RAM stability and write operations must be blocked, otherwise the RAM contents are lost as soon as random patterns are applied, which is generally the very first ATPG pattern. This means your RAM must behave like a ROM, which is probably not a restriction you want.

ROM Modeling

How do we model a ROM? Very simply, we create a RAM with no write ports and make sure it has an initialization file. Here's a simple ROM with a tristate output enable.

```
# example #24 : ROM with tristate output

module MY_ROM ( oe, addr, data_out );
  input oe; # output control
  input [3:0] addr; # 16 words
  output [7:0] data_out; # 8 bits per word
  reg [7:0] data_out; # output holding register
  reg [7:0] memory [0:15] ; # memory storage

  always @(oe or addr)
    if (!oe) data_out = memory[addr];
    else data_out = 8'bZZZZZZZZ;
  initial $readmemh("rom_image.dat", memory);

endmodule
```

Decoded Address Support

Can we support RAMS or ROMS with decoded address lines? Yes, but this gets a little more complicated. There are three new areas of syntax which need to be defined: (1) the 'ENCODE' function as shown in the following example; (2) each read or write port where a decoded address bus is desired should reference the 'ENCODE' function where normally the address itself would be placed; and (3) an additional check of the address bus for having a nonzero value must be added to any "if" clause qualifier.

```
# example #25 : RAM with decoded addresses

module MY_ATPG_RAM (read, write, data_in, data_out, ra, wa);
  parameter addrbits = 4, addrmax = 15, num_words = 16;
  parameter databits = 8;
  parameter XWORD = 8'bxxxxxxxx;
  input read, write;
  input [databits-1:0] data_in;
  input [addrmax:0] ra, wa;
  output [databits-1:0] data_out;
  reg [databits-1:0] data_out;
  reg [databits-1:0] memory [0:addrmax];
  event WRITE_OP;

  function [addrbits-1:0] ENCODE;      input [addrmax:0] addr;
  integer n;      begin      ENCODE = XWORD;      for (n=0; n <
```

```

num_words; n=n+1) begin
    addr[n] = 0;
num_words;
end
end
end
endfunction
always
@(posedge write)
    if (wa) begin
        memory[ENCODE(wa)] = data_in;
        #0; ->WRITE_OP;
    end

always @ (read or ra or WRITE_OP)
    if (!read && ra) data_out = memory[ENCODE(ra)];

endmodule

```

In the previous example, the bold text draws attention to the portions of the RAM definition related to the decoded address bus support. The function `ENCODE` has been defined, and to make it more convenient for reuse the address and data widths are parameterized. There is nothing special about the name of this function, we just picked one. For the write port, the additional `if (wa)` has been added and for the read port the `&& ra` has been added to an existing `if()` clause. This avoids a read or write if all address bits are zero which is desired behavior when decoded address busses are used. And finally the address bus references have been replaced with a call to the `ENCODE` function.

Memory Set/Reset Capability

It is not uncommon for DSP related RAMS or RAMS used in FIFOs or CAMS to have a global asynchronous reset which zeros all data bits in all words. TestMAX ATPG supports both an asynchronous set and clear capability. The following example shows additional statements added to asynchronously clear the memory if `CLR` is low.

```

# example #26 : RAM with asynchronous reset

module MY_ATPG_RAM (CLR, read, write, data_in, data_out, addr);
    input CLR;
    input read, write;
    input [7:0] data_in;
    input [3:0] addr;
    output [7:0] data_out;
    reg [7:0] data_out;
    reg [7:0] memory [0:15] ;
    integer i;
    event WRITE_OP;

    always @(posedge write) begin
        memory[addr] = data_in;
        #0; ->WRITE_OP;
    end

    always @(read or addr or WRITE_OP)
        if (!read) data_out = memory[addr];

```

```

always @ CLR if (!CLR) begin          for (i=0; i<16; i=i+1) memory[i] =
8'b0;          #0; ->WRITE_OP;      end

endmodule

```

If you require a synchronous set or reset, use an external DFF primitive to create this synchronization.

```

# example #27 : RAM with synchronous reset

_DFF u1 (0,0,write,CLR, sync_CLR); # synchronize CLR
always @ sync_CLR if ( !sync_CLR) begin
    for (i=0; i<16; i=i+1) memory[i] = 8'b0;
    #0; ->WRITE_OP;
end

endmodule

```

If you require a synchronous set or reset with an enable, use an external DFF + MUX primitive to create this synchronization.

```

# example #28 : RAM with synchronous reset and write enable

_MUX u1 (enable, sync_CLR, CLR, din);
_DFF u2 (0,0,write, din, sync_CLR); # synchronize CLR
always @ sync_CLR if ( !sync_CLR) begin
    for (i=0; i<16; i=i+1) memory[i] = 8'b0;
    #0; ->WRITE_OP;
end

endmodule

```

Debugging Your Models

Here are some tips for debugging any RAM/ROM modules you create:

- Keep your module under development in a separate file until you've finished testing it.
- Use the `-delete` option along with the `read_netlist` command to read your memory model and do the initial parsing. Pay attention to any and all warnings or errors and work to eliminate them. Watch out for N2 violations because this could mean your entire RAM or ROM has been rejected and replaced with a black box.
- Use the `run_build_model` command to build just your model by name. There should be no warnings or errors from the build.
- After building, use the `report_memory` command with both the `-all` and `-verbose` options to review data on your RAM or ROM model. Identify the `gate_id` of the ATPG RAM primitive and then use a `report_primitives` command to display information

on this gate ID. Review the reported contention behavior for the RAM primitive to make sure it is as you intended.

- After building, set the display mode to primitive and show a few module ports. Click net diamonds to expand the drawing until all of the RAM/ROM building blocks are in view. Review the graphical display to check that your address and data bus widths are correct, you have the correct number of read and write ports, and so forth. Consult the Simulation Primitives for a more complete description of the various RAM/ROM building blocks.
- For a rigorous test of the ATPG memory model construct a testbench into which you instantiate the RAM. The testbench should have both read address, write address, and data write lines coming to the RAM from: a) primary inputs, b) scan registers, c) nonscan registers. Similarly, the data outputs of the read port should go to: a) primary outputs, b) scan registers, c) nonscan registers which then feed either PO's or scan registers. This type of testbench provides almost all of the variations of address/data lines likely to be experienced in designs. Generate ATPG patterns under this environment and then simulate them to validate the RAM model is usable under all conditions.
- As a test of functional pattern behavior, generate and simulate your functional patterns in a Verilog simulator capable of create Extended VCD output. This VCD-E stimulus/response file can then be read into TestMAX ATPG as functional patterns, and by using the `run_simulation -sequential` command, the TestMAX ATPG behavior can be compared to the actual response of the Verilog model. Differences sometimes occur, but the prime goal is to have the ATPG model never generate a non-X value when the Verilog response is X. Having the ATPG model predict X when Verilog predicts non-X is often time a necessary conservative modeling approach.

Memory Modeling Syntax in Backus-Naur Form (BNF)

This section describes the BNF memory modeling syntax, which is a restricted subset of the Verilog language used for modeling of RAMs and ROMs. Constructs identical to the Verilog BNF definition are not described. Refer to IEEE Std 1364-1995 for the syntax definitions of standard Verilog.

```
memory_definition ::= { contention_mode } module_declaration
                   { contention_defaults }
contention_mode ::= rr_contention | rw_contention | ww_contention
module_declaration ::= module module_identifier [ list_of_ports ] ;
                   { module_item
                   } endmodule
contention_defaults ::= rr_undef | rw_undef | ww_undef
list_of_ports ::= (port_identifier {,port_identifier } )
module_item ::= module_item_declaration | read_port | write_port |
              set_port | reset_port | memory_initialization
              | gate_instantiation | udp_instantiation
```

```

module_item_declaration ::= data_out_reg_declaration | memory_declaration
    | parameter_declaration | input_declaration
    | output_declaration | inout_declaration | net_declaration |
    reg_declaration | integer_declaration | event_declaration
data_out_reg_declaration ::= reg data_range data_out_reg_identifier;
memory_declaration ::= reg data_range memory_name_identifier
    address_range ;
data_range ::= [ data_high_bit : data_low_bit ]
address_range ::= [ address_low : address_high ]
read_port ::= level_sensitive_read_port | edge_sensitive_read_port
write_port ::= level_sensitive_write_port | edge_sensitive_write_port
set_port ::= always @ set_control_net if ( [!] set_control_net ) 1_fill
reset_port ::= always @ reset_control_net if ( [!] reset_control_net )
    0_fill
level_sensitive_read_port ::= always @ ( read_sensitivity_list ) if ( [!]
    control_net
) read_assign [ else data_out_reg = bus_constant ; ]
edge_sensitive_read_port ::= always @ ( edge_read_control_net ) [ if
    ( [!]
control_net ) ] read_assign
level_sensitive_write_port ::= always @ ( write_sensitivity_list ) if
    ( [!] control_net
) write_assign
edge_sensitive_write_port ::= always @ ( edge_write_control_net ) [if
    ( [!]
control_net ) ] write_assign
read_assign ::= data_out_reg = memory_name [ address_net ] ;
write_assign ::= begin memory_name [ address_net ] = data_in_net ;#0;
->event_identifier; end
read_sensitivity_list ::= read_control_net [ or address_net ]
    [ orevent event_identifier
]
write_sensitivity_list ::= write_control_net or address_net or
    data_in_net
control_net ::= net_identifier
read_control_net ::= net_identifier
write_control_net ::= net_identifier
set_control_net ::= net_identifier
reset_control_net ::= net_identifier
address_net ::= bus_identifier
data_in_net ::= bus_identifier
memory_initialization ::= initial readmem_type ( filespec ,
    memory_name ) ;
readmem_type ::= $readmemh | $readmemb
filespec ::= " filepath "
0_fill ::= for ( i=0; i < max_address ; i=i+1 ) memory_name [ i ] =
    0_constant
;
1_fill ::= for ( i=0; i < max_address ; i=i+1 ) memory_name [ i ] =
    1_constant
;

```

```

edge ::= posedge | negedge
rr_contention ::= `define read_read rr_choices
rw_contention ::= `define read_write rw_choices
ww_contention ::= `define write_write ww_choices
rr_choices ::= normal | readx
rw_choices ::= mixed | new | readx | xfill | new_but_readx_across_ports
ww_choices ::= xbit | xword | xfill | dominance | forbidden
rr_undef ::= `undef read_read
rw_undef ::= `undef read_write
ww_undef ::= `undef write_write
data_high_bit ::= decimal_number
data_low_bit ::= decimal_number
address_low ::= decimal_number
address_high ::= decimal_number
max_address ::= decimal_number
bus_constant ::= 0_constant | 1_constant | x_constant | z_constant
0_constant ::= size base { 0 }+
1_constant ::= size base { 1 }+
x_constant ::= size base { x }+
z_constant ::= size base { z }+
size ::= decimal_digit { decimal_digit }
base ::= 'b | 'B | 'h | 'H
digit ::= dec_digit | hex_digit | sim_digit
decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_number ::= { decimal_digit }+
hex_digit ::= decimal_digit | a | A | b | B | c | C | d | D | e | E | f |
F
binary_digit ::= x | X | z | Z | 0 | 1
bus_identifier ::= net_identifier
net_identifier ::= <standard Verilog syntax>
module_identifier ::= <standard Verilog syntax>
port_identifier ::= <standard Verilog syntax>
parameter_declaration ::= <standard Verilog syntax>
input_declaration ::= <standard Verilog syntax>
output_declaration ::= <standard Verilog syntax>
inout_declaration ::= <standard Verilog syntax>
net_declaration ::= <standard Verilog syntax>
reg_declaration ::= <standard Verilog syntax>
integer_declaration ::= <standard Verilog syntax>
gate_instantiation ::= <standard Verilog syntax>
udp_instantiation ::= <standard Verilog syntax>
event_declaration ::= <standard Verilog syntax>

```

RAM and ROM Modeling Examples

This section contains a set of Verilog module definitions of functional RAMS recognized by TestMAX ATPG. For additional information on how to construct a memory model see "Memory Modeling."

Chapter 4: ATPG Modeling
Modeling Topics

```

#
# #1 Simple RAM with common address,
# edge sensitive read/write. If the read and
# write clocks come from a common source
# in the design, this will produce R/W
# contention behavior of old.
#
module ram1024x8 (wclk, rclk, a, din, dout);
  parameter databits = 8;
  parameter addrbits = 10;
  parameter addrmax = (1<<addrbits) - 1;
  input wclk, rclk;
  input [addrbits-1:0] a;
  input [databits-1:0] din ;
  output [databits-1:0] dout;
  reg [databits-1:0] mymem [0:addrmax];
  reg [databits-1:0] dout ;

  always @ (posedge wclk) mymem[a] = din;
  always @ (posedge rclk) dout = mymem[a];

endmodule

#
# #2 RAM with common address, level sensitive read/write,
# chip select, and active low write enable. If the read and
# write ports should both be active, the R/W contention behavior
# is to read the new value.
#
module ram128x32 (DO, DI, A, WEB, OE, CS);
  parameter databits = 32;
  parameter addrbits = 7;
  parameter addrmax = (1<<addrbits) - 1;
  output [databits-1:0] DO;
  input [databits-1:0] DI;
  input [addrbits-1:0] A;
  input WEB, OE, CS;
  reg [databits-1:0] memory [0:addrmax];
  reg [databits-1:0] DO;

  and u0 (OEN, CS, OE);
  and u1 (WEN, CS,!WEB);

  event WRITE_OP;

  always @ (WEN or A or DI) if (WEN) begin
    memory[A] = DI;
    #0; ->WRITE_OP;
  end

  always @ (OEN or A or WRITE_OP)
    if (OEN) DO = memory[A];

```

Chapter 4: ATPG Modeling Modeling Topics

```

        else DO = 32'hZ;

endmodule

#
# #3 Simple ROM
#
module rom16x24 (a, dout);
    parameter databits = 24;
    parameter addrbits = 4;
    parameter addrmax = (1<<addrbits) - 1;
    input [addrbits-1:0] a;
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] dout;

    initial $readmemh("rom16x24.dat", mymem);

    always @ (a)
        if (a <= addrmax) dout = mymem[a];

endmodule
# see also: Sample Memory Initialization Files

#
# #4 Simple ROM with output tri-state
#
module rom16x16 (ren, a, dout);
    parameter databits = 16;
    parameter addrbits = 4;
    parameter addrmax = (1<<addrbits) - 1;
    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("rom16x16.dat", mymem);

    always @ (ren or a)
        if (ren) dout = mymem[a];
        else dout = 16'bzzzz_zzzz_zzzz_zzzz ;

endmodule
# see also: Sample Memory Initialization Files

#
# #5 ROM with output hold
#
module rom32x16 (ren, a, dout);
    parameter databits = 16;
    parameter addrbits = 5;

```

Chapter 4: ATPG Modeling Modeling Topics

```

parameter addrmax = (1<<addrbits) - 1;
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("rom32x16.dat", mymem);

always @ (ren or a) if (ren) dout = mymem[a] ;

endmodule

#
# #6 Simple RAM with common address, active low level
# sensitive read, and an initialization file (in hex).
# R/W contention behavior of this example is new.
#
module ram500x8 (wclk, ren, a, din, dout);
parameter databits = 8;
parameter addrbits = 9;
parameter addrmax = 499;
input wclk, ren;
input [addrbits-1:0] a;
input [databits-1:0] din ;
output [databits-1:0] dout;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout ;
event WRITE_OP;

initial $readmemh("ram512x8.dat", mymem);

always @ (posedge wclk) begin
mymem[a] = din;
#0; ->WRITE_OP;
end

always @ (ren or a or WRITE_OP)
if (!ren) dout = mymem[a] ;

endmodule

#
# #7 Simple RAM with R/W direction and strobe
#
module ram64x8 (rwb, st, a, din, dout);
parameter databits = 8;
parameter addrbits = 6;
parameter addrmax = (1<<addrbits) - 1;
input rwb, st;
input [addrbits-1:0] a;
input [databits-1:0] din ;
output [databits-1:0] dout;

```

Chapter 4: ATPG Modeling Modeling Topics

```

reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout ;

always @ (posedge st) if (!rwb) mymem[a] = din;

# A potentially dangerous method to make 'RCLK', see example #10
# for a more accurate clock qualifier example.
#
and u1 (RCLK, st, rwb);
always @(posedge RCLK) dout = mymem[a] ;

endmodule

#
# #8 Simple RAM with read and write enable, a common
# read/write strobe, and common address
#
module ram9x32 (st, ren, wen, dout, din, a);
parameter databits = 32;
parameter addrbits = 4;
parameter addrmax = 8;
input st,ren,wen;
input [databits-1:0] din;
output [databits-1:0] dout;
input [addrbits-1:0] a;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout;

always @ (posedge st) if (wen) mymem[a] = din;

# A potentially dangerous method to make 'RCLK', see example #10
# for a more accurate clock qualifier example.
#
and u1 (RCLK, st, ren);
always @(posedge RCLK) dout = mymem[a];

endmodule

#
# #9 Simple RAM with separate R/W address
# The R/W contention behavior of this example is old
#
module ram256x4 (wclk, wa, din, rclk, ra, dout);
parameter databits = 4;
parameter addrbits = 8;
parameter addrmax = (1<<addrbits) - 1;
input wclk, rclk;
input [addrbits-1:0] wa, ra;
input [databits-1:0] din ;
output [databits-1:0] dout;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout ;

```

Chapter 4: ATPG Modeling Modeling Topics

```

    always @ (posedge wclk) mymem[wa] = din;
    always @ (posedge rclk) dout = mymem[ra] ;

endmodule

#
# #10 Clocked read port with clock qualifier
#
`define read_write new
module ram48x4 (CS, wclk, wen, wa, DI, RCLK, REN, RA, DO);
    parameter databits = 4;
    parameter addrbits = 6;
    parameter addrmax = 47;
    input wclk, RCLK, wen, REN, CS;
    input [addrbits-1:0] wa, RA;
    input [databits-1:0] DI ;
    output [databits-1:0] DO;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] DO;

    always @ (posedge wclk) if (wen) mymem[wa] = DI;

    and u2 (read_ena, CS, REN); # chip select & read enable
    always @ (posedge RCLK) if (read_ena) DO = mymem[RA];

endmodule
`undef read_write

#
# #11 Synchronous RAM with write-through behavior. A read
# operation occurs for every write operation and the
# new data written appears on the read port outputs.
#
# Note: Change R/W contention from new to mixed
# to have read port provide previous data during a
# write.
#
# Note 2: This write-through behavior is not supported for the
# fault grading of functional patterns, only for ATPG pattern
# generation.
#
`define read_write new
module ram192x6 (CLK, WEN, CS, ADDR, DI, DO);
    parameter databits = 6;
    parameter addrbits = 8;
    parameter addrmax = 191;
    input CLK, WEN, CS;
    input [addrbits-1:0] ADDR;
    input [databits-1:0] DI;
    output [databits-1:0] DO;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] DO ;

```

Chapter 4: ATPG Modeling Modeling Topics

```

#
# --- rising CLK with CS=1, WEN=0 causes a write
#
and ul (write_en, !WEN, CS); # write qualifier
always @ (posedge CLK) if (write_en) mymem[ADDR] = DI;
#
# --- rising edge on CLK with CS=1 always causes a read
#
always @ (posedge CLK) if (CS) DO = mymem[ADDR];

endmodule
`undef read_write

#
# #12 RAM with a write enable, separate R/W address
#
module ram256x8 (wclk, wen, wa, din, rclk, ra, dout);
parameter databits = 8;
parameter addrbits = 8;
parameter addrmax = (1<<addrbits) - 1;
input wclk, wen, rclk;
input [addrbits-1:0] wa, ra;
input [databits-1:0] din ;
output [databits-1:0] dout;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout ;

initial $readmemh("ram256x8.dat", mymem);

always @ (posedge wclk) if (wen) mymem[wa] = din;

always @ (posedge rclk) dout = mymem[ra] ;

endmodule

#
# #13 RAM with two edge sensitive R/W ports
#
module ram64x12 (w1,a1,d1, w2,a2,d2, r3,a3,d3, r4,a4,d4);
parameter databits = 12;
parameter addrbits = 5;
parameter addrmax = (1<<addrbits) - 1;
input w1,w2,r3,r4;
input [addrbits-1:0] a1, a2, a3, a4;
input [databits-1:0] d1, d2 ;
output [databits-1:0] d3, d4;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] d3, d4 ;

always @ (posedge w1) mymem[a1] = d1;
always @ (posedge w2) mymem[a2] = d2;

```

Chapter 4: ATPG Modeling Modeling Topics

```

        always @ (posedge r3) d3 = mymem[a3] ;
        always @ (posedge r4) d4 = mymem[a4] ;

endmodule

#
# #14 RAM with dual READ/WRITE ports and the READ
# ports are level sensitive
#
module ram64x6 (w1,a1,d1, w2,a2,d2, r3,a3,d3, r4,a4,d4);
    parameter databits = 6;
    parameter addrbits = 6;
    parameter addrmax = (1<<addrbits) - 1;
    input w1,w2,r3,r4;
    input [addrbits-1:0] a1, a2, a3, a4;
    input [databits-1:0] d1, d2 ;
    output [databits-1:0] d3, d4;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] d3, d4 ;
    event WRITE;

    always @ (posedge w1) begin
        mymem[a1] = d1; #0; ->WRITE; end
    always @ (posedge w2) begin
        mymem[a2] = d2; #0; ->WRITE; end

    always @ (r3 or a3 or WRITE)
        if (r3) d3 = mymem[a3] ;
    always @ (r4 or a4 or WRITE)
        if (r4) d4 = mymem[a4] ;

endmodule

#
# #15 Falling edge sensitive write ports, level sensitive
# read ports, separate tristate output, Chip Select
#
module ram50x20 (w1,a1,d1, w2,a2,d2, r3,a3,d3,oe3, r4,a4,d4,oe4, cs);
    parameter databits = 20;
    parameter addrbits = 6;
    parameter addrmax = 49;

    input w1,w2,r3,oe3,r4,oe4, cs;
    input [databits-1:0] d1, d2;
    input [addrbits-1:0] a1, a2, a3, a4;
    output [databits-1:0] d3, d4;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] d3, d3_reg, d4, d4_reg;
    event WRITE;

    /* internal control logic terms */

    and u1 (readena1, cs, !r3);

```

Chapter 4: ATPG Modeling Modeling Topics

```

and u2 (readena2, cs, !r4);
and u3 (outena1, cs, !oe3);
and u4 (outena2, cs, !oe4);

/* write ports, edge sensitive active high */

always @(posedge w1) if (cs) begin
    mymem[a1] = d1; #0; ->WRITE; end
always @(posedge w2) if (cs) begin
    mymem[a2] = d2; #0; ->WRITE; end

/* read ports, level sensitive */

always @(readena1 or a3 or WRITE)
    if (readena1) d3_reg = mymem[a3];
always @(readena2 or a4 or WRITE)
    if (readena2) d4_reg = mymem[a4];

/* output enables, qualified by chip select */

always @(outena1 or d3_reg)
    if (outena1) d3 = d3_reg; else d3 = 20'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZZ ;
always @(outena2 or d4_reg)
    if (outena2) d4 = d4_reg; else d4 = 20'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZZ ;

endmodule

#
# #16 RAM with asynchronous set & clear capability
#
module ram32x5 (set, rst, wclk, wa, din, rclk, ra, dout);
    parameter databits = 5;
    parameter addrbits = 4;
    parameter words = (1<<addrbits);
    parameter addrmax = words - 1;
    parameter ONES = 5'b11111;
    parameter ZEROS = 5'b0;
    input set, rst, wclk, rclk;
    input [addrbits-1:0] wa, ra;
    input [databits-1:0] din ;
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] dout ;
    integer i;
    event WRITE_OP;

    always @ rst if (rst) begin
        for (i=0; i<words; i=i+1) mymem[i] = ZEROS;
        #0; ->WRITE_OP;
    end

    always @ set if (set) begin
        for (i=0; i<words; i=i+1) mymem[i] = ONES;
    end

```


Chapter 4: ATPG Modeling Modeling Topics

```

    #0; ->WRITE_OP;
end

always @ (posedge wclk) begin
    mymem[wa] = din;
    #0; ->WRITE_OP;
end

always @ (rclk or ra or WRITE_OP)
    if (rclk) dout = mymem[ra] ;

endmodule

#
# #17 RAM with synchronous set & clear capability
#
module ram32x5_s (set, rst, wclk, wa, din, rclk, ra, dout);
    parameter databits = 5;
    parameter addrbits = 4;
    parameter words = (1<<addrbits);
    parameter addrmax = words - 1;
    parameter ONES = 5'b11111;
    parameter ZEROS = 5'b0;
    input set, rst, wclk, rclk;
    input [addrbits-1:0] wa, ra;
    input [databits-1:0] din ;
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] dout ;
    integer i;
    event WRITE_OP;

    _DFF u1 (1'b0, 1'b0, wclk, rst, srst);
    always @ srst if (srst) begin
        for (i=0; i<words; i=i+1) mymem[i] = ZEROS;
        #0; ->WRITE_OP;
    end

    _DFF u1 (1'b0, 1'b0, wclk, set, sset);
    always @ sset if (sset) begin
        for (i=0; i<words; i=i+1) mymem[i] = ONES;
        #0; ->WRITE_OP;
    end

    always @ (posedge wclk) begin
        mymem[wa] = din;
        #0; ->WRITE_OP;
    end

    always @ (rclk or ra or WRITE_OP)
        if (rclk) dout = mymem[ra] ;

endmodule

```

Chapter 4: ATPG Modeling
Modeling Topics

```

#
# #18 RAM that uses a core and a wrapper to accomplish
# bit-blasted address and data pins
#

# the inner core
module ramcore_16x6 (wba, wa, din, ra, dout);
  parameter databits = 6;
  parameter addrbits = 4;
  parameter addrmax = (1<<addrbits) - 1;
  input wba;
  input [addrbits-1:0] wa, ra;
  input [databits-1:0] din ;
  output [databits-1:0] dout;
  reg [databits-1:0] mymem [0:addrmax];
  reg [databits-1:0] dout;
  event WRITE_OP;

  always @ (wba or wa or din) if (!wba) begin
    mymem[wa] = din;
    #0; ->WRITE_OP;
  end
  always @ (wba or ra or WRITE_OP) if (wba) dout = mymem[ra];
endmodule

# the outer wrapper
module ram16x6( DO0, DO1, DO2, DO3, DO4, DO5,
  WA0, WA1, WA2, WA3, RA0, RA1, RA2, RA3,
  DI0, DI1, DI2, DI3, DI4, DI5, WBA);
  input WA0, WA1, WA2, WA3, RA0, RA1, RA2, RA3,
    DI0, DI1, DI2, DI3, DI4, DI5, WBA;
  output DO0, DO1, DO2, DO3, DO4, DO5;

  ramcore_16x6 u1 (
    .wba(WBA),
    .wa( {WA3,WA2,WA1,WA0} ),
    .din( {DI5, DI4, DI3, DI2, DI1, DI0} ),
    .ra( {RA3, RA2, RA1, RA0} ),
    .dout( {DO5, DO4, DO3, DO2, DO1, DO0} ) );
endmodule

#
# #19 RAM with data out off value of zero
#
module ram64x32 (wen, wa, din, ren, ra, dout);
  parameter databits = 32;
  parameter addrbits = 6;
  parameter addrmax = (1<<addrbits) - 1;
  parameter ZERO = 32'b0;
  input wen, ren;
  input [addrbits-1:0] wa, ra;
  input [databits-1:0] din ;

```

Chapter 4: ATPG Modeling Modeling Topics

```

output [databits-1:0] dout;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] dout ;
event WRITE_OP;

always @ (wen or wa or din) if (wen) begin
    mymem[wa] = din;
    #0; ->WRITE_OP;
end

always @ (ren or ra or WRITE_OP)
    if (ren) dout = mymem[ra] ;
    else dout = ZERO;

endmodule

#
# #20 RAM with data out offstate = 1, and tri-state output
#
module ram64x128 (wen, wa, din, ren, ra, dout, oe);
    parameter databits = 128;
    parameter addrbits = 6;
    parameter addrmax = (1<<addrbits) - 1;
    input wen, ren, oe;
    input [addrbits-1:0] wa, ra;
    input [databits-1:0] din ;
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] dout, dout_reg ;
    event WRITE_OP;

    always @ (wen or wa or din) if (wen) begin
        mymem[wa] = din;
        #0; -> WRITE_OP; /* signal event */
    end

    always @ (ren or ra or WRITE_OP)
        if (ren) dout_reg = mymem[ra] ;
        else dout_reg = 128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;

    always @ (oe or dout_reg)
        if (oe) dout = dout_reg;
        else dout = 128'bZ;

endmodule

#
# #21 a RAM with non-default contention behavior
#
`define read_read readx
`define read_write readx
`define write_write xword
module ram252x7 (w1,a1,d1, w2,a2,d2, r3,a3,d3, r4,a4,d4);

```

Chapter 4: ATPG Modeling Modeling Topics

```

parameter addrbits = 8;
parameter addrmax = 251;
parameter databits = 7;
input w1,w2,r3,r4;
input [addrbits-1:0] a1, a2, a3, a4;
input [databits-1:0] d1, d2;
output [databits-1:0] d3, d4;
reg [databits-1:0] mymem [0:addrmax];
reg [databits-1:0] d3, d4;
event WRITE_OP;

always @ (w1 or a1 or d1) if (w1) begin
    mymem[a1] = d1;
    #0; ->WRITE_OP; /* signal event */
end

always @ (w2 or a2 or d2) if (w2) begin
    mymem[a2] = d2;
    #0; ->WRITE_OP; /* signal event */
end

always @ (r3 or a3 or WRITE_OP) if (r3) d3 = mymem[a3] ;

always @ (r4 or a4 or WRITE_OP) if (r4) d4 = mymem[a4] ;

endmodule
`undef read_read
`undef read_write
`undef write_write

#
# #22 a RAM with decoded address bus
#
module RAM_decode (read, write, data_in, data_out, ra, wa);
parameter addrbits = 4, addrmax = 15, num_words = 16;
parameter databits = 8;
parameter XWORD = 8'bxxxxxxxx;
input read, write;
input [databits-1:0] data_in;
input [addrmax:0] ra, wa;
output [databits-1:0] data_out;
reg [databits-1:0] data_out;
reg [databits-1:0] memory [0:addrmax];
event WRITE_OP;

function [addrbits-1:0] ENCODE;
input [addrmax:0] addr;
integer n;
begin
    ENCODE = XWORD;
    for (n=0; n < num_words; n=n+1) begin
        if (addr[n]==1) begin
            addr[n] = 0;

```

Chapter 4: ATPG Modeling Modeling Topics

```

        if (!addr == 0) ENCODE = n;
            n = num_words;
        end
    end
end
endfunction

always @(posedge write)
    if (wa) begin
        memory[ENCODE(wa)] = data_in;
        #0; ->WRITE_OP;
    end

always @ (read or ra or WRITE_OP)
    if (!read && ra) data_out = memory[ENCODE(ra)];

endmodule

#
# #23 a RAM with a mixture of encoded and decoded address buses
#
module ram_decode2 (reset, w1,c1,a1,d1, w2,a2,d2, r3,a3,d3, r4,a4,d4);
    parameter addrbits = 4, addrmax = 15, num_words = 16;
    parameter databits = 8;
    parameter ZEROS = 8'h00, XWORD = 8'bx;

    input reset, w1, c1, w2, r3, r4;
    input [addrbits-1:0] a1, a3;
    input [addrmax:0] a2, a4; # decoded addresses
    input [databits-1:0] d1, d2;
    output [databits-1:0] d3, d4;

    reg [databits-1:0] mymem [0:addrmax], d3, d4;
    integer i;
    event WRITE_OP;

    function [addrbits-1:0] ENCODE;
        input [addrmax:0] addr;
        integer n;
        begin
            ENCODE = XWORD;
            for (n=0; n < num_words; n=n+1) begin
                if (addr[n]==1) begin
                    addr[n] = 0;
                    if (!addr == 0) ENCODE = n;
                    n = num_words;
                end
            end
        end
    endfunction

    always @ reset if (reset) begin
        for (i=0; i<num_words; i=i+1) mymem[i] = ZEROS;
    end
endmodule

```

Chapter 4: ATPG Modeling Modeling Topics

```

    #0; ->WRITE_OP;
end

always @ (posedge w1) if (c1) begin
    mymem[a1] = d1; #0; ->WRITE_OP; end

always @ (negedge w2) if (a2) begin
    mymem[ENCODE(a2)] = d2; #0; ->WRITE_OP; end

always @ (r3 or a3 or WRITE_OP)
    if (r3) d3 <= mymem[a3]; else d3 <= ZEROS;

always @ (r4 or a4 or WRITE_OP)
    if (!r4 && a4) d4 <= mymem[ENCODE(a4)];

endmodule

#
# #24 A level sensitive RAM with bitwise read/write control
#
module RAM16x8 ( CS, A0,A1,A2,A3,
    DI0, DI1, DI2, DI3, DI4, DI5, DI6, DI7,
    DO0, DO1, DO2, DO3, DO4, DO5, DO6, DO7,
    WE0, WE1, WE2, WE3, WE4, WE5, WE6, WE7 );

    input CS;
    input A0, A1, A2, A3;
    input DI0, DI1, DI2, DI3, DI4, DI5, DI6, DI7;
    output DO0, DO1, DO2, DO3, DO4, DO5, DO6, DO7;
    input WE0, WE1, WE2, WE3, WE4, WE5, WE6, WE7;

    wire [3:0] ADDR;
    assign ADDR = {A3,A2,A1,A0};

    RAM16x1 s10 (ADDR, DI0, DO0, WE0, CS);
    RAM16x1 s11 (ADDR, DI1, DO1, WE1, CS);
    RAM16x1 s12 (ADDR, DI2, DO2, WE2, CS);
    RAM16x1 s13 (ADDR, DI3, DO3, WE3, CS);
    RAM16x1 s14 (ADDR, DI4, DO4, WE4, CS);
    RAM16x1 s15 (ADDR, DI5, DO5, WE5, CS);
    RAM16x1 s16 (ADDR, DI6, DO6, WE6, CS);
    RAM16x1 s17 (ADDR, DI7, DO7, WE7, CS);

endmodule

# The core 1-bit RAM
module RAM16x1 (ADDR, DI, DO, WE, CS);
    input [3:0] ADDR;
    input DI, WE, CS;
    output DO;
    reg DO;
    reg [0:0] memory [0:15];

```

Chapter 4: ATPG Modeling Modeling Topics

```

event WRITE_OP;

and u1 (read_en, CS, !WE);
and u2 (write_en, CS, WE);

always@(write_en or ADDR or DI) if (write_en) begin
    memory[ADDR] = DI;
    #0; ->WRITE_OP;
end

always@(read_en or ADDR or WRITE_OP) if (read_en) DO = memory[ADDR];

endmodule

#
# #25 A level sensitive RAM with constantly enabled READ port
#
module ram512x11 (addr, din, wen, dout);
    parameter databits = 11;
    parameter addrbits = 9;
    parameter addrmax = (1<<addrbits)-1;
    input wen;
    input [addrbits-1:0] addr;
    input [databits-1:0] din;
    output [databits-1:0] dout;
    reg [databits-1:0] mymem [0:addrmax];
    reg [databits-1:0] dout;

    supply1 ren;
    event WRITE_OP;

    always @ (wen or addr or din) if (wen) begin
        mymem[addr] = din;
        #0; ->WRITE_OP;
    end

    always @ (ren or addr or WRITE_OP)
        if (ren) dout = mymem[addr];

endmodule

#
# #26 RAM with Data In to Data Out bypass using "assign"
# to describe a bank of MUX prims.
#
module bypass_RAM ( DO, RA, WA, DI, WE, RE, CLK, BYPASS);
    output [15:0] DO;
    input [15:0] DI;
    input [3:0] RA,WA;
    input CLK, WE, RE, BYPASS;
    reg [15:0] memory [0:15];
    reg [15:0] DO_REG;

```

```
always @(posedge CLK) if (WE) memory[WA] = DI;
always @(posedge CLK) if (RE) DO_REG = memory[RA];

assign DO = BYPASS ? DI : DO_REG ;

endmodule
```

Memory Data File Examples

A ROM model is required to have a memory initialization file. For a RAM, this file is optional. TestMAX ATPG accepts the data file format specified by Verilog. Some examples follow:

A memory initialization file for a 16x16 device, with data presented in binary.

```
000000000000000001
000000000000000010
0000000000000000100
00000000000000001000
000000000000010000
00000000000100000
0000000001000000
0000000010000000
0000000100000000
0000001000000000
0000010000000000
0000100000000000
0001000000000000
0010000000000000
0100000000000000
1000000000000000
```

The same data 16x16 presented in hex.

```
0001
0002
0004
0008
0010
0020
0040
0080
0100
0200
0400
0800
1000
2000
4000
8000
```


A more complicated Memory Initialization File which makes use of whitespace, comments, multiple data entries per line, underscores in data, and address skipping. This one is 64 words of 16 bits.

```
0001
0002
0004
0008

0010 0020 0040 0080
0100 0200 0400 0800
1000 2000 4000 8000

# address = 0010
A001 c401 e404 700a
3816 1c2c 2e58 07b0
23e0 07c0 25e0 0b70
363c 2c1c 7c0e b006

# address = 0020
8001 4002 2004 1008 8810 4421 2242 1184
0908 0650 0620 0950 1088

# skip to hex address 30
@30
fffe fffd fffb fff7 ffef fdfd ffbf ff7f
/* another comment */
feff fdff fbff f7ff # end of line comment
# underscores for readability
efff dfff bfff 7_f_f_f
```

Interpreting UDP Messages

Many of the detailed DRC violation messages associated with creating derived ATPG models for UDPs from vendor libraries need additional explanation on how the messages should be interpreted and whether further action is needed.

Variant #1

For example, one such commonly occurring message is something on the order of "Expected <string of chars with 't'> got <non 't'>. Users ask, "How do I interpret the 't', ':', and '.'?"

For the LAT2 UDP table shown below, TestMAX ATPG issues the following text:

```
underspecified UDP (Expected "tx....." got "?x?:?:-")
primitive LAT2 (q, d, gn, ntfy);
  output q;
  reg q;
```

```

input d, gn, ntfy;

table
# D GN ntfy : Q- : Q+
# --- --- --- --- ---
  ? 1 ? : ? : - ; #1
  0 0 ? : ? : 0 ; #2
  1 0 ? : ? : 1 ; #3
  1 x ? : 1 : 1 ; #4
  0 x ? : 0 : 0 ; #5
  ? p ? : ? : - ; #6 "p" includes (0x)
  ? ? * : ? : X ; #7
endtable
endprimitive

```

In this case the string Expected "tx...." contains a 't' meaning a test for 0 or 1 is expected. The periods "." occur for each position for which TestMAX ATPG does not care what the value is after the mismatch is detected, and the colons ':' occur as separators in the same position they hold in the table entries.

The 't' occurs in the first character position, so TestMAX ATPG is expecting the first character of the table entry to test for a 0 or 1. Instead it got "?x?:?:-", which can be used to help identify table entry #1 or #6 as the entry corresponding to the violation message. In this case you might need to get the Verilog reference manual out to decipher that the "p" is shorthand for the edge combinations of (01), (0x), (x1), (0z), and (z1). TestMAX ATPG is warning about the (0x) transition, which is similar to a steady state input on GN of "X" because the table entry implies that the output holds state when GN goes from 0 to X regardless of the value of D and Q-.

Let's rewrite table entry #6 and expand it into its variants, skipping those that have edge combinations which include Z.

```

primitive LAT2 (q, d, gn, ntfy);
output q;
reg q;
input d, gn, ntfy;

table
# D GN ntfy : Q- : Q+
# --- --- --- --- ---
  ? 1 ? : ? : - ; #1
  0 0 ? : ? : 0 ; #2
  1 0 ? : ? : 1 ; #3
  1 x ? : 1 : 1 ; #4
  0 x ? : 0 : 0 ; #5
  ? (01) ? : ? : - ; #6a expanded
  ? (x1) ? : ? : - ; #6b expanded
  ? (0x) ? : ? : - ; #6c expanded
  ? ? * : ? : X ; #7
endtable
endprimitive

```

This new variant of the UDP table has three lines to replace the original. If you read this with TestMAX ATPG and then perform a run build_model on it, you get the same violation message:

```
underspecified UDP (Expected "tx...." got "?x?:?:-")
```

Note the entry 6c which has the (0x) edge transition for GN and is the closest match to the error message showing "? x ? : ? : -". If the latch enable GN transitions to an X state, the table entry should not indicate a hold state, it would be better if it tested that D and Q- were identical, and if so then the output would be known. You can test this change by expanding entry 6c into two new lines as follows:

```
primitive LAT2 (q, d, gn, ntfy);
  output q;
  reg q;
  input d, gn, ntfy;

  table
    # D GN ntfy : Q- : Q+
    # --- --- --- --- ---
      ? 1 ? : ? : - ; #1
      0 0 ? : ? : 0 ; #2
      1 0 ? : ? : 1 ; #3
      1 x ? : 1 : 1 ; #4
      0 x ? : 0 : 0 ; #5
      ? (01) ? : ? : - ; #6a expanded
      ? (x1) ? : ? : - ; #6b expanded
      0 (0x) ? : 0 : 0 ; #6c1 expanded
      1 (0x) ? : 1 : 1 ; #6c2 expanded
      ? ? * : ? : X ; #7
  endtable
endprimitive
```

This new variant of the UDP table has two lines to replace the original 6c. If you read this with TestMAX ATPG and then perform a run build_model on it, you get a different violation this time:

```
unsupported UDP entry (Entry "??*?:X")
```

The reason the violation is different is because by default TestMAX ATPG shows only the single most significant violation for each module. By curing one violation message you expose the next. To see all of the violation messages at once issue the `set_netlist -nocheck_only_used_udps` command before reading the file with the modules being tested.

This N21 violation occurs any time a UDP has a notify column entry. There is no gate-level representation for the functionality described by the characters in the notify column and TestMAX ATPG is issuing a warning. Nearly all such models use this column for setting the output to an X as a result of a timing violation. For our ATPG functional model (timingless),

this violation can be ignored. At this point you have eliminated all the warnings that we can.

This example has been useful for demonstrating how to interpret UDP related warning messages and what action to take to try to reduce the warnings. Having led you through the process we'll now suggest some additional changes to the table that you can or cannot have noticed along the way.

The first change is that entry 6a and 6b are essentially redundant to entry 1. An edge transition on GN to a 1 for a level sensitive device is identical to a constant 1 on GN. So we could drop these entries from the table.

The second change is that entries 6c1 and 6c2 are essentially redundant to entries 4 and 5 by a similar argument.

So in the end we could have reduced warnings for this UDP by commenting out the original entry 6 rather than by expanding it into other lines. Each case is different and sometimes an idea for the final solution becomes more obvious by commenting out troublesome lines than by expanding them. Both methods should be considered when troubleshooting UDP messages. The reduced warning UDP is then:

```
primitive LAT2 (q, d, gn, ntfy);
  output q;
  reg q;
  input d, gn, ntfy;

  table
    # D GN ntfy : Q- : Q+
    # --- --- --- --- ---
      ? 1 ? : ? : - ; #1
      0 0 ? : ? : 0 ; #2
      1 0 ? : ? : 1 ; #3
      1 x ? : 1 : 1 ; #4
      0 x ? : 0 : 0 ; #5
      ? ? * : ? : X ; #6
  endtable
endprimitive
```

Variant #2

As another example, a message similar to "Expected <string of chars with 't'> got <string with 0 or 1 or X> is provided.

For the UDP table shown below TestMAX ATPG reports the following message:

```
underspecified UDP (Expected "1xr11?:t:." got "1xr11?:?:X")

primitive FJK (q, j, k, cp, cd, sd, ntfy);
  output q;
  reg q;
  input j,k, cp, cd, sd, ntfy;
```

```

table
# J K CP CD SD ntfy Q- : Q+
# --- : --- : ---
  0 0 r 1 1 ? : ? : - ; # 1. hold

  0 1 r 1 1 ? : ? : 0 ; # 2. clocked K
  0 1 r x 1 ? : ? : 0 ; # 3.
  ? ? ? x 1 ? : 0 : 0 ; # 4.

  1 0 r 1 1 ? : ? : 1 ; # 5. clocked J
  1 0 r 1 x ? : ? : 1 ; # 6.
  ? ? ? 1 x ? : 1 : 1 ; # 7.

  1 1 r 1 1 ? : 0 : 1 ; # 8.
  1 1 r 1 1 ? : 1 : 0 ; # 9.

  ? ? f 1 1 ? : ? : - ; # 10.

  0 0 (x1) 1 1 ? : ? : - ; # 11.
  0 1 (x1) 1 1 ? : 0 : 0 ; # 12.
  1 0 (x1) 1 1 ? : 1 : 1 ; # 13.
  0 0 (0x) 1 1 ? : ? : - ; # 14.
  0 1 (0x) 1 1 ? : 0 : 0 ; # 15.
  1 0 (0x) 1 1 ? : 1 : 1 ; # 16.

  * ? ? 1 1 ? : ? : - ; # 17.
  ? * ? 1 1 ? : ? : - ; # 18.

  ? ? ? 0 1 ? : ? : 0 ; # 19. clear
  ? ? ? 1 0 ? : ? : 1 ; # 20. set
  ? ? ? 0 0 ? : ? : 0 ; # 21. clear and set active
  ? ? ? 0 x ? : ? : 0 ; # 22. pessimism

  ? ? (?0) 1 1 ? : ? : - ; # 23. ignore falling clock.
  ? ? (1x) 1 1 ? : ? : - ; # 24.

  ? ? ? (?1) 1 ? : ? : - ; # 25. ignore changes on set and
  ? ? ? 1 (?1) ? : ? : - ; # 26. reset.
  ? ? ? ? ? * : ? : X ; # 27.
endtable
endprimitive

```

Here the message "Expected "1xr11?:t." got "1xr11?:?X" provides a useful hint. When we try to find the table entry to match the "1xr11?:?X" we won't be able to do so. Notice that the Q+ entry is an X and the only entry in our UDP table that explicitly sets the next state Q+ to X is entry 27. By Verilog default, all input combinations not explicitly defined by UDP table entries result in outputs set to X. So what TestMAX ATPG is trying to hint at is that a table entry is missing and it expects to see one of the form "1 x r 1 1 ? : t : . ;", where the t is replaced by 0 and 1, and an appropriate value for Q+ has been used of either 0/1/x/- .

Let us construct two additional entries and add them to the table by expanding the 't' into 0 and 1, and expanding the final output column, shown as a period, into its appropriate values given the input states:

```
# J K CP CD SD ntfy Q : Q+
```

```
# --- --- --- --- --- : --- : ---
```

```
1 x r 1 1 ? : 0 : 1 ; # A. Q=0, clock J=1
```

```
1 x r 1 1 ? : 1 : x ; # B. Q=1, clock K=X
```

When we insert these new entries we don't really need the ones in which the output is X as that is the default but it won't hurt for now. We also don't want to add these to the end of the table after all the entries with wildcard '?' or we won't get a match. We'll add our two new lines after entry #9 as lines A and B.

```
primitive FJK (q, j, k, cp, cd, sd, ntfy);
  output q;
  reg q;
  input j,k, cp, cd, sd, ntfy;

table
# J K CP CD SD ntfy Q- : Q+
# --- --- --- --- --- : --- : ---
  0 0 r 1 1 ? : ? : - ; # 1. hold

  0 1 r 1 1 ? : ? : 0 ; # 2. clocked K
  0 1 r x 1 ? : ? : 0 ; # 3.
  ? ? ? x 1 ? : 0 : 0 ; # 4.

  1 0 r 1 1 ? : ? : 1 ; # 5. clocked J
  1 0 r 1 x ? : ? : 1 ; # 6.
  ? ? ? 1 x ? : 1 : 1 ; # 7.

  1 1 r 1 1 ? : 0 : 1 ; # 8.
  1 1 r 1 1 ? : 1 : 0 ; # 9.

  1 x r 1 1 ? : 0 : 1 ; # A. Q=0, clock J=1
  1 x r 1 1 ? : 1 : x ; # B. Q=1, clock K=X

  ? ? f 1 1 ? : ? : - ; # 10.

  0 0 (x1) 1 1 ? : ? : - ; # 11.
  0 1 (x1) 1 1 ? : 0 : 0 ; # 12.
  1 0 (x1) 1 1 ? : 1 : 1 ; # 13.
  0 0 (0x) 1 1 ? : ? : - ; # 14.
  0 1 (0x) 1 1 ? : 0 : 0 ; # 15.
  1 0 (0x) 1 1 ? : 1 : 1 ; # 16.

  * ? ? 1 1 ? : ? : - ; # 17.
  ? * ? 1 1 ? : ? : - ; # 18.
```

```

? ? ? 0 1 ? : ? : 0 ; # 19. clear
? ? ? 1 0 ? : ? : 1 ; # 20. set
? ? ? 0 0 ? : ? : 0 ; # 21. clear and set active
? ? ? 0 x ? : ? : 0 ; # 22. pessimism

? ? (?0) 1 1 ? : ? : - ; # 23. ignore falling clock.
? ? (1x) 1 1 ? : ? : - ; # 24.

? ? ? (?1) 1 ? : ? : - ; # 25. ignore changes on set and
? ? ? 1 (?1) ? : ? : - ; # 26. reset.
? ? ? ? ? * : ? : X ; # 27.
endtable
endprimitive

```

After adding these new entries TestMAX ATPG now produces a different violation message of:

```
underspecified UDP (Expected "xtr11:..." got "x?r11?:?:X")
```

We have succeeded in eliminating one warning message by adding a table entry which reduces pessimism in the model behavior. By repeatedly analyzing the UDP messages in this manner and adjusting the table we can eventually eliminate all of the warning messages due to missing entries.

Variant #3

As another example, a violation message of N28, *unsupported priority* can occur. This is an indication of incomplete information in the table needed for asynchronous set or clear behavior of the UDP to match the TestMAX ATPG primitive's asynchronous set/clear behavior. N28 violations deal generally with prioritization between asynchronous set, clear, and clocks.

For the UDP table shown below TestMAX ATPG issues an N28 violation:

```
unsupported priority (reset "CD" has no priority over other clocks)
```

```

primitive TOGGLE (Q, CP, CD, ntfy);
  output Q;
  reg Q;
  input CP, CD, ntfy;

  table
  # CP CD ntfy: Q- : Q+
  # --- --- --- : --- : --- ;
    (01) 1 ? : 0 : 1 ; # 1. toggle
    (01) 1 ? : 1 : 0 ; # 2. toggle

    0 1 ? : ? : - ; # 3. hold

    ? 0 ? : ? : 0 ; # 4. async clear

```

```

(01) x ? : 1 : 0 ; # 5. reduce pessimism
    0 x ? : 0 : 0 ; # 6. potential clear
    ? ? * : ? : X ; # 7. go to X
endtable
endprimitive

```

What TestMAX ATPG is trying to indicate with the N28 message is that the pin CD has been identified as an asynchronous reset but that it has not been completely described to have priority over "other clocks", or in this case the CP pin. The TestMAX ATPG primitive for a DFF device models the behavior of the asynchronous reset pin to have priority over the clock pin, so the ATPG primitive we wish to use does not exactly match this table.

If you are interested in eliminating the N28 violation look for entries in the table which describe asynchronous set or reset functions. Generally they define the behavior when the "clock" pins are at steady states. You should add an entry that defines the async behavior in the presence of clock events. In our example line "4b" is added to produce the model below:

```

primitive TOGGLE (Q, CP, CD, ntfy);
  output Q;
  reg Q;
  input CP, CD, ntfy;

  table
  # CP CD ntfy: Q- : Q+
  # --- --- --- : --- : --- ;
  (01) 1 ? : 0 : 1 ; # 1. toggle
  (01) 1 ? : 1 : 0 ; # 2. toggle

      0 1 ? : ? : - ; # 3. hold

      ? 0 ? : ? : 0 ; # 4. async clear
      * 0 ? : ? : 0 ; # 4b. async clear

  (01) x ? : 1 : 0 ; # 5. reduce pessimism
      0 x ? : 0 : 0 ; # 6. potential clear
      ? ? * : ? : X ; # 7. go to X
  endtable
endprimitive

```

Line 4 specifies that the output is cleared whenever CD=0 for any steady state value on CP (? = 0, 1, or x). By adding line 4b we also define the behavior that any edge event on CP while CD=0 also produces Q=0. This then fully describes an asynchronous reset behavior with priority over clocks and matches our ATPG primitive. This eliminates the N28 violation.

Variant #4

As another example, a violation message of N23, inconsistent entry can occur. This is an indication that two entries in the table define conflicting behavior.

For the UDP table shown below TestMAX ATPG issues the N23 violation:

```
inconsistent UDP (Entry should have clocks off: *?1??:-)

primitive DFF (Q, D, CP, SD, ntfy );
output Q;
input D, CP, SD, ntfy;
reg Q;
table
# D CP SD ntfy: Q- : Q+
# --- --- --- --- : --- : --- ;
  1 r 1 ? : ? : 1 ; # 1. clock D
  0 r 1 ? : ? : 0 ; # 2.

  ? ? 0 ? : ? : 1 ; # 3. async set
  ? * 0 ? : ? : 1 ; # 4.

  ? 0 1 ? : ? : - ; # 5. hold
  ? (?0) 1 ? : ? : - ; # 6.

  * ? ? ? : ? : - ; # 7. ignore edge
  ? ? (?1) ? : ? : - ; # 8. ignore edge

  1 (0x) 1 ? : 1 : 1 ; # 9. possible clock with Q- = D
  0 (0x) 1 ? : 0 : 0 ; # 10.

  1 r x ? : ? : 1 ; # 11. possible SD
  ? ? x ? : 1 : 1 ; # 12. possible SD, with Q- = 1

  1 (0x) x ? : 1 : 1 ; # 13. possible set, possible clock

  ? ? ? * : ? : X ; # 14. go to X
endtable
endprimitive
```

Reviewing the violation message against the lines in the table we can identify line 7 as the closest match to "* ? 1 ? : ? : -". The difference between the violation message and line 7 is that the violation message shows SD=1. In reviewing line 7 we see that this line indicate the output is held for any transition on the D input and for any values of CP and SD, including SD=0!!! So TestMAX ATPG is suggesting we change this line to indicate SD=1 is required.

```
primitive DFF (Q, D, CP, SD, ntfy );
output Q;
input D, CP, SD, ntfy;
reg Q;
table
# D CP SD ntfy: Q- : Q+
# --- --- --- --- : --- : --- ;
  1 r 1 ? : ? : 1 ; # 1. clock D
  0 r 1 ? : ? : 0 ; # 2.
```

```

? ? 0 ? : ? : 1 ; # 3. async set
? * 0 ? : ? : 1 ; # 4.

? 0 1 ? : ? : - ; # 5. hold
? (?0) 1 ? : ? : - ; # 6.

* ? 1 ? : ? : - ; # 7. ignore edge
? ? (?1) ? : ? : - ; # 8. ignore edge

1 (0x) 1 ? : 1 : 1 ; # 9. possible clock
0 (0x) 1 ? : 0 : 0 ; # 10.

1 r x ? : ? : 1 ; # 11. possible SD
? ? x ? : 1 : 1 ; # 12.
1 (0x) x ? : 1 : 1 ; # 13. possible set, possible clock

? ? ? * : ? : X ; # 14. go to X
endtable
endprimitive

```

After changing CP=? in line 7 to SD=1 we've eliminated the N23 violation.

Debugging UDP-based Models

The following advice is beneficial if you are debugging violation messages encountered when reading UDP modules:

1. Put the UDP definition into its own file until debugging is completed. By doing so, the only warning or error messages are from the single UDP you are debugging.
2. Enable display of ALL messages in the model, not just the most serious ones by issuing "set netlist -nocheck_only_used_udps" before reading the file containing the UDP.
3. Within TestMAX ATPG define an alias to make the repeated steps easier. For example, if the file containing the UDP is named 'udp.v' then define an alias named 'go' similar to:


```
alias go clear ; build -f ; read net udp.v -del ; \ run build ; rep viol -all
```
4. Now type "go" and review the violation messages for guidance on what might need changed. Next, edit your source 'udp.v' file, save the edits and return to TestMAX ATPG and type "go" again, or "!!". Repeat this process until you've eliminated as many violation messages as possible.
5. Review the resulting derived ATPG model in the graphical schematic viewer:
 - a. Use the SHOW button and select ALL.
 - b. Compare the gate level functionality of the derived ATPG model with the intended functionality of the truth table.

6. If your UDP table is complex, or results in more gates than you expected then consider using the `set_netlist -noxmodeling` option before reading in the UDP. This avoids trying to explicitly model output=X states. After this is done you can use a `write_netlist` command to get a gate level implementation of the simplified function using ATPG primitives. This might be helpful in trying to understand how to modify your UDP table to produce fewer ATPG gates.

Modeling Examples

- [Optimistic MUX](#)
- [MUX, 4-to-1](#)
- [Latch](#)
- [Latch With Active Low Asynchronous Set/Reset](#)
- [Latch With Asynchronous Set/Reset](#)
- [Latch With Asynchronous Set/Reset Dominant Over EN](#)
- [Latch With Asynchronous Set/Reset Dominant Over EN, Reset Dominant](#)
- [Latch With Asynchronous Set/Reset Dominant Over EN, Set Dominant](#)
- [Dual Port Latch](#)
- [Positive-edge Clocked DFF With Notify](#)
- [DFF With Active Low Asynchronous Set/Reset and Notify](#)
- [DFF With Active Low Asynchronous Reset and Notify](#)
- [DFF With Active High Asynchronous Set/Reset](#)
- [DFF With Synchronous Reset and Notify](#)
- [Negative-Edge Clocked DFF With Active Low Asynchronous Clear and Notify](#)
- [DFF and Latch](#)
- [JK Flip-Flop With Active Low Asynchronous Set/Reset and Notify](#)
- [Bus Keeper Examples](#)

Optimistic MUX

```
//  
// --- Optimistic MUX:  
// Y=D0 if SL=0  
// Y=D1 if SL=1
```

Chapter 4: ATPG Modeling Modeling Topics

```
// if SL=x and D0=D1, then Y=D0=D1
//
primitive MUX (Y, SL, D0, D1);
  output Y;
  input SL, D0, D1;
  table
  // SL D0 D1 : Y
  // -- -- -- : --
    0 0 ? : 0 ;
    0 1 ? : 1 ;

    1 ? 0 : 0 ;
    1 ? 1 : 1 ;

    x 0 0 : 0 ; // reduce pessimism
    x 1 1 : 1 ; // reduce pessimism

  endtable
endprimitive
```

MUX, 4-to-1

```
//
// --- Four-input to one-output non-inverting digital multiplexer.
//
primitive MUX41 (y, d0, d1, d2, d3, s0, s1);

  input d0, d1, d2, d3, s0, s1;
  output y;

  table

  // D0 D1 D2 D3 S0 S1 : Y
  // -- -- -- -- -- -- : ---
    0 ? ? ? 0 0 : 0 ;
    1 ? ? ? 0 0 : 1 ;

    ? 0 ? ? 1 0 : 0 ;
    ? 1 ? ? 1 0 : 1 ;

    ? ? 0 ? 0 1 : 0 ;
    ? ? 1 ? 0 1 : 1 ;

    ? ? ? 0 1 1 : 0 ;
    ? ? ? 1 1 1 : 1 ;

    0 0 0 0 ? ? : 0 ;
    1 1 1 1 ? ? : 1 ;

    0 0 ? ? ? 0 : 0 ;
    1 1 ? ? ? 0 : 1 ;
```

Chapter 4: ATPG Modeling Modeling Topics

```

    ? ? 0 0 ? 1 : 0 ;
    ? ? 1 1 ? 1 : 1 ;

    0 ? 0 ? 0 ? : 0 ;
    1 ? 1 ? 0 ? : 1 ;

    ? 0 ? 0 1 ? : 0 ;
    ? 1 ? 1 1 ? : 1 ;

    endtable
endprimitive

```

See Also

- Modeling Examples

Latch

```

primitive LATCH (Q, D, G);
output Q; reg Q;
input D, G;

table
// D G : Q- : Q+
// --- --- :----: ---
    ? 0 : ? : - ; // hold

    0 1 : ? : 0 ; // pass 0
    1 1 : ? : 1 ; // pass 1

    0 x : 0 : 0 ; // reduce pessimism
    1 x : 1 : 1 ; // reduce pessimism

endtable
endprimitive

```

See Also

- Modeling Examples

Latch With Active Low Asynchronous Set/Reset

```

//
// --- active high level sensitive latch with active
// low set and reset.
//
primitive LAT_SB_RB (Q, D, G, SB, RB);
output Q; reg Q;
input D, G, SB, RB;
table
// D G SB RB : Q- : Q+

```

```

// --- :---: ---
  ? 0 1 1 : ? : - ; // hold

  0 1 1 1 : ? : 0 ; // pass 0
  1 1 1 1 : ? : 1 ; // pass 1

  ? 0 0 1 : ? : 1 ; // async set
  ? 0 1 0 : ? : 0 ; // async clear

  1 1 ? 1 : ? : 1 ; // G and SB active
  0 1 1 ? : ? : 0 ; // G and RB active

  1 x 0 1 : ? : 1 ; // G=X and SB
  0 x 1 0 : ? : 0 ; // G=X and RB

  0 x 1 1 : 0 : 0 ; // G=X, Q=D
  1 x 1 1 : 1 : 1 ;

  ? 0 x 1 : 1 : 1 ; // SB=X, Q=1
  ? 0 1 x : 0 : 0 ; // RB=X, Q=0

  1 x x 1 : 1 : 1 ; // G=SB=X
  0 x 1 x : 0 : 0 ; // G=RB=X

  endtable
endprimitive

```

See Also

- Modeling Examples

Latch With Asynchronous Set/Reset

```

//
// --- active high level sensitive latch with active
// high set and reset.
//
primitive DLAT1 (q, set, rst, en, data);
  output q; reg q;
  input set, rst, en, data;
  table
  // set rst en d : Q- : Q+
  // --- :---: ---
    0 0 0 ? : ? : - ; // hold

    0 0 1 0 : ? : 0 ; // pass 0
    0 0 1 1 : ? : 1 ; // pass 1

    1 0 0 ? : ? : 1 ; // async set
    0 1 0 ? : ? : 0 ; // async clear
  endtable
endprimitive

```

Chapter 4: ATPG Modeling Modeling Topics

```

    ? 0 1 1 : ? : 1 ; // ena and set active
    0 ? 1 0 : ? : 0 ; // ena and rst active

    1 0 x 1 : ? : 1 ; // ena=X and set
    0 1 x 0 : ? : 0 ; // ena=X and rst

    0 0 x 0 : 0 : 0 ; // ena=X, Q=D
    0 0 x 1 : 1 : 1 ;

    x 0 0 ? : 1 : 1 ; // set=X, Q=1
    0 x 0 ? : 0 : 0 ; // rst=X, Q=0

    x 0 x 1 : 1 : 1 ; // ena=set=X
    0 x x 0 : 0 : 0 ; // ena=rst=X

    endtable
endprimitive

```

See Also

- Modeling Examples

Latch With Asynchronous Set/Reset Dominant Over EN

```

//
// --- active high level sensitive latch with active
// high set and reset dominant over EN.
//
primitive DLAT2 (q, set, rst, en, data);
output q; reg q;
input set, rst, en, data;
table
// set rst en d : Q- : Q+
// --- --- --- --- :----: ---
    0 0 0 ? : ? : - ; // hold

    1 0 ? ? : ? : 1 ; // async set
    0 1 ? ? : ? : 0 ; // async clear

    0 0 1 0 : ? : 0 ; // pass 0
    0 0 1 1 : ? : 1 ; // pass 1

    ? 0 x 1 : 1 : 1 ; // ena=X, Q=D
    0 ? x 0 : 0 : 0 ;

    x 0 0 ? : 1 : 1 ; // set=X, Q=1
    0 x 0 ? : 0 : 0 ; // rst=X, Q=0

    x 0 x 1 : 1 : 1 ; // ena=set=X
    0 x x 0 : 0 : 0 ; // ena=rst=X

```

```
    endtable
endprimitive
```

See Also

- Modeling Examples

Latch With Asynchronous Set/Reset Dominant Over EN, Reset Dominant

```
//
// --- active high level sensitive latch with active
// high set and reset dominant over EN, RST dominates SET.
//
primitive DLAT3 (q, set, rst, en, data);
    output q; reg q;
    input set, rst, en, data;
    table
    // set rst en d : Q- : Q+
    // --- --- --- --- :----: ---
        0 0 0 ? : ? : - ; // hold

        ? 1 ? ? : ? : 0 ; // async clear, dominant
        1 0 ? ? : ? : 1 ; // async set

        0 0 1 0 : ? : 0 ; // pass 0
        0 0 1 1 : ? : 1 ; // pass 1

        ? 0 x 1 : 1 : 1 ; // ena=X, Q=D
        0 ? x 0 : 0 : 0 ;

        x 0 0 ? : 1 : 1 ; // set=X, Q=1
        0 x 0 ? : 0 : 0 ; // rst=X, Q=0

        x 0 x 1 : 1 : 1 ; // ena=set=X
        0 x x 0 : 0 : 0 ; // ena=rst=X

    endtable
endprimitive
```

See Also

- Modeling Examples

Latch With Asynchronous Set/Reset Dominant Over EN, Set Dominant

```
//
// --- active high level sensitive latch with active
```



```
// high set and reset dominant over EN, SET dominates RST.
//
primitive DLAT4 (q, set, rst, en, data);
  output q; reg q;
  input set, rst, en, data;
  table
  // set   rst   en   d   :   Q- :   Q+
  // ---   ---   --- --- :   --- :   ---
    0     0     0   ?   :   ? :   - ; // hold

    1     ?     ?   ?   :   ? :   1 ; // async set, dominant
    0     1     ?   ?   :   ? :   0 ; // async clear

    0     0     1   0   :   ? :   0 ; // pass 0
    0     0     1   1   :   ? :   1 ; // pass 1

    ?     0     x   1   :   1 :   1 ; // ena=X, Q=D
    0     ?     x   0   :   0 :   0 ;

    x     0     0   ?   :   1 :   1 ; // set=X, Q=1
    0     x     0   ?   :   0 :   0 ; // rst=X, Q=0

    x     0     x   1   :   1 :   1 ; // ena=set=X
    0     x     x   0   :   0 :   0 ; // ena=rst=X

  endtable
endprimitive
```

See Also

- Modeling Examples

Dual Port Latch

```
//
// --- dual port latch. D1 enabled by G1 and D2 enabled
// by G2.
//
primitive DPLATCH (Q, D1, G1, D2, G2);
  output Q; reg Q;
  input D1,G1,D2,G2;
  table
  // D1 G1 D2 G2 Q- Q+
  // --- --- --- --- : --- : ---
    ? 0 ? 0 : ? : - ; // hold

    0 1 ? 0 : ? : 0 ;
    1 1 ? 0 : ? : 1 ;

    ? 0 0 1 : ? : 0 ;
    ? 0 1 1 : ? : 1 ;
  endtable
endprimitive
```

```

    0 1 0 1 : ? : 0 ;
    1 1 1 1 : ? : 1 ;

    0 x 0 1 : ? : 0 ;
    1 x 1 1 : ? : 1 ;
    0 x ? 0 : 0 : 0 ;
    1 x ? 0 : 1 : 1 ;

    0 1 0 x : ? : 0 ;
    1 1 1 x : ? : 1 ;

    ? 0 0 x : 0 : 0 ;
    ? 0 1 x : 1 : 1 ;

    0 x 0 x : 0 : 0 ;
    1 x 1 x : 1 : 1 ;

endtable
endprimitive

```

See Also

- Modeling Examples

Positive-edge Clocked DFF With Notify

```

// FUNCTION : POSITIVE EDGE TRIGGERED D FLIP-FLOP WITH NOTIFY.
//
primitive DFF_N (q, d, clk, ntfy);
  output q;
  input d, clk, ntfy;
  reg q;

  table
  // d   clk   ntfy : q- : q+
  // --- --- --- : --- : ---
    *   ?     ?   : ?   : - ; // hold
    ?   (?0)  ?   : ?   : - ; // hold
    ?   (1?)  ?   : ?   : - ; // hold

    0   (01)  ?   : ?   : 0 ; // clock 0
    1   (01)  ?   : ?   : 1 ; // clock 1

    0   (0x)  ?   : 0   : 0 ; // clk=X with D=Q=0
    1   (0x)  ?   : 1   : 1 ; // clk=X with D=Q=1

    0   (x1)  ?   : 0   : 0 ; // possibly clk
    1   (x1)  ?   : 1   : 1 ;

    ?   ?     *   : ?   : x ;

```

```
endtable
endprimitive
```

See Also

- Modeling Examples

DFF With Active Low Asynchronous Set/Reset and Notify

```
//
// --- positive edge clocked DFF with active low
// asynchronous set and reset and also a notify
// function.
//
primitive DFF_SB_RB_N (Q, D, CLK, RB, SB ,notifier);
output Q;
input D, CLK, RB, SB, notifier;
reg Q;

table
// D CLK RB SB noti : q- : q+
// --- --- --- --- ---- : --- : ---
* ? 1 1 ? : ? : - ; // hold
? (?0) 1 1 ? : ? : - ; // hold
? (1?) 1 1 ? : ? : - ; // hold
? ? (?1) 1 ? : ? : - ; // hold
? ? 1 (?1) ? : ? : - ; // hold

0 (01) ? 1 ? : ? : 0 ; // clock 0
1 (01) 1 ? ? : ? : 1 ; // clock 1

? ? 1 0 ? : ? : 1 ; // active low preset
? ? 0 1 ? : ? : 0 ; // active low clear

? ? 1 x ? : 1 : 1 ; // preset=X with Q=1
? ? x 1 ? : 0 : 0 ; // clear=X with Q=0

0 (0x) ? 1 ? : 0 : 0 ; // clk=X with D=Q=0
1 (0x) 1 ? ? : 1 : 1 ; // clk=X with D=Q=1

D=Q=0
0 (x1) ? 1 ? : 0 : 0 ; // possible clk with
D=Q=1
1 (x1) 1 ? ? : 1 : 1 ; // possible clk with

? ? ? ? * : ? : x ; // this line is
unsupported; it references a // notify construct and
generates
```

```

TestMAX ATPG
endtable
endprimitive
// N21 warnings in

```

See Also

- Modeling Examples

DFF With Active Low Asynchronous Reset and Notify

```

// FUNCTION : POSITIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
// ASYNCHRONOUS CLEAR WITH NOTIFY.
//
primitive DFF_RB_N (q, d, clk, rb, ntfy);
    output q;
    input d, clk, rb, ntfy;
    reg q;

    table
// d    clk  rb  ntfy : q-  : q+
// ---  ---  ---  ---- : ---  : ---
    *   ?   ?   ?   : ?   : - ; // hold
    ?   (?0) ?   ?   : ?   : - ; // hold
    ?   (1?) ?   ?   : ?   : - ; // hold
    ?   0   (?1) ?   ?   : ?   : - ; // hold

    0   (01) ?   ?   : ?   : 0 ; // clock 0
    1   (01) 1   ?   : ?   : 1 ; // clock 1

    ?   ?   0   ?   : ?   : 0 ; // clear

    0   (01) x   ?   : ?   : 0 ; // clock 0 with RB=x
    ?   ?   (?x) ?   : 0   : 0 ; // RB=X with Q=0

    0   (0x) ?   ?   : 0   : 0 ; // clk=X with D=Q=0, possible clear
    1   (0x) 1   ?   : 1   : 1 ; // clk=X with D=Q=1

    0   (x1) ?   ?   : 0   : 0 ; // possible clock
    1   (x1) 1   ?   : 1   : 1 ;

    ?   ?   ?   *   : ?   : x ;

    endtable
endprimitive

```

See Also

- Modeling Examples

DFF With Active High Asynchronous Set/Reset

```
//
// --- positive edge clocked DFF with active high
// asynchronous set and clear.
//
primitive DFF_S_R (q, set, rst, clk, data);
  output q;
  input set, rst, clk, data;
  reg q;

  table
  // set   rst   clk   data   :   q-   :   q+
  // ---   ---   ---   ----   :   ---   :   ---
    0     0     ?     *     :   ?     :   - ; // hold
    0     0     (?0)  ?     :   ?     :   - ; // hold
    0     0     (1?)  ?     :   ?     :   - ; // hold
    (?0)  0     ?     ?     :   ?     :   - ; // hold
    0     (?0)  ?     ?     :   ?     :   - ; // hold

    1     0     ?     ?     :   ?     :   1 ; // set
    0     1     ?     ?     :   ?     :   0 ; // reset

    0     ?     (01)  0     :   ?     :   0 ; // clock data
    ?     0     (01)  1     :   ?     :   1 ; // clock data

    0     ?     (0x)  0     :   0     :   0 ; // possible clk, D=Q=0
    ?     0     (0x)  1     :   1     :   1 ; // possible clk, D=Q=1

    0     ?     (x1)  0     :   0     :   0 ; // possible clk
    ?     0     (x1)  1     :   1     :   1 ;

    0     x     ?     ?     :   0     :   0 ; // rst=X with Q=0
    x     0     ?     ?     :   1     :   1 ; // set=X with Q=1

  endtable
endprimitive
```

See Also

- Modeling Examples

DFF With Synchronous Reset and Notify

```
//
// --- Positive edge clocked DFF with active low synchronous// reset and
// a notify function.
//
primitive DFF_SR_N (Q, D, CLK, SR ,notifier);
  output Q;
  input D, CLK, SR, notifier;
```

```

reg Q;

table
// D CLK SR noti : q- : q+
// --- : --- : ---
? (?0) ? ? : ? : - ; // hold
? (1?) ? ? : ? : - ; // hold
* ? ? ? : ? : - ; // ignore changes on D
? ? * ? : ? : - ; // ignore changes on SR

? (01) 0 ? : ? : 0 ; // synchronous clear
0 (01) ? ? : ? : 0 ; // clock D=0
1 (01) 1 ? : ? : 1 ; // clock D=1

? (0x) 0 ? : 0 : 0 ; // clk=X with RB=Q=0
0 (0x) ? ? : 0 : 0 ; // clk=X with D=Q=0
1 (0x) 1 ? : 1 : 1 ; // clk=X with D=Q=1

? (x1) 0 ? : 0 : 0 ; // possible clk
0 (x1) ? ? : 0 : 0 ;
1 (x1) 1 ? : 1 : 1 ;

? ? ? * : ? : x ;

endtable
endprimitive

```

See Also

- Modeling Examples

Negative-Edge Clocked DFF With Active Low Asynchronous Clear and Notify

The following is an example of a negative-edge triggered D flip-flop with active low asynchronous clear and notify.

```

// FUNCTION : NEGATIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
// ASYNCHRONOUS CLEAR AND NOTIFY.
//
primitive NDDFF_RB_N (q, d, clk, rst, ntfy);
output q;
input d, clk, rst, ntfy;
reg q;

table

// d clk rst ntfy : q- : q+
// --- : --- : ---
* ? 1 ? : ? : - ; // hold
? (?1) 1 ? : ? : - ; // hold

```

Chapter 4: ATPG Modeling Modeling Topics

```

? (0?) 1 ? : ? : - ; // hold
? ? (?1) ? : ? : - ; // hold

0 (10) ? ? : ? : 0 ; // clock 0
1 (10) 1 ? : ? : 1 ; // clock 1

? ? 0 ? : ? : 0 ; // clear
? * 0 ? : ? : 0 ; // clk while reset

0 (10) x ? : 0 : 0 ;
? ? (?x) ? : 0 : 0 ; // reset=X with Q=0

0 (1x) ? ? : 0 : 0 ; // clk=X with D=Q=0
1 (1x) 1 ? : 1 : 1 ; // clk=X with D=Q=1

0 (x0) ? ? : 0 : 0 ; // possible clk
1 (x0) 1 ? : 1 : 1 ;

? ? ? * : ? : x ;

endtable
endprimitive

```

See Also

- Modeling Examples

DFF and Latch

```

//
// --- mixed positive edge clocked input plus level
// sensitive latched input.
// D1,CLK is edge sensitive
// D2,G is level sensitive
//
primitive FLOPLATCH (Q, D1,CLK, D2,G);
output Q;
input D1, CLK, D2, G;
reg Q;

table
// D1   CLK   D2   G   :   q-   :   q+
// ---   ---   ---   --- :   ---   :   ---
  *     0     ?     0   :   ?     :   - ; // hold for data changes
  ?     0     *     0   :   ?     :   - ; // hold for data changes

  ?     0     ?     0   :   ?     :   - ; // hold for CK off

  ?     ?     0     1   :   ?     :   0 ; // latch enabled, D2=0
  ?     ?     1     1   :   ?     :   1 ; // latch enabled, D2=1

```

```

0   (01)   ?   0   :   ?   :   0   ;   // clock 0
1   (01)   ?   0   :   ?   :   1   ;   // clock 1

0   (01)   0   1   :   ?   :   0   ;   // clock with latch active
1   (01)   1   1   :   ?   :   1   ;   // clock with latch active

0   (01)   1   1   :   ?   :   1   ;   // latch overrides clock
1   (01)   0   1   :   ?   :   0   ;

0   (01)   0   x   :   ?   :   0   ;   // clock with possible latch
1   (01)   1   x   :   ?   :   1   ;

?    0    0   x   :   0   :   0   ;   // latch=X, D2=Q=0
?    0    1   x   :   1   :   1   ;   // latch=X, D2=Q=1

0   (0x)   0   1   :   ?   :   0   ;   // latch on, possible clock
1   (0x)   1   1   :   ?   :   1   ;

0   (0x)   1   1   :   ?   :   1   ;   // latch overrides clock
1   (0x)   0   1   :   ?   :   0   ;

0   (0x)   ?   0   :   0   :   0   ;   // possible clock, D1=Q=0
1   (0x)   ?   0   :   1   :   1   ;

0   (0x)   0   x   :   0   :   0   ;   // D1=Q=0, possible clocks
1   (0x)   1   x   :   1   :   1   ;

    endtable
endprimitive

```

See Also

- Modeling Examples

JK Flip-Flop With Active Low Asynchronous Set/Reset and Notify

```

//
// --- Positive edge clocked JK flip-flop with active low
// asynchronous set/rst plus a notify function.
//
`celldefine
module JK_SCAN_SB_RB_N (Q, J, K, TI, TE, CP, CD, SD, notifier);
    output Q;
    reg Q;
    input J, K, TI, TE, CP, CD, SD, notifier;

    MUX m1 (jknet, Q, J, !K);
    MUX m2 (din , TE, jknet, TI);
    DFF_SB_RB_N r1 (Q, din, CP, CD, SD, notifier);
endmodule

```



```
`endcelldefine
```

See Also

- Modeling Examples
- [UDP Example: MUX](#)
- [UDP Example: DFF_SB_RB_N](#)

Bus Keeper Examples

TestMAX ATPG attempts to automatically identify modules that are acting in a bus keeper or bus hold manner. However, it is sometimes useful to explicitly assist in this identification process. This is done by adding the "BUSK0", "BUSK1", or "BUSK01" net attribute into the module definition as in buskeep2/3/4 below. You can also model a bus keeper using an empty module with a single bidirectional pin and the BUSK attribute as in buskeep1 below.

The "BUSK01" holds both a 0 and 1. The "BUSK1" holds just a 1, and the "BUSK0" holds just a 0.

```
#
# #1 - null module
#
module buskeep1 (X);
  inout X;
  _BUSK01 X;
endmodule

#
# #2 - back-to-back inverters
#
module buskeep2 (X);
  inout X;
  wire fb;
  _BUSK01 X; # this line is optional
  not u1 (fb, X);
  not (weak0,weak1) u2 (X, fb);
endmodule

#
# #3 - back-to-back buffers
#
module buskeep3 (X);
  inout X;
  _BUSK01 X; # this line is optional
  buf u1 (fb, X);
  buf (weak0,weak1) u2 (X, fb);
endmodule
```

```

#
# #4 - single buffer feedback loop
#
module buskeep4 (X);
  inout X;
  _BUSK01 X; # this line is optional
  buf (weak0,weak1) u1 (X,X);
endmodule

```

See Also

- [Modeling Examples](#)

Scan Cell Models

This chapter contains the following topics:

- [Scan Cell Models - MUX Flop Scan](#)
- [Scan Cell Models - Master Slave Latch](#)
- [Scan Cell Models - MUX Latch Scan](#)
- [Scan Cell Models - Clocked Scan Flip-Flop](#)
- [Scan Cell Models - Clocked Scan Latch](#)
- [Scan Cell Models - Single Latch LSSD](#)
- [Scan Cell Models - Double Latch LSSD](#)
- [Scan Cell Models - Clocked LSSD](#)
- [Scan Cell Models - Auxiliary Clocked LSSD](#)
- [Scan Cell Models - Retention Cell](#)

Scan Cell Models - MUX Flop Scan

The MUX Flip-Flop Scan uses a MUX in front of the data input to an edge sensitive D-flop. In this example, the "SE" pin selects either "D" or "SDI" input and the flop is clocked on the rising edge of "CLK".

```

primitive mux_flop (Q, SE, D, SI, CLK);
  output Q; reg Q;
  input SE, D, SI, CLK;

  table
  // SE D SI CLK : Q- : Q+
  // --- --- --- ----- : --- : ---

```

Chapter 4: ATPG Modeling

Scan Cell Models

```

    0 0 ? (01) : ? : 0 ;
    0 1 ? (01) : ? : 1 ;
    1 ? 0 (01) : ? : 0 ;
    1 ? 1 (01) : ? : 1 ;

// MUX select is X
    x 0 0 (01) : ? : 0 ;
    x 1 1 (01) : ? : 1 ;
    x 0 0 (0x) : 0 : 0 ;
    x 1 1 (0x) : 1 : 1 ;

// stable clock, but data changes

    (??) ? ? ? : ? : - ;
    ? (??) ? ? : ? : - ;
    ? ? (??) ? : ? : - ;

// X->1 transitions of clock

    0 0 ? (x1) : 0 : 0 ;
    0 1 ? (x1) : 1 : 1 ;

    1 ? 0 (x1) : 0 : 0 ;
    1 ? 1 (x1) : 1 : 1 ;

// transitions of clock to 0

    ? ? ? (10) : ? : - ;
    ? ? ? (x0) : ? : - ;

// transitions of clock to X
    x ? ? (1x) : ? : - ;
    ? ? ? (1x) : ? : - ;

    0 0 ? (0x) : 0 : 0 ;
    0 1 ? (0x) : 1 : 1 ;

    1 ? 0 (0x) : 0 : 0 ;
    1 ? 1 (0x) : 1 : 1 ;

    ? ? ? (0x) : ? : x ;
endtable
endprimitive

```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, and test_setup procedures.

```

// clk has been defined as an active high clock
Procedures {
  load_unload {
    V { test=1; scan_en=1; bidi_en=0; clk=0; bidi_port=Z; }
    V { bidi_port=1; }
  }
}

```

```

Shift {
  V { _si=#; _so=#; clk=P; }
}
}

MacroDefs {
  test_setup {
    V { bidi_en=0; test=1; bidi_port=Z; clk=0; }
  }
}

```

Scan Cell Models - Master Slave Latch

The Master-Slave Latch features a MUX preceding a pair of latches. Each latch has a separate enable. In this example, the "SE" pin selects either the "D" or "SDI" input into the master latch, which is enabled by "MCLK". The output of the master latch feeds the slave latch which is enabled by "SCLK".

```

module mux_ms_latch( D, SDI, SE, MCLK, SCLK, Q);
  input D, SDI, SE, MCLK, SCLK;
  output Q;
  MUX mx1 (.D0(D), .D1(SDI), .SL(SE), .Y(din));
  LATCH master (.G(MCLK), .D(din), .Q(mq));
  LATCH slave (.G(SCLK), .D(mq), .Q(Q));
endmodule

```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, and test_setup procedures.

```

// mclk and sclk have been defined to be active high clocks
Procedures {
  load_unload {
    V { mclk=0; sclk=0;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
    V { bidi_port=1; }
    Shift {
      V { _si=#; _so=#; mclk=P; sclk=0; }
      V { mclk=0; sclk=P; }
    }
    V { sclk=0; }
  }

  master_observe {
    V { sclk=P; }
  }
}

```

```
MacroDefs {
  test_setup {
    V { mclk=0; sclk=0;
      bidi_en=0; test=1; bidi_port=Z; }
  }
}
```

Scan Cell Models - MUX Latch Scan

The MUX Latch is similar to the Master Slave Latch style of scan with the exception that it uses a single clock. The slave latch uses the opposite polarity of the clock for the master. In this example the "SE" pin selects either the "D" or "SDI" pin for data input and the "CLK" pin is the latch enable. Because of the clocking scheme this style of scan is functionally identical to the MUX Flop Scan.

```
module mux_latch (D, SDI, SE, CLK, Q, SDO);
  input D, SDI, SE, CLK;
  output Q, SDO;
  not (clkb, CLK);
  MUX mx1 (.D0(D), .D1(SDI), .SL(SE), .Y(din));
  LATCH master (.G(CLK), .D(din), .Q(Q));
  LATCH slave (.G(clkb), .D(Q), .Q(SDO));
endmodule
```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, and test_setup procedures.

```
// clk has been defined as an active high clock
Procedures {
  load_unload {
    V { test=1; scan_en=1; bidi_en=0; clk=0; bidi_port=Z;}
    V { bidi_port=1;}
    Shift {
      V { _si=#; _so=#; clk=P; }
    }
  }
}

MacroDefs {
  test_setup {
    V { bidi_en=0; test=1; bidi_port=Z; clk=0; }
  }
}
```

Scan Cell Models - Clocked Scan Flip-Flop

The Clocked Scan Flip-Flop uses a latch for both the "D" and "SDI" inputs with separate latch enables, and then feeds the outputs of these master latches into a single dual port slave latch. The slave latch enables use the compliment of the master latch enables. In this example, "CLK" clocks the "D" input to the "Q" output and "SCLK" clocks the "SDI" input to the "Q" output. Care must be taken that both clocks are not active at the same time.

```
primitive clocked_scan_flop_1 (Q, SDI, SCLK, D, CLK);
  output Q; reg Q;
  input SDI, SCLK, D, CLK;

  table
  // SDI SCLK D CLK : 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 : ? : - ;

  // clock recovers from X
    ? ? 0 (x1) : 0 : 0 ;
    ? ? 1 (x1) : 1 : 1 ;
    0 (x1) ? ? : 0 : 0 ;
    1 (x1) ? ? : 1 : 1 ;

  // clock goes to X
    ? ? 0 (?x) : 0 : 0 ;
    ? ? 1 (?x) : 1 : 1 ;
    0 (?x) ? ? : 0 : 0 ;
    1 (?x) ? ? : 1 : 1 ;

  // clock with other clock is active or X
    0 r ? x : 0 : 0 ;
    1 r ? x : 1 : 1 ;

    0 r ? 1 : 0 : 0 ;
    1 r ? 1 : 1 : 1 ;

  endtable
endprimitive
```

Chapter 4: ATPG Modeling

Scan Cell Models

```

module clocked_scan_flop_2 (Q, SDI, SCLK, D, CLK);
  output Q; reg Q;
  input SDI, SCLK, D, CLK;

  not u1 (ckb, CLK);
  not u2 (skb, SCLK);

  LATCH rm1 (n1, D, ckb);
  LATCH rm2 (n2, SDI, skb);

  DPLATCH rs (Q, n1,CLK, n2,SCLK);
endmodule

```

For designs using this type of scan cell model here is a representative test procedure file showing the load_unload, shift, and test_setup procedures.

```

// clk and sclk have been defined as active high clocks
Procedures {
  load_unload {
    V { clk=0; sclk=0;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
    V { bidi_port=1; }
    Shift {
      V { _si=#; _so=#; sclk=P; }
    }
  }
}

MacroDefs {
  test_setup {
    V { bidi_en=0; test=1; bidi_port=Z; clk=0; sclk=0; }
  }
}

```

Scan Cell Models - Clocked Scan Latch

The Clocked Scan Latch is a hybrid device. The "D" to "Q" path passes through a latch enabled by "G" but the "SDI" to "Q" path passes through master-slave latches to create an edge clocked behavior controlled by "SCLK". It is important to ensure that "G" is not asserted when using "SCLK".

```

module clocked_scan_latch_1 (Q, D, G, SDI, SCLK);
  input D, G, SDI, SCLK;
  output Q;
  FLOPLATCH r1 (Q, SDI,SCLK, D,G);
endmodule

```

Chapter 4: ATPG Modeling

Scan Cell Models

```

module clocked_scan_latch_2 (Q, D, G, SDI, SCLK);
  output Q;
  input D, G, SDI, SCLK;
  not ul (scb, SCLK);
  LATCH rlat (q1, SDI, scb);
  DPLATCH rdff (Q, D,G, q1,SCLK);
endmodule

```

For designs using this type of scan cell model here is a representative set of load_unload, shift, and test_setup procedures.

```

// latch_en and sclk have been defined as active high clocks
Procedures {
  load_unload {
    V { latch_en=0; sclk=0;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
    V { bidi_port=1; }
  Shift {
    V { _si=#; _so=#; sclk=P; }
  }
}

MacroDefs {
  test_setup {
    V { bidi_en=0; test=1; bidi_port=Z; latch_en=0; sclk=0; }
  }
}

```

Scan Cell Models - Single Latch LSSD

The Single Latch LSSD uses a dual port latch for a front end. The "D" to "Q" path passes through a single latch enabled by "G". The scan data path has an additional latch on the output with a separate enable "SCKB". In normal mode the "G" enables data flow. In scan mode the combination of non-overlapping "SCKA" and "SCKB" shift data from "SDI" to "SDO".

```

module single_latch_lssd (Q, SDO, D, G, SDI, SCKA, SCKB);
  output Q, SDO;
  input D, G, SDI, SCKA, SCKB;

  DPLATCH rdff (Q, D,G, SDI,SCKA);
  LATCH rlat (SDO, Q,SCKB);
endmodule

```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, master_observe, and test_setup procedures.


```
// latch_clk, scka, and sckb have been defined as active high clocks
Procedures {
  load_unload {
    V { latch_clk=0; scka=0; sckb=0;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
    V { bidi_port=1; }
    Shift {
      V { _si=#; _so=#; scka=P; sckb=0; }
      V { scka=0; sckb=P; }
    }
    V { sckb=0; }
  }

  master_observe {
    V { sckb=P; }
  }
}

MacroDefs {
  test_setup {
    V { latch_clk=0; scka=0; sckb=0;
      bidi_en=0; test=1; bidi_port=Z; }
  }
}
```

Scan Cell Models - Double Latch LSSD

The Double Latch LSSD is similar to the Single Latch LSSD with the exception that the "D" to "Q" path passes through two latches instead of one. In normal mode, data is moved from "D" to "Q" by the application of non-overlapping "G" followed by "SCKB". In scan mode data is moved from "SDI" to "Q" by the application of "SCKA" followed by "SCKB".

```
module double_latch_lassd (Q,D, G, SDI, SCKA, SCKB);
  output Q;
  input D, G, SDI, SCKA, SCKB;

  DPLATCH rdff (q1, D,G, SDI,SCKA);
  LATCH rlat (Q, q1,SCKB);
endmodule
```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, master_observe, and test_setup procedures.

```
// latch_clk, scka, and sckb have been defined as active high clocks
Procedures {
  load_unload {
    V { latch_clk=0; scka=0; sckb=0;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
  }
}
```

```

    V { bidi_port=1; }
    Shift {
        V { _si=#; _so=#; scka=P; sckb=0; }
        V { scka=0; sckb=P; }
    }
    V { sckb=0; }
}

master_observe {
    V { sckb=P; }
}

}

MacroDefs {
    test_setup {
        V { latch_clk=0; scka=0; sckb=0;
            bidi_en=0; test=1; bidi_port=Z; }
    }
}

```

Scan Cell Models - Clocked LSSD

In the Clocked LSSD model, the "D" to "Q" path exhibits clocked behavior controlled from the "CLK" input. The "SDI" to "Q" path uses master-slave latching with separate enables "SCKA" and "SCKB".

```

module clocked_lssd_1 (Q, D, CLK, SDI, SCKA, SCKB);
    output Q;
    input D, CLK, SDI, SCKA, SCKB;
        FLOPLATCH r1 (n1, D,CLK, SDI,SCKA);
        FLOPLATCH r2 (Q, D,CLK, n1, SCKB);
endmodule

module clocked_lssd_2 (Q, D, CLK, SDI, SCKA, SCKB);
    output Q;
    input D, CLK, SDI, SCKA, SCKB;

        not u1 (clk, CLK);
        LATCH rmd (d1, D,clkb);
        LATCH rms (d2, SDI,SCKA);
        DPLATCH rs (Q, d1,CLK, d2,SCKB);
endmodule

module clocked_lssd_3 (Q, SDO, D, CLK, SDI, SCKA, SCKB);
    output Q, SDO;
    input D, CLK, SDI, SCKA, SCKB;
        not u1 (clkb, CLK);
        LATCH rm (d1, D,clkb);

```

Chapter 4: ATPG Modeling

Scan Cell Models

```

        DPLATCH rs1 (Q, d1,CLK, SDI,SCKA);
        LATCH rs2 (SDO, Q,SCKB);
    endmodule

module clocked_lssd_4 (Q, SDO, D, CLK, SDI, SCKA, SCKB);
    output Q, SDO;
    input D, CLK, SDI, SCKA, SCKB;
    not u1 (clkb, CLK);
    not u2 (sckab, SCKA);
    and u3 (sck, sckab, CLK);
    DPLATCH rm (d1, D,clkb, SDI,sck);
    LATCH rs1 (Q, d1,CLK);
    LATCH rs2 (SDO, Q,SCKB);
endmodule

module clocked_lssd_5 (Q, D, CLK, SDI, SCKA, SCKB);
    output Q;
    input D, CLK, SDI, SCKA, SCKB;
    LATCH r1 (n1, SDI,SCKA);
    FLOPLATCH r2 (Q, D,CLK, n1,SCKB);
endmodule

module clocked_lssd_6 (Q, SDO, D, CLK, SDI, SCKA, SCKB);
    output Q, SDO;
    input D, CLK, SDI, SCKA, SCKB;
    FLOPLATCH r2 (Q, D,CLK, SDI,SCKA);
    LATCH r1 (SDO, Q,SCKB);
endmodule

```

For designs using this type of scan cell model here is a representative sample of load_unload, shift, master_observe, and test_setup procedures.

```

// clk, scka, and sckb have been defined as active high clocks
Procedures {
    load_unload {
        V { clk=0; scka=0; sckb=0;
            test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
        V { bidi_port=1; }
        Shift {
            V { _si=#; _so=#; scka=P; sckb=0; }
            V { scka=0; sckb=P; }
        }
        V { sckb=0; }
    }

    master_observe {
        V { sckb=P; }
    }
}

```

```

}

MacroDefs {
  test_setup {
    V { clk=0; scka=0; sckb=0;
      bidi_en=0; test=1; bidi_port=Z; }
  }
}

```

Scan Cell Models - Auxiliary Clocked LSSD

The Auxiliary Clocked LSSD features a "D" to "Q" path that exhibits clocked behavior using either "CLK" or "TCK" with SCKA=SCKB=0. For scan mode, "TCK" and "SCKB" are brought high and data is shifted from "SDI" to "Q" by non-overlapping enables "SCKA" (active high) followed by "SCKB" (active low).

```

module auxiliary_clocked_lssd (Q, D,CLK, TCK, SDI,SCKA,SCKB);
  output Q;
  input D, CLK, TCK, SDI, SCKA, SCKB;

  or u1 (c1, CLK, TCK);
  not u2 (c1b, c1);
  not u3 (bcb, SCKB);
  and u4 (c2, bcb,c1);
  DPLATCH rm (n1, D,c1b, SDI,SCKA);
  LATCH rs (Q, n1,c2);
endmodule

```

For designs using this type of scan cell model here is a representative test procedure file.

```

// clk and sckb have been define as active low clocks
// scka has been defined as an active high clock
// tck is not defined as a clock
//
Procedures {
  load_unload {
    V { clk=1; tck=1; scka=0; sckb=1;
      test=1; scan_en=1; bidi_en=0; bidi_port=Z; }
    V { bidi_port=1; }
    Shift {
      V { _si=#; _so=#; scka=P; sckb=1; }
      V { scka=0; sckb=P; }
    }
    V { scka=0; sckb=1; }
  }

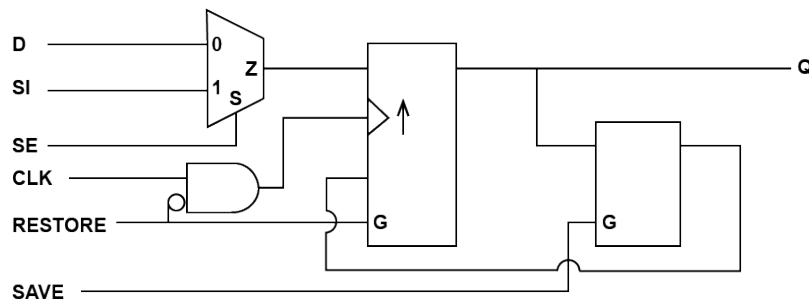
  master_observe {
    V { sckb=P; }
  }
}

```

```
}  
  
MacroDefs {  
  test_setup {  
    V { clk=1; tck=1; scka=0; sckb=1;  
      bidi_en=0; test=1; bidi_port=Z; }  
  }  
}
```

Scan Cell Models - Retention Cell

Scan cells that use SAVE and RESTORE functions are directly addressed by the TestMAX ATPG retention cell testing flows, as shown in the following example.



The behavior of this scan cell is as follows:

- When the SAVE and RESTORE functions are deasserted, the cell behaves as a normal scan flip-flop.
- When the retention function is required, the SAVE sequence stores the flip-flop value in the retention latch.
- After power is restored to the flip-flop portion of the cell, the RESTORE sequence disables the flip-flop clock and any asynchronous controls, and loads the value from the retention latch into the flip-flop.

ATPG Simulation Primitives

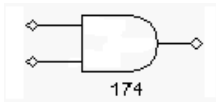
See Also

- [AND](#)
- [ADRBUS](#)

- BUF (Buffer)
- BUSK (Bus Keeper)
- BUS
- CMUX (Conservative Multiplexer)
- DATABUS
- DFF (Flipflop)
- DLAT (Latch)
- EQUIV
- INV (Inverter)
- MEMORY
- MOUT
- MUX (Multiplexer)
- NAND
- NOR
- OR
- PI (Primary Input)
- PIO (Primary Inout)
- PO (Primary Output)
- RPORT
- SEL01
- SEL1
- SW (Switch)
- TIE0
- TIE1
- TIEX
- TIEZ
- TSD (Tristate Driver)

- [WIRE](#)
- [XNOR](#)
- [XOR](#)
- [ADRBUS Primitive \(Address Bus\)](#)

AND Primitive



Description

The AND primitive has two or more inputs on the left side and one output on the right side. The inputs are identified starting with the topmost as input 0, then 1, and so forth. The numeric [gate ID](#) appears below the symbol.

Simulation Behavior

```
I0 I1 : out
--- --- : ---
0 ? : 0
? 0 : 0
1 1 : 1
X 1 : X
1 X : X
X X : X
```

The AND primitive provides as the output a 1 if all inputs are 1, or a 0 if any input is zero. Any other input conditions result in an output of X. A Z on an input is treated as an X. The inputs might be inverted and this is displayed as an inversion bubble on the symbol graphics.

Verilog Netlist Usage

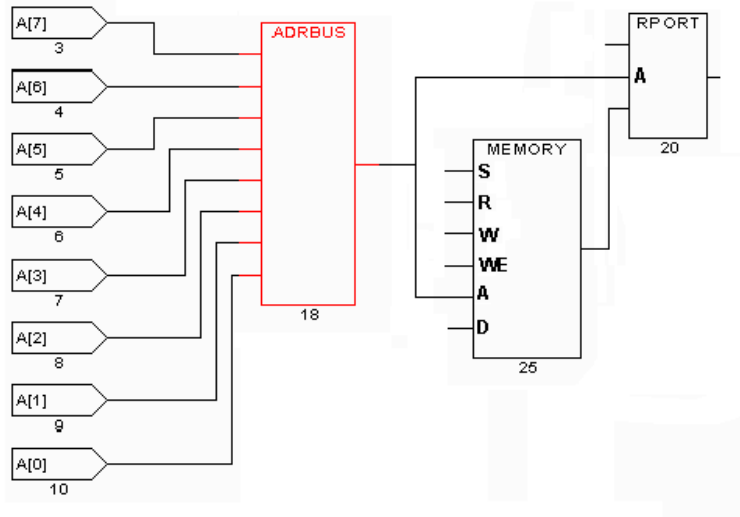
```
_AND u1 (in0, in1, [inN,]... out);
or
```

```
and ul (out, in0, in1 [,inN]... );
```

See Also

- [ATPG Modeling Primitive Summary](#)

ADRBUS Primitive (Address Bus)



Description

The ADRBUS primitive has one or more inputs on its left side, and a single bussed output on its right side. The inputs are identified starting with the topmost input, which is input I0, then I1, then I2, and so forth. The address lines are connected with the MSB bit at the top and the LSB bit at the bottom. Any input might be inverted, in which case it is drawn with an inversion bubble on its input. The output is a bussed net with the same number of bits as on the left side of the ADRBUS gate. The output of the ADRBUS gate can connect only to a MEMORY primitive or an RPORT primitive. Multiple connections are possible.

Simulation Behavior

There is no simulation behavior of the ADRBUS gate. It provides a grouping of individual address nets into a bussed net for connection to the MEMORY primitive or an RPORT gate.

Any Z on an input is treated as an X.

Any input which is an X causes the output to be X across all bits.

The number of address inputs must match the defined address range of the associated MEMORY primitive.

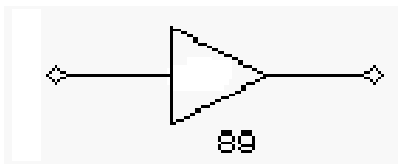
Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG during the flattening process in response to the presence of an ATPG RAM or ROM model.

See Also

- [TestMAX ATPG Memory Modeling](#)

BUF Primitive (Buffer)



Description

The BUF primitive has one input on the left side and one output on the right side. The input cannot be inverted. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
in : out
--- : ---
0 : 0
1 : 1
X : X
Z : X
```

The BUF primitive provides as the output the same value as the input for inputs of 0 or 1. For all other inputs the output is X.

Verilog Netlist Usage

```
_BUF u1 (in, out);

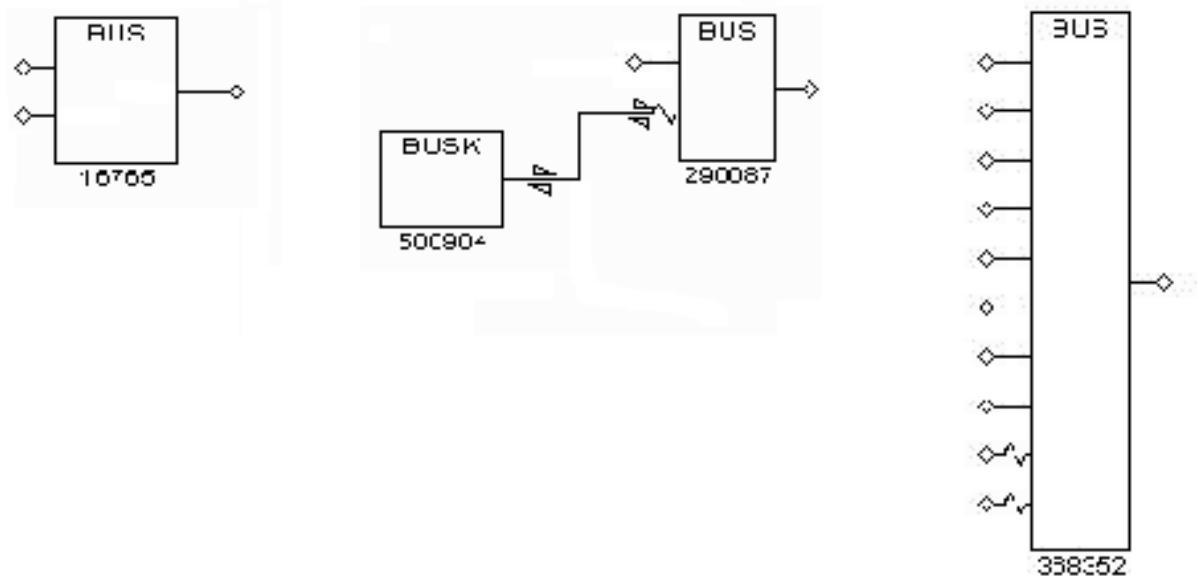
or

buf u1 (out, in);
```

See Also

- [ATPG Modeling Summary](#)

BUS Primitive



Description

The BUS primitive has two or more inputs or bidirectional connections on its left side, and a single output on its right side. However, the output is not always connected when one of the connections on the left side is a bidirectional connection. The inputs are identified starting with the topmost as input 0, then 1, and so forth. A [gate ID](#) appears below the symbol.

Input connections can be inverted and shown with a bubble. Bidirectional connections and connections to BUSK ([Bus Keeper](#)) primitives cannot be inverted.

Input connections can be weak and are shown with a resistor symbol on the input wire.

Simulation Behavior

The BUS primitive represents a net resolution function for multiple driver nets when the drivers are tristateable. All inputs and bidirectional connections are strong or weak.

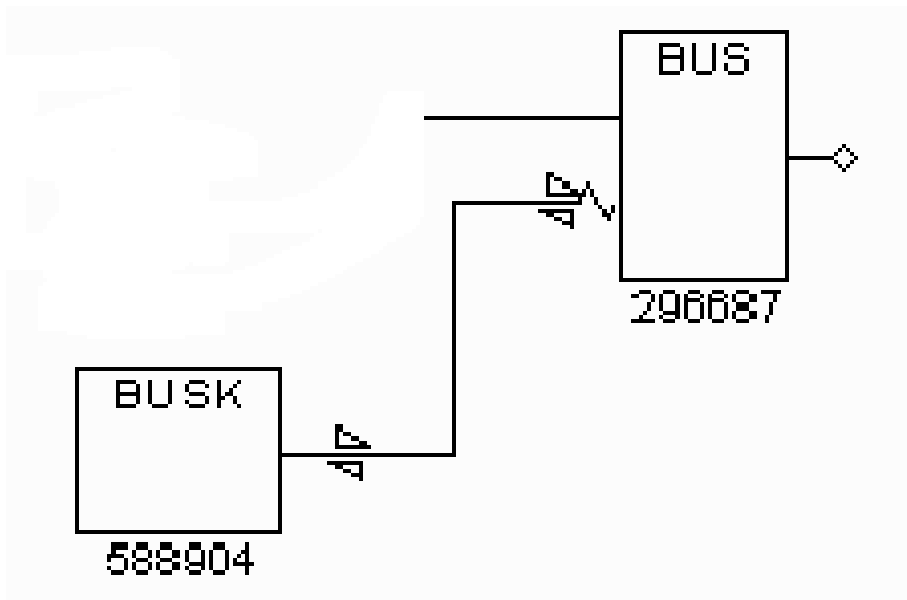
1. If all inputs are Z, then the output is Z.
2. If at least one input is a strong 0 and no other input is a strong 1 or X, then the output is 0.

3. If at least one input is a strong 1 and no other input is a strong 0 or X, then the output is 1.
4. If at least one input is a strong 1 or X and at least one other input is a strong 0 or X, then the output is X. This condition is called a bus contention.
5. If all strong inputs are Z and weak inputs exist, then rules #1 through #4 are checked using weak inputs in place of strong inputs.
6. If all strong inputs are Z and all weak inputs are Z, then rules #1 through #4 are checked using weak inputs sourced from **BUS keepers**.

Verilog Netlist Usage

The BUS primitive cannot be directly referenced in a netlist. It is automatically inserted by TestMAX ATPG during flattening when a net resolution function is required.

BUSK Primitive (Bus Keeper)



Description

The BUSK (**bus keeper**) primitive has one weak bidirectional connection that connects to a BUS primitive. The numeric **gate ID** appears under the symbol.

Simulation Behavior

If the value of the associated BUS primitive's output is Z, then the BUSK outputs its prior value, otherwise the BUSK's output matches that of the BUS and it stores this value for its next evaluation.

Verilog Netlist Usage

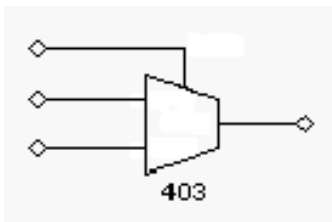
A bus keeper function is automatically determined from most Verilog module definitions. However, if you need to explicitly define a bus keeper, you can do so in the same way you declare a wire or tri net:

```
_BUSK0 net1; # keep only 0  
_BUSK1 net2; # keep only 1  
_BUSK01 net3; # keep 0 and 1
```

See Also

- [BUS Keeper Examples](#)
- [ATPG Modeling Summary](#)

CMUX Primitive (Conservative Multiplexer)



Description

The CMUX primitive is a conservative version of the MUX primitive. It has one select input on the top side. There are two inputs on the left side, the first input is I0 and the second is I1. Any input can be inverted, in which case it has a bubble shown on the symbol. The numeric [gate ID](#) appears under the symbol.

The CMUX primitive differs from the MUX primitive only in its behavior when $S=X$ and $D0=D1$. Under those conditions, the output of the CMUX is X, while the output of the MUX is the same value as D0 and D1. In all other conditions, the behavior of the CMUX is intended to be the same as the behavior of the MUX.

Simulation Behavior

```
S I0 I1 : out
--- --- --- : ---

0 0 ? : 0
0 1 ? : 1
0 X ? : X

1 ? 0 : 0
1 ? 1 : 1
1 ? X : X

X 0 0 : X
X 0 1 : X
X 1 0 : X
X 1 1 : X
```

The CMUX primitive provides as the output the I0 input when S=0, or the I1 input when S=1.

A z on an input is treated as an x.

The `-conservative_mux` option of the `set_netlist` command enables you to specify how TestMAX ATPG should handle a conservative MUX. The default is for TestMAX ATPG to extract conservative MUXes from combinational UDPs, but not from sequential UDPs. You can also specify TestMAX ATPG to handle either all or none of the conservative MUXes.

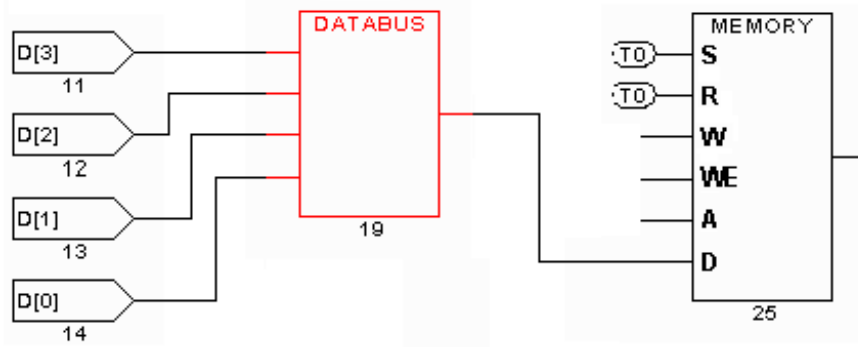
Verilog Netlist Usage

```
_CMUX u1 (S, I0, I1, out);
```

See Also

- [ATPG Modeling Summary](#)

DATABUS Primitive (Data Bus)



Description

The DATABUS primitive has one or more inputs on its left side, and a single bussed output on its right side. The inputs are identified starting with the topmost input, which is input I0, then I1, then I2, and so forth. The data lines are connected with the MSB bit at the top and the LSB bit at the bottom. Any input can be inverted, in which case it is drawn with an inversion bubble on its input. The output is a bussed net with the same number of bits as on the left side of the DATABUS gate. The output of the DATABUS gate can connect only to a MEMORY primitive data input. Multiple connections are possible.

Simulation Behavior

There is no simulation behavior of the DATABUS gate. It provides a grouping of individual data nets into a bussed net for connection to the MEMORY primitive.

Any Z on an input is treated as an X.

The number of data inputs must match the defined data width of the associated MEMORY primitive.

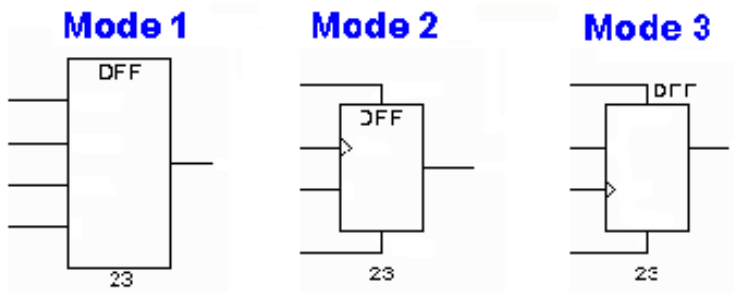
Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG during the flattening process in response to the presence of an ATPG RAM or ROM model.

See Also

- [Memory Modeling](#)

DFF Primitive



Description

The DFF primitive has one output, one asynchronous set input, one asynchronous clear input, and one or more pairs of clock and data inputs. The numeric Gate ID appears under the symbol.

For Mode 1 graphics, the inputs are identified on the left starting at the top as `set`, `reset`, and then clock/data pairs in the order `clock`, then `data`. The example shows only one clock/data pair but additional ones would cascade down the left side.

For Mode 2 graphics, the input on the top is the `set`, the input on the bottom is the `reset`, and the inputs on the left side are clock/data pairs in the order `clock`, then `data`.

For Mode 3 graphics, the input on the top is the `set`, the input on the bottom is the `reset`, and the inputs on the left side are clock/data pairs in the order `data`, then `clock`.

All inputs are active high. Clock inputs are active on a 0 to 1 transition. However, all inputs can be inverted, in which case they are shown with an inversion bubble in the graphical schematic viewer.

Approximate Simulation Behavior

```
#
# for the single clock/data case
#
set rst clk data : q- : out
--- --- --- : --- : ---
? ? b * : ? : - ; # data event, hold unless clk=x
? ? (?0) ? : ? : - ; # clk to 0, hold

0 ? (1x) 0 : 0 : 0 ; # clk to x, hold
? 0 (1x) 1 : 1 : 1 ; #

(?0) 0 b ? : ? : - ; # set to off, hold
0 (?0) b ? : ? : - ; # rst to off, hold

0 1 ? ? : ? : 0 ; # async reset
```

```

1 0 ? ? : ? : 1 ; # async set

0 ? (01) 0 : ? : 0 ; # clock data
? 0 (01) 1 : ? : 1 ; # clock data

0 ? (0x) 0 : 0 : 0 ; # possible clk, D=Q=0
? 0 (0x) 1 : 1 : 1 ; # possible clk, D=Q=1

0 ? (x1) 0 : 0 : 0 ; # possible clk, D=Q=0
? 0 (x1) 1 : 1 : 1 ; # possible clk, D=Q=1

0 ? x 0 : 0 : 0 ; # metastable clk, D=Q=0
? 0 x 1 : 1 : 1 ; # metastable clk, D=Q=1

0 (?x) b ? : 0 : 0 ; # rst=X with Q=0
(?x) 0 b ? : 1 : 1 ; # set=X with Q=1

0 (?x) x 0 : 0 : 0 ; # rst=X with Q=0
(?x) 0 x 1 : 1 : 1 ; # set=X with Q=1

# 0 0 x * : ? : x ; # data change with clk=metastable

```

Note 1: The `-xclock_gives_xout` option of the `set_simulation` command controls behavior when a clock changes to X. By default a 1->X transition causes no change while a 0->X transition is processed as a potential clock. Specifying this option causes any transition to X or steady state value of X to cause the output to be set to X.

Set and Reset inputs are active high level sensitive. Clock inputs are activated by a 0 to 1 transition. For the clock pin, the label "active" should be considered the same as have a rising edge event.

A z on any input is treated the same as an x.

Textual Simulation Behavior

1. >If set and reset inputs are off and all clocks are stable non-X values, then the output retains its previous state.
2. If set is on and reset is off then the output is set to 1. The asynchronous set is dominant over a clock edge event.
3. If reset is on and set is off then the output is set to 0. The asynchronous reset is dominant over a clock edge event.
4. If a single clock line is active and the set, reset, and other clocks are off, then the output becomes the value of the associated data input at the time of the capturing clock edge. If multiple clocks are active, then the output is reviewed for whether the captured value would be the same from all capturing inputs. If the output would be the same it is set to that value, otherwise the output is set to X.
5. If set and reset are both on, the output is set to X.

6. If any set or reset input is at X and all clocks are non-X, then the output is reviewed for whether it would change if that X were really a 0 and 1. If the output would be unchanged, then it is kept; otherwise, it is set to X.
7. If a possible clock transition has occurred (0->X, X->1), then the output is reviewed for whether it would change if the clock did occur. If the output would be unchanged, then it is kept; otherwise, it is set to X.
8. If any clock is at X and its corresponding data input is X, then the output is set to X.
9. If any clock is at X (with set/reset off) and the data input changes during this time (0->1, 1->0), then the output is set to X.
10. If any clock is at X and the `-xclock_gives_xout` option of the `set_simulation` command has been selected, then the output is set to X.

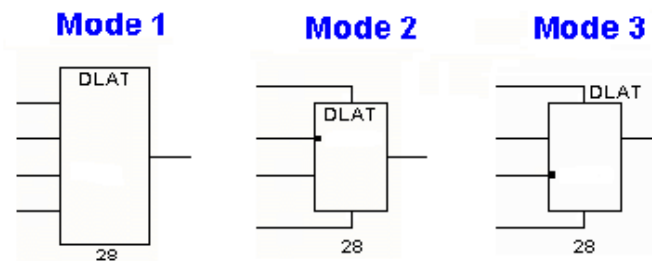
Verilog Netlist Usage

```
_DFF ul (set, rst, clk, data, [clkN, dataN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

DLAT Primitive



Description

The DLAT primitive has one output, one asynchronous set input, one asynchronous clear input, and one or more pairs of clock and data inputs. The numeric [gate ID](#) appears under the symbol.

Note: The `-xclock_gives_xout` option of the `set_simulation` command controls behavior when a clock changes to X. By default, a clock value of X causes no change while D=Q. Specifying this option causes any transition to X, or steady state value of X, to immediately set the output to X.

For Mode 1 graphics, the inputs are identified on the left starting at the top as `set`, `reset`, and then clock/data pairs in the order `clock`, then `data`. The example shows only one clock/data pair but additional ones would cascade down the left side.

For Mode 2 graphics, the input on the top is the `set`, the input on the bottom is the `reset`, and the inputs on the left side are clock/data pairs in the order `clock`, then `data`.

For Mode 3 graphics, the input on the top is the `set`, the input on the bottom is the `reset`, and the inputs on the left side are clock/data pairs in the order `data`, then `clock`.

All inputs are active high. However, all inputs can be inverted, in which case they are shown with an inversion bubble.

Simulation Behavior

```

set rst clk data : q- : out
--- --- --- ---- : --- : ---
0 0 0 ? : ? : - ; # clocks off

0 ? 1 0 : ? : 0 ; # latch open, D=0
? 0 1 1 : ? : 1 ; # latch open, D=1

1 0 0 ? : ? : 1 ; # async set
0 1 0 ? : ? : 0 ; # async reset

1 0 1 ? : ? : x ; # clk while async set
0 1 1 ? : ? : x ; # clk while async reset

x 0 0 ? : 1 : 1 ; # set=X, but Q=1
0 x 0 ? : 0 : 0 ; # reset=X, but Q=0

? 0 x 1 : 1 : 1 ; # possible clk of D=1, but Q=1
0 ? x 0 : 0 : 0 ; # possible clk of D=0, but Q=0

```

Set, Reset, and Clock inputs are active high level sensitive. if:

- all clock/set/reset inputs are inactive, the output retains its previous state.
- the set line is active and the reset and clocks are inactive, then out=1.
- the reset line is active and the set and clocks are inactive, then out=0.
- a single clock line is active and the other set/reset/clocks are inactive, then the output becomes the value of the associated data input.
- any clock/set/reset input is at X then the output is reviewed for whether it would change if that X were really a 0 and 1. If the output would be unchanged then it is kept, otherwise it is set to X.
- set and reset are both active, the output is X.

If multiple clock/set or clock/reset or clock/clock inputs are active, then the output is reviewed for whether the captured value would be the same from all active inputs. If the output would be the same, it is kept; otherwise, the output is set to X.

A Z on any input is treated as an X.

Verilog Netlist Usage

```
_DLAT u1 (set, rst, clk, data, [clkN, dataN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

EQUIV Primitive (Equivalence)

There is no graphic for this primitive. It is a virtual gate added in response to the creation of an ATPG primitive and does not show up in the schematic view.

Description

The EQUIV gate has two or more inputs and one output. The inputs can be inverted.

Simulation Behavior

```
I0 I1 I2 : out
--- --- --- : ---
0 0 0 : 1
1 1 1 : 1
0 1 ? : 0
? 0 1 : 0
1 ? 0 : 0
0 0 X : X
X 0 0 : X
0 X 0 : X
1 1 X : X
```

```
X 1 1 : X
1 X 1 : X
```

A Z on any input is treated as an X.

If all inputs are 0, the output is 1.

If all inputs are 1, the output is 1.

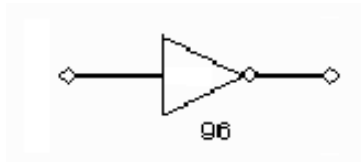
If at least one input is 0 and one input is 1, then the output is 0; otherwise, the output is X.

Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG as a result of the `add_atpg_primitives` command.

INV Primitive (Inverter)

The following figure shows the INV primitive inverter:



Description

The INV primitive has one input on the left side and one output on the right side. The input cannot be inverted. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
in : out
--- : ---
0  : 1
1  : 0
X  : X
Z  : X
```

The INV primitive provides on the output the inverted value of the input for inputs of 0 or 1. For all other inputs the output is X.

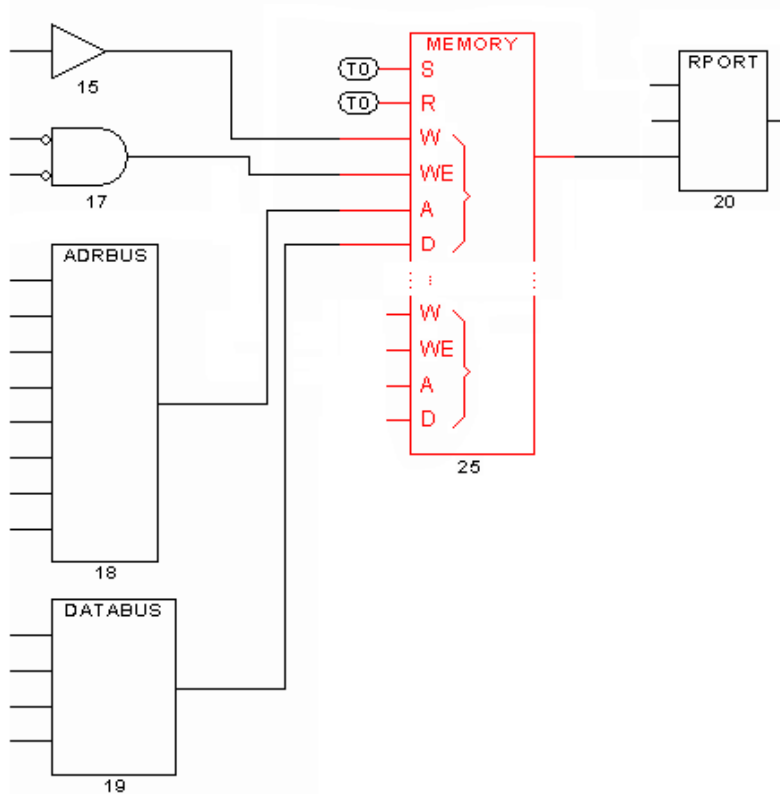
Verilog Netlist Usage

```
_INV u1 (in, out);  
  
or  
  
not u1 (out, in);
```

See Also

- [ATPG Modeling Summary](#)

MEMORY Primitive (RAM/ROM Memory)



Description

The MEMORY gate has inputs on its left side and a single output on its right side. Starting from the topmost input on the left side, the first input is the active high `set` control. The second input is the active high `reset` control. Next, there are zero or more groups of four inputs designated "write ports". The order of the four inputs for a write port are, from top to

bottom: `write_control`, `write_enable`, `address_bus`, and `data_bus`. Beneath the symbol is the [gate ID](#).

The `set/reset` inputs, as well as the `write_control` and `write_enable` inputs of any write port, can be inverted, in which case they are drawn with inversion bubbles.

The `address_bus` input can only connect to the output of an ADRBUS gate.

The `data_bus` input can only connect to the output of a DATABUS gate.

The MEMORY gate's output can only connect to one or more RPORT gates.

Simulation Behavior

A Z state on any input is treated as an X.

The `set` and `reset` control inputs are level sensitive, active high controls. The `write_control` input is either level sensitive (active high) or edge triggered (active on 0 to 1 edge). The write operation requires that the `write_control` be asserted with the `write_enable` line at 1.

When the `set` control is asserted and no other `reset` or `write_control` operations are active, then the memory contents are all set to 1. If the `set` control is X, then the entire memory contents are set to X.

When the `reset` control is asserted and no other `set` or `write_control` operations are active, then the memory contents are all set to 0. If the `reset` control is X, then the entire memory contents are set to X.

When both the `set` and `reset` controls are simultaneously asserted, the memory contents are all set to X.

If all `write_operations/set/reset` are inactive, the memory contents remain unchanged.

If a single `write_operation` is active and the other `set/reset/write_operations` are inactive then a write operation occurs to the corresponding memory word determined by the `address_bus` using the data values on `data_bus`. If the associated `address_bus` has a value of X, then the entire memory contents are set to X.

If any `write_control` is X while its corresponding `write_enable` is high, then the data value of the currently addressed memory location is set to X.

If multiple `write_operation/set/reset` lines are active then TestMAX ATPG determines if all possible data writes to the associated address would produce the same value. If this is true, the value is written; otherwise, the memory location is set to X's.

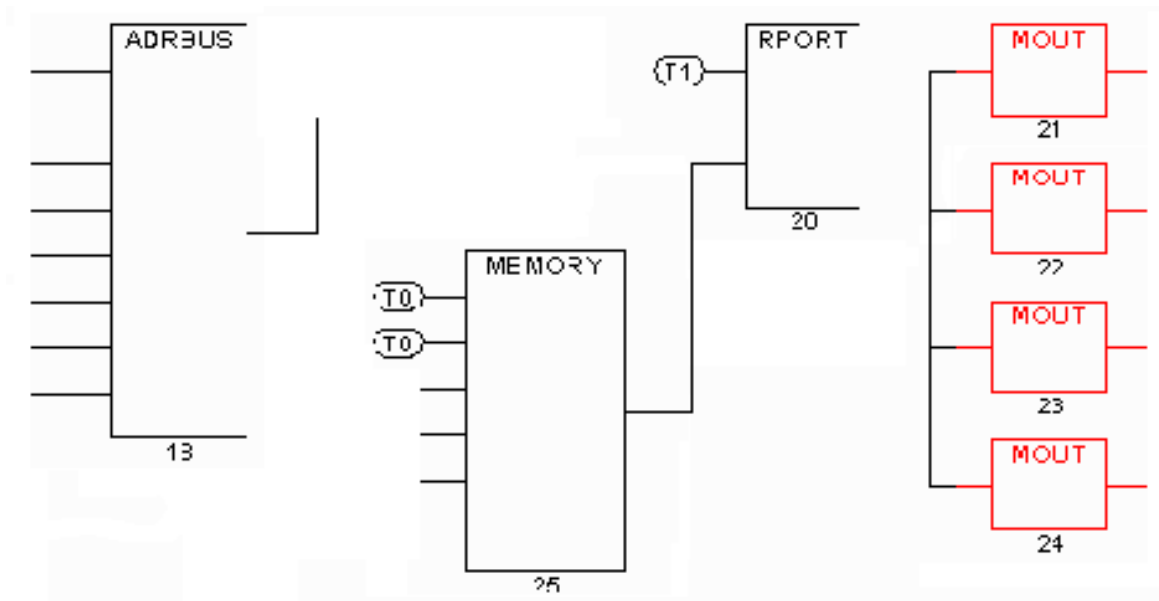
Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG during the flattening process in response to the presence of an ATPG RAM or ROM model.

See Also

- [Memory Modeling](#)

MOUT Primitive (Macro Output)



Description

An MOUT gate has one input on its left side and one output on its right side. The input is a bussed net and the output is a single bit of that net. The gate acts as a bus splitter. The [gate ID](#) appears beneath the symbol.

An MOUT gate can connect only to the data_bus output of an RPORT gate.

Simulation Behavior

The MOUT gate has no simulation behavior. It functions to hold the value for each data bit from the associated RPORT gate. Selecting the MOUT gate in the schematic viewer and opening the Gate Info window will report which bit of the bus is being split off.

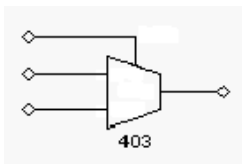
Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG during the flattening process in response to the presence of an ATPG RAM or ROM model.

See Also

- [Memory Modeling](#)

MUX Primitive (Multiplexer)



Description

The MUX primitive has one select input on the top side. There are two inputs on the left side, the first input is I0 and the second is I1. Any input can be inverted, in which case it has a bubble shown on the symbol. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
S I0 I1 : out
--- --- --- : ---
0 0 ? : 0
0 1 ? : 1
0 X ? : X

1 ? 0 : 0
1 ? 1 : 1
1 ? X : X

X 0 0 : 0
X 0 1 : X
X 1 0 : X
```



```
X 1 1 : 1
```

The MUX primitive provides as the output the I0 input when S=0, or the I1 input when S=1. If S=X and I0=I1, then the out=I0=I1; otherwise the output is X.

A Z on an input is treated as an X.

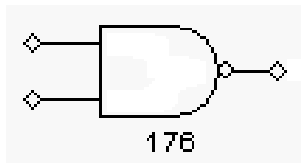
Verilog Netlist Usage

```
_MUX u1 (S, I0, I1, out);
```

See Also

- [ATPG Modeling Summary](#)

NAND Primitive



Description

The NAND primitive has two or more inputs on the left side and one output on the right side. The inputs are identified starting with the topmost as input 0, then 1, and so forth. The numeric [gate ID](#) appears below the symbol.

Simulation Behavior

```
I0 I1 : out
--- --- : ---
0 ? : 1
? 0 : 1
1 1 : 0
X 1 : X
1 X : X
X X : X
```

The NAND primitive provides as the output a 0 if all inputs are 1, or a 1 if any input is zero. Any other input conditions result in an output of X.

A Z on an input is treated as an X.

The inputs can be inverted and this is displayed as an inversion bubble on the symbol graphics.

Verilog Netlist Usage

```
_NAND u1 (in1, in2, [inN,]... out);
```

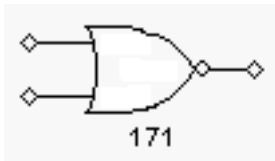
or

```
nand u1 (out, in1, in2 [,inN]... );
```

See Also

- [ATPG Modeling Summary](#)

NOR Primitive



Description

The NOR primitive has two or more inputs on it's left side and one output on its right side. The inputs on the left are identified going from top to bottom as I0, I1, I2, and so forth. Any input can be inverted in which case it is drawn with an input bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
I0 I1 : out
```

```
--- --- : ---
```

```
0 0 : 1
```

```
0 1 : 0
```

```
1 0 : 0
```

```
1 1 : 0
```

```
X 0 : X
X 1 : 0
0 X : X
1 X : 0
X X : X
```

The NOR primitive provides as its output the inverted OR function of its inputs. When any input is 1, the output is 0. When all inputs are 0 the output is 1. For all other combinations the output is X.

Any Z input is treated as an X.

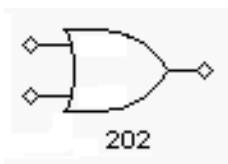
Verilog Netlist Usage

```
_NOR u1 (in0, in1, [,inN]... out);
    or
nor u1 (out, in0, in1, [,inN]... );
```

See Also

- [ATPG Modeling Summary](#)

OR Primitive



Description

The OR primitive has two or more inputs on its left side and one output on its right side. The inputs on the left are identified going from top to bottom as I0, I1, I2, and so forth. Any input can be inverted, in which case it is drawn with an input bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
I0 I1 : out
--- --- : ---

0 0 : 0
0 1 : 1
1 0 : 1
1 1 : 1

X 0 : X
X 1 : 1
0 X : X
1 X : 1
X X : X
```

The OR primitive provides as its output the OR function of its inputs. When any input is 1, the output is 1. When all inputs are 0, the output is 0. For all other combinations the output is X.

Any Z input is treated as an X.

Verilog Netlist Usage

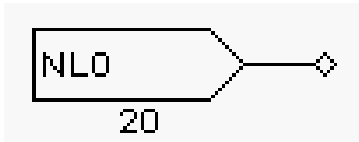
```
_OR u1 (in0, in1, [,inN]... out);

or
    or u1 (out, in0, in1, [,inN]... );
```

See Also

- [ATPG Modeling Summary](#)

PI Primitive (Primary Input)



Description

The PI primitive has one output. It represents a primary input for the top level of the design. The name of the top level port appears within the PI symbol. The numeric [gate ID](#) appears under the symbol.

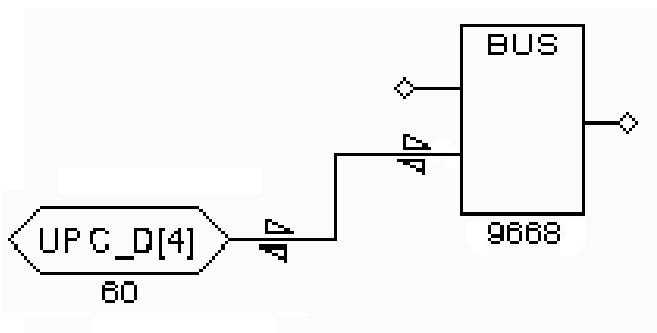
Simulation Behavior

There is no simulation behavior associated with a PI gate. It is a holding place for values of 0/1/X/Z applied as inputs.

Verilog Netlist Usage

Direct access to a PI primitive is not supported in a netlist. TestMAX ATPG inserts PI primitives during flattening to correspond to top level module inputs declared.

PIO Primitive (Primary Input/Output)



Description

The PIO primitive has one bidirectional connection. It represents a primary input and output for the top level of the design. The name of the top level port appears within the PIO symbol. The numeric [gate ID](#) appears under the symbol.

The PIO primitive always attaches to a corresponding BUS primitive.

Simulation Behavior

There is no simulation behavior associated with a PIO gate. It is a holding place for values of 0/1/X/Z applied as inputs and a place to observe the output value from its companion BUS gate.

Verilog Netlist Usage

Direct access to a PIO primitive is not supported in a netlist. TestMAX ATPG inserts PIO primitives during flattening to correspond to top level module inouts declared.

PO Primitive (Primary Output)



Description

The PO primitive has one input. It represents a primary output for the top level of the design. The name of the top level port appears within the PO symbol. The numeric [gate ID](#) appears under the symbol.

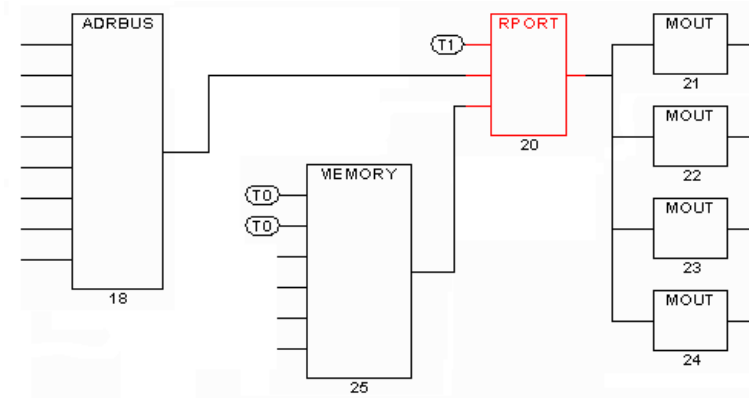
Simulation Behavior

There is no simulation behavior associated with a PO gate. It is an observation point for values from the attached output net.

Verilog Netlist Usage

Direct access to a PO primitive is not supported in a netlist. TestMAX ATPG inserts PO primitives during flattening to correspond to top level module outputs declared.

RPORT Primitive (Read Port)



Description

This primitive with its connecting ADRBUS, MEMORY, and MOUT primitives represent a read port which is used to model RAMs and ROMs.

The RPORT primitive has three inputs on the left side and one output on the right side. Starting with the topmost input, the first input is the active high read_control input, the second input is the address_bus input, and the third input is the data_bus input. A [gate ID](#) appears below the symbol.

The data_bus input can connect only to the output of a MEMORY gate. This is a bussed input.

The address_bus input can connect only to the output of an ADRBUS gate of appropriate width for the MEMORY to which the RPORT is associated. This is a bussed output.

The RPORT's output is a bussed net having the same data width as its data_bus input. It can connect only to MOUT gates.

Simulation Behavior

If the read_control input is 1, the MOUT primitives attached to the RPORT's output are set to the data values of the memory location specified by the address_bus. If the address_bus is X, the MOUT primitives are set to X.

If the read_control input is 0, the MOUT primitives are set to the read_off value of the associated MEMORY primitive (set by attributes when the model is defined).

If the read_control input is X, the MOUT primitives are set to X.

Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG during the flattening process in response to the presence of an ATPG RAM or ROM model.

See Also

- [Memory Modeling](#)

SEL01 Primitive

There is no graphic for this primitive. It is a virtual gate added in response to the creation of an ATPG primitive and does not show up in the schematic view.

Description

The SEL01 gate has two or more inputs and one output. The inputs can be inverted.

Simulation Behavior

```
I0 I1 I2 : out
--- --- --- : ---
0 0 0 : 1
0 0 1 : 1
0 1 0 : 1
1 0 0 : 1

0 1 1 : 0
1 0 1 : 0
1 1 0 : 0
1 1 1 : 0

X ? ? : X
? X ? : X
? ? X : X
```


If a single input is at 1 with all other inputs 0, then the output is 1.

If all inputs are 0, then the output is also 1.

If more than one input is 1, then the output is 0.

If any input is X, the output is X. A Z on any input is treated as an X.

Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG as a result of the `add_atpg_primitives` command.

SEL1 Primitive

There is no graphic for this primitive. It is a virtual gate added in response to the creation of an ATPG primitive and does not show up in the schematic view.

Description

The SEL1 gate has two or more inputs and one output. The inputs can be inverted.

Simulation Behavior

```
I0 I1 I2 : out
--- --- --- : ---
0 0 1 : 1
0 1 0 : 1
1 0 0 : 1

0 0 0 : 0
0 1 1 : 0
1 0 1 : 0
1 1 0 : 0
1 1 1 : 0

X ? ? : X
? X ? : X
? ? X : X
```

A Z on any input is treated as an X.

If a single input is at 1 and all other inputs are at 0, the output is 1.

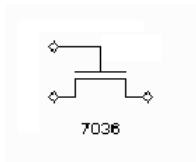
If more than one input is at 1 or all inputs are 0, the output is 0.

If any input is at X, the output is an X.

Verilog Netlist Usage

This primitive cannot be directly referenced in a netlist. It is inserted by TestMAX ATPG as a result of the `add_atpg_primitives` command.

SW Primitive (Switch)



Description

The SW primitive has one control input on its top side, one data input on its left side and one output on its right side. The control input can be inverted, in which case it is drawn with an inversion bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

C	in	:	out
---	---	:	---
0	?	:	Z
1	0	:	0
1	1	:	1
1	X	:	X
1	Z	:	Z
X	?	:	X

The SW primitive provides as the output its data input value when the control input is 1, a Z value when its control input is 0, and X for all other combinations. It can pass a Z value.

Verilog Netlist Usage

```
_SW ul (control, in, out);
```

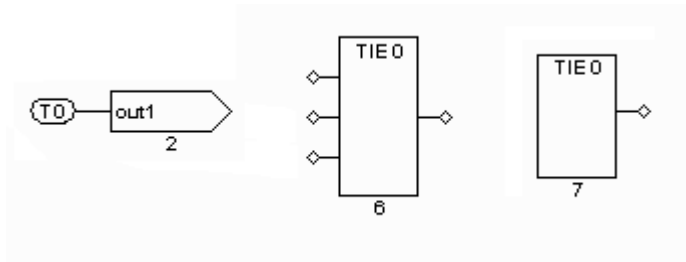
or

```
nmos ul (out, in, control);
```

See Also

- [ATPG Modeling Summary](#)

TIE0 Primitive



Description

The TIE0 primitive has one output on the right side and zero or more optional inputs on the left side. The symbol can appear as both a small oval with the letters T0 inside and as a rectangular block. For a rectangular block, the numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
I0 : out
```

```
--- : ---
```

```
? : 0
```

The TIE0 primitive provides a constant zero output.

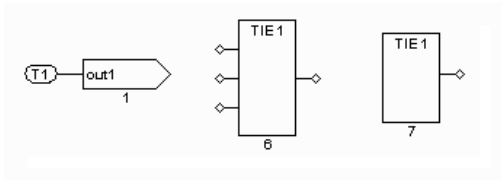
Verilog Netlist Usage

```
_TIE0 u1 ( [inN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

TIE1 Primitive



Description

The TIE1 primitive has one output on the right side and zero or more optional inputs on the left side. The symbol can appear as both a small oval with the letters T1 inside and as a rectangular block. For a rectangular block, the numeric [gate ID](#) appears under the symbol.

Simulation Behavior

I0 : out

--- : ---

? : 1

The TIE1 primitive provides a constant high output.

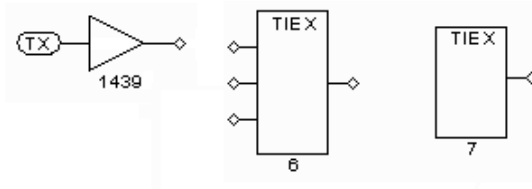
Verilog Netlist Usage

```
_TIE1 u1 ( [inN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

TIEX Primitive



Description

The TIEX primitive has one output on the right side and zero or more optional inputs on the left side. The symbol can appear as both a small oval with the letters `TX` inside and as a rectangular block. For a rectangular block, the numeric [gate ID](#) appears under the symbol.

Note that while the TIEX gate provides a constant X value in simulation behavior, it can be treated differently for purposes of analysis. The TIEX gate is considered to be capable of either a 0 or 1 value and it might show up as producing a 0 or 1 in certain kinds of analyses, including analyses of certain types of design rule violations.

Simulation Behavior

I0 : out

--- : ---

? : X

The TIEX primitive provides a constant X output.

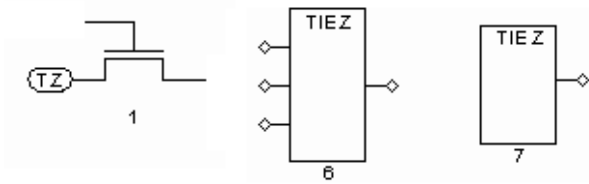
Verilog Netlist Usage

```
_TIEX u1 ( [inN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

TIEZ Primitive



Description

The TIEZ primitive has one output on the right side and zero or more optional inputs on the left side. The symbol can appear as both a small oval with the letters `TZ` inside and as a rectangular block. For a rectangular block, the numeric [gate ID](#) appears under the symbol.

Simulation Behavior

`I0` : out

--- : ---

? : Z

The TIEZ primitive provides a constant Z output.

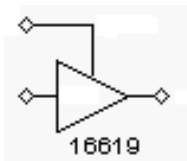
Verilog Netlist Usage

```
_TIEZ u1 ( [inN,]... out);
```

See Also

- [ATPG Modeling Summary](#)

TSD Primitive (tristate Device)



Description

The TSD primitive has one control input on its top side, one data input on its left side and one output on its right side. The control or data inputs can be inverted, in which case they are drawn with an inversion bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
C in : out
--- --- : ---
0 ? : Z
1 0 : 0
1 1 : 1
1 X : X
1 Z : X
X ? : X
```

The TSD primitive provides as the output its data input value when the control input is 1, a Z value when its control input is 0, and X for all other combinations.

A Z input is treated as an X.

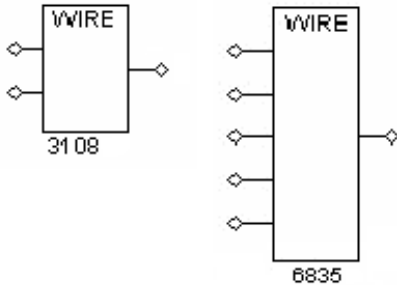
Verilog Netlist Usage

```
_TSD u1 (control, in, out);
or
bufif1 u1 (out, in, control);
```

See Also

- [ATPG Modeling Summary](#)

WIRE Primitive



Description

The WIRE primitive has two or more inputs on its left side, and a single output on its right side. The inputs are identified starting with the topmost as input 0, then 1, and so forth. A [gate ID](#) appears below the symbol.

Input connections cannot be inverted.

Simulation Behavior

I0	I1	:	out
---	---	:	---
0	0	:	0
1	1	:	1
0	1	:	X
1	0	:	X
X	?	:	X
?	X	:	X

The WIRE primitive represents a net resolution function for multiple driver nets when the drivers are non-tristatable. If all inputs are the same, then the output is that value. Otherwise, the output is X.

An input of Z is treated as an X.

Verilog Netlist Usage

```
_WIRE u1 (in1, in2, [inN,... out);
```

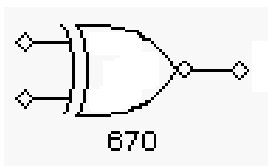

or

```
buf u1 (in1, out);  
buf u2 (in2, out);
```

See Also

- [ATPG Modeling Summary](#)

XNOR Primitive (Exclusive NOR)



Description

The XNOR primitive has two or more inputs on the left side and one output on the right side. Any input can be inverted, in which case it is shown with an inversion bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

I0	I1	: out
---	---	: ---
0	0	: 1
0	1	: 0
1	0	: 0
1	1	: 1
X	?	: X
?	X	: X

The XNOR primitive provides as the output the inverted XOR function of its inputs. When an odd number of inputs are 1 and no inputs are X then the output is 0. When an even number of inputs are 1 and no inputs are X, then the output is 1. If any input is X or Z, the output is X.

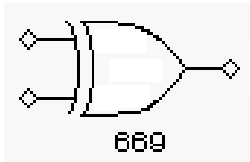
Verilog Netlist Usage

```
_XNOR u1 (in0, in1, [inN,]... out);  
or  
xnor u1 (out, in0, in1 [,inN,]... );
```

See Also

- [ATPG Modeling Summary](#)

XOR Primitive (Exclusive OR)



Description

The XOR primitive has two or more inputs on the left side and one output on the right side. Any input might be inverted, in which case it is shown with an inversion bubble. The numeric [gate ID](#) appears under the symbol.

Simulation Behavior

```
I0 I1 : out  
--- --- : ---  
  
0 0 : 0  
0 1 : 1  
1 0 : 1  
1 1 : 0  
  
X ? : X  
? X : X
```

The XOR primitive provides as the output the XOR function of its inputs. When an odd number of inputs are 1 and no inputs are X then the output is 1. When an even number of inputs are 1 and no inputs are X, then the output is 0. If any input is X or Z, the output is X.

Verilog Netlist Usage

```
_XOR ul (in0, in1, [inN,]... out);  
or  
xor ul (out, in0, in1 [,inN,]... );
```

See Also

- [ATPG Modeling Summary](#)

5

Command Interface

TestMAX ATPG provides an interactive command interface in the TestMAX ATPG GUI, and menus and buttons in the TestMAX ATPG GUI. The command interface includes a command language that you can use to execute command sequences in batch mode.

Online Help is available on commands, error messages, design flows, and many other TestMAX ATPG topics.

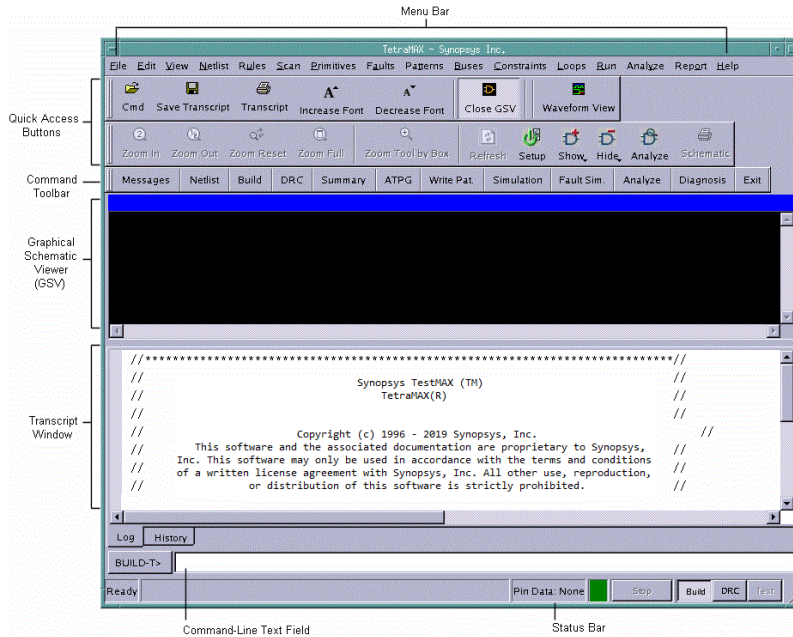
The following sections describe the components of the TestMAX ATPG command interface:

- [TestMAX ATPG GUI](#)
- [Command Entry](#)
- [Transcript Window](#)
- [Interacting with the TestMAX ATPG GUI](#)
- [Using Online Help](#)

TestMAX ATPG GUI

Figure 1 shows the main window of the TestMAX ATPG 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 text field, and the status bar.

Figure 11 TestMAX ATPG GUI Main Window



The GSV window is not displayed when you start TestMAX ATPG. It first appears when you execute a command that requests a schematic display. For more information on the GSV window, see [Using the Graphical Schematic Viewer](#).

The status bar, located at the very bottom of the main window, contains the STOP button and displays the state of TestMAX ATPG (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, see [Command Mode Indicator](#).

See Also

- [Using the Graphical Schematic Viewer](#)
- [Using the Hierarchy Browser](#)
- [Using the Simulation Waveform Viewer](#)

Command Entry

The main window provides three ways to interactively enter commands:

- Select from pull-down menus at the top of the main window.
- Use the command buttons in the command toolbar or the graphical schematic viewer (GSV) toolbar.
- Type commands in the command-line window.

The pull-down menus and command buttons let you specify the command options in dialog boxes. The command-line window uses a command-line-based entry method.

The following sections describe how to perform command entry:

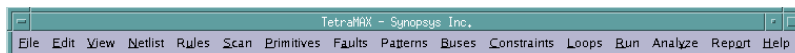
- [Menu Bar](#)
- [Command Toolbar and GSV Toolbar](#)
- [Command-Line Window](#)
- [Commands From a Command File](#)
- [Command Logging](#)

Menu Bar

The menu bar consists of a set of pull-down menus you use to select a required action. These menus provide the most comprehensive set of command selections.

The following figure shows the menu bar.

Figure 12 Menu Bar

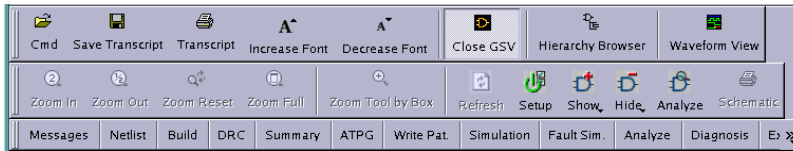


Command Toolbar and GSV Toolbar

The command toolbar is a collection of buttons you use to run TestMAX ATPG 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.

The following figure shows the command toolbar and GSV toolbar.

Figure 13 Command Toolbar and GSV Toolbar



By default, the command toolbar is displayed at the top of the main window, just below the menu bar. The GSV toolbar is displayed on the left side of the GSV window. Both toolbars are “dockable” — that is, you can move and “dock” them to any four sides of the GSV window, or use them 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 required location, and release the mouse button.

Command-Line Window

The command-line window is located between the transcript window and the status bar. The following components comprise the command-line window:

- [Command Mode Indicator](#)
- [Command-Line Entry Field](#)
- [Command Continuation](#)
- [Command History](#)
- [Stop Button](#)

The following figure shows the command-line window.

Figure 14 Command-Line Window



Command Mode Indicator

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

To change the command mode, use the Build, DRC, and Test buttons, located at 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 TestMAX ATPG 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 TestMAX ATPG 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.

You can use the arrow keys to queue the command line. If you are in Tcl mode, TestMAX ATPG includes automatic command completion feature. This feature also applies to directory and file name completion in both native mode and Tcl mode.

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.

The following example shows the `add_atpg_primitives` command usage in Tcl mode. This command defines an ATPG primitive connected to multiple pins. Note the use of curly brackets in Tcl mode for specifying lists. Each pin path name is presented on a separate line using the backslash character. All five lines are treated as a single command.

Command continuation across multiple lines (Tcl mode)

```
BUILD-T> add_atpg_primitives spec_atpg_prim1 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 }
```

The following example shows the same command example in native mode.

Command continuation across multiple lines (native mode)

```
BUILD> add_atpg_primitives spec_atpg_prim1 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 required 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 TestMAX ATPG is idle, the Stop button is dimmed. If TestMAX ATPG is busy processing a command (and the command mode indicator displays <Busy>), the button is active and is labeled "Stop." Click this button to halt processing of the current command. TestMAX ATPG might take several seconds to halt the activity.

You can interrupt a multicore ATPG process by clicking the Stop button. At this point, the master process sends an abort signal to the slave processes and waits for the slaves to finish any ongoing interval tasks. If this takes an extended period of time, you can click the Stop button twice; this action causes the master process to send a kill signal to the slaves, and the prompt will immediately return. Note that the clicking the Stop button twice will terminate all slave processes without saving any data gathered since the last communication with the master. For more information on multicore ATPG, see [Running Multicore ATPG](#).

Commands From a Command File

You can submit a list of commands as a file and have TestMAX ATPG execute those commands in batch mode. In Tcl mode, any line starting with # is treated as a comment and is ignored. In native mode, any line starting with a double-slash (//) 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, specfile .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](#).

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 spec_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 spec_logfile.log -replace  
> set_messages log spec_logfile.log -append
```

If you intend to use the log file as an executable command file, use the `-leading_comment` option of the `set_messages` command. In this case, TestMAX ATPG writes out the comment lines starting with either a pound sign(`#`) in Tcl mode, or a double slash in Native mode, so that those lines are ignored when you use the log file as a command file.

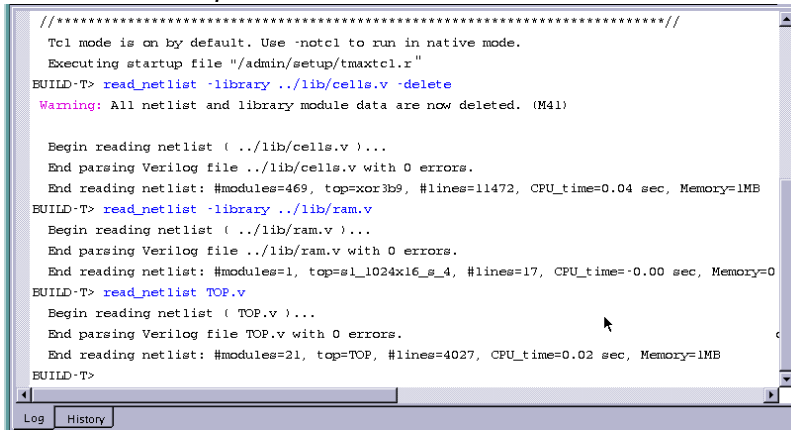
Transcript Window

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

The following sections describe the transcript window:

- [Setting the Keyboard Focus](#)
- [Using the Transcript Text](#)
- [Selecting Text in the Transcript](#)
- [Copying Text From the Transcript](#)
- [Finding Commands and Messages in the Transcript](#)
- [Saving or Printing the Transcript](#)
- [Clearing the Transcript Window](#)

Figure 15 Transcript Window



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 once.
- *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 required text, drag to the end of the required text, and release the mouse button. The selected text is highlighted.

Copying Text From the Transcript

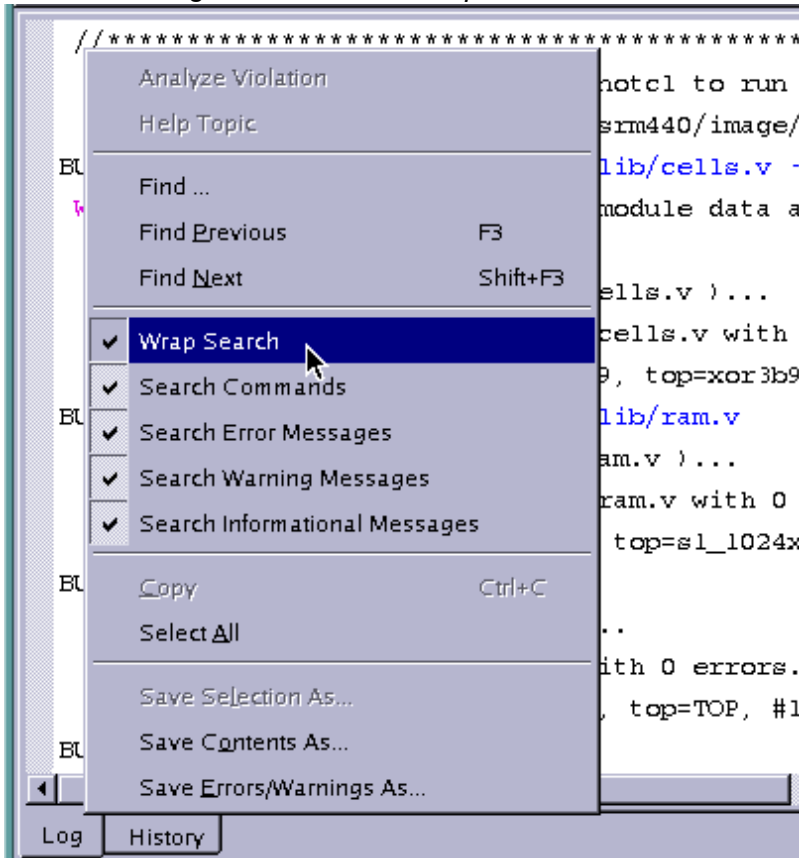
You can copy selected text from the transcript window to the Clipboard. Use the keyboard shortcut *Control-c*, or choose Copy from the pop-up menu that appears when you press the right mouse button. The Copy command is disabled if no text has been selected.

Finding Commands and Messages in the Transcript

To find commands and messages in the transcript window, right-click inside the Transcript window, and then click the box or boxes that reference the type of search you want to perform (that is, Wrap Search, Search Commands, Search Error Messages, Search Informational Messages). Click “Find Next” to find the next occurrence of a command or message in the transcript after the current cursor position or “Find Previous” to find the previous occurrence of a command or message.

The following figure shows how to find text in the transcript.

Figure 16 Finding Text in the Transcript



Saving or Printing the Transcript

To save selected text from the transcript window, select the text you want to save, then right-click anywhere in the Transcript window, and choose “Save Selection As...” in the pop-up dialog. To save the entire contents of the Transcript, right-click in the Transcript window, and choose “Save Contents As...” To print the transcript, use the keyboard shortcut Control-p.

Clearing the Transcript Window

To clear the transcript window, choose Edit> Clear All. 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.

Interacting with the TestMAX ATPG GUI

The following sections provide an overview on interacting with the TestMAX ATPG GUI:

- [Using Keys in the Command Line](#)
- [Using the Graphical Schematic Viewer](#)
- [Using the Transcript Window](#)
- [Saving Preferences](#)

Using Keys in the Command Line

`Ctrl-V` - Pastes the current copy buffer into the command input.

`Ctrl-X` - Cuts selected text in the paste buffer.

Using the Graphical Schematic Viewer

Note: If you try one of the keys or mouse actions listed below and nothing happens, make sure you click in the graphical schematic window.

Left click - Selects graphic objects of gates or nets. To select an object, click it. TestMAX ATPG deselects existing selections and selects the object you clicked. To deselect the object, click it a second time. To deselect all selections, click over an area with no objects.

Shift-Left click - Adds to the existing selection set. This is identical to clicking the left mouse button except selected items remain selected even when you click new objects.

Ctrl-Left click - Toggles selected items. To select an object and add it to the selected set, click it. To remove the object from the selected set, click the object (already selected).

Delete - Hides all currently selected objects.

Insert - Shows all currently selected objects.

Ctrl-right click over net diamond - Provides a shortcut to the Unconnected Fanout list. Normally left-clicking a net diamond draws the next connection. If there are many connections exist or if you want a specific connection, select it using the Unconnected Fanout list.

`Ctrl-right click over gate` - Updates the contents of the Block Info window if it is open.

`Block Info on Top` - After the Block Info window is open, it moves to the back of all the windows when you do anything in the TestMAX ATPG main window. To keep the Block

Info window always on top, select the option in the Edit > Environment dialog under the Info Viewer tab.

Arrow Keys - The arrow keys scroll the graphical window by one grid. One grid is the distance between two pins on a gate.

Shift-Arrow Keys - Moves your view into the GSV by 1/2 window.

Ctrl-Arrow Keys - Moves your view into the GSV by 1 window.

Page Down - Moves your view into the GSV by one full window.

Page Up - Moves your view into the GSV by one full window.

End - Moves your view into the GSV to the far right.

Home - Moves your view into the GSV to the far left.

Ctrl-F - Performs a ZOOM FULL.

Ctrl-B - Initiates a ZOOM BY BOX. This changes the cursor and allows you to draw a box to zoom into.

Using the Transcript Window

Note: If you try one of the keys or mouse actions listed below and nothing happens, make sure you click in the transcript window.

F3 - Starts a Find Next operation in the transcript.

Left mouse button - Selects text when you left-click at the beginning of the text range, hold, and drag the cursor to the end of the range and release. TestMAX ATPG displays the selected text in reverse highlight. If you drag the cursor outside of the transcript window, the window automatically scrolls.

Ctrl-S - Displays the Save Text dialog and to save either the selected text or all text.

Ctrl-P - Displays the Print menu to print the transcript.

Ctrl-Delete - Clears the transcript window.

Arrow Keys - Move the cursor one character position in the direction selected (up, down, left, right). If necessary, scrolls the window.

Page Up, Page Down - The Page Up key scrolls the transcript up by one page, and the Page Down key scrolls the transcript down by one page. A page is the number of lines that can be displayed in the current window.

Home, End - The Home key moves the cursor to the beginning of the current line and scrolls to the left if necessary. The End key moves the cursor to the end of the current line.

`Ctrl-Home`, `Ctrl-End` - Scrolls the transcript window to the beginning of the first line in the transcript, or the end of the last line in the transcript, respectively.

Saving Preferences

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

When you invoke the TestMAX ATPG GUI, it reads some of the default GSV preferences from the `tmax.rc` file. The TestMAX ATPG GUI has a Preferences dialog 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, TestMAX ATPG saves your changes in the `$(HOME)/.config/Synopsys/tmaxgui.conf` file before it exits. The next time you invoke TestMAX ATPG, it does the following:

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

Using Online Help

TestMAX ATPG provides Online Help in the following forms:

- [Browser-Based Online Help](#) on commands, design flows, error messages, design rules, fault classes, and many other topics.
- [Text-Only Help](#) on TestMAX ATPG commands, displayed in the transcript window. In Tcl mode, enter the command name, followed by the `-help` option. In non-Tcl (native) mode, use the `help` command in the command-line window.

Browser-Based Online Help

You can view detailed help on a wide range of TestMAX ATPG topics by using browser-based Online Help. This section describes the following topics related to Online Help:

- [Setting Up Online Help in Linux](#)
- [Launching Online Help](#)

- [Installing and Running Stand-Alone Online Help in Windows](#)
- [How to Browse, View, and Copy Scripts](#)

Setting Up Online Help in Linux

Note the following when configuring Online Help in Linux:

- If you are starting Online Help for the first time, and you have an existing Netscape profile, Mozilla will initially try to convert your profile. In this case, select "Do Not Convert."
- To set up a default browser for Help, do the following:
 1. If you want to use Firefox, specify the following:

```
alias Firefox '</usr/bin>/firefox'
```

(where </usr/bin> is a path on your network)
 2. If you want to use Mozilla, specify the following:

```
alias mozilla '</usr/bin/>mozilla'
```
 3. Specify the following environment variable to use Firefox as your default browser for running Online Help:

```
setenv USER_HELP_BROWSER /usr/bin/firefox
```
- Your browser's preferences for page setup (ie: open page-links in a new window, most recent viewed window, or new tab/window), can affect the display of popup windows in Online Help. It is recommended that you use the browser's default preference settings for page setup.

Launching Online Help

You can launch TestMAX ATPG Help by doing any of the following:

Selecting the GUI Help Menu

From the menu bar in the GUI, choose Help > Table of Contents, or select a particular topic to open (that is, Command Summary, Getting Started, Fault Classes, and so forth.).

Figure 17 Accessing Help Through the GUI Menu Bar



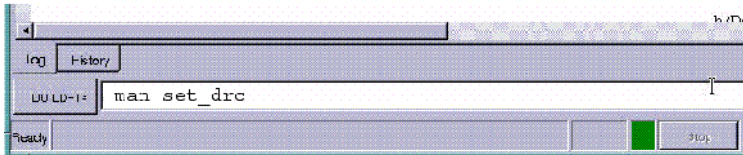
Entering the man Command in GUI Text Field

In the command-line text field of the GUI, use the following syntax to open a topic related to either a specific command (`set_drc`) or a message (that is, M401):

```
> man command | message_id
```

See the following figure for an example.

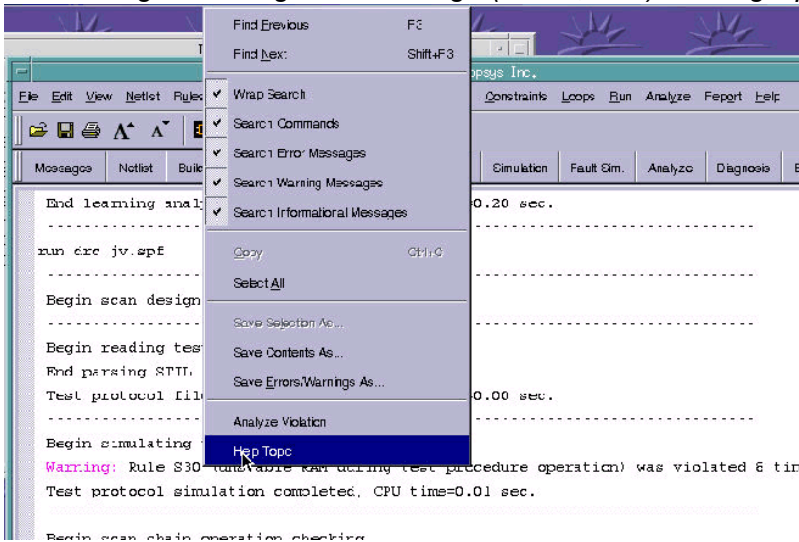
Figure 18 Opening a Specific Topic in TestMAX ATPG Help From the Command-line Text Field



Right-Clicking On a Command Or Message

Right-click a particular command or message in the console window, then select Help Topic. The Help topic for the command on message will appear. See Figure 3 for an example.

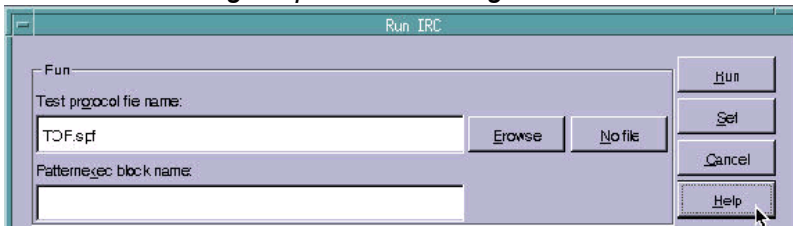
Figure 19 Right-Clicking on a Message (that is, S30) to Bring Up Online Help



Click the Help Button in Dialog Box

Click the Help button within a dialog box to bring up a Help topic that describes the active dialog box. See Figure 4.

Figure 20 Accessing Help From a Dialog Box



Installing and Running Stand-Alone Online Help in Windows

There are several ways you can run a stand-alone version of TestMAX ATPG Online Help in Windows:

- In the first method (advised), you download a .zip file from a SolvNet article and unzip the file to your desktop.
- You can also copy a .tar file containing TestMAX ATPG Online Help from the product installation tree and extract the file on your desktop
- Your third option is to download a Microsoft Help (CHM) version of TestMAX ATPG Online Help from a SolvNet article.

How to Download a .zip file and Install TestMAX ATPG Online Help

1. Go to the following URL:

<https://solvnetplus.synopsys.com/s/article/How-to-Set-Up-Browser-Based-TetraMAX-Online-Help-in-Windows-1576148274956>

2. In the SolvNet article that appears (How to Set Up Browser-Based TestMAX ATPG Online Help in Windows), click on the tmax_olh.zip link.

How to Set Up Browser-Based TetraMAX Online Help in Windows

Doc Id: 1467037 Product: TetraMAX Last Modified: 03/03/2017

Average User Rating: ☆☆☆☆☆ (0) Rate Article: ☆☆☆☆☆ Send comment

 Save Article  Tag Article  Print  Email

You can set up and install TetraMAX Online Help in Windows so it will run in a standard HTML browser (Internet Explorer, Firefox, Chrome, etc.) on your desktop.

The following PDF attachment shows you how to set up and install TetraMAX Help:

[how_to_set_up_tmax_olh.pdf](#)

The following .zip file contains an archived version of TetraMAX Help in HTML format:

[tmax_olh.zip](#)

3. Save the tmax_olh.zip file to a local directory.
4. Extract the .zip file.

5. Create a shortcut to TestMAX ATPG Help from the Default.htm file in the top level of the extracted directory. (To do this, right-click on the Default.htm file and select "Create Shortcut" from the menu.)
6. Drag and drop the shortcut icon to your desktop.

How to use a .tar file from the Product Installation and Install TestMAX ATPG Online Help

1. Copy the tmax_olh.tar file from the following location in the TestMAX ATPG installation directory to a Windows machine:

\$SYNOPSISYS_install_path/doc/test/tmax_olh.tar

2. Extract the contents of the tmax_olh.tar file.
3. To create a shortcut for TestMAX ATPG Online Help, follow steps 5 and 6 in the previous section, "How to Download a .zip file and install TestMAX ATPG Online Help."

How to Download and install a Microsoft Help (CHM) version of TestMAX ATPG Online Help

1. Go to the following URL:

<https://solvnet.synopsys.com/retrieve/1466979.html>

2. In the SolvNet article that appears (TestMAX ATPG Help File in CHM Format), click the "TestMAX ATPG Online Help CHM File" link.

TetraMAX Help File in CHM Format

Doc Id: 1466979 Product: TetraMAX Last Modified: 06/26/2017

Average User Rating: ★★★★★ (5) Rate Article: ☆☆☆☆☆ Send comment

Save Article Tag Article Print Email

TetraMAX Online Help is available in Microsoft Compiled HTML Help (CHM) format.

The attached tmax_olh.chm file can be downloaded directly to your Windows desktop and will run as a Windows-based application as long as you have Internet Explorer installed on your system. This CHM file includes the TetraMAX User Guide and the Test Pattern Validation User Guide.

In some cases, the content in the CHM file might be blocked. This is because extra security settings have been turned on for your system. To fix this issue, right-click on the file name or icon, and select "Properties" from the pop-up menu. In the Properties dialog box, click the "General" tab, then go to the "Security" section (located at the bottom), and unblock all security parameters.

[TetraMAX Online Help CHM File](#)

3. Save the file to a local directory.

Important: Do not run the .chm file from a network location. It will not work.

4. Double-click the executable tmax_olh.chm file.

TestMAX ATPG Online launches as a stand-alone CHM application.

How to Browse, View, and Copy Scripts

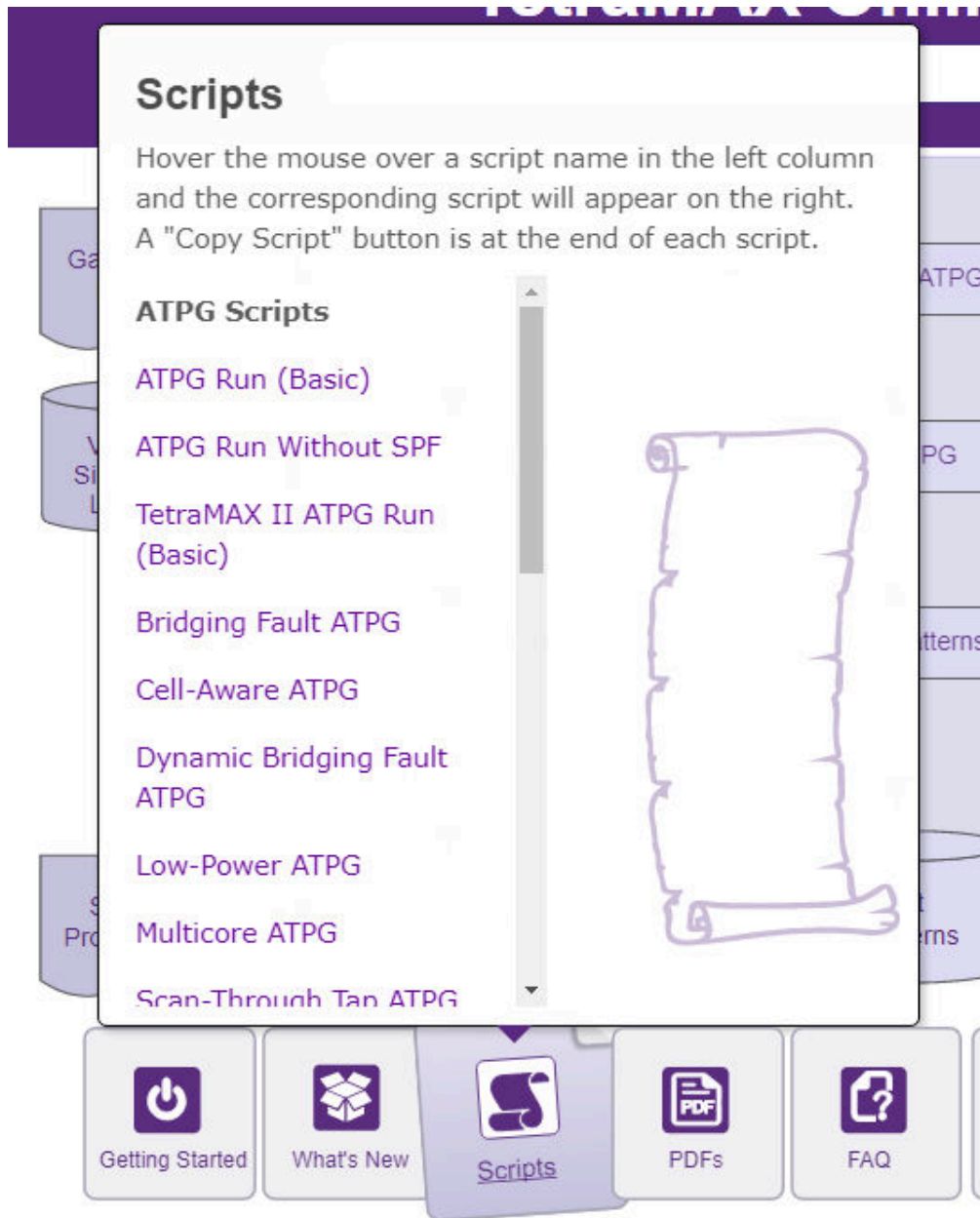
There are several ways you can access and copy scripts from TestMAX ATPG Online Help:

- The "Scripts" button at the bottom of the home page
- The "Scripts" menu at the top over every topic
- The "Scripts" chapter of the PDF TestMAX ATPG User Guide.

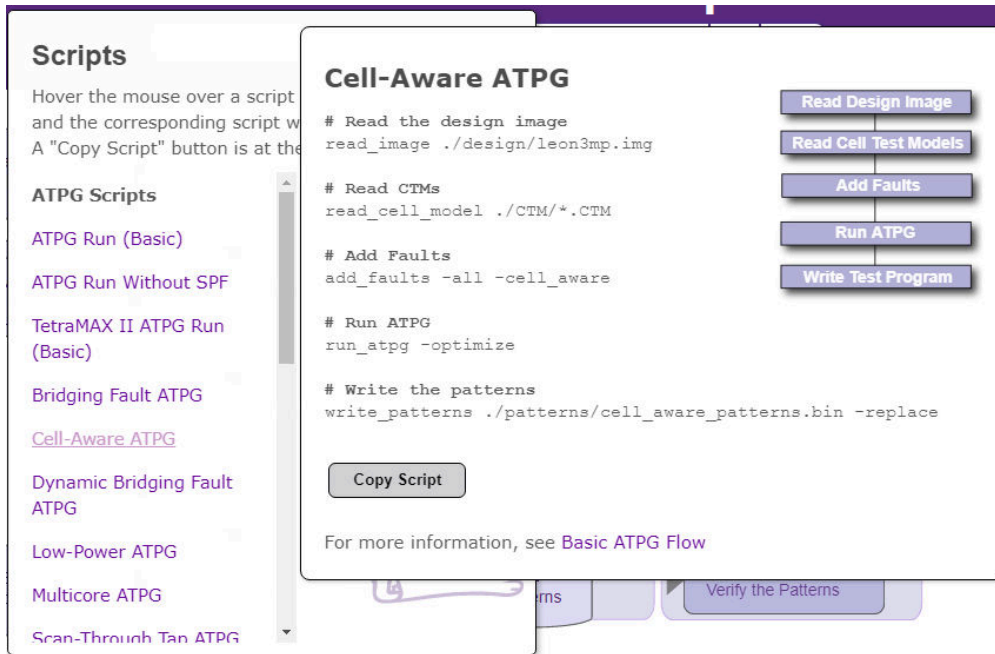
To use the Scripts button at the bottom of the home page:

1. Hover your mouse over the Scripts button.

A list of scripts appears in a popup window.



2. Hover your mouse and browse through various scripts in the scroll-down list.
The script associated with each hovered list item appears in a popup window on the right side of the list.



3. Move your mouse into the script popup window. You can manually select and copy the script or use the "Copy Script" button at the bottom of the window to automatically select the entire script and copy it to your clipboard.


```
# Read the design image
read_image ./design/leon3mp.img

# Read CTMs
read_cell_model ./CTM/*.CTM

# Add Faults
add_faults -all -cell_aware

# Run ATPG
run_atpg -optimize

# Write the patterns
write_patterns ./patterns/cell_aware_patterns.bin -replace
```

Read Design Image

Read Cell Test Models

Add Faults

Run ATPG

Write Test Program

Copy Script

For more information, see [Basic ATPG Flow](#)

4. Move your mouse anywhere outside the popup window to close it.

Text-Only Help

To access text-only help on a command in Tcl mode, enter the name of the command, followed by the `-help` option of the command-line window. In non-Tcl (native) mode, use the `help` command, followed by the name of the command.

A Tcl mode text-only help example is as follows:

```
BUILD-T> set_workspace_sizes -help
Usage: set_workspace_sizes
[-connectors]      (maximum number of fanout connections supported)
[-decisions]       (maximum active decisions)
[-drc_buffer_size] (maximum DRC buffer size)
[-line]           (maximum line length)
[-string]         (maximum string length)
[-command_line]   (command line length)
[-command_words]  (command line words)
```

A native mode text-only help example is as follows:

```
BUILD> help set workspace sizes  
Set Workspace Sizes [-Atpg_gates d]  
[-CONNECTors d] [-Decisions d] [-DRC_buffer_size d] [-Line d]  
[-String d] [-COMMAND_Line d] [-COMMAND_Words d]
```

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

```
BUILD-T> report_commands -all  
add_atpg_constraints          add_atpg_primitives  
add_capture_masks           add_cell_constraints  
add_clocks                   add_display_proc  
add_delay_paths              add_display_gates  
add_distributed_processors    add_equivalent_nofaults  
add_faults                   add_net_connections  
add_nofaults                 add_pi_constraints  
...
```

For a list of available options associated with a command, type the name of the command, followed by a dash and the TAB key:

```
BUILD-T> report_buses -  
all          clock      keepers    noverbose  verbose  
behavior     contention max      pull       weak  
bidis        gate_id   names     summary    zstate
```

6

Using the Graphical Schematic Viewer

The graphical schematic viewer (GSV) 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.

The following sections describe how to use the GSV for interactive analysis and correction of test design rule checking (DRC) violations and test pattern generation problems:

- [Getting Started With the GSV](#)
- [Displaying Pin Data](#)
- [Analyzing a Feedback Path](#)
- [Checking Controllability and Observability](#)
- [Analyzing DRC Violations in the GSV](#)
- [Analyzing Buses](#)
- [Analyzing ATPG Problems](#)
- [Printing a Schematic to a File](#)

Getting Started With the GSV

The following sections describe how to get started using the GSV:

- [Using the SHOW Button to Start the GSV](#)
- [Starting the GSV From a DRC Violation or Specific Fault](#)
- [Navigating, Selecting, Hiding, and Finding Data](#)
- [Expanding the Display From Net Connections](#)
- [Hiding Buffers and Inverters in the GSV Schematic](#)
- [ATPG Model Primitives](#)

- [Displaying Symbols in Primitive or Design View](#)
 - [Displaying Instance Path Names](#)
-

Using the SHOW Button to Start the GSV

The following steps describe how 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.)

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.

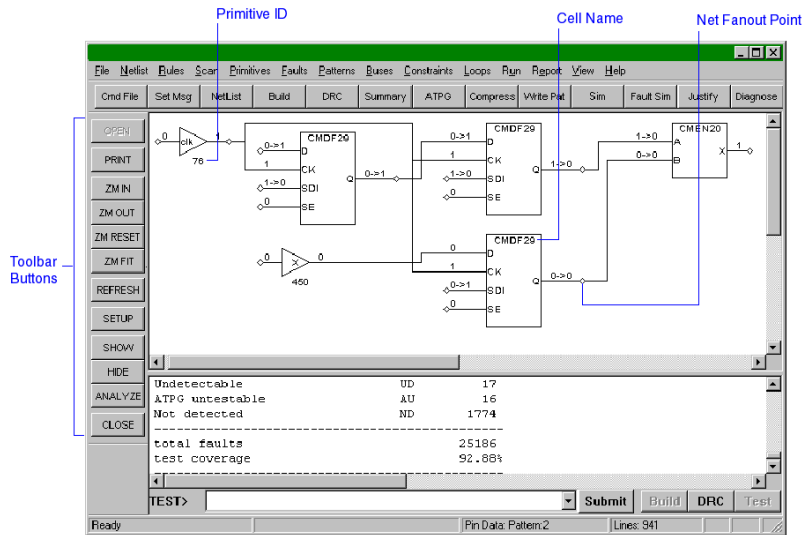
Your entry is added to the list box.

5. Repeat steps 3 and 4 to add all the parts of the design that you want to view.

6. Click OK

The following figure shows the TestMAX ATPG GUI main window split by the movable divider. The top window shows a GSV schematic containing the specified objects. The bottom window contains the transcript.

Figure 21 GSV in the TestMAX ATPG GUI Main Window



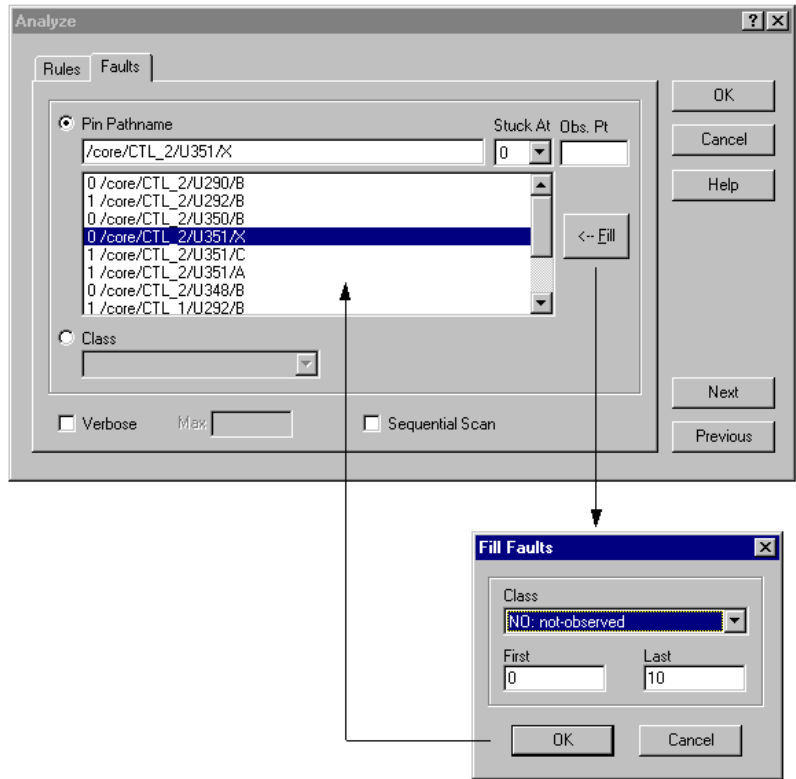
Starting the GSV From a DRC Violation or Specific Fault

You can start the GSV and view a specific DRC violation by using the Analyze dialog box, as shown in the following steps:

1. Click the ANALYZE button in the GSV toolbar.

The Analyze dialog box appears as shown in the following figure.

Figure 22 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.

The Fill Faults dialog box opens.

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.
6. Click OK to fill in the list box in the Analyze window, as shown in the following figure.
7. From the list, select the specific fault you would like displayed, such as “0 /core/CTL_2/U351/X”.

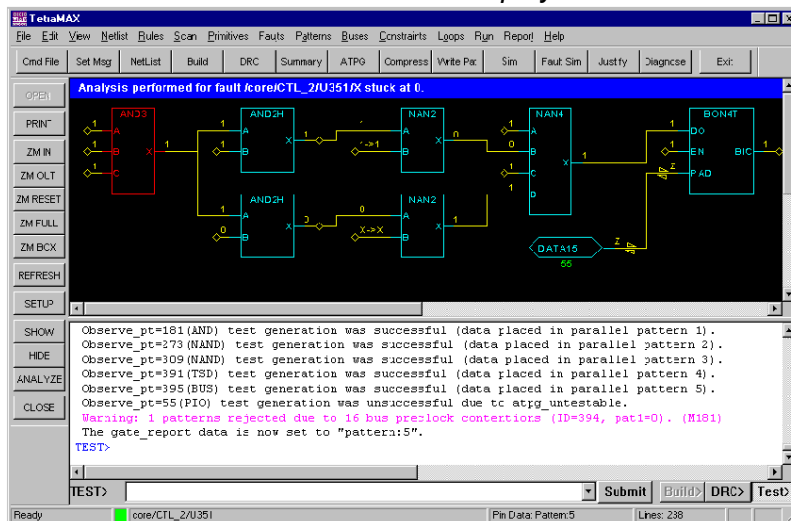
The fields at the top of the dialog box are filled in automatically from your selection.

8. Click OK.

The Analyze dialog box closes and the GSV displays the logic associated with the selected fault location.

The following figure 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 23 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 the following example.

Transcript of Not-Observed Analysis

```

TEST-T> 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 in the [Analyzing a NO Fault](#) section.

See Also

- [Performing Design Rule Checking](#)
- [Fault Lists and Faults](#)

Navigating, Selecting, Hiding, and Finding Data

Within the GSV, you can navigate to different locations and views, select objects, hide objects, and find specific data for various objects.

The following sections describe each of these actions:

- [Navigating Within the GSV](#)
- [Selecting Objects in the GSV Schematic](#)
- [Hiding Objects in the GSV Schematic](#)
- [Using the Block ID Window](#)

Navigating Within the GSV

To navigate within the GSV window, use the horizontal or vertical slider; the arrow keys on the keyboard; and the ZM IN, ZM OUT, ZM RESET, ZM FULL, and ZM BOX buttons.

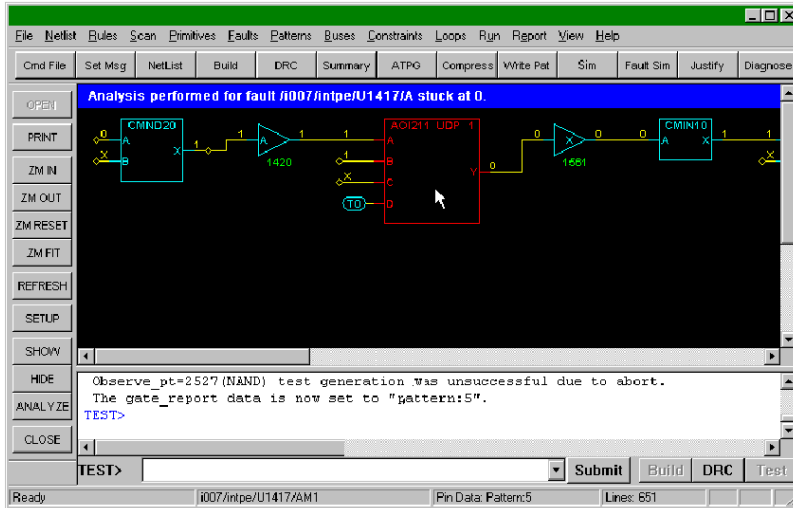
To zoom in to a specific area, click the ZM BOX button and then drag a box around the area to be magnified.

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 the following figure.

To deselect the object, click it again. To select more than one object, hold down the Shift key and click each object.

Figure 24 Selected Object Name



Hiding Objects in the GSV Schematic

The following steps describe how 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. The following steps show you how 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 of the menu.

The Block ID window appears with information about the selected object.

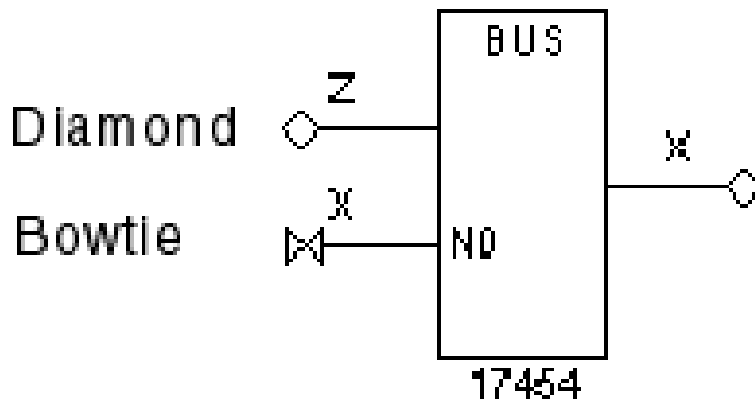
4. To display information for other objects, with the Block ID window still open, click each object with the right mouse button while holding down the Control key.

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 the following figure:

- The diamond symbol represents a unidirectional net connection
- The bow tie symbol represents a bidirectional net connection

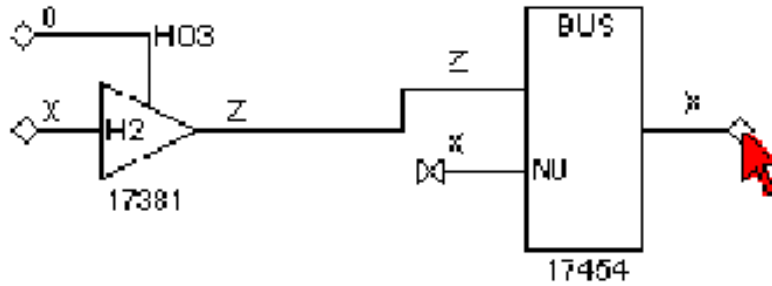
Figure 25 Net Expansion Symbols: Diamond and Bow Tie



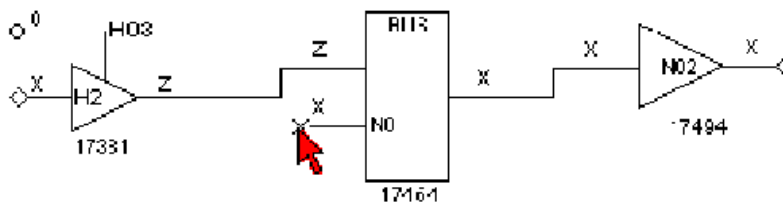
To expand the display from a specific connection, click the diamond or bow tie that represents the connection of interest. The schematic expands to include the next gate or component forward or backward from the selected connection. Each click adds one component to the display. If a net has multiple additional components, you can click repeatedly and display more components until the diamond or bow tie no longer appears.

The following steps show an example of the results obtained by clicking the diamond and bow tie connection points:

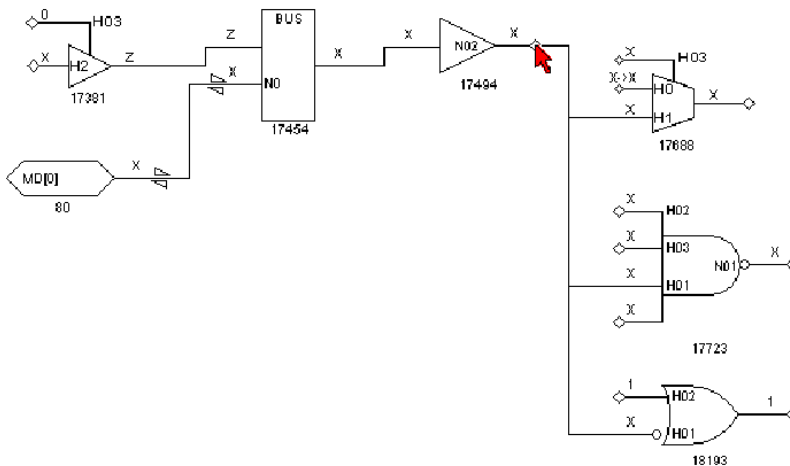
1. Click the diamond.



2. Click the boxtie on gate 17454.



3. Click three times on diamond on gate 17454.



The following steps describe how 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.

For information about the `add_net_connections` command, see the man pages.

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 instead display them as double slashes and bubbles.

The following steps describe how 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](#).)

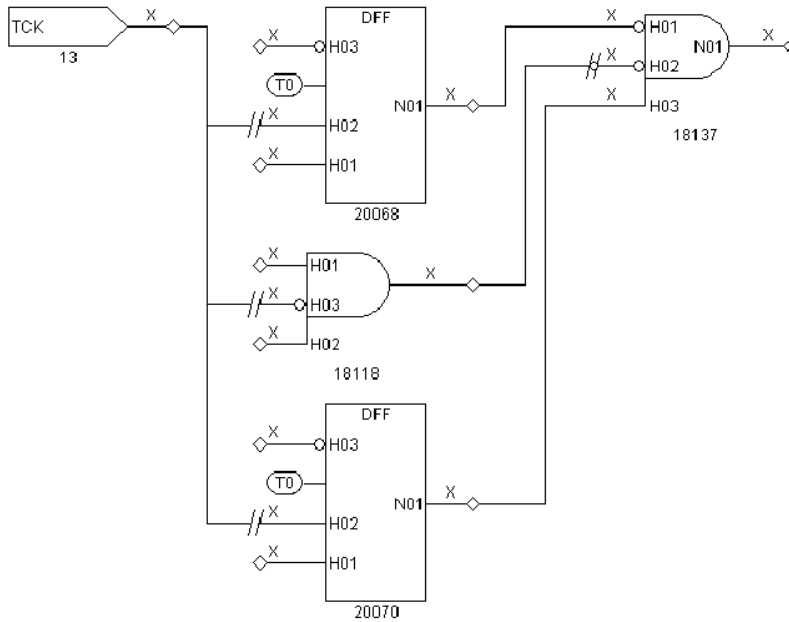
2. Select the BUF/INVs check box in the Hide section.
3. Click OK.

TestMAX ATPG 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 the following figure. 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 26 Schematic With Buffers and Inverters Hidden



ATPG Model Primitives

This section describes the set of TestMAX ATPG primitives that are used in GSV displays when Primitive is selected in the GSV Setup dialog box. If Design is selected, see [Displaying Symbols in Primitive or Design View](#).

The primitives include the following:

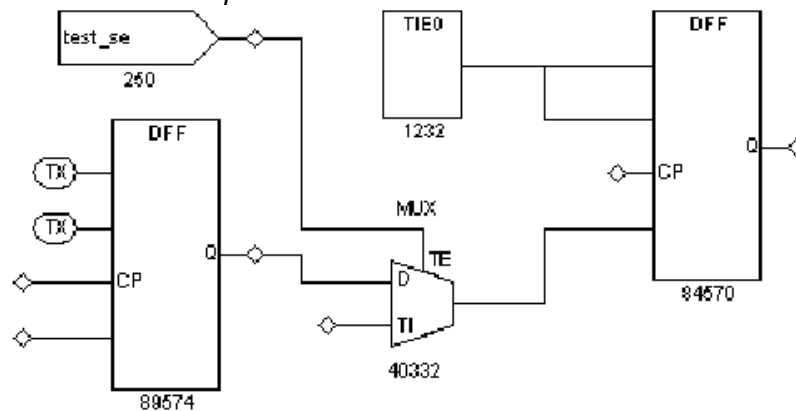
- [Tied Pins](#)
- [Primary Inputs and Outputs](#)
- [Basic Gate Primitives](#)
- [Additional Visual Characteristics](#)
- [RAM and ROM Primitives](#)

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 the following figure, 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 the following figure, 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 27 GSV Representation of Tied Pins

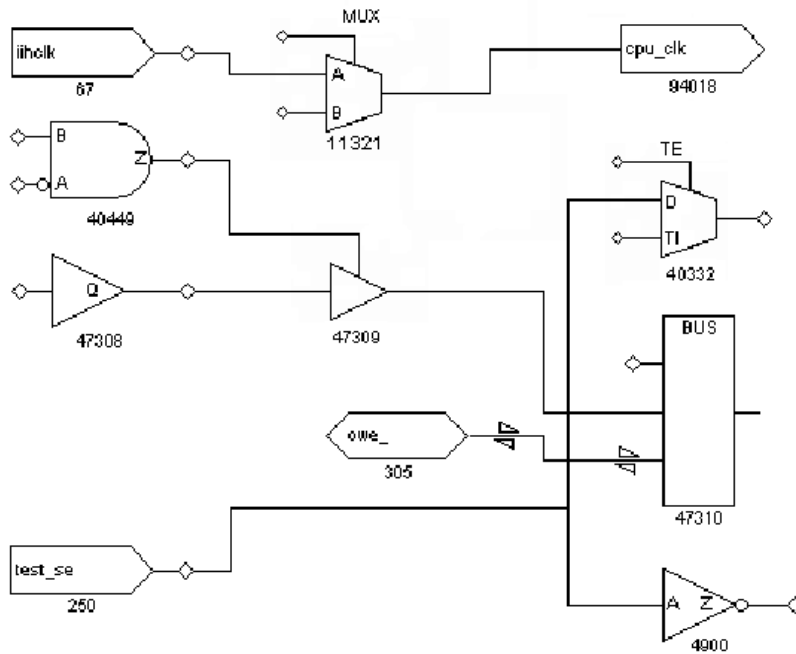


Primary Inputs and Outputs

Primary (top-level) inputs and outputs are identified in the following ways:

- Primary inputs are identified with the symbol shown for gates 67 and 250 in the following figure. 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 the following figure. 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 the following figure. 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.

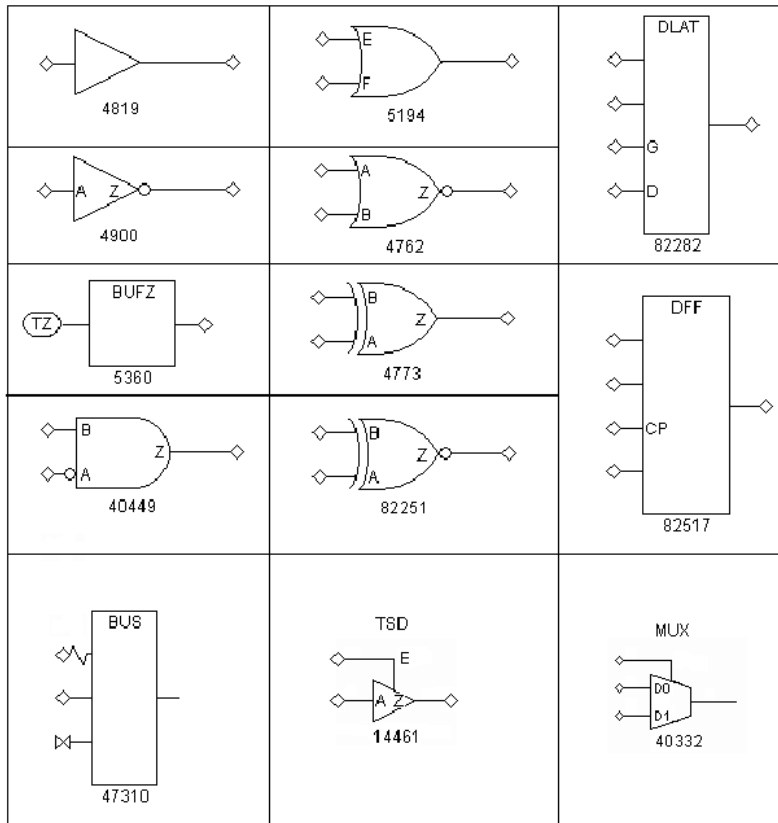
Figure 28 Primary I/O and Bidirectional Port Symbols



Basic Gate Primitives

The following figure shows representative symbols for many of the more commonly used TestMAX ATPG 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 Help reference topic “ATPG Simulation Primitives.”

Figure 29 Some Basic Gate Primitives



Additional Visual Characteristics

Some additional visual characteristics of ATPG primitives are described as follows:

- **Merged inverters:** Inverters can be merged into the drawn symbol to make the schematic more compact. For example, in the preceding figure, 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 the preceding figure, 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 the preceding figure, 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 TestMAX ATPG

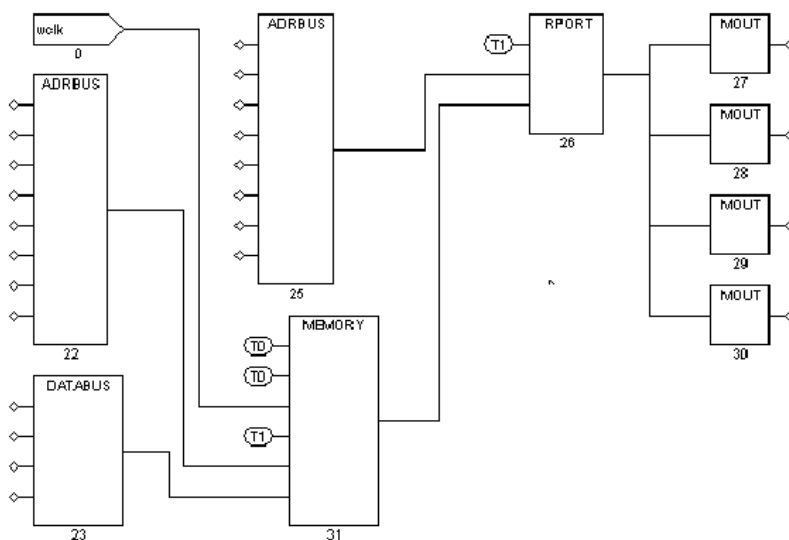
primitives. A single module can be represented by several TestMAX ATPG primitives; in that case, the labels do not all appear on the same TestMAX ATPG primitive.

- Pin order: The order of pins on the TSD, DLAT, DFF, and MUX primitives is significant. Refer to the preceding figure; 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 and 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 the following figure. 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 4: RAM and ROM Primitives



The RAM and ROM primitives are described as follows:

- **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.

See Also

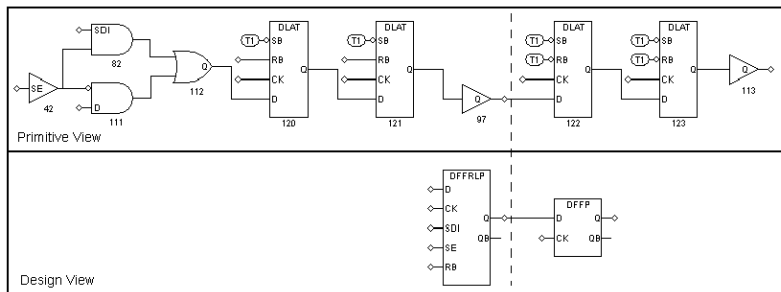
- [Creating Custom ATPG Models](#)

Displaying Symbols in Primitive or Design View

You can choose to display a schematic using the TestMAX ATPG 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 1 shows two different views of a design.

Figure 30 Comparison of Primitive and Design Views



Note that the schematic labeled *Primitive View* uses TestMAX ATPG primitives; the schematic labeled *Design View* uses cells in the technology library.

Displaying Instance Path Names

You can display the instance path name above each instance in the schematic, as described in the following steps:

1. Select Edit > Environment in the menu bar.
The Environment dialog box appears.
2. In the Environment dialog box, click the Viewer tab.
3. Select the Display Instance Names check box.
4. Click OK.

See Also

- [Masking Scan Cell Inputs and Outputs](#)

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, TestMAX ATPG 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.

The following sections describe how to display pin data:

- [Using the Setup Dialog Box to Display Pin Data](#)
- [Pin Data Types](#)
- [Displaying Clock Cone Data](#)

- [Displaying Clock Off Data](#)
- [Displaying Constrain Values](#)
- [Displaying Load Data](#)
- [Displaying Shift Data](#)
- [Displaying Test Setup Data](#)
- [Displaying Pattern Data](#)
- [Displaying Tie Data](#)

Using the Setup Dialog Box to Display Pin Data

The following steps describe how to display pin data on the schematic using the Setup dialog box:

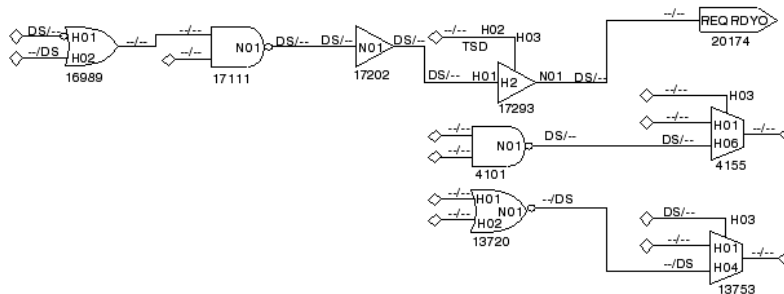
1. With a schematic displayed in the GSV (for example, as shown in the following figure), click the SETUP button on the GSV toolbar.

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.

TestMAX ATPG redraws the schematic using the new pin data type.

Figure 31 GSV Display With Pin Data Type Set to Tie Data



To set the pin data display mode from the command line, use the `set_pindata` command. For example:

```
TEST-T> set_pindata -clock_cone CLK
```

For complete syntax and option descriptions, see Online Help for the `set_pindata` command.

Pin Data Types

The following table lists each pin data type, a description of the data displayed in the GSV, and its typical use. You can find additional related information in the description of the `set_pindata` command in Online Help.

Table 2 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 On	Simulated values when all clocks are held in on state	Debugging clock (C) violations
Clock Off	Simulated values when all clocks are held in off state	Debugging clock (C) violations
Constraint 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 load_unload procedure	Debugging problems in a STIL load_unload macro

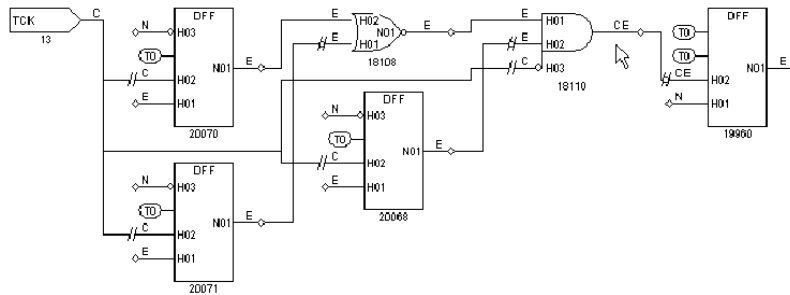
Table 2 Pin Data Types (Continued)

Pin Data Type	Data Displayed	Typical Use
Master Observe	Simulated values for the master_observe procedure	Debugging problems in a STIL master_observe 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 shadow_observe procedure	Debugging problems in a STIL shadow_observe procedure
Shift	Simulated values for the Shift procedure	Debugging DRC T (scan chain tracing) violations
Stability Patterns	Simulated values for the load_unload, Shift, and capture procedures	Analysis of classification of nonscan cells
Test Setup	Simulated values for the test_setup macro	Debugging problems in a STIL test_setup 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 in the following figure. This example shows the clock cones and effect cones of the TCK clock port.

Figure 32 GSV Display: Pin Data Type Set to Clock Cone



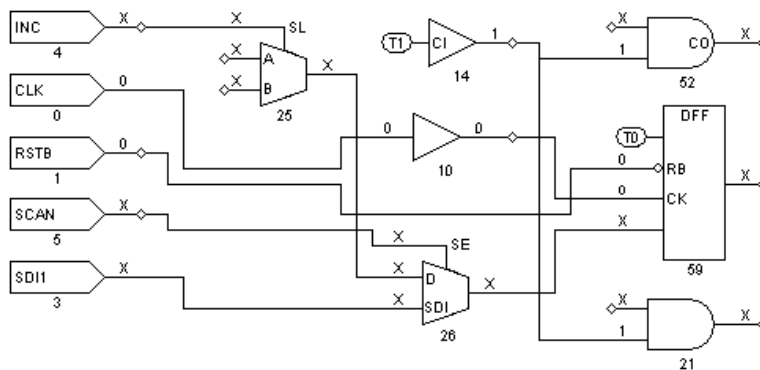
Note the following:

- 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 the following figure.

Figure 33 GSV Display: Pin Data Type Set to Clock Off



In the preceding figure, 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.

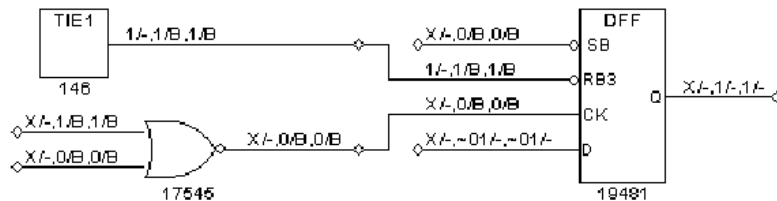
In this design, the clock ports are CLK and RSTB and their nets have values of 0. The 0 value from the CLK net is propagated to the input of gate 10, the output of gate 10, and the CK input of gate 59 (the DFF). The 0 value of RSTB is propagated to the RB input of the same DFF, gate 59. Notice that the 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. The following figure shows a schematic displaying the constrain values.

Figure 34 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:

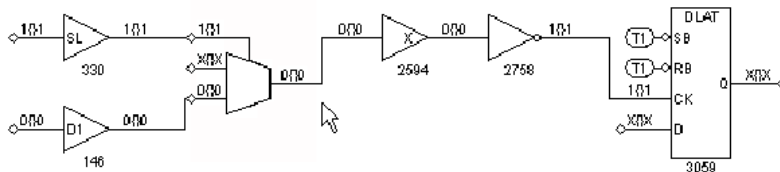
- T is the pin's value that is a result of tied circuitry, if any exists. An "X" indicates that there is no value due to tied logic.
- B1 indicates whether faults are blocked on the pin because of the tied value "T." A value of "B" indicates that the fault is blocked; a dash (-) indicates that the fault 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. The following figure shows a schematic displaying the load data.

Figure 35 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. TestMAX ATPG 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.

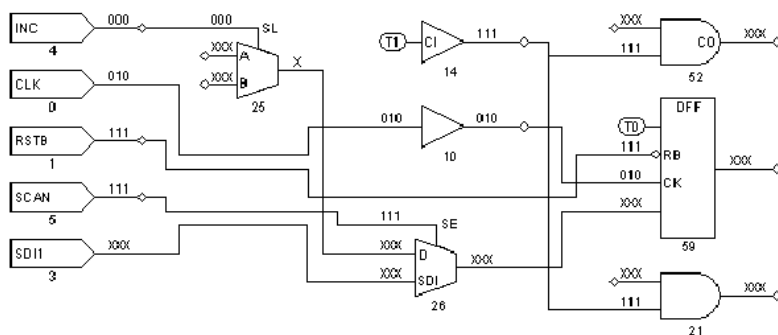
If no test cycles in the load_unload procedure occur after the Shift procedure, BB is an empty string. Otherwise, the string displayed for BB contains characters: one character for each test cycle that can be represented with a single time event, and multiple characters for any test cycles that require 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 the following figure.

In the following figure, 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 36 GSV Display: Pin Data Type Set to Shift



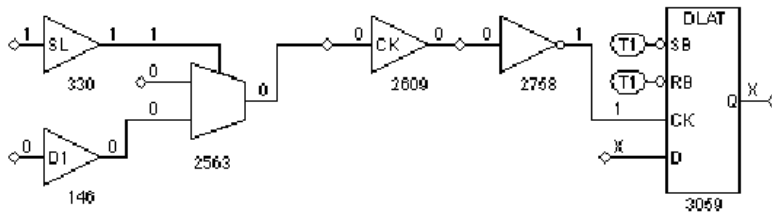
These are all appropriate values for the STIL Shift procedure shown in the following example:

```
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 the following figure.

Figure 37 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-T> drc
DRC-T> set_drc -store_setup
DRC-T> 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.

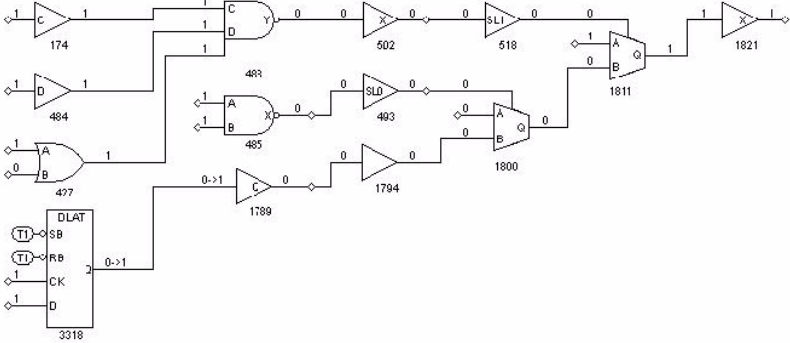
To display logic values for a specific pattern:

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 the following figure.

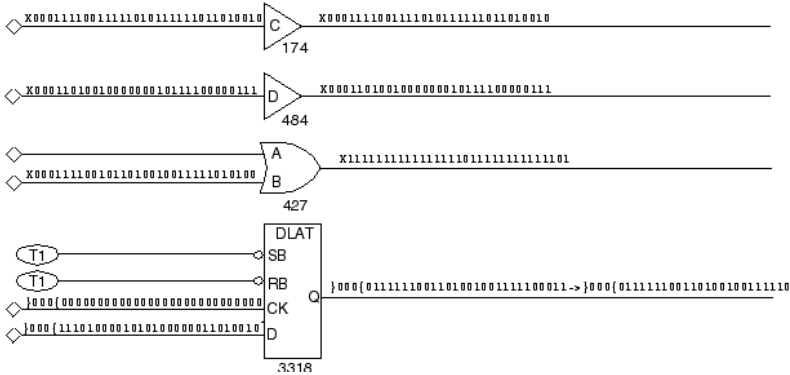
Figure 38 GSV Display: Pin Data Type Set to a Pattern Number



The logic values shown with an arrow, as in 0->1, show the pre-clock state on the left and the post-clock state on the right. A logic state shown as a single character represents the pre-clock state. For a clock pin, a single character represents the clock-on state.

To display logic values for all patterns, choose All Patterns in the GSV Setup dialog box and click OK. The following figure 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 39 GSV Display: Pin Data Type Set to Pattern All



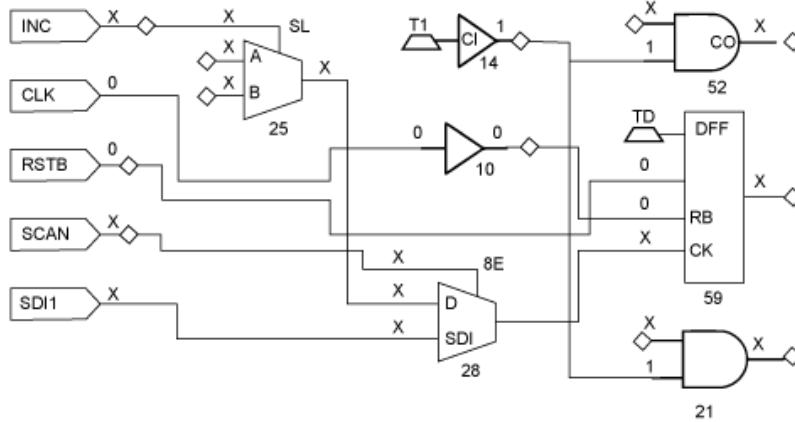
To examine a pattern that is not in the final 32 patterns processed, choose Good Sim Results in the GSV Setup dialog box and click OK.

For an additional method of viewing the logic values from a specific pattern, see “[Running Logic Simulation](#) .” 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 the following figure.

Figure 40 Displaying Tie Pin Data in the GSV



In the preceding figure, 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.

Analyzing a Feedback Path

You can use the GSV to review combinational feedback loops in the design. The following example 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.

Report Feedback Paths Transcript

```
TEST-T> 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
```

Chapter 6: Using the Graphical Schematic Viewer Checking Controllability and Observability

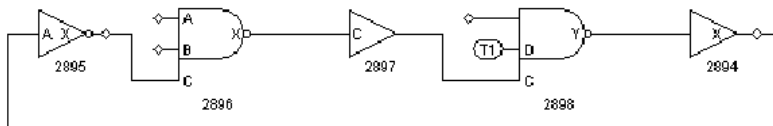
```

TEST-T> 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. The following figure shows the resulting schematic display for feedback path number 6 in the preceding example.

Figure 41 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 if:

- a single internal point is controllable and observable
- a single internal point is controllable and observable within existing ATPG constraints
- multiple points can be set to required states simultaneously

You specify one or more internal pin states to achieve. TestMAX ATPG 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 running 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 of 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. TestMAX ATPG attempts the justification and reports the results.

Using the run_justification Command

The following example 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.

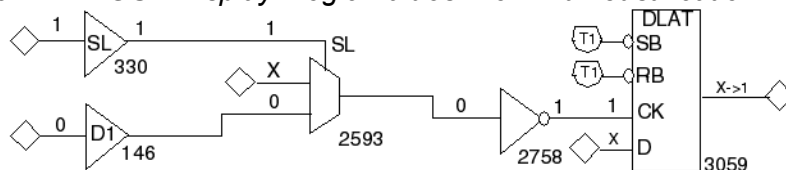
Using the run_justification Command

```
TEST-T> run_justification -set 330 1 -set 146 0 -store
```

Successful justification: pattern values available in pattern 0.

The following figure shows a schematic that displays the data for pattern number 0 in the preceding example. Gate 146 is at logic 0 state and gate 330 is at logic 1, as requested. Justification was successful, and TestMAX ATPG was able to create a pattern to satisfy the list of set points.

Figure 42 GSV Display: Logic Values From Run Justification



See Also

- [Analyzing the Cause of Low Test Coverage](#)

Analyzing DRC Violations in the GSV

To analyze DRC violations in the GSV:

1. Run DRC. For details, see [Starting Test DRC](#).
2. Click the ANALYZE button in the command toolbar of the GSV.
3. Click the Rules tab and select a violation from the displayed list or enter a specific violation occurrence number in the Rule Violation field.
4. Click OK.
5. Determine the cause of the violation and correct it. For details, see [Output from the run_drc Command](#).
6. Run DRC again using the Run DRC Dialog Box.
7. List the violations of the same rule, verify the absence of the violation you just corrected, and examine the remaining violations. (Sometimes, correcting a violation corrects others as well. But it also might create new violations.)
8. Return to Step 2 and repeat the same process until all violations of the rule have been corrected.

The following topics show how to troubleshoot some typical DRC violations:

- [Troubleshooting a Scan Chain Blockage](#)
- [Troubleshooting a Bidirectional Contention Problem](#)

Troubleshooting a 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.

The following example shows the transcript message for violation S1-13.

S1-13 Violation Message

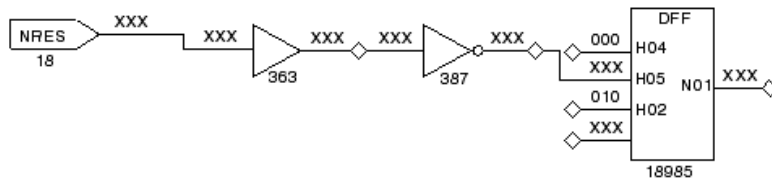
```
Error: Chain c16 blocked at DFF gate /spec_asic/alu/bits/AD_DATIN/
ff_reg (18985)
after tracing 3 cells. (S1-13)
```


The following steps show you how to view the violation:

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 the following figure 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 43 GSV Display: DRC Violation S1-13



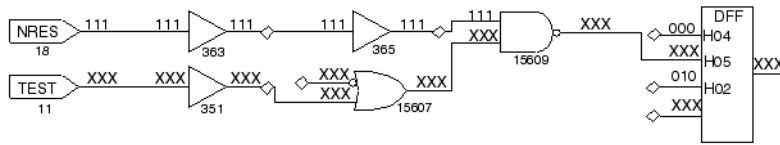
The following steps show you how to find the signal blocking the scan chain at gate 18985:

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.

The NRES input 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 should investigate these possibilities and correct the problem, then execute the `run_drc` command and examine the new list of violations.

After correcting the NRES input problem and executing the `run_drc` command, you select violation S1-9 from the list of remaining S1 violations. As before, you display the violation using the GSV. The following figure shows the resulting schematic with Shift pin data displayed.

Figure 44 GSV Display: DRC Violation S1-9

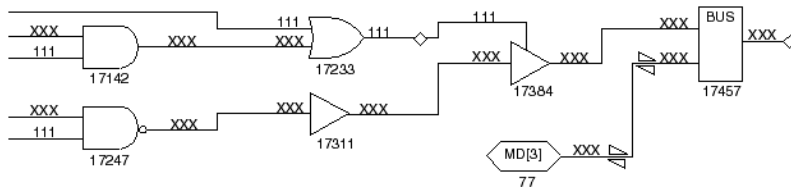


In the preceding figure, although the NRES input is now correctly defined as a clock with an off state of 1, there is a problem with the reset pin, pin H05, on gate 19766 (DFF). Tracing the XXX values back as in the previous example, you find that the source is the primary input TEST. In this case, TEST was not defined as a constrained port in the STIL file.

To correct the problem, you need to 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 the following figure.

Figure 45 GSV Display: Another S1 Violation



This time, the problem is associated with the bus device, which is a gate inserted by TestMAX ATPG 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 [STIL Procedure Files](#)).

After you make this correction, you will need to execute the run_drc command again and find no further S1 violations.

Troubleshooting a Bidirectional Contention Problem

Bidirectional contention issues on ports and internal pins are checked by the Z rules. In Example 1, you use the report_rules command to get a listing of Z rules that have failed. This particular 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

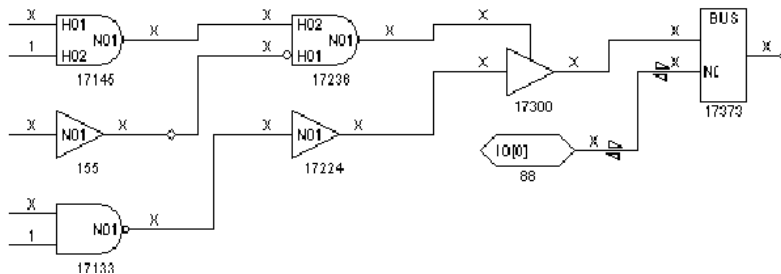
list of five violations, as shown in the following example. From those, you select the Z4-1 violation to troubleshoot first.

report_rules Listing of Violation Messages

```
TEST-T> report_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-T> report_violations z4 -max 5
Warning: Bus contention on /spec_asic/L030 (17373)
occurred at time 0 of test_setup procedure. (Z4-1)
Warning: Bus contention on /spec_asic/L032 (17374)
occurred at time 0 of test_setup procedure. (Z4-2)
Warning: Bus contention on /spec_asic/L034 (17375)
occurred at time 0 of test_setup procedure. (Z4-3)
Warning: Bus contention on /spec_asic//L036 (17376)
occurred at time 0 of test_setup procedure. (Z4-4)
Warning: Bus contention on /spec_asic/L038 (17377)
occurred at time 0 of test_setup procedure. (Z4-5)
```

According to the violation error message in the preceding example, the problem is bus contention at time 0 of the test_setup macro. You display the violation using the GSV, as shown in the following figure. The schematic shows the test_setup data.

Figure 46 GSV Display: DRC Violation Z4-1

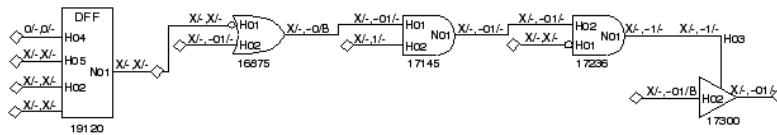


The schematic display shows a bidirectional port, IO[0], which is at an X state. In addition, BUS has both inputs driven at X; at least one should be a Z value. Tracing back from BUS, you find a three-state driver TSD (gate 17300) whose enable and data values are both X. There appear to be numerous potential 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 the following figure.

Figure 47 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:

1. 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.
2. 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.
3. Adjust the contention checking to monitor bus contention both before and after clock events. TestMAX ATPG 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, TestMAX ATPG analyzes bus and wire gates to determine if 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 data from the analysis is used to prevent issuing false contention violations for the STIL procedures.

The following sections describe how to analyze buses:

- [BUS Contention Status](#)
- [Understanding the Contention Checking Report](#)
- [Reducing Aborted Bus and Wire Gates](#)
- [Causes of Bus Contention](#)

BUS Contention Status

Based on the results of DRC 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 TestMAX ATPG 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. TestMAX ATPG 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.

See Also

- [Contention Analysis](#)

Understanding the Contention Checking Report

After the contention check is complete, TestMAX ATPG displays a report similar to the following example. This report identifies the number of bus and wire gates and the number of gates that were placed into each contention and Z-state category.

DRC Report for Contention Checking

```
SETUP> run_drc spec_stil_file.spf
# -----
# Begin scan design rule checking...
# -----
```

```
# Begin reading test protocol file spec_stil_file.spf...
# End parsing STIL file spec_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

Reducing Aborted Bus and Wire Gates

Bus gates associated with aborted contention checking are still checked for contention during ATPG. If contention checking is aborted for some gates, you should increase the effort used to classify as pass, fail, or bidirectional, rather than abort. You can do this using the Analyze Buses dialog box, or you can use the `set_atpg -abort_limit` and `analyze_buses` commands on the command line, as shown in the following example.

Using `set_atpg -abort_limit` and `analyze_buses`

```
TEST-T> report_buses -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-T> set_atpg -abort 50
TEST-T> analyze_buses -all -update
Bus Contention results: #pass=257, #bidi=31, #fail=289, #abort=0,
CPU time=0.00
TEST-T> analyze_buses -zstate -all -update
Bus Zstate ability results: #pass=160, #bidi=128, #fail=289,
#abort=0, CPU
time=0.80
```

```
TEST-T> report_buses -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_buses` Commands

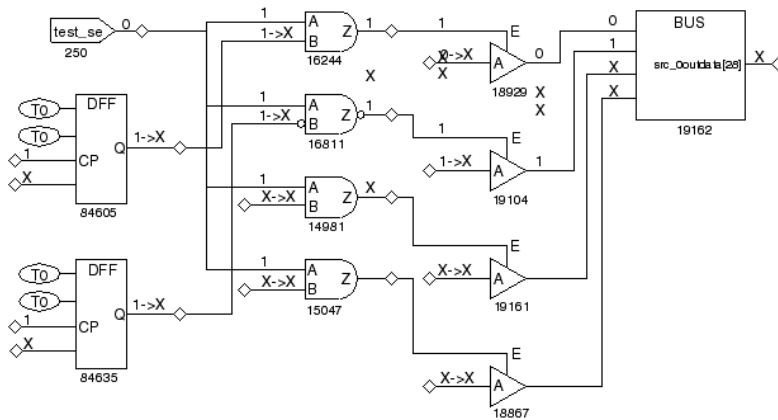
To reduce the number of aborted bus and wire gates from the command, use the `set_atpg -abort_limit` and `analyze_buses` commands.

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.

The following figure 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, TestMAX ATPG 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 48 GSV Display: DRC Violation Z1



You can deal with the Z1 and Z2 violations in one of the following ways:

- Ignore the warnings, with the following consequences:
 1. Contention will probably occur during pattern generation, and TestMAX ATPG will discard those patterns that result in contention, possibly increasing the runtime.
 2. The resulting test coverage could be reduced. TestMAX ATPG might be forced to discard patterns that would otherwise detect certain faults.
 3. 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

The following steps show you how to analyze ATPG problems that appear as fault sites classified as untestable:

1. View the fault list by opening the Analyze dialog box and clicking the Faults tab. You can also use the `report_faults` command or write a fault list.
2. Select a specific fault class and fault location from the fault list.

3. Display the fault in the GSV using the Analyze dialog box, or use the `analyze_faults` command.
4. View the schematic and transcript to determine the cause of the problem.

The following examples demonstrate the process of analyzing ATPG problems:

- [Analyzing an AN Fault](#)
- [Analyzing a UB Fault](#)
- [Analyzing a NO Fault](#)

Analyzing an AN Fault

This example shows how to perform an analysis on an AN (ATPG untestable–not detected) fault identified as follows:

```
/amd2910/stack/U948/D1
```

The following example shows a transcript of an `analyze_faults` command for this fault. TestMAX ATPG 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.

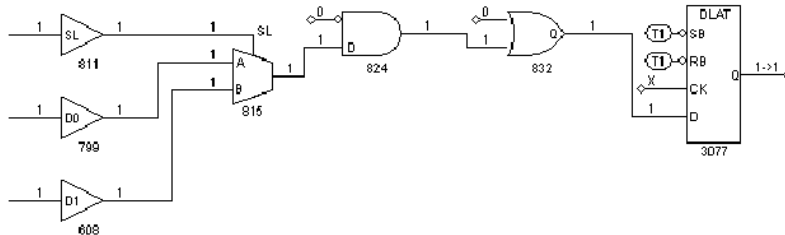
Transcript of `analyze_faults` Results for an AN Fault

```
TEST-T> 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(DIAT) test generation was unsuccessful due to
atpg_untestable.
```

The following figure shows the GSV schematic display of the untestable fault location. From the schematic and the messages in the preceding example, you can make the following conclusions:

- 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 49 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 in [“Checking Controllability and Observability.”](#)

Analyzing a UB Fault

This example shows how to analyze a UB (undetectable-blocked) 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.

TestMAX ATPG analyzes the fault selected and displays in the transcript window the equivalent analyze_faults command and the results of the analysis, as shown in the following example.

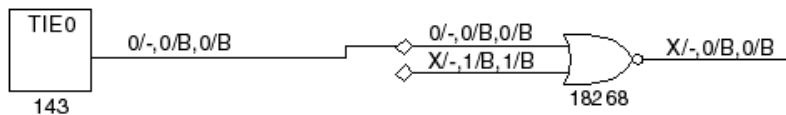
Transcript of analyze_faults Results for a UB Fault

```
TEST-T> 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).
```

The following figure shows the graphical representation of the section of the design associated with the fault.

Figure 50 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.”](#)

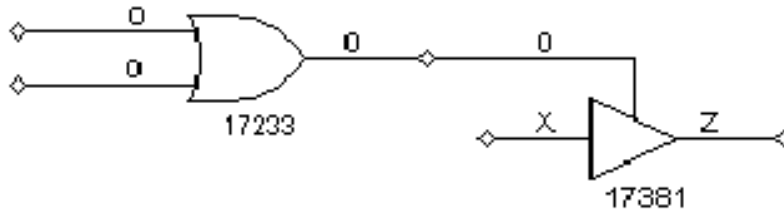
In the example in the preceding figure, TestMAX ATPG has analyzed a stuck-at-0 fault on the H02 pin in the schematic. The transcript shows that this fault is UB, that the blockage point is gate 18268, and that the source of the blockage is gate 143.

You review the GSV display in the preceding figure 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.

Analyzing a NO Fault

The following figure shows the schematic for an NO (not-observed) fault class.

Figure 51 GSV Display: A NO Fault



The fault report in the following example 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.

analyze_faults Report for a NO Fault

```
TEST-T> 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 a Schematic to a File

You can use the `gsv_print` command to create a grayscale PostScript file, which captures the schematic displayed in the graphical schematic viewer (GSV):

```
gsv_print -file file -banner string Y
```

You can add the `gsv_print` command to your TestMAX ATPG scripts and automatically capture schematic output. The computing host must have PostScript drivers installed (usually with `lpr/lp`). You can enclose the arguments in double quotation marks (" ").

7

Using the Hierarchy Browser

The Hierarchy Browser displays a design's basic hierarchy and enables graphical analysis of coverage issues. It is launched as a standalone window that sits on top of the TestMAX ATPG GUI main window.

Before you start using the Hierarchy Browser, you should familiarize yourself with the graphical schematic viewer (GSV). See [Using the Graphical Schematic Viewer](#) for more information.

The Hierarchy Browser does not display layout data. It is intended to supplement the GSV so you can analyze graphical test coverage information while browsing through a design's hierarchy.

The following topics describe how to use the Hierarchy Browser:

- [Launching the Hierarchy Browser](#)
- [Basic Components of the Hierarchy Browser](#)
- [Performing Fault Coverage Analysis](#)
- [Exiting the Hierarchy Browser](#)

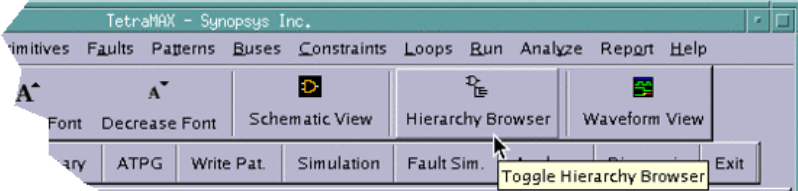
Launching the Hierarchy Browser

To launch the Hierarchy Browser, you first need to begin the ATPG flow and start the TestMAX ATPG GUI, as described in the following steps:

1. Follow the initial test pattern generation steps described in [ATPG Design Flow](#).
2. Launch the TestMAX ATPG GUI. For details, see [Controlling TestMAX ATPG Processes](#).
3. After the DRC process is completed, start the Hierarchy Browser by clicking the Hierarchy Browser button in the TestMAX ATPG GUI, as shown in the following figure.

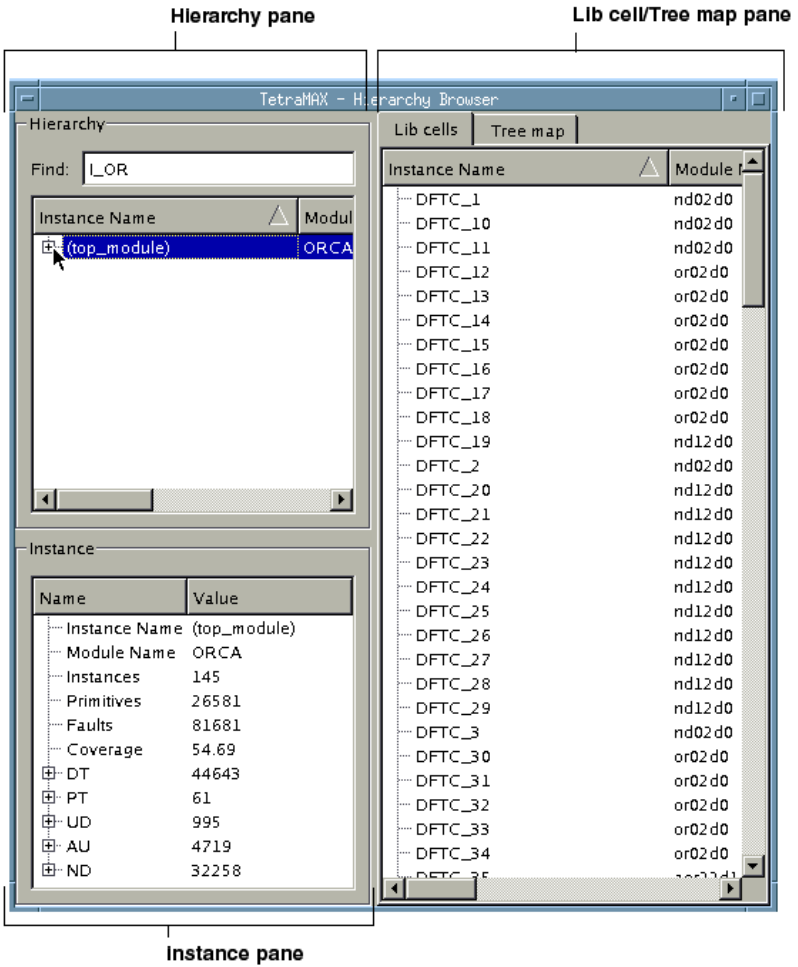
Chapter 7: Using the Hierarchy Browser
Launching the Hierarchy Browser

Figure 52 Hierarchy Browser Button in the TestMAX ATPG GUI



The Hierarchy Browser appears as a new window, as shown in the following figure.

Figure 53 Initial Display of the Hierarchy Browser



See Also

- [Exiting the Hierarchy Browser](#)

Basic Components of the Hierarchy Browser

The Hierarchy Browser is comprised of the following main components:

- *Hierarchy Pane* — Located in the top left portion of the browser, this area displays an expandable view of the design hierarchy and test coverage data.
- *Instance Pane* — Located in the bottom left portion of the browser, this area displays test coverage data associated with the module selected in the Hierarchy pane.
- *Lib Cell/Tree Map Pane* — Located in the right portion of the browser, this area toggles between library cell data and a graphical display of all submodules associated with the selected instance in the Hierarchy pane.

Using the Hierarchy Pane

The Hierarchy pane displays the overall design hierarchy, including the number of instances, the test coverage, the number of faults, and the main fault types. It also controls the data displayed in the Instance pane and Lib cell/Tree map pane.

Using the Hierarchy pane, you can expand and collapse the hierarchical display of a design's submodules and view the associated test coverage information.

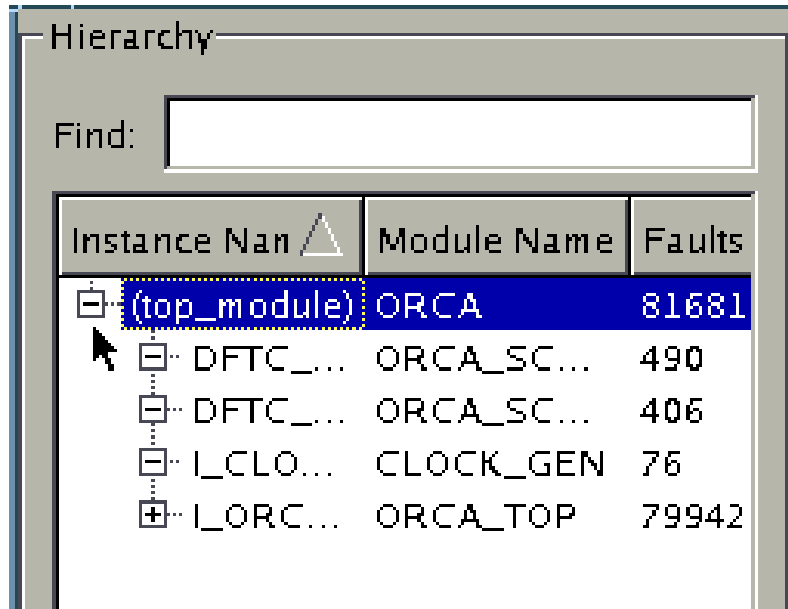
The following steps describe how to display coverage data in the Hierarchy pane:

1. Click the

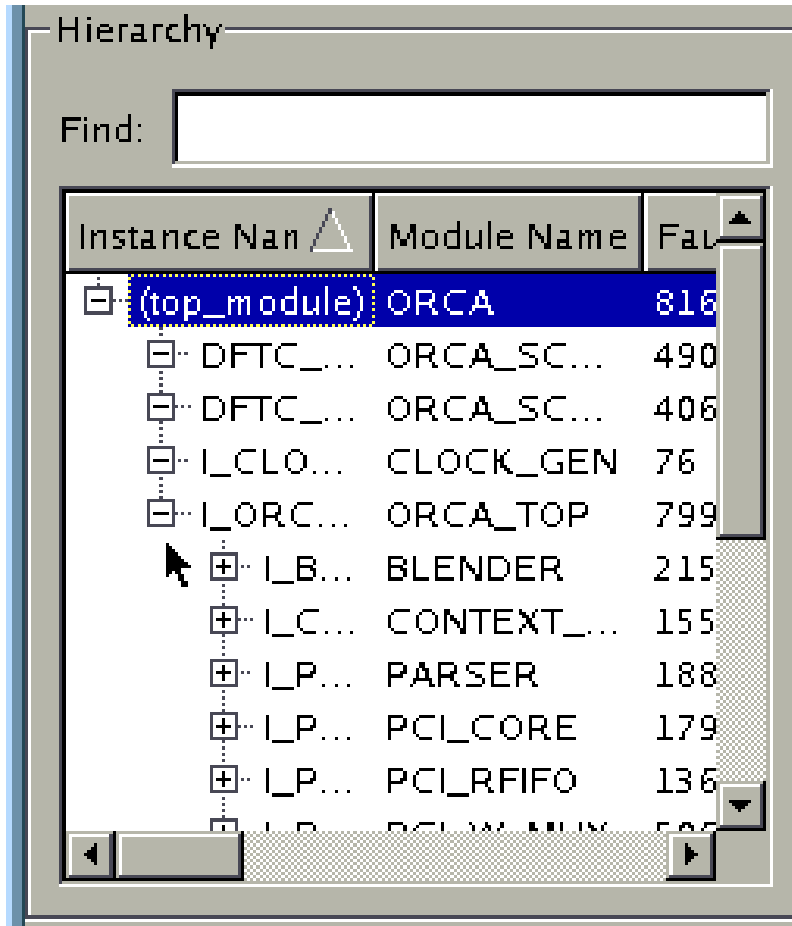
+

symbol next to the top instance name.

The design hierarchy expands, and the submodules and related test coverage information for the top instance name are displayed.



2. Continue to expand the hierarchy, as needed.



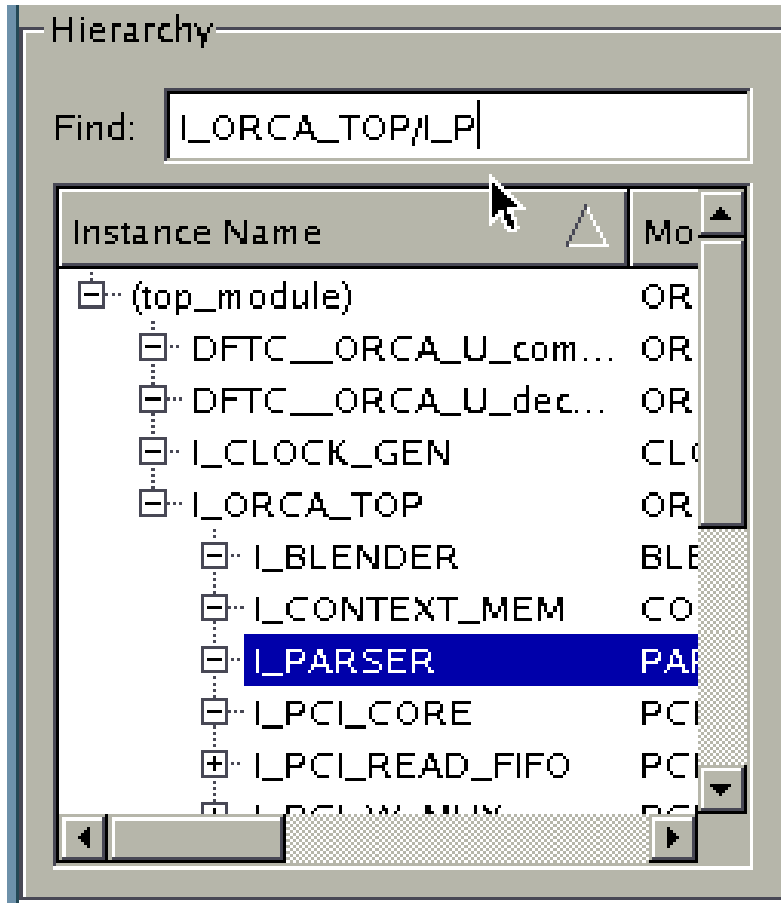
3. Move the slider bar, located at the bottom of the pane, to view coverage details and fault information associated with the instance names in the left column.

Hierarchy

Find:

Coverage	DT	PT	UD	AU
54.69	44643	61	995	4719
93.88	460	0	0	0
45.32	184	0	0	0
30.26	20	6	0	48
54.21	43309	55	990	4647
33.73	7276	0	5	798
8.51	132	0	342	778
50.74	956	0	0	0
73.87	13259	0	0	604
52.35	712	0	0	32
0.00	0	0	0	0

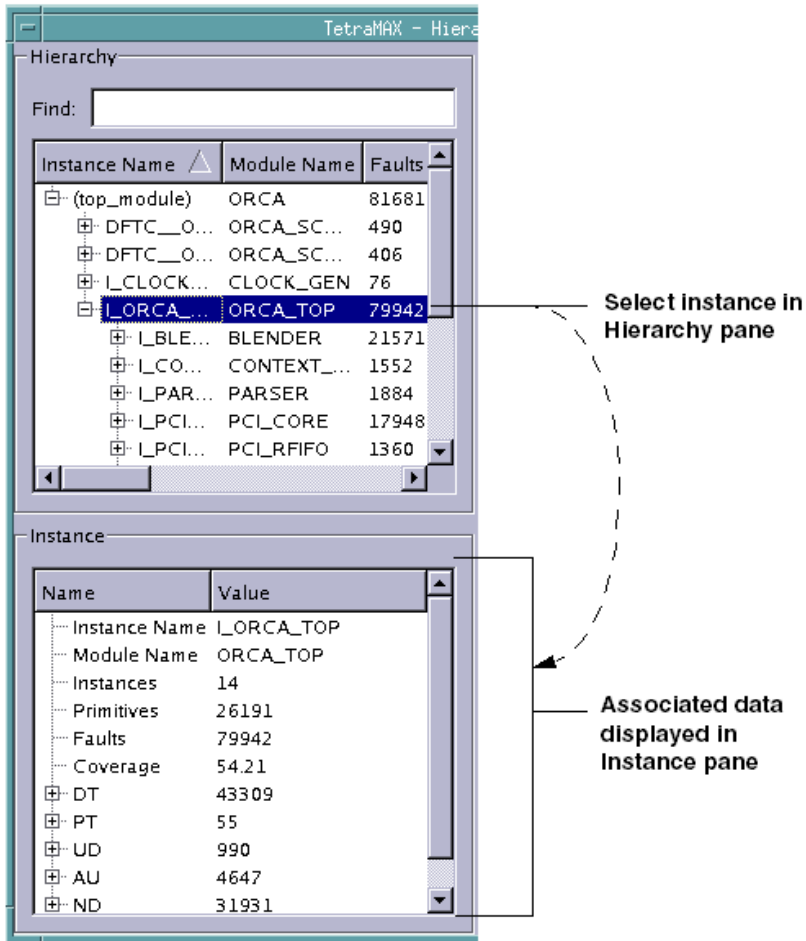
4. Find data for a particular instance using the Find text field.



Viewing Data in the Instance Pane

The Instance pane displays coverage data for the instance selected in the Hierarchy pane, as shown in the following figure.

Figure 54 Displaying Information in the Instance Pane

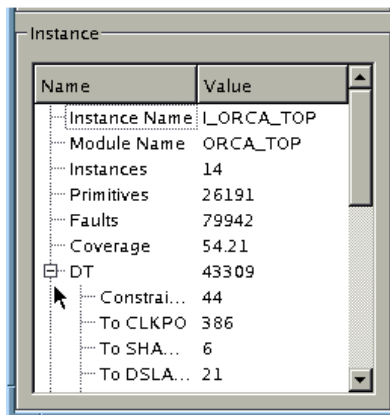


You can expand the display of information for a fault type in the Instance pane by clicking the

+

symbol next to a fault class, as shown in the following figure.

Figure 55 Expanding the Display of Data for the DT Fault Class



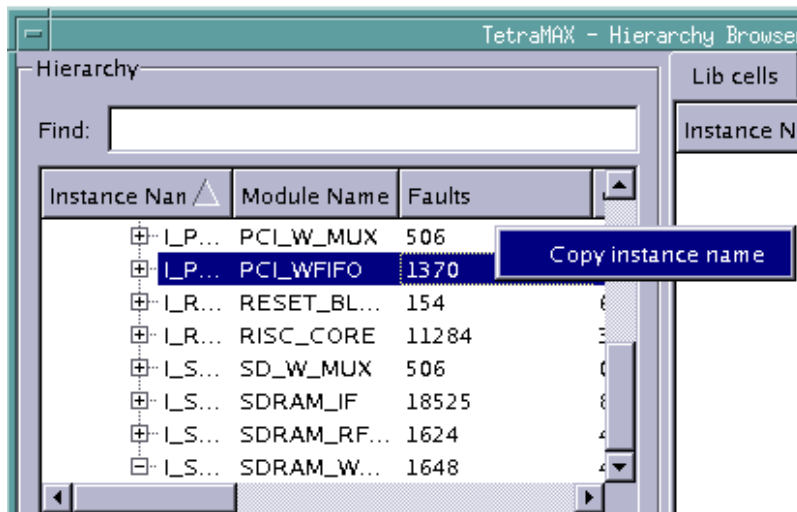
Copying an Instance Name

You can copy the full instance name from anywhere in the Hierarchy Browser and paste it in the Find text field, or use it for reference purposes in other applications.

To copy an instance name:

- Right-click on an instance name anywhere in the Hierarchy Browser, and select Copy Instance Name.

Figure 56 Copying An Instance Name



Viewing Data in the Lib Cells/Tree Map Pane

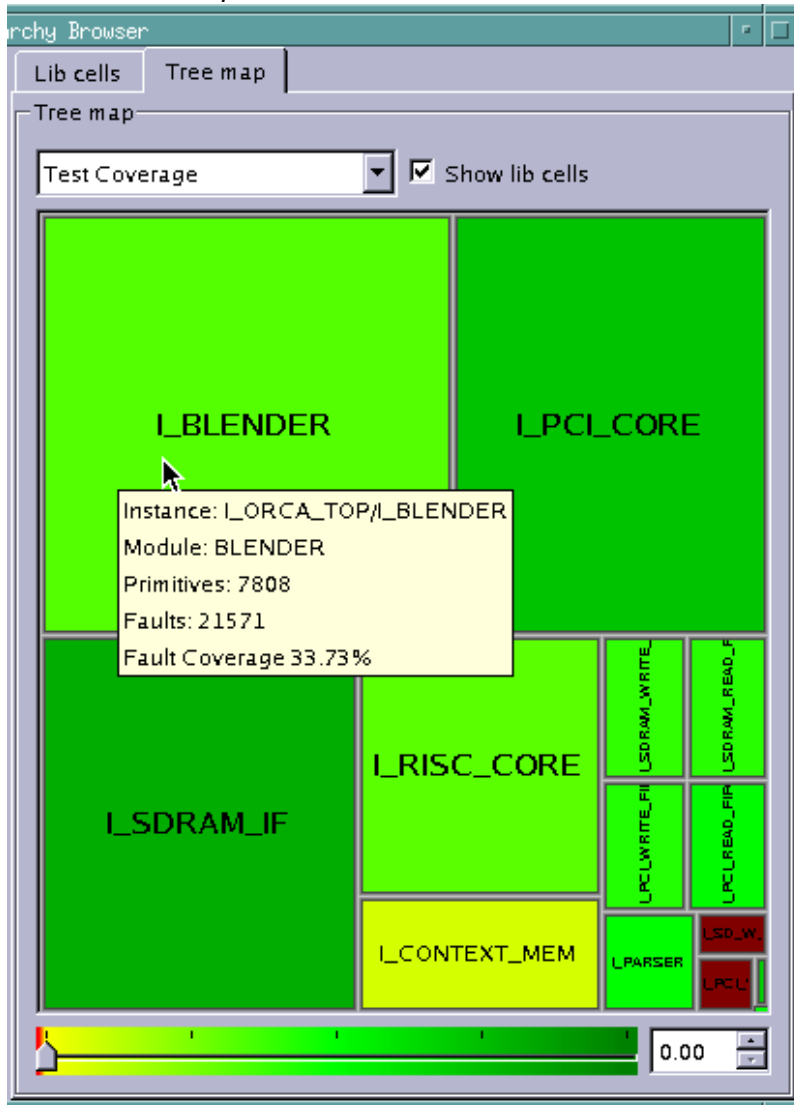
The Lib cells/Tree map pane toggles between two tabs:

- *Lib cells* — Displays module names, primitives, faults, test coverage, and fault class data for library cells, which include all non-hierarchical cells in a selected module. An example is shown in [Figure 57](#).
- *Tree map* — Displays a design’s hierarchical graphical test coverage. An example is shown in [Figure 58](#).

Figure 57 Display of Library Cell Information

Instance Name	Module Name	Primitives	Faults	Coverage	DT	PT	UD	AU	ND
count_int_reg_0_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_0_1	sdcrq1	2	12	66.67	8	0	0	0	4
count_int_reg_1_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_1_1	sdcrq1	2	12	66.67	8	0	0	0	4
count_int_reg_2_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_2_1	sdcrq1	2	12	66.67	8	0	0	0	4
count_int_reg_3_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_3_1	sdcrq1	2	12	66.67	8	0	0	0	4
count_int_reg_4_0	sdcrq1	2	12	100.00	12	0	0	0	0
count_int_reg_4_1	sdcrq1	2	12	91.67	11	0	0	0	1
count_int_reg_5_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_5_1	sdcrq1	2	12	91.67	11	0	0	0	1
count_int_reg_6_0	sdcrq1	2	12	75.00	9	0	0	0	3
count_int_reg_6_1	sdcrq1	2	12	91.67	11	0	0	0	1
DFTC_1	aor22d1	3	10	50.00	5	0	0	4	1
empty_int_reg1	sdcrq1	2	12	91.67	11	0	0	0	1
full_int_reg	sdcrq1	2	12	66.67	8	0	0	0	4
LOCKUP	lanlq1	2	6	100.00	6	0	0	0	0
SD_WFIFO_RAM	ram32x64	265	94	0.00	0	0	0	94	0
sync_reg_0_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_0_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_1_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_1_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_2_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_2_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_3_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_3_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_4_0	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_4_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_5_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_5_1	sdcrq1	2	12	66.67	8	0	0	0	4
sync_reg_6_0	sdcrq1	2	12	75.00	9	0	0	0	3
sync_reg_6_1	sdcrq1	2	12	66.67	8	0	0	0	4

Figure 58 Tree Map View



As shown in preceding figure, the data displayed in the Tree map is color-coded according to the test coverage. Dark green indicates the maximum coverage, light green is slightly lower coverage, yellow is minimal coverage, and dark red is coverage below the minimum threshold.

When you hold your pointer over a particular instance, a pop-up window will display detailed coverage information for that instance.

Additional details on using the Tree map are provided in the following section, [Performing Fault Coverage Analysis](#).

Performing Fault Coverage Analysis

You can access and adjust the display of a variety of interactive test coverage data in the Hierarchical Browser. The following sections show you how to display various types of data that helps you perform fault coverage analysis:

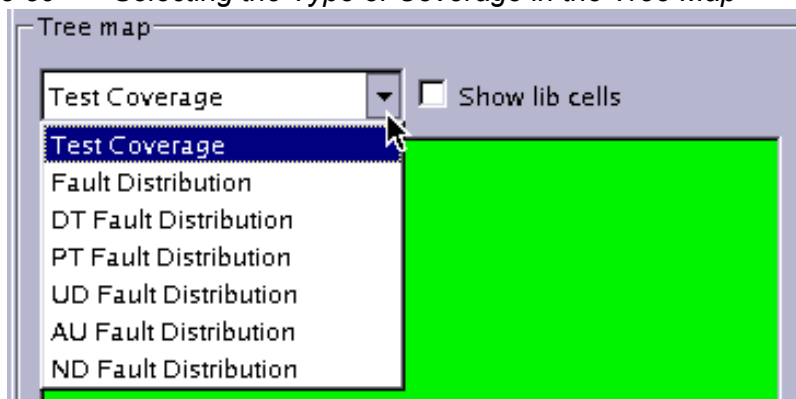
- [Understanding the Types of Coverage Data](#)
- [Expanding the Design Hierarchy](#)
- [Viewing Library Cell Data](#)
- [Adjusting the Threshold Slider Bar](#)
- [Identifying Fault Causes](#)
- [Displaying Instance Information in the GSV](#)

Understanding the Types of Coverage Data

You can view coverage data in the Tree map based on the overall test coverage, the fault distribution, or the fault class distribution. Fault classes include the DT (detected), PT (possibly detected), UD (undetectable), AU (ATPG untestable), and ND (not detected) classes.

As shown in the following figure, you use the drop-down menu to select the type of coverage data you want to display.

Figure 59 Selecting the Type of Coverage in the Tree Map



The formula to calculate the Test Coverage displayed in the Hierarchy Browser is as follows:

$$\langle \text{displayed area} \rangle = (DT + PT_CREDIT * PT) / \text{Faults}$$

The formulas to calculate the various categories of coverage data provided by the Hierarchy Browser are as follows:

- *Fault Distribution:* <displayed area> = <number of (DT+PT+AU+ND) faults >
- *DT Fault Distribution:* <displayed area> = <number of DT faults>
- *PT Fault Distribution:* <displayed area> = <number of PT faults>
- *UD Fault Distribution:* <displayed area> = <number of UD faults>
- *AU Fault Distribution:* <displayed area> = <number of AU faults>
- *ND Fault Distribution:* <displayed area> = <number of ND faults>

See Also

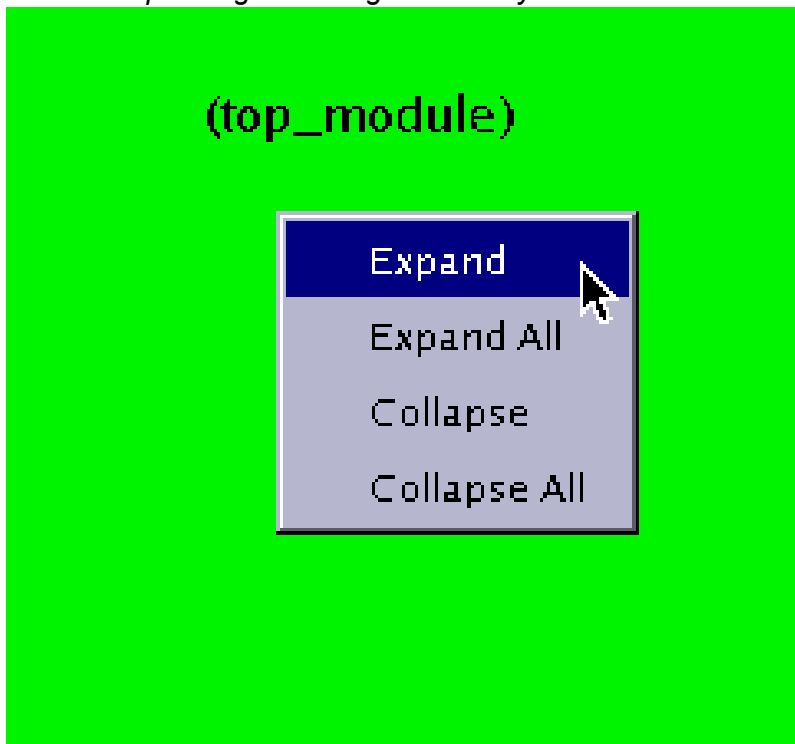
- [Fault Categories and Classes](#)

Expanding the Design Hierarchy

When the Hierarchy Browser is initially invoked, the Tree map displays only the top-level instance in the design. The following steps show you how to expand the display of the design hierarchy:

1. Right-click your mouse in the Tree map, then select Expand to expand the display of one level of the design hierarchy.

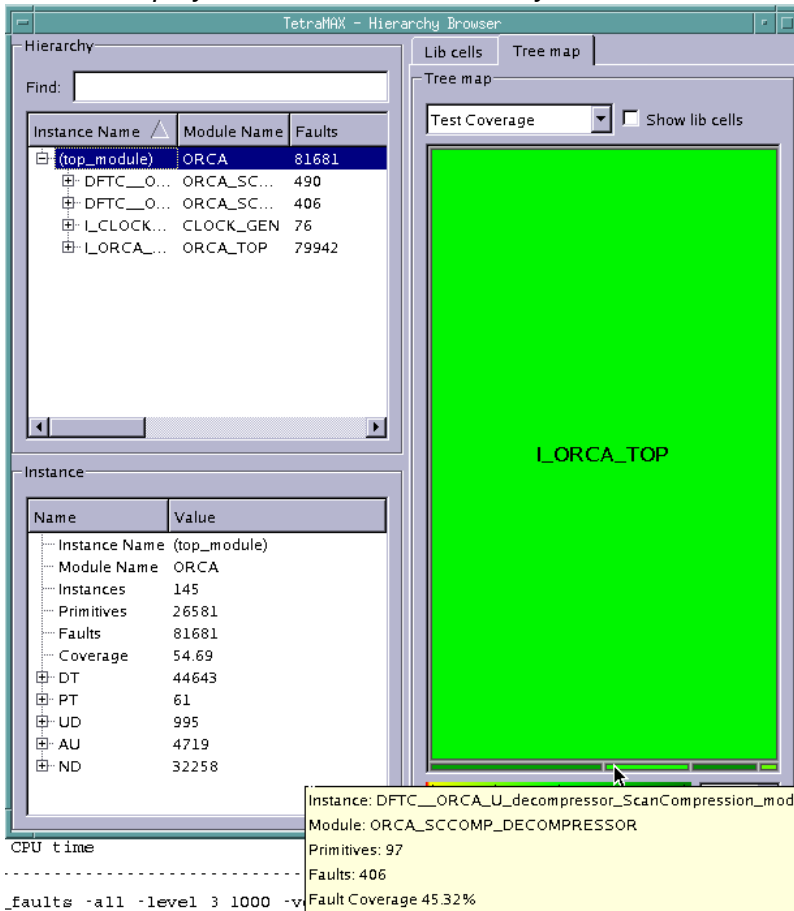
Figure 60 Expanding the Design Hierarchy



After selecting Expand, the next level of hierarchy is displayed in the Tree map, as shown in the following figure.

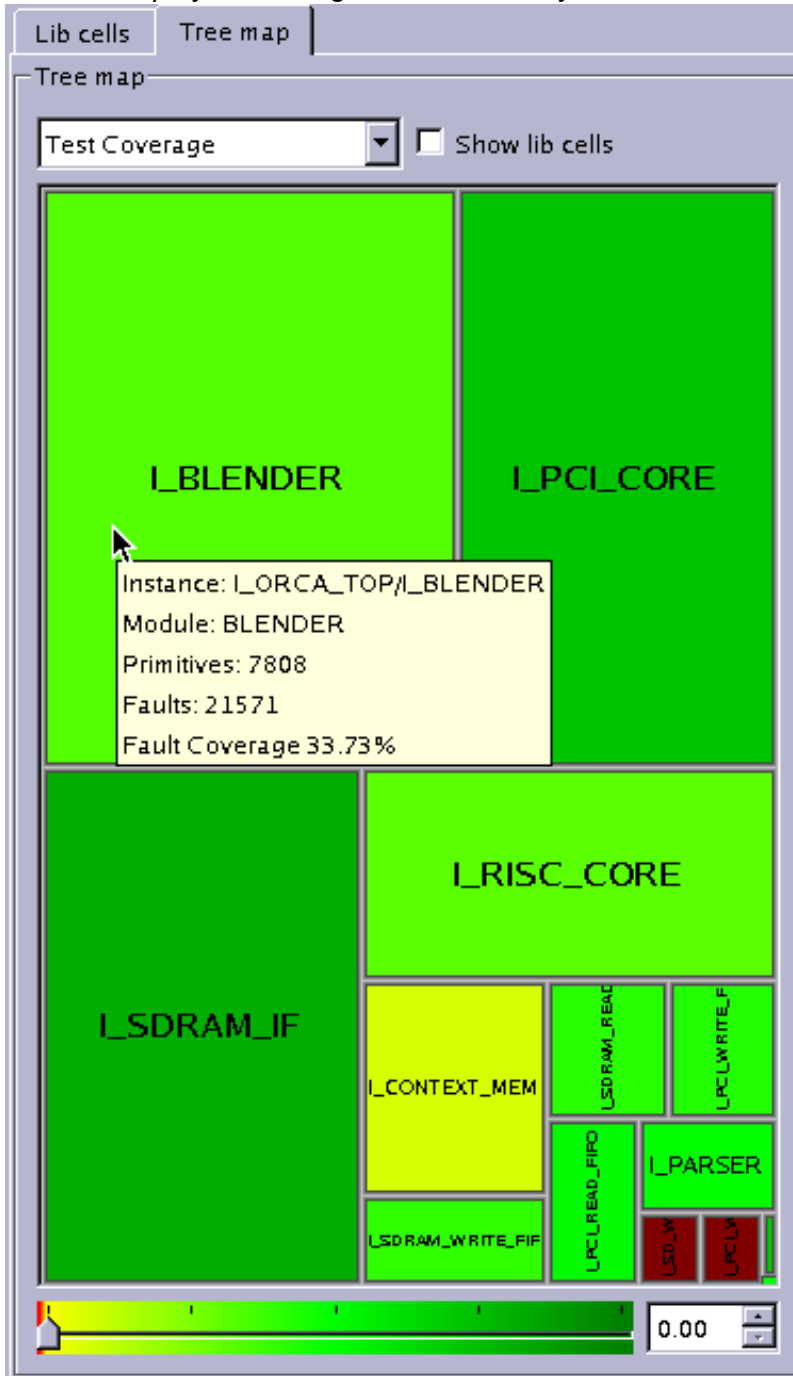
The Hierarchy Browser is not a layout viewer. The size of each graphically represented instance is based on the number of primitives for that instance in proportion to the other instances. In the following figure, the largest displayed instance is I_ORCA_TOP. Below that instance are four smaller instances. The pointer at the bottom of the window is highlighting the DFTC_ORCA_U instance. Also note that the data in the Hierarchy pane, in the upper left portion of the window, expands to coincide with the Tree map view.

Figure 61 Display of First Level of Hierarchy



2. You can continue to expand the design hierarchy one level at a time by right-clicking and selecting Expand, or by selecting Expand All to expand the entire design hierarchy. The following figure shows the full display of a design's hierarchy.

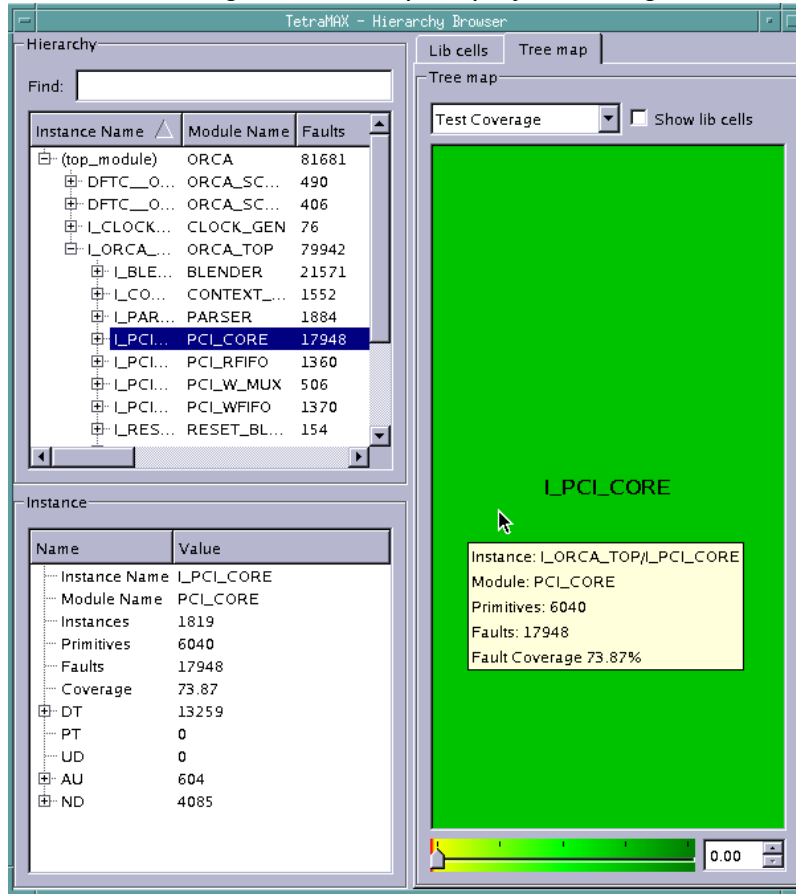
Figure 62 Display of a Design's Full Hierarchy



3. You can further focus the display of data for a particular instance by clicking on the instance in the Tree map or in the Hierarchy pane, as shown in the following figure.

Figure 4: Focusing the Tree Map Display on a Single Instance

Figure 63 Focusing the Tree Map Display on a Single Instance

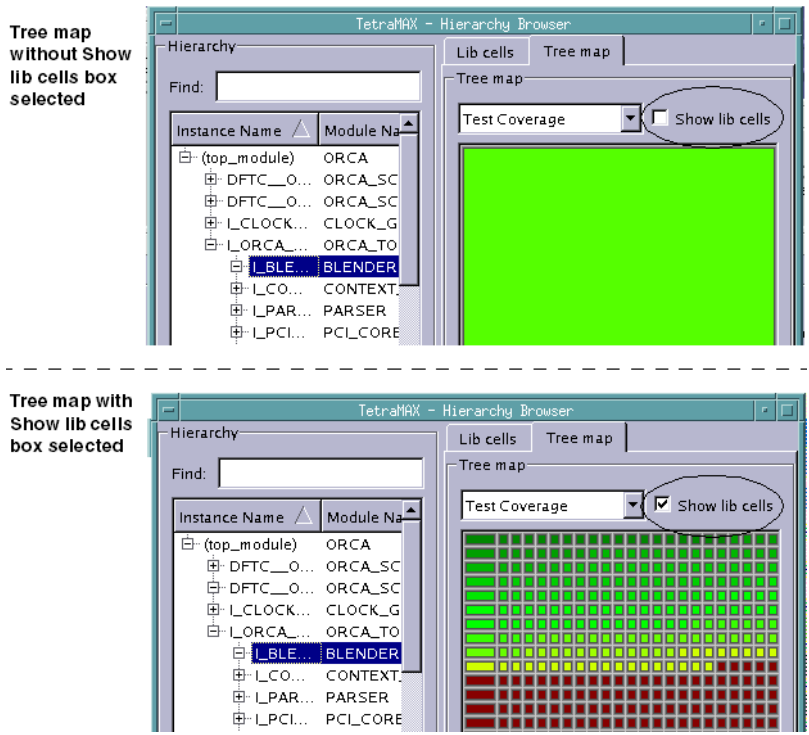


4. To collapse the display of the design hierarchy, right-click anywhere in the Tree map and select Collapse or Collapse All.

Viewing Library Cell Data

You can view a graphical representation of library cells associated with a particular instance by clicking the Show lib cells check box in the Tree map. Figure 1 shows how selecting this check box affects the display of data in an example instance.

Figure 64 How Selecting the Show Lib Cells Box Affects the Tree Map Display

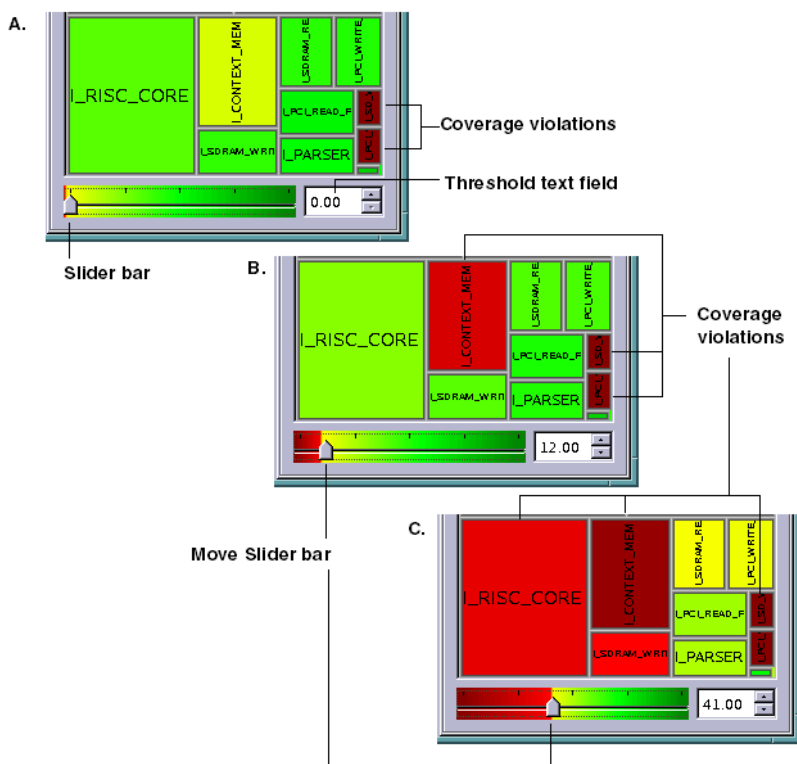


Adjusting the Threshold Slider Bar

The threshold slider bar is located at the bottom of the Tree map. You can use this bar to change the threshold for the color spectrum display of fault coverage. By default, the threshold is set to 0% coverage, which means that any instances with 0% coverage is displayed in red.

To change the threshold, either move the slider bar or enter a different value in the threshold text field. Figure 1 shows the comparative effect of moving the threshold slider bar.

Figure 65 Effect of Moving Threshold from A (0%) to B (12%) to C (41%)



Identifying Fault Causes

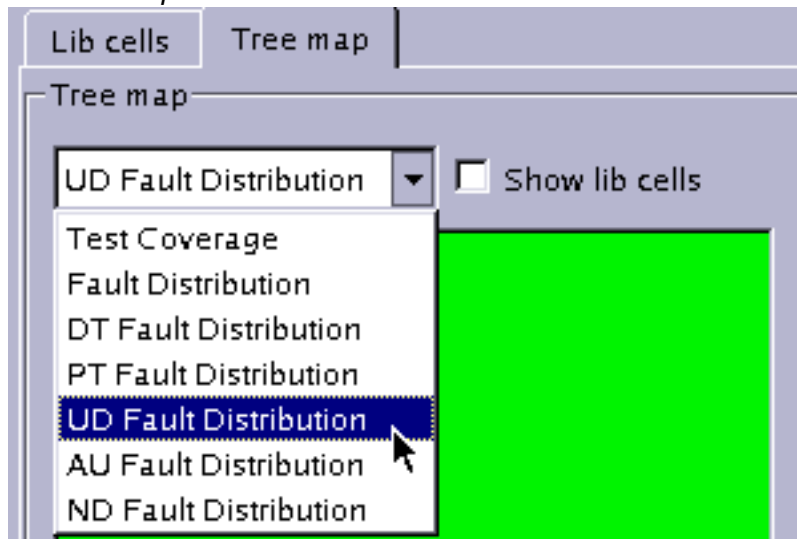
The Hierarchy Browser enables you to identify causes of various faults. The four basic fault causes are as follows:

- Constrain Values
- Constrain Value Blockage
- Connected to <value>
- Connected from <value>

The following steps show you how to identify fault causes for a specific fault class in a specific instance:

1. In the drop-down menu located the top of the Tree map, select the type of coverage you want to display. For example, select UD Fault Distribution.

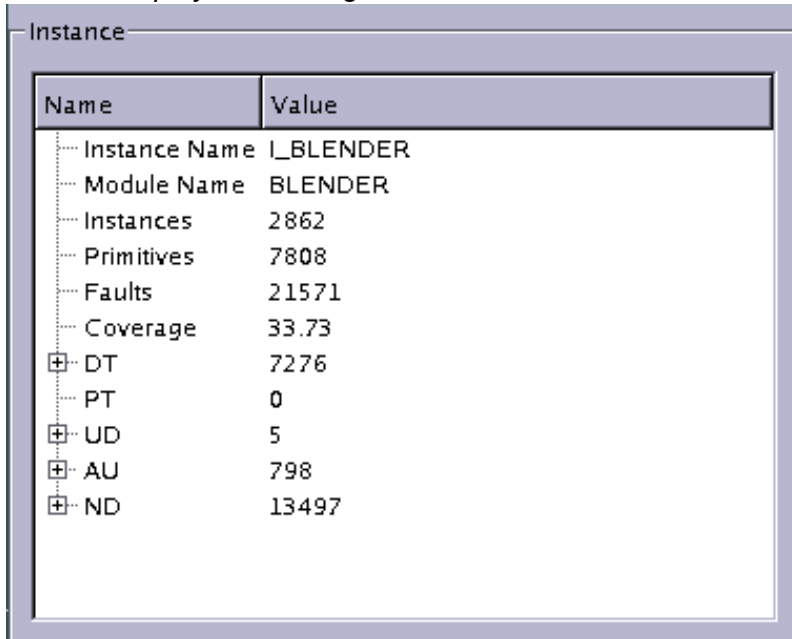
Figure 66 Drop-Down Menu



2. If required, click the Show lib cells check box to view all the cells in the instance.
3. Expand the display of the design's hierarchy, as needed.
4. Click an instance of interest in the Tree map or select an instance in the Hierarchy pane.

The Instance pane displays the name of the selected instance and its related coverage data, as shown in Figure 2.

Figure 67 Display of Coverage Information for Selected Instance in Instance Pane

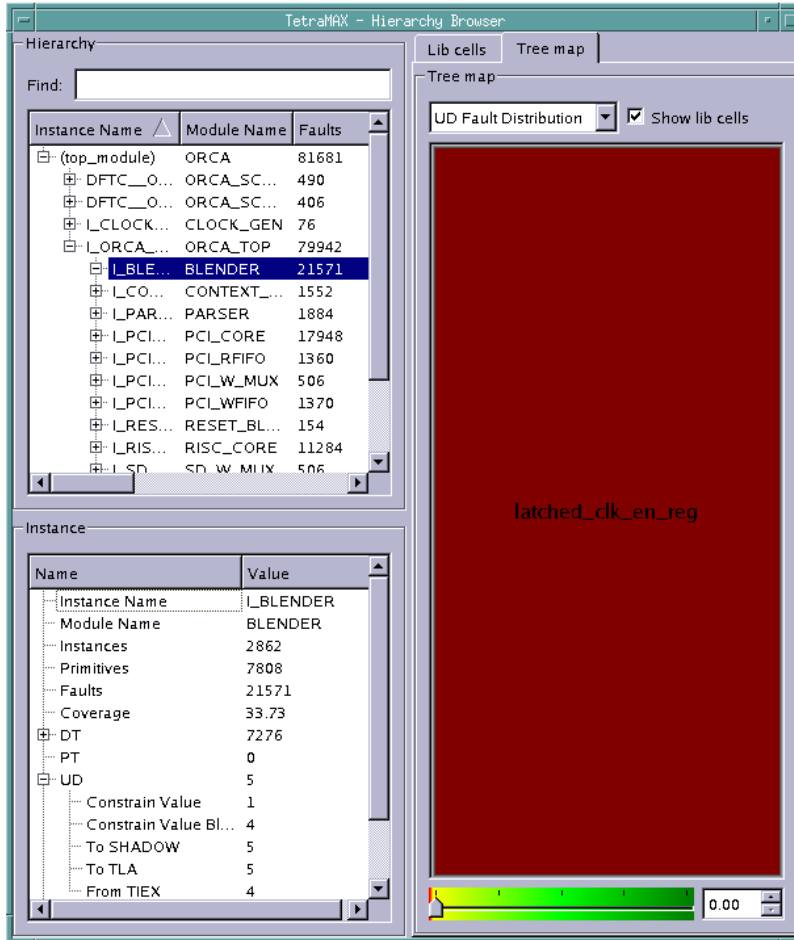


The screenshot shows a window titled "Instance" containing a table with two columns: "Name" and "Value". The table lists various metrics for an instance named "I_BLENDER". The metrics include Instance Name, Module Name, Instances, Primitives, Faults, Coverage, and several fault types (DT, PT, UD, AU, ND) with their respective counts.

Name	Value
Instance Name	I_BLENDER
Module Name	BLENDER
Instances	2862
Primitives	7808
Faults	21571
Coverage	33.73
DT	7276
PT	0
UD	5
AU	798
ND	13497

- Expand the display of the fault class of interest in the Instance pane. Figure 3 shows the expansion of the UD fault class and display of the related fault causes.

Figure 68 Display of Fault Class and Related Fault Causes



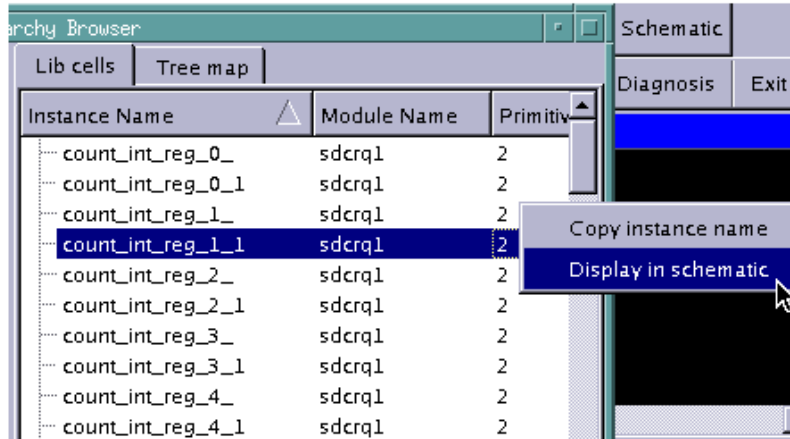
Displaying Instance Information in the GSV

You can select an instance name anywhere in the Hierarchy Browser and display it in the graphical schematic viewer (GSV).

To display a selected instance from the Hierarchy Browser in the GSV:

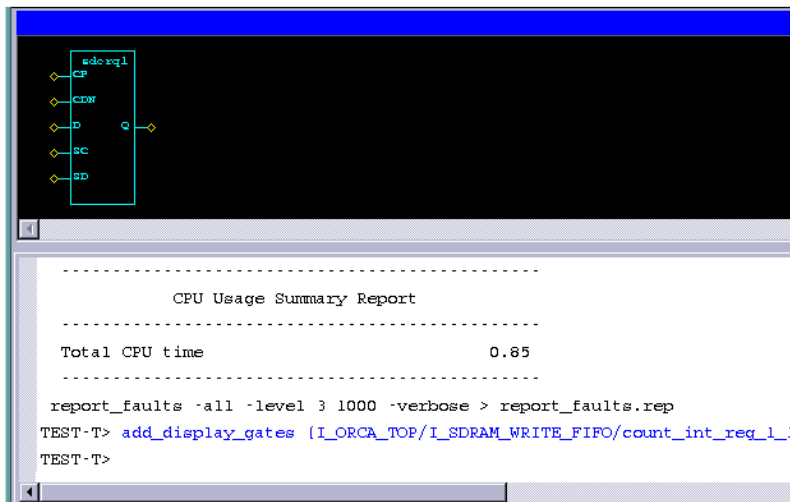
- Right-click an instance name in the Hierarchy Browser, and select Display in schematic, as shown in the following figure.

Figure 69 Selecting an Instance Name



The selected instance will display in the GSV, as shown in the following figure.

Figure 70 Display of Selected Instance in GSV

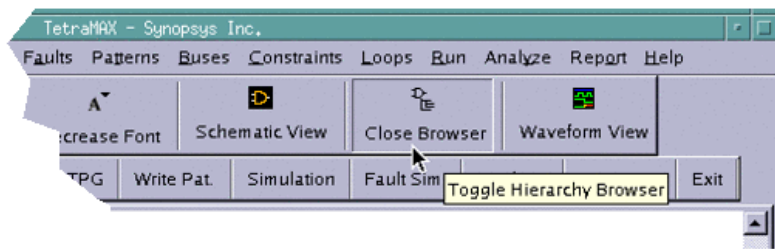


Exiting the Hierarchy Browser

To exit the Hierarchy Browser, click the Close Browser button in the TestMAX ATPG GUI.

Chapter 7: Using the Hierarchy Browser Exiting the Hierarchy Browser

Figure 71 Exiting the Hierarchy Browser



See Also

- [Launching the Hierarchy Browser](#)

8

Using the Simulation Waveform Viewer

You can use the TestMAX ATPG Simulation Waveform Viewer (SWV) to debug internal, external, and imported functional pattern mismatches by displaying the failing simulation values and TestMAX ATPG simulated values of the test_setup procedure.

The following topics describe how to use the SWV:

- [Getting Started With the SWV](#)
- [Understanding the SWV Color Codes](#)
- [Supported Pin Data Types and Definitions](#)
- [Invoking the SWV](#)
- [Using the SWV Interface](#)

Getting Started With the SWV

Before you start using the Simulation Waveform Viewer (SWV), you should familiarize yourself with the graphical schematic viewer (GSV). For more information, see [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 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:

- Create or parse the STL procedure file, edit the STL procedure file, and rerun DRC
- Use the GSV to identify and resolve shift errors, and also to view test_setup
- 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

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:

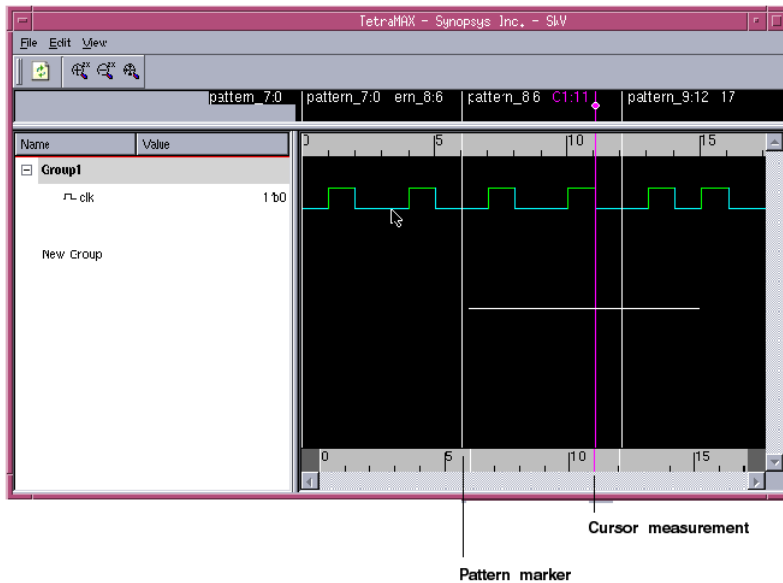
```
set_environment viewer -max_pindata_length d
```

Understanding the SWV Color Codes

The SWV uses the following color codes:

- Red — Insertion line
- Pink — Cursor measurement
- White — Pattern marker
- Green — Load signal
- Yellow — Capture signal

Figure 1: SWV Colors



Supported Pin Data Types and Definitions

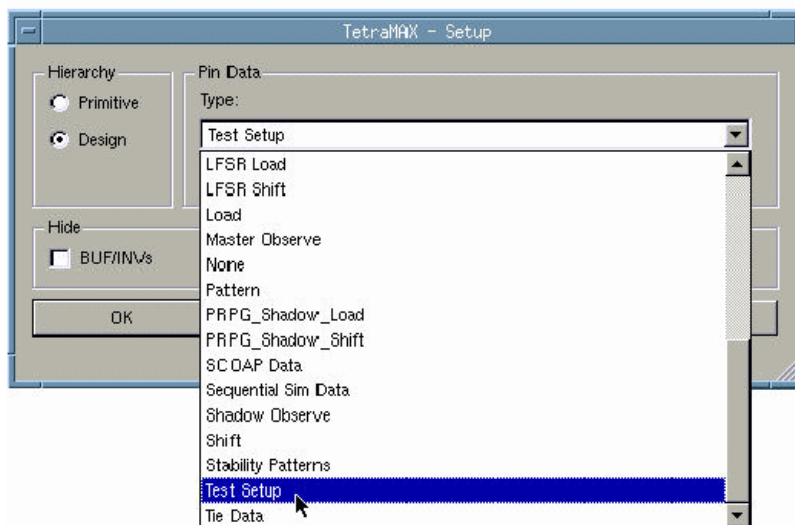
The following pin data types are supported by the Simulation Waveform Viewer (SWV):

- *Test Setup* (test_setup) — Displays simulated values for the test_setup macro displaying debugging problems in a STIL test_setup macro.
- *Debug Sim Data* (debug_sim_data) — Displays imported external simulator values used for debugging golden simulation vector mismatches
- *Sequential Sim Data* (seq_sim_data) — Displays currently stored sequential simulation data used for displaying results of sequential fault simulation (for advanced users of fault simulation)

Note that the SWV does not support all pin data types upon initialization; it supports only test_setup, debug_sim_data, and sequential_sim_data. Several other pin data types are supported after starting the SWV in one of the initial three pin data types. You can choose test_setup after SWV is opened, and then change to any pin data type, such as shift.

The following figure shows the TestMAX ATPG pin data type setup menu.

Figure 72 Setting the Pin Data Type



Two of the pin data types require data to be stored internally in TestMAX ATPG.

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 the DRC setting using the `set_drc` command, then rerun the DRC analysis as follows:

```
TEST-T> drc
DRC-T> set_drc -store_setup
DRC-T> run_drc
```

The `test_setup` pin data type requires the `set_drc -store` command.

Sequential simulation data typically comes 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 `seq_sim_data`, as shown in the following example:

```
TEST-T> set_simulation -data 85 89
TEST-T> run_simulation -sequential
```

The `seq_sim_data` pin data type requires the output of the `set_simulation -data` command.

See Also

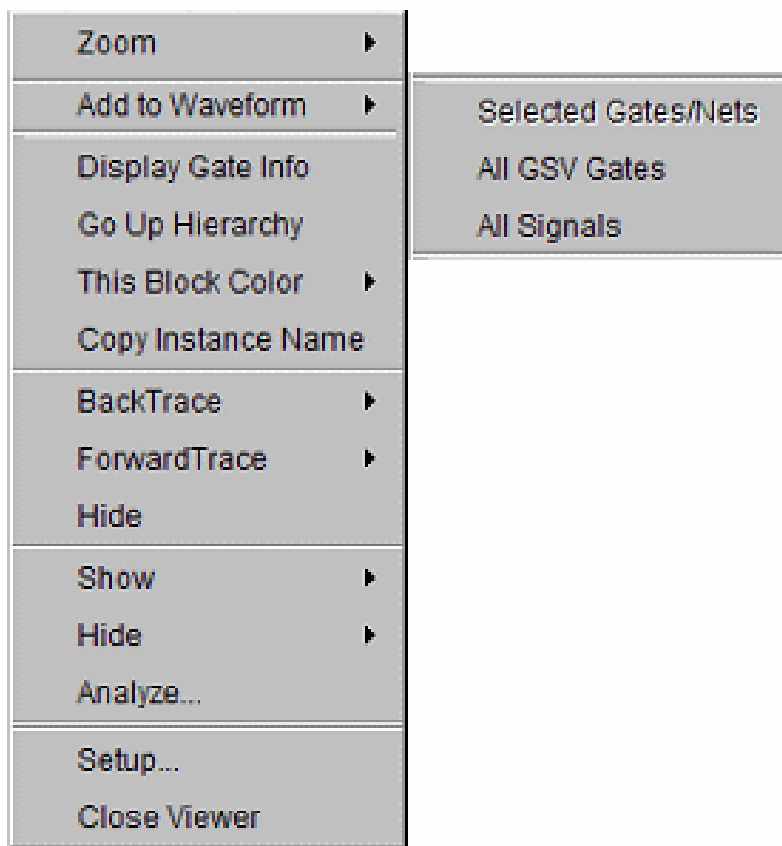
- [Defining the test_setup Macro](#)

Invoking the SWV

You can specify commands, select buttons, or use your right mouse button to open menus that cause TestMAX ATPG to launch the SWV either directly from the GSV or without the GSV.

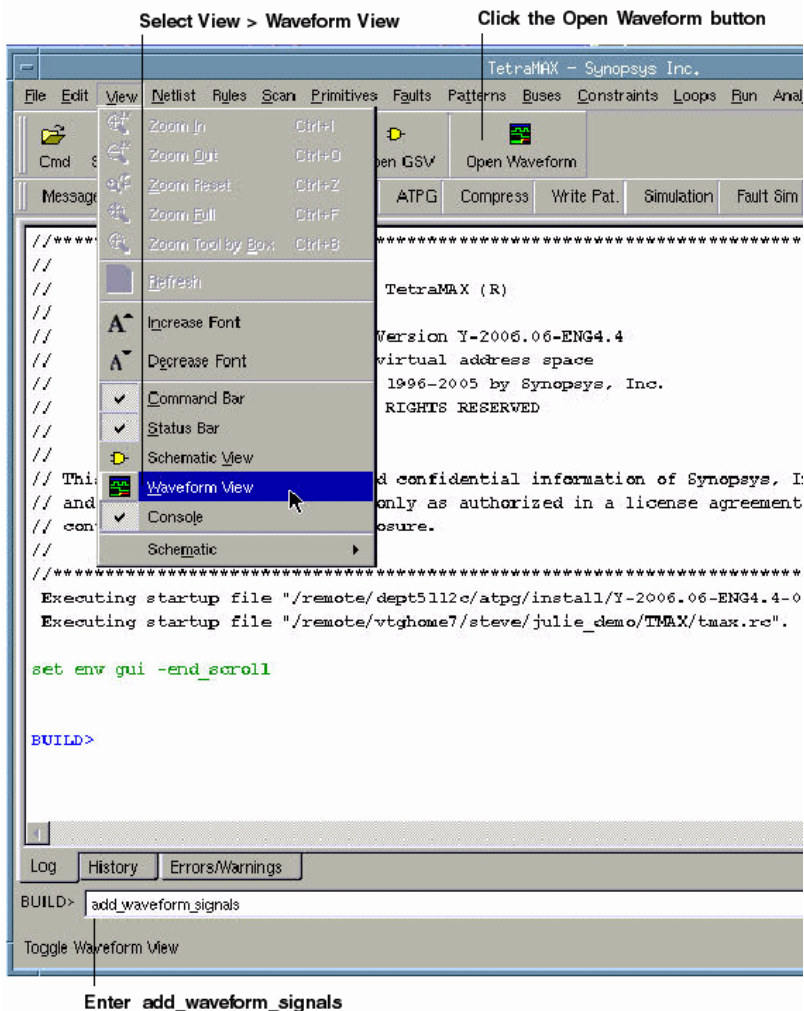
The following figure shows the SWV menu that appears when you right-click after selecting the nets and or gates. You can add signals, gates, and nets to the waveform using this menu.

Figure 73 Opening the SWV Using Your Right Mouse Button



The following figure shows the three ways to invoke the SWV from the GSV.

Figure 74 Three Ways to Open the SWV



Using the SWV Interface

The following topics describe the basic features of the SWV interface:

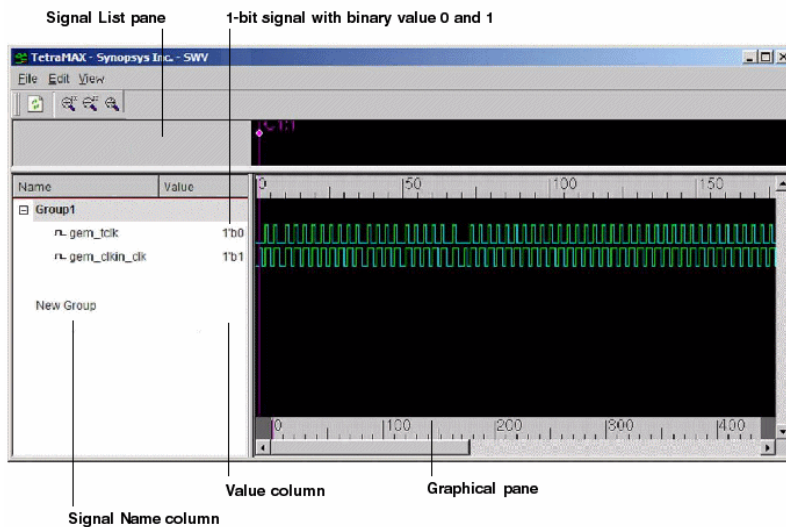
- [Understanding the SWV Layout](#)
- [Manipulating Signals](#)
- [Identifying Signal Types in the Graphical Pane](#)
- [Using the Time Scales](#)
- [Using the Marker Header Area](#)

- [Using the SWV With the GSV](#)
- [Using the SWV Without the GSV](#)
- [SWV Inputs and Outputs](#)
- [Analyzing Violations](#)

Understanding the SWV Layout

The layout of the SWV is shown in Figure 1.

Figure 75 SWV Layout



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.

Refreshing the View

To refresh the view (similar to the GSV), click the Refresh button or select Edit > Refresh View.

Manipulating Signals

The following sections show you how to manipulate signals:

- [Using the Signal List Pane](#)
- [Adding Signals](#)
- [Deleting Signals](#)
- [Inserting Signals](#)

Using the Signal List Pane

You can manipulate signals using the Signal List pane, which is located on the left side of the SWV. This pane 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 cannot edit or make any changes to the design.

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.

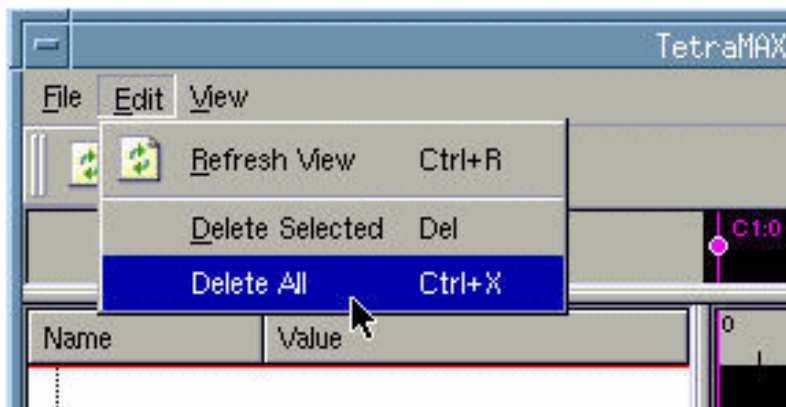
To add a signal, middle-click the signal from the waveform to select it. The red insertion line appears around the signal. Then drag it to the required group.

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 1 shows the Edit menu.

Figure 76 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 77 shows an empty waveform table, and Figure 78 illustrates signal insertion.

Figure 77 Empty Waveform Table

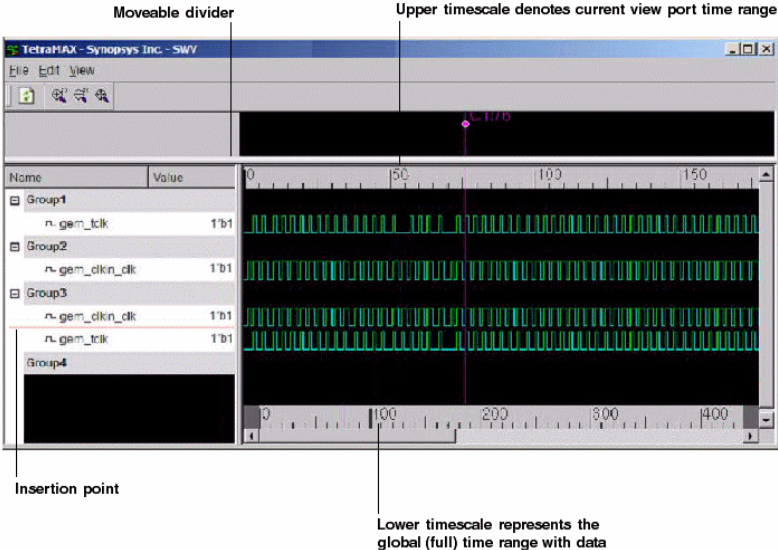
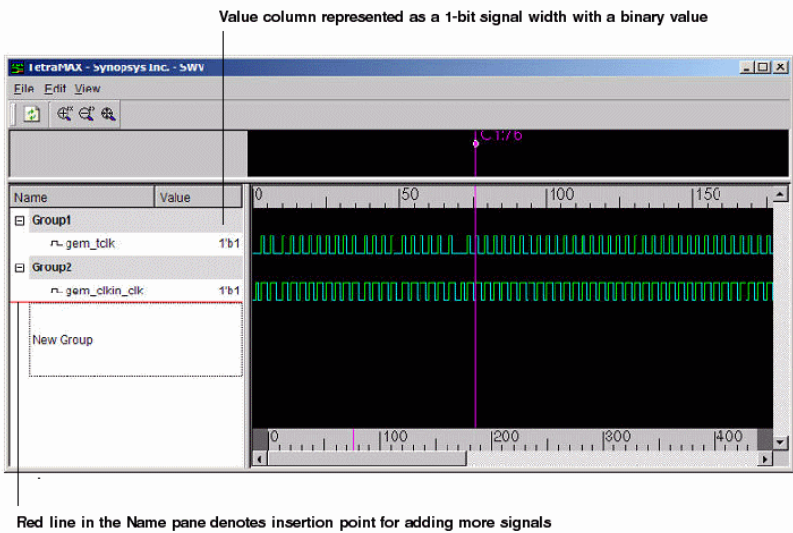


Figure 78 Inserting Signals



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

The SWV displays two types of time scales:

- *Upper Time Scale*

This 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*

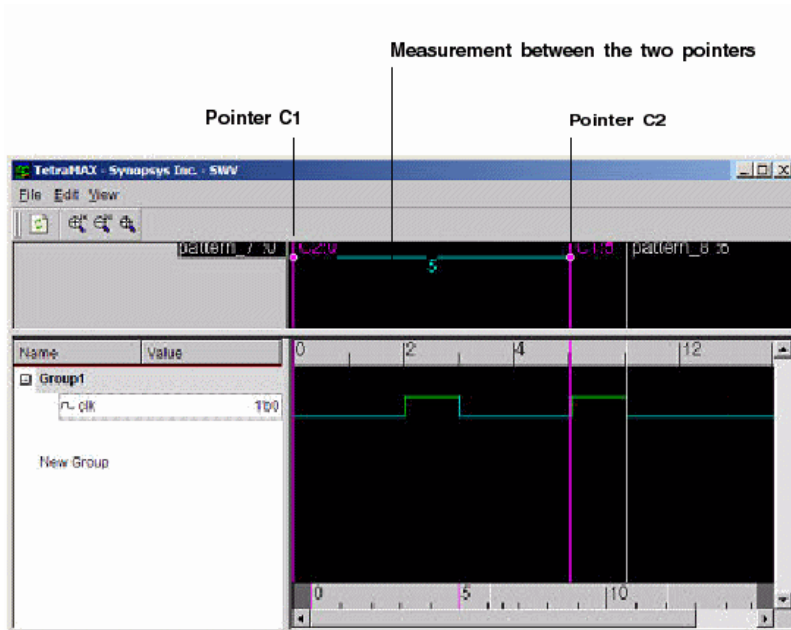
This area shows the full time range the data occupies. You can control zoom operation using your left mouse button, which causes an adjustment in the current view time range. The width of the scroll thumb in the horizontal scroll bar shows the approximate view area in proportion to the full data time 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 pointers: 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 measure between events (as shown in the following figure).

Figure 79 Reference Pointers



The following sections show you how to use the marker header area:

- [Adding and Deleting Pointers](#)
- [Moving a Marker Pointer](#)
- [Measuring Between Two 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.

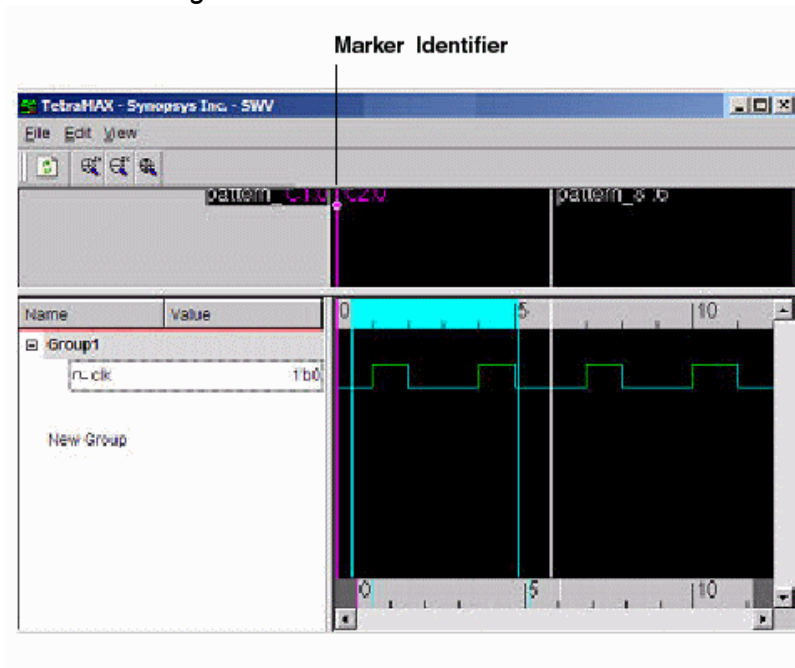
You can delete all markers by first selecting the markers, and then choosing the Delete Selected command (or the Delete This Marker command if you selected only one marker).

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 2).
- Drag the left marker, then click to release it.

Figure 80 Moving a Marker Cursor



Measuring Between Two Pointers

As shown in [Figure 79](#), 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 primary component of the TestMAX ATPG GUI is the graphical schematic viewer (GSV), which displays annotated simulation values during DRC (for details see [Using the Graphical Schematic Viewer](#)). You can expand the GSV to ease DRC debugging. Patterns are displayed in a logic cone view: that is, logic from each design derived by tracing back from a pair of matched points. Logic cones appear when there are DRC warnings and error messages.

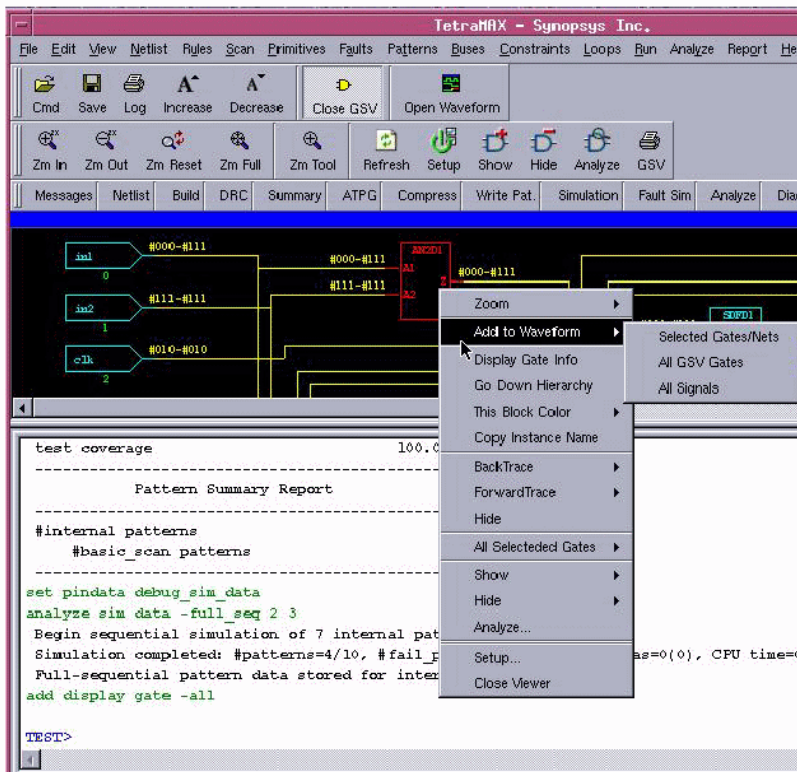
The following steps show you how to launch the SWV window from a selected logic cone view:

1. Select View > Waveform View > Setup.
2. Select "Pin Data Type" as "Test Setup."
3. Click your right mouse button and select 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 81 Using the SWV With the GSV

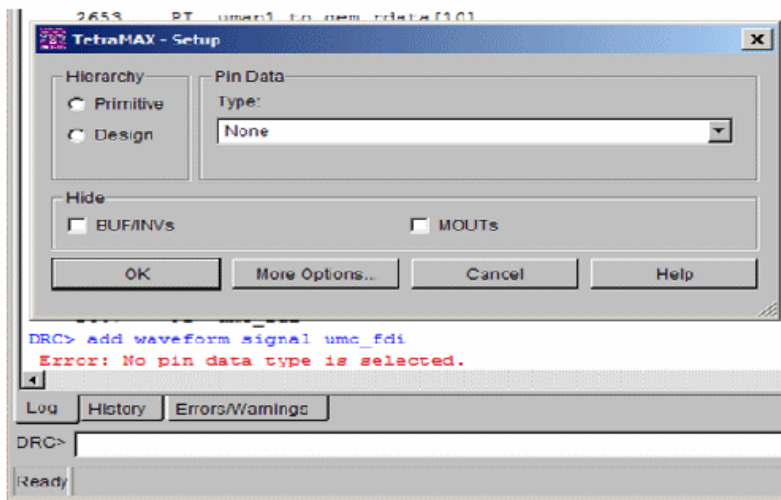


The data values displayed are generated either by DRC or by ATPG. Data values generated by DRC correspond to the simulation values used by DRC in simulating the STIL test_setup protocol to check conformance to the test protocol. Data values generated by ATPG are the actual logic values resulting from a specific ATPG pattern.

When you analyze a rule violation or a fault, TestMAX ATPG 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 the [Supported Pin Data Types and Definitions](#) section. Figure 2 shows an error caused when you do not select a valid pin data type that is supported by the SWV.

Figure 82 Example of Selecting an Invalid Pin Data Type



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 TestMAX ATPG) and you want to see the patterns for an overall evaluation of how TestMAX ATPG 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 TestMAX ATPG.

The following examples show some sample flows using the SWV. Enter these commands in a command file.

Example Flow

```
set_pindata -test_setup # test_setup is one of the many pin_data_types
add_display_gates -all # this invokes the GSV containing the gates of
interest.
set_pindata -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 might 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_pindata -seq_sim_data # required for SWV
```

```
add_waveform_signals <> # this invokes the SWV containing the  
# waveforms for the I/Os of the patterns 85 through 89
```

Example 3

```
set_patterns -external patterns.stil  
analyze_simulation_data patsl.vcd -fast 1  
add_display_gates < >  
set_pindata -debug_sim_data # should be the default setting
```

SWV Inputs and Outputs

The SWV has two input flows:

- 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 includes messages, warnings, and errors.

Analyzing Violations

The various TestMAX ATPG error messages related to the SWV are described as follows:

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 TestMAX ATPG using the `set_drc -store_setup` or the `set_simulation -data` command. See the [Supported Pin Data Types and Definitions](#) section.

Error: Invalid argument "TOP_template_DW_tap_inst/U34/QN". <M1>

This message 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 appears 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 appears 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

Using Tcl With TestMAX ATPG

The following sections describe how to use the TestMAX ATPG Tcl command interface:

- [Converting TestMAX ATPG Command Files to Tcl Mode](#)
- [Converting a Collection to a List in Tcl Mode](#)
- [Tcl Syntax and TestMAX ATPG Commands](#)
- [Redirecting Output in Tcl Mode](#)
- [Using Command Aliases in Tcl Mode](#)
- [Interrupting Tcl Commands](#)
- [Using Command Files in Tcl Mode](#)
- [An Introduction to the TestMAX ATPG Tcl API](#)

For a general guide on how to use Tcl with Synopsys tools, see *Using Tcl With Synopsys Tools*, available through SolvNet at the following URL:

https://solvnet.synopsys.com/dow_retrieve/latest/tclug/tclug.html

In Tcl Mode, it is possible to use Tcl API commands to access, and then manipulate TestMAX ATPG data. For a complete description, see “An Introduction to the TestMAX ATPG Tcl API” in TestMAX ATPG Online Help.

Converting TestMAX ATPG Command Files to Tcl Mode

You can use the `native2tcl.pl` translation script to convert existing native mode TestMAX ATPG command files to Tcl mode TestMAX ATPG command files. This script is in the installation tree at the following location:

```
$SYNOPSYS/auxx/syn/tmax/native2tcl.pl
```

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, TCL.
<code>[- -r dir]</code>	Accepts input from STDIN or from the specified directory path.

For example, assuming that the native mode script to be converted is located under `/user/TMAX`, the command-line entry would appear as follows:

```
native2tcl.pl -t .TCL -r /user/TMAX
```

Converting a Collection to a List in Tcl Mode

TestMAX ATPG Tcl API netlist query commands, such as `get_clocks` and `get_ports`, return a collection of design objects, but not a Tcl list of named objects. You can use the `get_object_name` procedure to convert a collection to a Tcl list. For example, you can convert a collection of ports to a list of port names.

You can define the `get_object_name` procedure using the following command:

```
source [getenv SYNOPSIS]/auxx/syn/tmax/get_object_name.tcl
```

After the `get_object_name` procedure is sourced within the Tcl environment, it is available for use with various TestMAX ATPG collections. An example is as follows:

```
TEST-T> set coll [get_ports test_si*]
{test_si1 test_si2 test_si3 test_si4 test_si5 test_si6 test_si7}
TEST-T> echo $coll
_ssel2
TEST-T> set tcllist [get_object_name $coll]
test_si1 test_si2 test_si3 test_si4 test_si5 test_si6 test_si7
```

Tcl Syntax and TestMAX ATPG Commands

The TestMAX ATPG user interface is based on Tcl version 8.4. Using Tcl, you can extend the TestMAX ATPG 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.

There are two types of TestMAX ATPG commands:

- Application commands
- Built-in commands

Each type is described in the following sections. Other aspects of Tcl version 8.4 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.

The following sections describe Tcl syntax and TestMAX ATPG Commands:

- [Specifying Lists in Tcl Mode](#)
- [Abbreviating Commands and Options in Tcl Mode](#)
- [Using Tcl Special Characters](#)
- [Using the Result of a Tcl Command](#)
- [Using Built-In Tcl Commands](#)
- [TestMAX ATPG Extensions and Restrictions in Tcl Mode](#)

Specifying Lists in Tcl Mode

In Tcl mode, you can specify lists in commands within curly braces (`{ }`), or within brackets (`[]`) if preceded by the `list` keyword.

In the following example, curly braces are used in the `add_pi_constraints` command to specify a list of ports:

```
DRC-T> add_pi_constraints 1 {TEST_MODE TICK CLK}
DRC-T> report_pi_constraints
port_name  constrain_value
-----  -
/TEST_MODE 1
/TICK      1
/CLK       1
```

Alternatively, you can specify a list of ports in the `add_pi_constraints` command using the keyword `list` and brackets:

```
DRC-T> add_pi_constraints 1 [list TEST_MODE TICK CLK]
DRC-T> report_pi_constraints
port_name  constrain_value
-----  -
/TEST_MODE 1
/TICK      1
/CLK       1
```


In Tcl mode, a list format is required when multiple arguments follow an option. For example:

```
set_build -instance_modify {spechbuffer TIEX}
```

Tcl Mode and Backslashes

In Tcl mode, a backslash character (\) specified at the end of a line represents a line continuation. Any single backslash specified in the middle of a word escapes the character following it. The following examples show how to overcome this situation when you want to specify a backslash within a Tcl list:

Use a double-backslash, for example:

```
add_clocks 0 {\A[0] \B[0]}
```

Use two levels of curly braces, for example:

```
add_clocks 0 {{\A[0]} {\B[0]}}
```

As an alternative, you can remove backslashes entirely. In this case, TestMAX ATPG commands automatically match specified identifiers that have no backslashes to identifiers in the database that have backslashes.

The following examples show various methods for specifying escaped names for a list argument:

```
add_faults {{\abccdef/hij/U1/A}}
add_faults {\abccdef/hij/U1/A}
add_faults {abccdef/hij/U1/A}
add_faults [list {\abccdef/hij/U1/A} ]
add_faults [list \abccdef/hij/U1/A ]
add_faults [list abccdef/hij/U1/A ]
```

Using Positional Arguments

Positional arguments must be specified within a Tcl list using curly braces. For example:

```
run_simulation -pin { ucore/freg/u540 01 }
```

However, if multiple specifications of the same argument are required, you must use a separate set of lists, as shown in the following example:

```
run_simulation -pin { ucore/freg/u540 0 } -pin { ucore/alu/u27 1 }
```

Abbreviating Commands and Options in Tcl Mode

Application commands are specific to TestMAX ATPG. 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 is `Anywhere`, in the site setup 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, TestMAX ATPG attempts to help you find the correct command.

For example, the following command is ambiguous:

```
> report_scan_c
Error: ambiguous command 'report_scan_c' matched 2 commands:
(report_scan_cells, report_scan_chains) (CMD-006).
```

TestMAX ATPG 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 1 have special meaning for Tcl in certain contexts.

Table 3 Special Characters

Character	Description
\$	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.

Table 3 Special Characters (Continued)

Character Description	
{ }	Denotes rigid quoting. There are no substitutions.
;	Ends a command.
#	Begins a comment.

Using the Result of a Tcl Command

TestMAX ATPG 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

The following example uses a result:

```
if {[expr $a + 11] <= $b} {  
  echo "Done"  
  return $b  
}
```

Using Built-In Tcl Commands

Most built-in commands are intrinsic to Tcl. Their arguments do not necessarily conform to the TestMAX ATPG argument syntax. For example, many Tcl commands have options that do not begin with a dash, but do have a value argument.

For example, the Tcl string command has a compare option that you use as follows:

```
string compare string1 string2
```

A log file of the TestMAX ATPG session can be created using the `set_messages -log <file>` command, as with native mode. However, some Tcl built-in commands might not be able to write to the log file. For example, the `puts` command cannot write to the TestMAX ATPG log file; use the `echo` command instead.

TestMAX ATPG Extensions and Restrictions in Tcl Mode

Generally, TestMAX ATPG implements all the Tcl built-in commands. However, TestMAX ATPG adds semantics to some Tcl built-in commands and imposes restrictions on some elements of the language. The differences are as follows:

- 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 non-standard 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 TestMAX ATPG 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 in Tcl Mode

You can direct the output of a command, procedure, or a script to a specified file using the `redirect` command or by 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.

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
```

The Tcl built-in command puts does not respond to redirection of any kind. Instead, use the TestMAX ATPG command echo, which responds to redirection.

The following sections describe in detail how to redirect output:

- [Using the redirect Command in Tcl Mode](#)
- [Getting the Result of Redirected Tcl Commands](#)
- [Using Redirection Operators in Tcl Mode](#)

Using the redirect Command in Tcl Mode

In an interactive session, the result of a redirected command that does not generate a Tcl error is an empty string, as shown in the following 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, as shown in the following 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 the following command:

```
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 Tcl 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.

An example is as follows:

```
redirect p.out {
  set rnet [catch {read_netlist h4c.lib }]
}
if {$rnet == 1} {
  echo "read_netlist failed! Returning..."
  return
}
```

Using Redirection Operators in Tcl Mode

Because Tcl is a command-driven language, traditional operators usually have no special meaning unless a particular command (such as `expr`) imposes some meaning. TestMAX ATPG commands respond to `>` and `>>` but, unlike UNIX, TestMAX ATPG 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, as shown in the following example:

```
echo $spec_variable >> file.out; # Right
echo $spec_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 in Tcl Mode

You can use aliases to create short forms for the commands you commonly use. For example, the following command duplicates the function of the `dc_shell` include command when using TestMAX ATPG:

```
> alias include "source -echo -verbose"
```

After creating the alias in the previous example, you can use it by entering the following command:

```
> include commands.cmd
```

When you use aliases, keep the following points in mind:

- TestMAX ATPG 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 TestMAX ATPG 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 Tcl 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 might take awhile before responding to an interrupt request, but TestMAX ATPG commands will eventually respond to the interruption.

If TestMAX ATPG is processing a command file (see [Using Command Files](#)), and you interrupt one of the file's commands, script processing is interrupted and TestMAX ATPG does not process any more commands in the file.

If you press Control-c three times before a command responds to your interrupt, TestMAX ATPG is interrupted and exits with the following message:

```
Information: Process terminated by interrupt.
```

There are a few exceptions to this behavior, which are documented with the applicable commands.

Using Command Files in Tcl Mode

You can use the source command to execute scripts in TestMAX ATPG. A script file, also called a command file, is a sequence of commands in a text file.

The syntax is as follows:

```
> 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 TestMAX ATPG command. The file can be simple ASCII or gzip compressed.

The following sections describe how to use command files:

- [Adding Comments](#)
- [Controlling Command Processing When Errors Occur](#)
- [Using a Setup Command File](#)

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, as shown in the following 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, TestMAX ATPG 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 TestMAX ATPG 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 might not perform as expected if a command fails due to syntax or semantic errors (for example, an invalid option).

The `sh_script_stop_severity` variable has no effect if the `sh_continue_on_error` variable is set to `true`.

To get TestMAX ATPG 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` 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 TestMAX ATPG will automatically execute it at startup. The default setup file is located in the following directory:

```
$SYNOPSIS_TMAX/admin/setup/tmaxtcl.rc
```

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 TestMAX ATPG was started or in your home directory.

An Introduction to the TestMAX ATPG Tcl API

The Tcl Application Programming Interface (API) creates a script to manipulate TestMAX ATPG data. Most TestMAX ATPG data is stored in an internal data structure. This data is updated after the execution of the commands that manipulate it. For example, diagnostics fault candidates are stored in a data structure updated at the end of each `run_diagnosis` command. To access this data, you use a set of Tcl API commands.

For a list of all TestMAX ATPG-specific API commands, see `tcl_api_commands` in TestMAX ATPG Online Help.

Retrieving Information

Tcl API commands enable you to retrieve information for a specific object. The Tcl API includes several classes of objects, for example: `class cell`, `class pattern`, and `class fault candidate`. To retrieve information on these objects, you use a set of Tcl commands that begin with the `"get_"` prefix.

When data is available, a query of a class returns a collection (which is different as a Tcl list) of objects. If the data is unavailable, the query return is empty.

If you have a large design, a query of a collection of all cells in the design could take excessive time. You can avoid this by using matching expressions to filter the objects that are returned for each `"get_"` command.

Using the -filter Option

Some commands have a `-filter` option. This option filters a command query based on specified attributes. The `-filter` option accepts only wildcards for filtering and not the regular expression syntax. Note that you cannot use the `-filter` and `-regexp` options at the same time.

Examples of the `-filter` option are as follows:

- To get a collection of all diagnostics fault candidates with the `pinpath` attribute containing the letter "A," specify the following:

```
TEST-T> set candid_col [get_candidate -filter "pinpath=~*A*"]
```

- You can specify complex filter expressions using `&&` or `||` operators. For example:

```
TEST-T> set cell_col [get_cells -filter "chain_name == c0 &&  
scan_position== 2"]
```

Using the -regexp Option

You can also use the `-regexp` to filter a command query. This option applies only to the object type name. For example, if you use the `-regexp` option to get cells, the pattern will only match the cell name.

Examples of the `-regexp` option are as follows:

```
TEST-T> set cell_col [get_cells {[A-U]28} -regexp]  
TEST-T> set cell_obj [get_cells {U28} -regexp]
```

10

Design Netlists and Library Models

TestMAX ATPG builds a netlist that is optimized for ATPG.

The [Preparing a Netlist](#), [Reading a Netlist](#), and [Reading Library Models](#) sections provide specific information on how to specify netlists and library models. The following sections provide additional information on reading and processing design netlists and library models:

- [Netlist Format Requirements](#)
- [About Reading a Netlist](#)
- [Using Wildcards to Read Netlists](#)
- [About Reading Library Models](#)
- [Controlling Case-Sensitivity](#)
- [Setting Parameters for Learning](#)
- [About Building the ATPG Model](#)
- [Processes That Occur When Building the ATPG Model](#)
- [Flattening Optimization for Hierarchical Designs](#)
- [Identifying Missing Modules](#)
- [Removing Unused Logic](#)
- [Using Black Box and Empty Box Models](#)
- [Handling Duplicate Module Definitions](#)
- [Creating Custom ATPG Models](#)
- [Condensing ATPG Libraries](#)
- [Assertions](#)
- [Memory Modeling](#)

Netlist Format Requirements

TestMAX ATPG can read netlists in Electronic Design Interchange Format (EDIF), Verilog, and VHDL formats. It can read non-encrypted as well as Synenc-encrypted netlists. Some minimal preprocessing might be necessary to make the netlist compatible with TestMAX ATPG.

The following sections describe the netlist requirements for TestMAX ATPG:

- [EDIF Netlist Requirements](#)
- [Verilog Netlist Requirements](#)
- [VHDL Netlist Requirements](#)

EDIF Netlist Requirements

To ensure EDIF netlists are compatible with TestMAX ATPG, you must review all power and ground logic connections. The following sections describe how to handle these situations:

- [Logic 1/0 Using Global Nets](#)
- [Logic 1/0 by Special Library Cell](#)

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, TestMAX ATPG issues warnings, such as “floating internal net,” 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 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 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 required 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 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 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. Use the following guidelines to ensure that your Verilog netlist is compatible with TestMAX ATPG:

- 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.
- Verilog is case-sensitive, although many tools ignore case and treat “specNet” and “specnet” 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.

[ATPG Modeling Primitive Summary](#)

VHDL Netlist Requirements

The following guidelines apply when using a VHDL netlist with TestMAX ATPG:

- VHDL designs 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.

About Reading a Netlist

TestMAX ATPG automatically determines the format of a referenced netlist. It reads the file in hierarchical order, starting with the library leaf cells and ending with the top-level module.

The netlist data passed into ATPG must contain structural design constructs. These constructs are instances of modules comprised of leaf-level Boolean and sequential logic primitives. A design is defined and fault-graded based on these Boolean and sequential elements.

TestMAX ATPG supports limited use of complex logical expressions within the design environment. All extracted design structures must be thoroughly reviewed. For optimal performance, make sure that minimal design constructs represent the design. For example, passing in design libraries containing unused or redundant elements directly impact overall netlist processing.

Netlists should not contain power and ground connections since they cannot be used in the test operation. The power and ground information cause large and often bidirectional networks that directly affect netlist processing in both runtime and memory requirements. This assumes the networks are consumed without exceeding TestMAX ATPG processing limitations.

There are several different options you can use when reading a netlist:

- By default, TestMAX ATPG treats [Verilog netlists](#) as case-sensitive, and [EDIF](#) and [VHDL](#) netlists as case-insensitive. You can override the default using the `-sensitive` or `-insensitive` option of the `read_netlist` command or by selecting Sensitive or Insensitive in the "Case sensitivity" drop-down menu of the Read Netlist dialog box in the TestMAX ATPG GUI.
- TestMAX ATPG issues a warning if a module is defined more than one time and uses the module definition from the last loaded netlist. You can identify a module as a master module and it will not be replaced if TestMAX ATPG encounters a module with the same name. Use the `set_netlist -redefined_module` command to change the module definition to the first or last module.

- Use the `-define` option of the `read_netlist` command to define any variable function or expression. This option is equivalent to the 'define Verilog statement.
- Use the `-delete` option of the `read_netlist` command to delete any netlists currently stored in memory.
- The `set_commands noabort` command prevents TestMAX ATPG from terminating if it encounters an error when reading multiple netlists.

For more information, see [Reading a Netlist](#).

Using Wildcards to Read Netlists

If your library cells are stored in multiple individual files, you can read them all using wildcards. TestMAX ATPG supports the asterisk (*) to match occurrences of any character, and the question mark (?) to match any single character.

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

```
BUILD-T> read_netlist speclib/*.v
```

To read in all files in `speclib`, enter the following command:

```
BUILD-T> read_netlist speclib/*
```

To read in all files that begin with `DF` and end with `.udp`, in all subdirectories in `speclib` that end in `_lib`, enter the following command:

```
BUILD-T> read_netlist speclib/*_lib/DF*.udp
```

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

```
BUILD-T> read_netlist DF???.V
```

You can also use wildcards in the Read Netlist 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 the following options:

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

About Reading Library Models

TestMAX ATPG creates ATPG models based on the functional portion of Verilog simulation models. These models include user-defined primitives (UDPs), which are essential for describing particular library cells. Behavioral models are not recognized.

TestMAX ATPG recognizes the following Verilog language attributes:

```
`define
`ifdef
`include
`celldefine
`suppress_faults
`enable_portfaults
```

You must read in all library models referenced by your design. You can read in one model at a time, or you can read in the entire library with a single command. If your design already contains a module that has the same name as one of the library modules, when you read in the library, the library model overwrites your module.

See Also

- [Reading Library Models](#)
- [Reading a Netlist](#)

Controlling Case-Sensitivity

Netlist formats differ in whether or not the instance, pin, net, and module names are case-sensitive. When TestMAX ATPG 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 `speclib`, using case-insensitive rules, use the following command:

```
BUILD-T> read_netlist speclib/*.V -insensitive
```

Setting Parameters for Learning

When TestMAX ATPG builds an ATPG model, it also performs a circuit learning process to determine information useful for performing simulation and test generation.

This learning process performs the following tasks:

- 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 and 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 (those with identical inputs)
- Identifies implication relationships between gates

Learned Behavior Types

During the learning process, each gate is assigned a learned behavior. The possible types of learned behavior are as follows:

Blocked - A gate whose fault effects are blocked from detection by tied circuitry.

Common Input - A gate that has a common source for two or more of its inputs.

Common Tied Input - A gate that is equivalent to a tied gate due to some logical relationship between its inputs. For example, an XOR gate with both inputs attached to the same net is equivalent to a tied-to-0; or an AND gate with a net and its inverted value as inputs will also be equivalent to a tied-to-0.

Constrained - A gate with an input constraint resulting in an output that can never achieve a 0, 1, or Z, or some combinations of these logic values.

Constrained Blocked - A gate whose fault effects are blocked from detection by constraints.

Equivalence - Two gates whose outputs are equivalent or complementary to each other at all times. For example, a NAND and an AND gate with the same input connections always have opposite values on their outputs.

Implications - Two gates whose behavior has been learned to have an implied relationship, such as "gate A at value J implies gate B at value K".

Inverted Inputs - A gate that has an inverted input function. In other words, an inverter has been merged into the input of an otherwise standard gate such as AND, NAND, OR, or NOR.

Learn BUF - A gate whose function is equivalent to a BUF, such as an AND gate with its inputs tied together.

Learn INV - A gate whose function is equivalent to an INV, such as a NAND gate with its inputs tied together.

Learn Tied Gate - A gate whose function is equivalent to a tied 0/1/Z/X gate.

Tied - Any gate learned to be always tied to 0/1/Z/X.

Weak - Any gate with a WEAK input. This is generally a BUS device.

You can view most learned data for a given gate by setting the `-verbose` option for the `set_pindata` command and running the `report_primitives` command for the selected gate.

Controlling the ATPG Learning Algorithm

You can control the ATPG learning algorithms using the `set_learning` command or the Run Build Model dialog box.

The following example uses the `set_learning` command to specify the ATPG equivalence algorithm for learning:

```
BUILD-T> set_learning -atpg_equivalence
```

To use the Run Build Model dialog box to control the learning algorithms:

1. From the command toolbar, click the Build button.

The Run Build Model dialog box appears.

2. Select or enter the appropriate options in the Set Learning section. For descriptions of these controls, see the description of the `set_learning` command in TestMAX ATPG Help.
3. Click OK.

About Building the ATPG Model

To build the ATPG model, TestMAX ATPG compiles a set of netlist and library models into a single in-memory image. For more information on reading netlists and library models, see [Reading a Netlist](#) and [Reading a Library Model](#).

When you build an ATPG model of a hierarchical design, TestMAX ATPG flattens the hierarchy to make a single-level, in-memory model of the design. Several different optimization methods are used to reduce the number of gates and simplify the design. You can control many of these processes using the `set_build` command, as described in [Controlling the Build Process](#).

During the process of building a model, TestMAX ATPG also performs a circuit learning process to determine information useful for performing simulation and test generation. The parameters you can set for this learning process are described in [Setting Parameters for Learning](#).

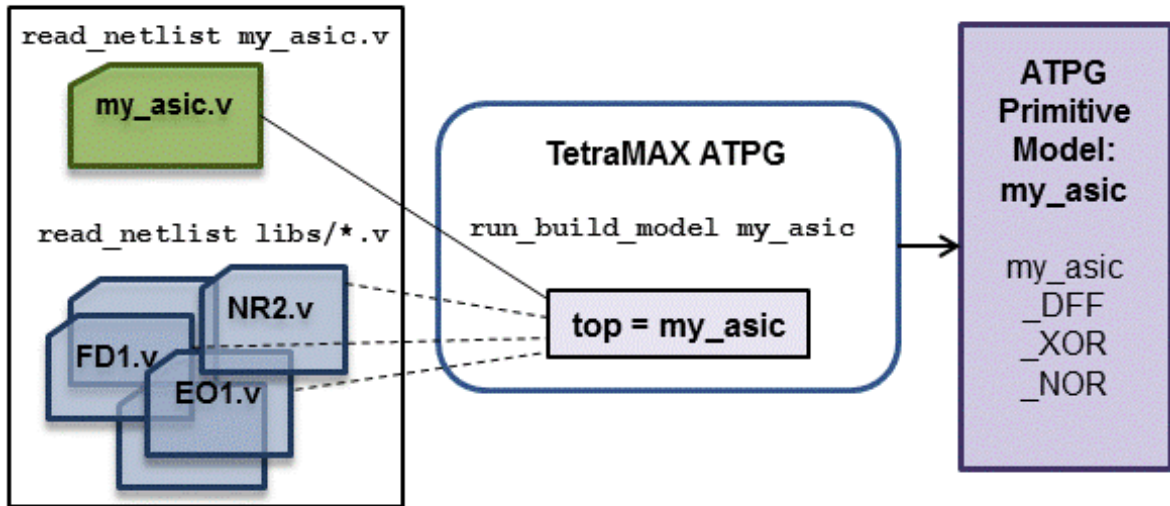
You can specify several parameters that control and optimize the process of building an ATPG model, including:

- Add a buffer gate between any latch or flip-flop gate directly connected to another latch or flip-flop gate
- Specify and remove modules designated as black boxes, empty boxes, design library cells
- Preserve pin names of models that would normally be flattened
- Add pulldown and pullup drivers and bus keepers to BUS gates
- Keep or delete unused gates
- Specify an alternate hierarchical delimiter
- Modify or replace selected instances
- Define certain input signals in the top-level module as bidirectional signals
- Limit the number of fanouts for gates
- Specify various model flattening optimization algorithms
- Specify how net connections affect the flattened ATPG model
- Specify parameters for modeling undriven bidirectional nets

For details on these options, see the description of the `set_build` command in TestMAX ATPG Help.

The following figure shows the process for building the ATPG model:

Figure 83 Building the ATPG Design Model



See Also

- [Building the ATPG Model](#)

Processes That Occur When Building the ATPG Model

During the 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.

The following example shows an example transcript of the `run_build_model` command.

Transcript of `run_build_model` Command Output

```
BUILD-T> 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
-----
```

Flattening Optimization for Hierarchical Designs

When you build a model of a hierarchical design (using the `run_build_model` command), TestMAX ATPG flattens the hierarchy to make a single-level, in-memory model of the design. Several different optimization methods are used to reduce the number of gates and simplify the design. Some of these methods are always performed, while others are enabled or disabled with the `set_build` command.

TestMAX ATPG can perform 16 different types of optimization. Each optimization method is described, including the user commands for enabling or disabling the method, where

applicable. The default configuration (either enabled or disabled) is marked with an asterisk in each such description.

1. *BUF elimination*

Always enabled; no user control

During the flattening process, buffers are eliminated wherever this is possible without eliminating any fault sites.

2. *INV elimination*

Always enabled; no user control

During the flattening process, inverters are eliminated wherever this is possible without eliminating any fault sites.

3. *Switches(SW) as BUFs or BUFZs*

Always enabled; no user control

During the flattening process, each SW primitive found that has its control gate held constantly on is replaced with a BUFZ device; or if the propagation of a Z value is not needed, it is replaced with a BUF device. These BUF/BUFZ device can be removed later by the BUF elimination method (#1 above). This optimization can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

4. *DLATs as BUFs*

Always enabled; no user control

During the flattening process, for each DLAT primitive found that has its gate/clock input held on and its set and reset lines held off so that the latch is always transparent, the DLAT is replaced with a BUF device. This optimization can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

5. *DFFs as DLATs*

Always enabled; no user control

During the flattening process, each DFF primitive found that has its clock permanently off, but able to use its asynchronous set or reset input, is replaced with a latch device.

6. *Unused Gates*

To enable: `set_build -delete_unused_gates` (default)

To disable: `set_build -nodelete_unused_gates`

An unused gate is one which has no output connections to other gates, including black box or empty box gates. When this optimization method is enabled, unused gates are

removed during the flattening process. It is possible for fault sites to be dropped as these gates are removed.

7. TIE propagation

To enable: `set_build -merge tied_gates_with_pin_loss`

To disable: `set_build -merge notied_gates_with_pin_loss (default)`

TIE propagation optimization identifies nets and pins tied high or low and attempts to propagate this constant value through logic to reduce the number of gates. When disabled, TIE propagation is still performed, but only where it does not cause testable fault sites to be dropped. When enabled, TIE propagation occurs even where it causes testable fault sites to be dropped. If any fault sites are dropped, they are reported in a summary message; for example:

```
There were 38240 primitives and 6318 faultable pins removed during model
optimizations
```

8. Cascaded Gates

To enable: `set_build -merge cascaded_gates_with_pin_loss`

To disable: `set_build -merge noscascaded_gates_with_pin_loss (default)`

Cascaded gate optimization is performed by identifying two gates in series that can be logically merged into a single gate. An example of this is two 2-input AND gates in series, which can be replaced by a single 3-input AND gate. When disabled, cascaded gate optimization is still performed, but only where it does not cause fault sites to be dropped. When enabled, cascade optimization is performed even where it causes fault sites to be dropped. If any fault sites are dropped, they are reported as B22 violations.

9. Bus Keepers

To enable: `set_build -merge Bus_keepers (default)`

To disable: `set build -merge NOBus_keepers`

Bus keeper recognition is performed by searching for small, constantly enabled combinational loops with a weak driver. This recognition considers paths including BUF and INV, as well as SW and TSD devices used to form bus keepers that hold only one state. After identified, these loops are replaced with BUSK ATPG primitives. When enabled, bus keeper optimization can result in the dropping of faults sites. If any fault sites are dropped, they are reported as B22 violations.

10. Feedback Paths

To enable: `set_build -merge feedback_paths (default)`

To disable: `set_build -merge nofeedback_paths`

Feedback path optimization is done by searching for combinational loops that do not perform any testable function. One of example is a loop involving a BUS with a weak driver and at least one strong, non-three-state driver. The loop through the weak driver can be removed. Another example is a three-state net where all the potential drivers come from top-level primary inputs (strong drivers) and the feedback path is again through a weak driver. Elimination of these feedback paths can cause fault sites to be dropped. If this happens, the dropped faults are reported as B22 violations.

11. MUX Recognition

To enable: `set_build -merge mux_from_gates`

To enable: `set_build -merge muxpins_from_gates (default)`

To enable: `set_build -merge muxx_from_gates`

To disable: `set_build -merge nomux_from_gates`

The MUX recognition optimization method is done by searching for discrete gates that can be combined to create MUX behavior. The most common form is two 2-input AND gates followed by an OR gate. Additional variants are also recognized, such as pass-transistor MUXes. There are three variations of this optimization method:

`Mux_from_gates` - When enabled, discrete-gate forms of MUX behavior are replaced with TestMAX ATPG MUX primitives. During this optimization, it is possible that fault sites is dropped. If any fault sites are dropped, they are reported as B22 violations.

`Muxpins_from_gates` - When enabled, discrete-gate forms of MUXes are replaced, but only if no fault sites are dropped as a result.

`Muxx_from_gates` - When enabled, discrete-gate forms of MUXes are replaced, but only if no fault sites are dropped as a result, and only for "optimistic MUX" behavior. Gates which form the "pessimistic MUX" behavior are left unchanged.

An "optimistic MUX" produces an output equal to the data inputs when the select line is X and both inputs are identical. The "pessimistic MUX" produces an output of X when the select line is X, even when the data inputs are identical. The TestMAX ATPG MUX primitive implements the "optimistic MUX" behavior.

12. XOR/XNOR Recognition

To enable: `set_build -merge Xor_from_gates`

To enable: `set_build -merge XORPins_from_gates (default)`

To disable: `set_build -merge NOXor_from_gates`

XOR/XNOR recognition optimization is done by searching for discrete gates that form either the XOR or XNOR function. There are two variations of this optimization:

`Xor_from_gates` - When enabled, discrete-gate forms of XOR/XNOR are replaced with TestMAX ATPG XOR/XNOR primitives. This optimization can cause fault sites to be dropped. If this occurs, the dropped fault sites are reported as B22 violations.

`Xorpins_from_gates` - When enabled, discrete-gate forms of XOR/XNOR are replaced with TestMAX ATPG XOR/XNOR primitives, but only where no fault sites are dropped as a result.

13. Equivalent DLAT/DFF

To enable: `set_build -merge equivalenten_dlat_dff (default)`

To enable: `set_build -merge equivalenten_initialized_dlat_dff`

To disable: `set_build -merge noequivalenten_dlat_dff`

This optimization method identifies equivalent DLAT and DFF devices, and merges the equivalent functions into a single device. The `equivalenten_initialized_dlat_dff` setting, if enabled, will assume the devices are initialized to their steady state values before determining if they can be merged into a single device. Two DLAT or two DFF devices are equivalent if they share common input connections, including all clock, set, reset, and data inputs. The outputs of the two devices can be identical or complementary to each other. This optimization method replaces one equivalent device with a BUF or INV connected to the output of the other equivalent device. During this process, fault sites might be dropped, in which case they are reported as B22 violations.

14. DLAT pairs as DFF

To enable: `set_build -merge flipflop_from_dlat (default)`

To enable: `set_build -merge flipflop_cell_from_dlat`

To disable: `set_build -merge noflipflop_from_dlat`

This optimization method finds each pair of D-latches that operate together as a D flip-flop, and replaces them with a DFF primitive. This occurs when two DLAT devices are connected serially from the Q output of one to the D input of the other, share common set, reset, and complementary clocks from the same source. When this optimization occurs, the two DLAT devices are replaced with a DFF primitive, possibly causing fault sites to be dropped. If any fault sites are dropped, they are reported as B22 violations.

`flipflop_cell_from_dlat` - When enabled, merging master-slave latches into a flip-flop is limited to those latch pairs that are part of the same design-level cell. This option should be used by TestMAX DFT and when necessary to avoid pin loss due to merging latches to flip-flops.

15. WIRE and BUS gates

To enable: `set_build -merge Wire_to_buffer` (default)

To disable: `set_build -merge NOWire_to_buffer`

This optimization identifies and optimizes WIRE and BUS gates having common-source inputs with buffer gates. Such WIRE and BUS gates are like those found common in clock and scan-enable repowering networks. The buffer gates thus created can be further removed by other optimizations. This optimization can result in faultable pin losses; however, the lost faults are untestable anyway. In many designs, this optimization results in fewer primitives in the final model, particularly fewer WIRE gates; in many cases the number of WIRE gates is reduced to 0, which also results in faster DRC. The default is `-wire_to_buffer` (optimization is enabled).

16 Tied inputs and MUX gates

To enable: `set_build -merge Global_tie_propagate` (default)

To disable: `set_build -merge NOGlobal_tie_propagate`

This optimization identifies and optimizes global tie 0/1 value propagations and replaces certain [N]AND, [N]OR and MUX gates with buffers/inverters or eliminates them completely. The analysis ensures that all faults eliminated are either undetectable-redundant (UR) or equivalent to other faults that are preserved. The memory and CPU time required by the flattening process are not measurably affected by this analysis.

This optimizations might change the reported test coverage, because:

- Eliminated UR faults could have been classified as ATPG-untestable (AU).
- Eliminated equivalent faults might change equivalence classes size and affect uncollapsed coverage.

The following optimizations are performed during analysis:

- [N]AND, [N]OR gate with controlling tied inputs (T0/T1): replaced with T0/T1 if no output fault and no faults on the tied inputs. All faults lost are classified UR.
- [N]AND, [N]OR gate with non-controlling tied inputs (T1/T0): replaced with buffer/inverter if only one input is not tied. Faults lost are either classified UR or equivalent to the corresponding output fault.
- MUX gate with tied select input: replaced with buffer/inverter from selected data input if no fault on the select input or data lines cannot have complementary values. All faults lost are classified UR.

- MUX gate with data inputs driven by common gate, with same inversion: replaced with buffer/inverter from a data input if no faults on data inputs. All faults lost are classified UR.
- MUX gate with data inputs driven by T0/T1: replaced with buffer/inverter from select line. Faults lost are either classified UR or equivalent to the corresponding output fault.

Disabling this optimization could be desirable in the following cases:

- If pins targeted by an `add_net_connections` command or by reading in external fault files are eliminated by the new analysis.
- If design-level viewing is limited because instances of interest have been "flattened-down" (these are tracked by B22 violations).

The current value of all optimization settings can be reviewed by using the `report_settings build` command. An example of a report is as follows:

```

BUILD> report settings build
build = add_buffer=yes, delete_unused_gates=yes, fault_boundary=lowest,
        hierarchal_delimiter='/', pin_assign=256, undriven_bidi=PIO,
        net_connections_change_netlist=yes,
merge: bus_keepers=yes
        cascaded_gate_with_pin_loss=no
        equivalent_dlat_dff=on
        feedback_paths=yes
        flipflop_from_dlat=on
        mux_from_gates=pin-preserve
        tied_gates_with_pin_loss=no
        global_tie_propagate=yes
        wire_to_buffer=yes
        xor_from_gates=pin-preserve

```

During the flattening process, gate optimization details are reported if expert-level messages have been enabled with the `set_messages -level expert` command. In addition, a summary of optimization results is available at any time after the build process is completed by using the `report_summaries optimizations` command, as shown in the following example.

```

TEST> report_summaries optimizations
          Optimizations Report
-----
optimization #occurrences #primitives #pins #modules
type          eliminated  lost      optimized
-----
unused gates 15905          15905      2552      133
tied gates   0                   42         0         0
buffers     44152             44152      0         313
inverters   10601             10601      0         100
cascaded gates 529              529        0         2
SWs as BUFs  42                0          0         1

```

Chapter 10: Design Netlists and Library Models

Identifying Missing Modules

DLATs as BUFs	0	0	0	0
MUXs	3261	16242	8	19
XORs	0	0	0	0
equiv. DLAT/DFE	1831	0	0	8
DLATs as DFFs	0	0	0	0
DFFs as DLATs	0	0	0	0
BUS keepers	60	0	0	1
feedback paths	18	36	18	1

total	76399	87507	2638	322

If, during the optimization process, faults sites are eliminated as gates are removed, those faults sites are identified as B22 violations. Use the `report_violations b22` command to get a detailed list of fault sites removed during optimization or the `report_rules b22` command to get a summary count.

Identifying Missing Modules

If your design references undefined modules, TestMAX ATPG sends you error messages during execution of the `run_build_model` command. To identify all currently referenced undefined modules, you can use the Netlist > Report Modules menu command, or you can enter the `report_modules -undefined` command at the command line, for example:

```
BUILD-T> report_modules -undefined
```

An example of such a report is shown in the following figure.

Figure 84 Report Modules Window Listing Undefined Modules

module name	pins				inst	refs (def'd)	used
	tot	i	o	io			
ICNH	0	0	0	0	0	32 (N)	0
ON4	0	0	0	0	0	13 (N)	0
AND2	0	0	0	0	0	7 (N)	0
DFFLP	0	0	0	0	0	6 (N)	0
DFFRP	0	0	0	0	0	5 (N)	0
EXNOR	0	0	0	0	0	1 (N)	0
OR2	0	0	0	0	0	4 (N)	0
ICN	0	0	0	0	0	5 (N)	0
BICN	0	0	0	0	0	4 (N)	0
BON4T	0	0	0	0	0	4 (N)	0
INC4H	0	0	0	0	0	1 (N)	0
MUX2H	0	0	0	0	0	4 (N)	0
DFFP	0	0	0	0	0	4 (N)	0
OR2H	0	0	0	0	0	2 (N)	0
NAN2	0	0	0	0	0	6 (N)	0
AND3	0	0	0	0	0	1 (N)	0
NOR2	0	0	0	0	0	1 (N)	0
DFFLP	0	0	0	0	0	2 (N)	0
EXOR	0	0	0	0	0	2 (N)	0
NAN3	0	0	0	0	0	1 (N)	0
INV	0	0	0	0	0	3 (N)	0
MUX2A	0	0	0	0	0	1 (N)	0

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, see TestMAX ATPG Online Help.

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, TestMAX ATPG 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-T> set_rules 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, TestMAX ATPG 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-T> 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.

Removing Unused Logic

Designs can contain unused logic for several reasons:

- 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.

The following example 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 Module With Extra Logic

```
module sdfdr (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);
```

Chapter 10: Design Netlists and Library Models

Removing Unused Logic

```

    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 the `report_modules -verbose` command for module `sdffr` in the following example shows each primitive in the TestMAX ATPG model, with the timing check gates present.

Example 2: Module Report Showing Unused Gates

```
BUILD-T> report_modules 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

```

Chapter 10: Design Netlists and Library Models Removing Unused Logic

```

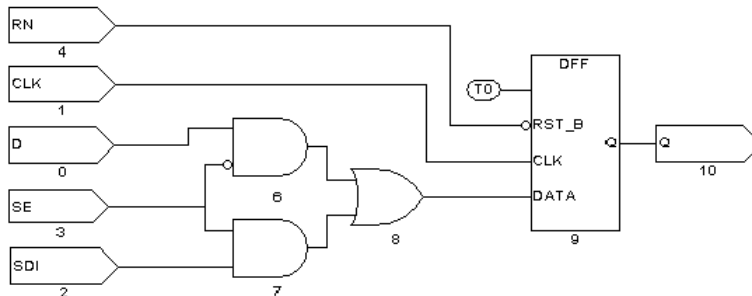
I:notify )
timing_check_1: not conn=( O:seb I:SE
)
timing_check_2: and conn=(
O:rn_and_SE I:RN I:SE )
timing_check_3: and conn=(
O:rn_and_seb I:RN I:seb
)

```

By default, TestMAX ATPG 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.

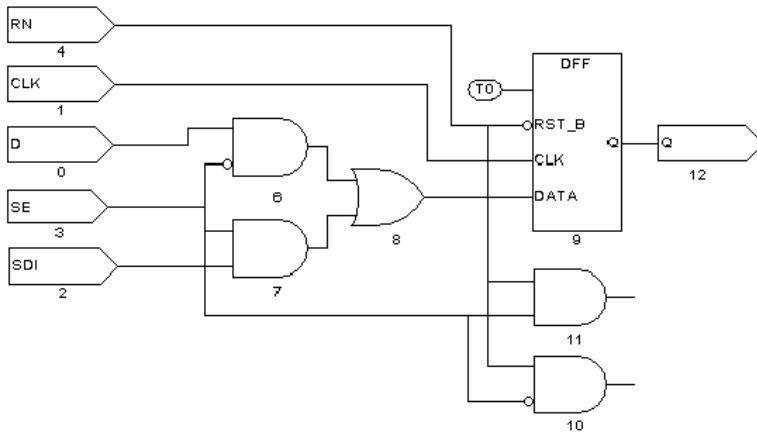
The following figure shows the GSV display of the schematic created when the “Delete unused gates” option is selected. The extra gates do not appear in the schematic.

Figure 85 Design Schematic With Delete Unused Gates On



To keep the extra gates, deselect the “Delete unused gates” option of the Set Build dialog box. The following figure 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 86 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-T> set_build -nodelete_unused_gates
```

Using Black Box and Empty Box Models

You might prefer not to perform ATPG on some blocks in a design – referred to as *black boxes* or *empty boxes*.

You can declare any block in the design to be a black box or an empty box, including 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.

The following sections describe how to use of black box and empty box models:

- [Declaring Black Boxes and Empty Boxes](#)
- [Behavior of RAM Black Boxes](#)

See Also

- [Binary Image Files](#)
- [Excluding Vectors from Simulation](#)

Declaring Black Boxes and Empty Boxes

You can declare black box or any empty box by using one of the following commands:

```
set_build -black_box module_name  
set_build -empty_box module_name
```

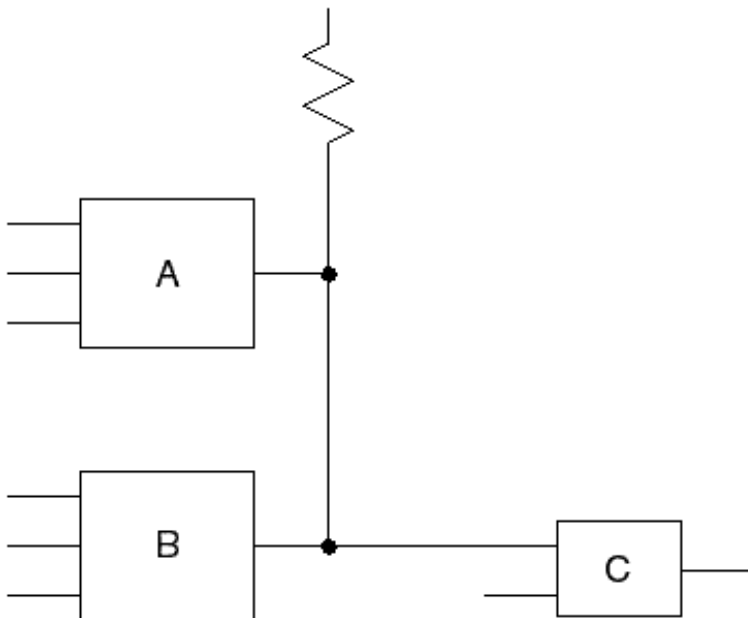
If you declare a block to be a black box, TestMAX ATPG ignores the contents of the block when you build the model with the `run_build_model` command. Instead, it terminates the block inputs and connects TIE X primitives to the outputs. Thus, the block outputs are unknown (X) for ATPG.

An empty box is the same as a black box, except that the outputs are connected to TIE Z rather than TIE X 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 the following figure. If the enabling logic is working properly, no more than one RAM is enabled at any given time, thus preventing contention at the outputs.

Figure 1: RAM Blocks Modeled As Empty Boxes



If you declare blocks A and B to be black boxes, their outputs are 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 is disabled during test, you can declare these two blocks to be empty boxes. In that case, their outputs is 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, bidi1, bidi2);
input i1, i2;
output o1, o2;
inout bidi1, bidi2;
_TIEX (i1, i2, o1); // terminate inputs & drive
output
_TIEX (o2);
_TIEX (bidi1);
_TIEX (bidi2);
endmodule
```

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

```
module EMPTY (i1,i2, o1, o2, bidi1, bidi2);
input i1, i2;
output o1, o2;
inout bidi1, bidi2;
_TIEZ (i1, i2, o1);
_TIEZ (o2);
_TIEZ (bidi1);
_TIEZ (bidi2);
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, bidi1, bidi2);
input i1, i2;
output o1, o2;
inout bidi1, bidi2;
endmodule
```

If you use such a model, TestMAX ATPG 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 following six cases revolve around how the memory module is or is not in the netlist, and how TestMAX ATPG treats that memory device. Additionally, pros and cons are provided for each case.

Case 1

Netlist Contains: No module definition for memory

TestMAX ATPG 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 specRAM` command to tell TestMAX ATPG 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. TestMAX ATPG 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

TestMAX ATPG 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 specRAM` command to instruct TestMAX ATPG 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 TestMAX ATPG might 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 TIEEX that might have exposed a contention problem will go unnoticed.

Case 3

Netlist Contains: Null module definition for memory

TestMAX ATPG 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 specRAM` 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 TestMAX ATPG 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. TestMAX ATPG will have a false environment where it sees no contention, but there could really be contention occurring.

Here's an example null module:

```
module specRAM (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
```

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 dumspec net.

Case 4

Netlist Contains: Null module definition

TestMAX ATPG 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 specRAM` command to instruct TestMAX ATPG that the outputs of the module should be connected to TIEZ 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 TestMAX ATPG 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, TestMAX ATPG might 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 TestMAX ATPG 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 TestMAX ATPG 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 example is a memory module with OEN modeling:

```
module specRAM (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 TestMAX ATPG.

Pros: Eliminates all problems of Cases 1 through 5.

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

The following example shows a memory module with full functional modeling (see [Memory Modeling](#) for additional examples):

```
//
// --- level sensitive RAM with active high chip select, read,
//      write, and output enable controls.
//
module specRAM (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

You should double-check the following specific items when you see unexplained behavior from your RAM block box are described next:

1. Did you follow the guidelines in the previous cases in terms of how the memory module was (or was not) defined in the netlist, as well as what command was issued in TestMAX ATPG?
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 spec_ram1024x8 # spec_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 TestMAX ATPG 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 this is the case, then perform the following steps:

1. 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. Therefore, an empty box model would be more appropriate here.
2. If you determine that a black box is still required for a RAM having inout ports used as outputs, then you have some choices to make because there is no way that TestMAX ATPG can determine whether a particular inout should be an "in" or an "out" given only the null module declaration in the netlist:
 - Make a TestMAX 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, `spec_RAM1_BBmodel.v`, which will in essence redefine the module `spec_RAM1` to now have these TIEX primitives on its inout ports). The RAM inouts will now act as outputs driving out 'X' values. The TestMAX ATPG graphical schematic viewer (GSV) will show you only the TIEXs representing the RAM at this point, not the RAM itself.
 - Make a TestMAX ATPG mode similar to the previous example, 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 is not only a RAM whose inout ports now drive out “X”, but also a RAM that is visible in the GSV.

Handling Duplicate Module Definitions

You can read a module definition more than one time. By default, TestMAX ATPG 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 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-T> set_rules n5 error
```

With a severity setting of error, the process stops when TestMAX ATPG 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.

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, TestMAX ATPG primitives, and other defined modules. For a list of TestMAX ATPG primitives, see “ATPG Modeling Primitives Summary” in TestMAX ATPG Online Help.

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.

The following example uses TestMAX ATPG 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.

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
    
```

The following example uses Verilog primitives to implement the same functions.

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
    
```

The following example 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.

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
```

Condensing ATPG Libraries

TestMAX ATPG attempts to condense each module's functionality in a netlist into a gate-level representation using TestMAX ATPG 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 TestMAX ATPG 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 the following example illustrates the creation of a condensed library file, which is a two-step process:

1. Read in all required modules.

In the following example, 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 1: Creating a Condensed Library File

```
BUILD-T> 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-T> write_netlist parts_lib.gz -compress gzip
```

```
End writing Verilog netlist, CPU_time = 1.13 sec, \  
  File_size = 47571  
  
BUILD-T> 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.

Assertions

Assertions are user-defined library modeling checks that reduce or eliminate VCS simulation failures for library cells. These failures generally come from the modeling limitations of memories, PLLs, power controllers, and other behavioral Verilog models.

To prevent VCS simulation failures, TestMAX ATPG performs a DRC simulation specifically to verify that all assertions in instantiated cell instances are met during either the shift or capture operations or both. If an assertion is not met during DRC, TestMAX ATPG issues a DRC rule violation, such as A1, A2, or A3.

The following topics describe how to implement assertions and descriptions of each assertion:

- [Implementing Assertions](#)
- [Using Assertions with PLLs and Memories](#)
- [Assertion Descriptions](#)
- [Limitations](#)

Implementing Assertions

Assertions are defined on the input signals of select library models in the Verilog library. For example, you can configure TestMAX ATPG DRC to check that signals are held at 0, 1, and either 0 or 1 but not X during shift and/or capture operations.

The following example shows how to define two assertions: `tmax_shift_assert_0 SD` and `tmax_capture_assert_0 SD`. The first assertion sets a check on the SD input signal of all A1_256X8 instances to remain at 0 during the shift operation. The second assertion sets a check on the SD input signal of all A1_256X8 instances to remain at 0 during the capture operation.

```
`celldefine
```

Chapter 10: Design Netlists and Library Models Assertions

```

`define read_write new

module A1_256X8 ( Q, CLK, CEN, WEN, A, D, EMA, SD, PUDELAY_SD );
  parameter BITS = 8;
  parameter word_depth = 256;
  parameter addr_width = 8;
  output [BITS-1:0] Q;
  input CLK;
  input CEN;
  input WEN;
  input [addr_width-1:0] A;
  input [BITS-1:0] D;
  input [2:0] EMA;
  input SD;
  output PUDELAY_SD;

  reg [BITS-1:0] mem [word_depth-1:0];
  reg [BITS-1:0] Q;

  and u0 (read_en, !CEN);
  and u1 (write_en, !WEN, !CEN);

  `tmax_shift_assert_0 SD
  `tmax_capture_assert_0 SD

  buf ub_SD (b_SD, SD);
  buf ub_PUDELAY_SD (PUDELAY_SD, b_SD);

  always @ (posedge CLK)
    if (write_en)
      mem[A] = D;
  always @ (posedge CLK)
    if (read_en)
      Q = mem[A];

endmodule
`undef read_write
`endcelldefine

```

A violated assertion causes a DRC A rule violation. For example, if the

``tmax_shift_assert_0 SD` assertion fails during DRC, the following rule violation is displayed:

```
A1: Assert 0 during shift on Gate XYZ failed (A1-1)
```

For a complete list of all DRC A rule violations, see [Category A – Assertion Rules](#).

If you do not apply a proper constraint or design change to avoid an assertion failure prior to ATPG, the outputs of all instances where the assertion is violated are masked. For example, a RAM with a failing assertion causes it to generate Xs during ATPG.

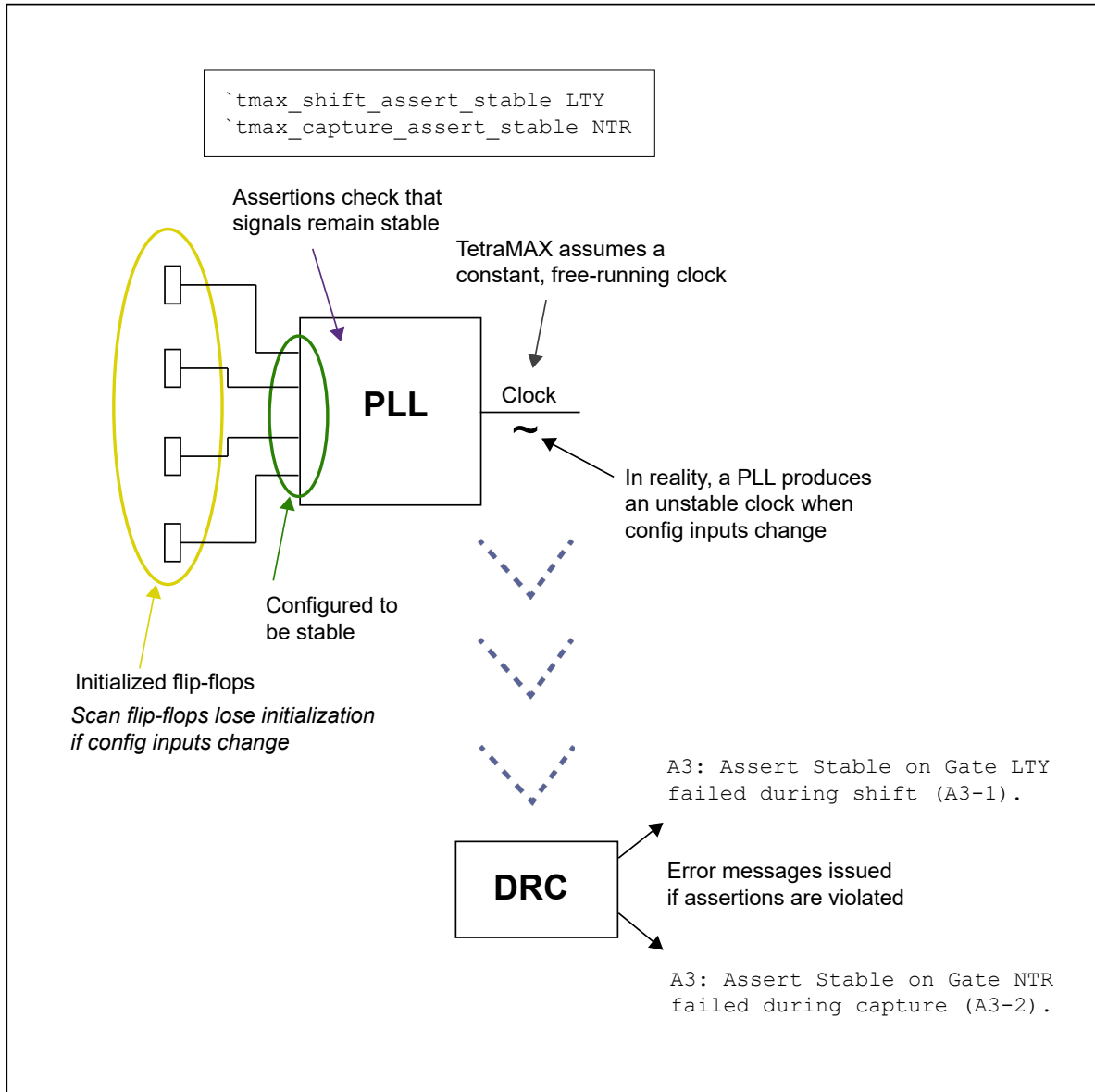
Note: Downgrading an error to a warning affects the behavior of ATPG and propagates an X from the output of the cell.

Using Assertions with PLLs and Memories

TestMAX ATPG cannot directly read and interpret behavioral models. Instead, the tool makes certain assumptions that do not always match a VCS simulation. For example, the output of a PLL must always be a free-running clock. However, a PLL produces an unstable clock anytime its configuration inputs change during test. If these signals change because of a design error, the error may not be discovered until late in the design process.

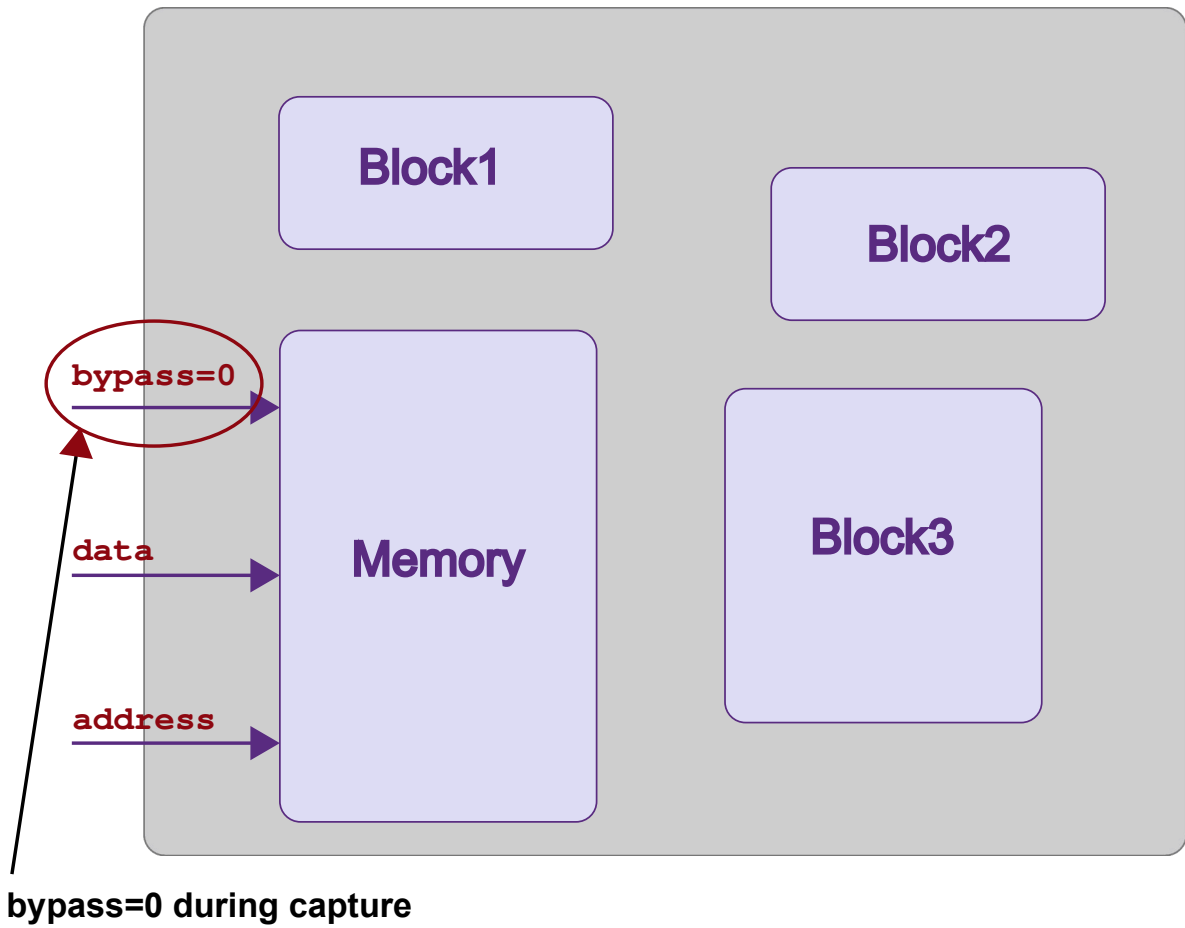
To formally express the intended PLL behavior, you can use the

``tmax_shift_assert_stable` and ``tmax_capture_assert_stable` assertions to set particular library cell to remain stable throughout the shift or capture operations.

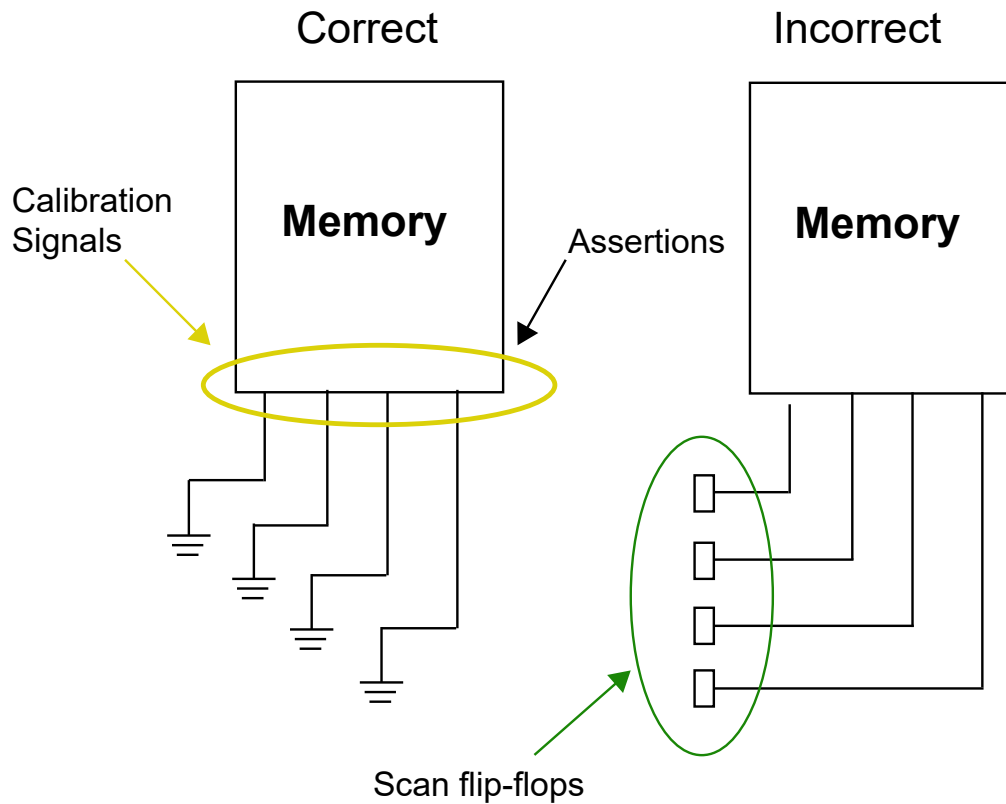


Memories also require special handling. You can use a mux to model the bypass behavior of a memory. If a mux switches during shift, TestMAX ATPG always ignores it. A memory model can potentially invalidate the memory by replacing all content with X values. In this case, you can use an assertion to ensure that the bypass is constant during shift, as shown in the following example and figure.

```
`tmax_capture_assert_0 bypass
```



The following figure shows correct and incorrect implementations of memories for using assertions:



Assertion Descriptions

Assertions can apply to either shift or capture operations, or both.

You can also specify assertions that check an X value from driving any instance of a cell during a shift or capture operation.

Shift assertions are described in the following table:

Assertion	Description
<code>`tmax_shift_assert_0 lib_cell</code>	Each instance of the specified library cell is checked to always be set to 0 during shift.

Assertion	Description
<code>`tmax_shift_assert_1 lib_cell</code>	Each instance of the specified library cell is checked to always be set to 1 during shift.
<code>`tmax_shift_assert_stable lib_cell</code>	Each instance of the specified library cell is checked to always be set to either 0 or 1 throughout shift. The value must be the same for all patterns. If this assertion is used with the <code>`tmax_capture_assert_stable</code> assertion, the stability value must be the same for both shift and capture across all patterns
<code>`tmax_shift_assert_not_X lib_cell</code>	Each instance of the specified library cell is checked so that it cannot be driven by an X value throughout shift

Capture assertions are described in the following table:

Assertion	Description
<code>`tmax_capture_assert_0 lib_cell</code>	Each instance of the specified library cell is checked to always be set to 0 during capture.
<code>`tmax_capture_assert_1 lib_cell</code>	Each instance of the specified library cell is checked to always be set to 1 during capture.
<code>`tmax_capture_assert_stable lib_cell</code>	Each instance of the specified library cell is checked to always be set to either 0 or 1 throughout capture. The value must be the same for all patterns. If this assertion is used with the <code>`tmax_shift_assert_stable</code> assertion, the stability value must be the same for both shift and capture across all patterns.
<code>`tmax_capture_assert_not_X lib_cell</code>	Each instance of the specified library cell is checked so that it cannot be driven by an X value throughout capture.

The following table describes the shift and capture stable assertion:

Assertion	Description
<code>`tmax_assert_stable libCell</code>	Each instance of the specified library cell is checked so that it must always be set to either 0 or 1 throughout shift and capture. The value must be the same for all patterns, and the stability value must be the same for both shift and capture across all patterns.

Limitations

The following limitations apply to assertions:

- Image files are not supported
- You cannot apply assertion directives on single bit of BUS ports
- Conditional assertions are not supported for IDDQ testing patterns or mixed shift assertion (SA) patterns that state that only two-clock patterns are tested and not basic SA patterns.

Memory Modeling

You can define RAM and ROM models using a simple Verilog behavioral description. The following sections describe memory modeling:

- [Memory Model Functions](#)
- [Basic Memory Modeling Template](#)
- [Initializing RAM and ROM Contents](#)
- [Improving Test Coverage for RAMs](#)

Memory Model Functions

Memory models can have the following 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 reset capability
- Memory initialization files

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

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.

TestMAX ATPG 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 “TestMAX ATPG Memory Modeling” topic in the Online Help. 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.

Basic Memory Modeling Template

The following example is a basic template for a 16-word by 8-bit RAM that can be applied to a ROM.

Basic Memory Modeling Template

```
module spec_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-through
    ...memory port definitions...
endmodule
```

The template consists of a Verilog module definition in which you make the following definitions:

- 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 might 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

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 TestMAX ATPG, 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. The following example shows a sample hexadecimal memory initialization file.

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 “TestMAX ATPG Memory Modeling” topic in Online Help.

Default Initialization

To establish the default memory initialization file, specify its file name in the module definition of the ROM. The following example defines a Verilog module for a ROM that has 16 words of 16 data bits.

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] specmem [0:addrmax];
reg [databits-1:0] dout ;
initial $readmemh("rom_init.dat", specmem);
always @ ren if (ren) dout <= specmem[a] ;
endmodule
```

The `initial $readmemh` statement in this example indicates that the data in the `rom_init.dat` file is used to initialize the memory core `specmem`. The `$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 `specmem` core. This order is based on how you define the `specmem` index, as follows:

- The format `specmem[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 `specmem[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 the following example, the following line indicates that the first data word is loaded into address 0 and the last data word is loaded into the address specified by `addrmax`:

```
reg [databits-1:0] specmem [0: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.

1. To use the Read Memory File dialog box to override the memory initialization file specification in the module definition for a specific ROM instance, perform the following steps:
2. From the menu bar, choose the Primitives > Read Memory File. The Read Memory File dialog box appears.
3. Enter the instance and then enter or browse to the memory initialization file.

For more information about the controls in this dialog box, see Online Help for the `read_memory_file` command.

4. Click OK.

You can also override the memory initialization file specification in the module definition using the `read_memory_file` command. For example:

```
DRC-T> read_memory_file i007/u1/mem/rom1/rom_core i007.d3 -hex
```

The following example indicates that the instance `/TOP/BLK1/rom1/rom_core` is to be initialized using the hexadecimal file `U1_ROM1.dat`.

```
DRC-T> read_memory_file /BLK1/rom1/rom_core U1_ROM1.dat -hex
```

In responding to the `read_memory_file` command, TestMAX ATPG 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.

Improving Test Coverage for RAMs

Test patterns for RAMs intrinsically require more clock cycles than most other types of tests. Also, a RAM usually requires the justification of considerably more values (all address bus bits, data bus bits, and enable signals) than most combinational gates. In addition, the behavior of RAMs is more complex than the behavior of other circuit elements, which may increase the difficulty of getting tests for these faults.

TestMAX ATPG minimizes the complexity of memory test generation by separating the various memory operations into different scan chain loads. For example, if a test for a RAM fault involves two write operations and one read operation, TestMAX ATPG will generally do the following:

1. Scan chain load 1
2. Write operation 1
3. Scan chain load 2
4. Write operation 2

5. Scan chain load 3

6. Read operation

A RAM must be load stable to make use of multiple scan chain loads. This means all RAM operations must be disabled during the scan chain load-unload procedure. You can do this by gating the RAM clock with the scan-enable signal or by turning off the RAM enable signals (including Chip Select, if such a signal exists) during the scan chain load. A load-stable RAM enables TestMAX ATPG to maximize its efficiency when generating tests for RAM faults, however the tool still cannot generate tests for all RAM faults.

11

STIL Procedures

The STIL language describes scan-shifting protocol, test procedures, and ATPG signal, timing, and data information. STIL procedures provide information TestMAX ATPG uses as a basis to perform design rule checking (DRC).

You can provide a set of STIL procedures to TestMAX ATPG through a file, called STIL procedure file (SPF). You can use an existing SPF written by a tool, such as DFT Compiler, or you can create a new SPF. TestMAX ATPG supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data as part of the STIL procedure file definitions. If you use an existing SPF, make sure it meets the parameters recognized by TestMAX ATPG, as described in [STIL Language Support](#).

If you create an SPF, you can initially define the minimum information needed by TestMAX ATPG to run DRC. If you are using the TestMAX ATPG GUI, you can provide this information via the QuickSTIL tab in the DRC dialog box.

The following sections describe the guidelines for using STIL procedures:

- [STIL Procedure File Guidelines](#)
- [Creating a New STIL Procedure File](#)
- [Defining STIL Procedures](#)
- [Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller](#)
- [Specifying Internal Clocking Procedures](#)
- [JTAG/TAP Controller Variations for the load_unload Procedure](#)
- [Multiple Scan Groups](#)
- [DFTMAX Compression with Serializer](#)

STIL Procedure File Guidelines

TestMAX ATPG can read and write a properly formatted SPF. Any STIL files written by TestMAX ATPG contain an expanded form of the minimum information and may also contain pattern and response data produced by the ATPG process. After an SPF is generated for a design, TestMAX ATPG can read it again at a later time to recover the

clock, constraint, and chain data, and the pattern and response data, or both. You can also use several TestMAX ATPG commands to supplement or provide the same or similar information as STIL procedures.

The following general guidelines, tips, and shortcuts help you efficiently and accurately work with STIL procedure files:

- To save time and avoid typing errors, use the `write_drc_file` command to create the STIL template. The more information that you provide to TestMAX ATPG before the `write_drc_file` command, the more TestMAX ATPG will provide in the template. If possible, build your design model and define all clock and constrained inputs before you create the STIL template.
- 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 TestMAX DFT 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; } }
```

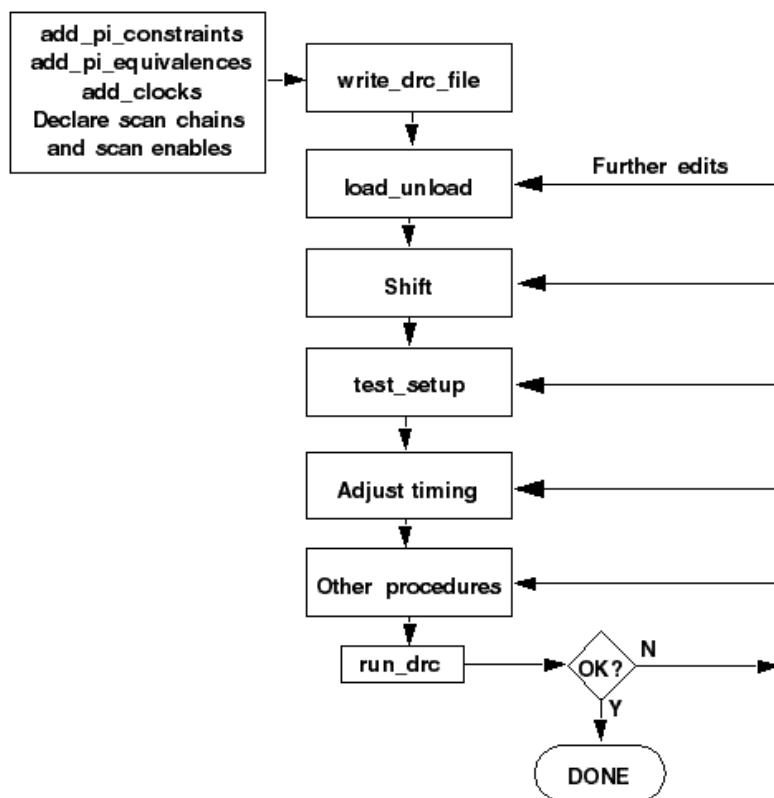
Creating a New STIL Procedure File

To create a new STIL procedure file, you first need to define the primary input (PI) constraints, clocks, and scan chain information using a series of TestMAX ATPG commands. You can then use the `write_drc_file` command to create a STIL template file, and edit this file to define the required STIL procedures and port timing.

The following sections describe how to create an STL procedure file with no prior input:

- [Declaring Primary Input Constraints](#)
- [Declaring Clocks](#)
- [Declaring Scan Chains and Scan Enables](#)
- [Writing the SPF Template](#)

Figure 1 Flow for Creating an Initial STL Procedure File With No Prior Input



See Also

- [Defining STIL Procedures](#)

Declaring Primary Input Constraints

In most design-for-test (DFT) scenarios, a design shifts into ATPG mode based on the top-level ports. The success of the ATPG algorithm usually requires that these ports are held to a constant state.

You can use STIL procedures to force a constrained port to a state other than the requested constrained value for a limited number of tester cycles, and then return the port to its constrained value. For example, you might want to hold a global reset port to an off state for general ATPG patterns, but then allow it to be asserted to initialize the design (for more information, see [Defining the test_setup Macro](#)).

You can declare a port using the Add PI Constraints dialog box in the TestMAX ATPG GUI, the `add_pi_constraints` command, or by defining it in the STL procedure file. For more information on using the STL procedure file to define PI constraints, see [Defining Constrained Primary Inputs](#).

The following sections show you how to use TestMAX ATPG to declare primary input constraints:

- [Using the Add PI Constraints Dialog Box](#)
- [Using the `add_pi_constraints` Command](#)

A port that enables a test mode for a design is different from the `scan_enable` port 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 use the `add_pi_constraints` command to declare a PI constraint. For example:

```
DRC-T> add_pi_constraints 1 TEST_MODE
```

Declaring Clocks

You can declare a port as a clock only if the port affects the state of flip-flops and latches or controls RAM/ROM read or write ports. You declare a clock in terms of its natural off state. 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 in the TestMAX ATPG GUI using the Add Clocks dialog box, the Edit Clocks dialog box, or the DRC dialog box. You can also use the `add_clocks` command to declare a clock, or edit the timing block in the STL procedure file. For more information on declaring clocks in the timing block of the STL procedure file, see [Defining Basic Signal Timing](#).

The following sections show you how to declare clocks:

- [Using the Edit Clocks Dialog Box](#)
- [Using the `add_clocks` Command](#)
- [Asynchronous Set and Reset Ports](#)

Using the Edit Clocks Dialog Box

To declare clocks using 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 defaults 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-T> add_clocks 0 CLK1 -timing 200 50 80 40 -unit ns -shift
```

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, TestMAX ATPG 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. See [Cells With Asynchronous Set/Reset Inputs](#) for details.

Declaring Scan Chains and Scan Enables

You can use the DRC dialog box in the TestMAX ATPG GUI or enter a command at the command line to declare the scan chains and scan enable inputs. You can also declare scan chains in the STIL Procedure file, as described in [Defining Scan Chains](#).

The following sections describe how to declare scan chains and scan enables in TestMAX ATPG:

- [Using the DRC Dialog Box](#)
- [Declaring Scan Chains at the Command Line](#)

Using the DRC Dialog Box

To use the DRC dialog box to declare scan chains:

1. Click the DRC button in the command toolbar at the top of the TestMAX ATPG 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.

If you select the Clocks view, the Edit Clocks dialog box appears, as described in [Declaring Clocks](#).

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 running the scan chain and scan enable information, click OK.

Declaring Scan Chains at the Command Line

You can use the following commands to declare, report, and remove scan chains and scan enables at the command line:

- `add_scan_chains`
- `add_scan_enables`
- `report_scan_chains`
- `report_scan_enables`
- `remove_scan_chains`
- `remove_scan_enables`

Writing the SPF Template

You can create an SPF template file after executing the `run_build_model` command. This template includes all clocks, PI equivalences, PI constraints, or scan chain information you have previously specified.

To create an SPF template from the TestMAX ATPG GUI:

1. Click the DRC button in the command toolbar at the top of the TestMAX ATPG GUI main window.

The DRC dialog box appears.

2. Click the Write tab in the DRC dialog box.
3. In the Name field, enter the name of the STIL procedure file you want to create.
4. Click the Write button.

The following example shows how to create a STIL template using the `write_drc_file` command:

```
write_drc_file template.spf
```

Example SPF Template File

The following example shows an STL procedure file template file created from the `write_drc_file` command:

```
STIL 1.0 {
    Extension Design P2011;
}
Header {
    Title " TestMAX ATPG  2010.06-i000622_173054 STIL output";
    Date "Wed Dec 31 17:21:05 2011";
    History { }
}
Signals {
    CLK In; RSTB In; SDI2 In; SDI1 In; INC In; SCAN In; HACKIN In; si4
In;
    six In; D0 InOut; D1 InOut; D2 InOut; D3 InOut; SDO2 Out; COUT Out;
    HACKOUT Out; so4 Out; sox Out;
}
SignalGroups {
    _pi = 'D0 + D1 + D2 + D3 + CLK + RSTB + SDI2 + SDI1 + INC +
SCAN + HACKIN + si4 + six';
    _default_Clk1_Timing_ = 'RSTB';
    _io = 'D0 + D1 + D2 + D3' { WFCMap 0X->0; WFCMap 1X->1; WFCMap
ZX->Z; WFCMap NX->N; }
    _po = 'SDO2 + COUT + D0 + D1 + D2 + D3 + HACKOUT + so4 + sox';
    _default_In_Timing_ = 'D0 + D1 + D2 + D3 + CLK + RSTB + SDI2 +
SDI1 + INC + SCAN + HACKIN + si4 + six';
```


Chapter 11: STIL Procedures

Creating a New STIL Procedure File

```

    _default_Out_Timing_ = 'SDO2 + COUT + D0 + D1 + D2 + D3 + HACKOUT
    + so4 + sox';
    _default_Clk0_Timing_ = 'CLK';
}
ScanStructures {
    # Uncomment and modify the following to suit your design
    # ScanChain chain_name { ScanIn chain_input_name; ScanOut
chain_output_name; }
}
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_Clk1_Timing_ { P { '0ns' U; '50ns' D; '80ns' U; } }
            _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; } }
        }
    }
}
PatternBurst _burst_ { PatList {
    _pattern_ {
    }
}}
PatternExec {
    PatternBurst _burst_;
}
Procedures {
    capture_CLK {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
        pulse: V { CLK=P; _po=\j \r9 X ; }
    }
    capture_RSTB {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
        pulse: V { RSTB=P; _po=\j \r9 X ; }
    }
    capture {
        W _default_WFT_;
        forcePI: V { _pi=\r13 # ; _po=\j \r9 X ; }
        measurePO: V { _po=\r9 # ; }
    }
}

# Uncomment and modify the following to suit your design

```

```
# PRE_CLOCK_MEASURE Procedures {
# load_unload {
#   W _default_WFT_;
#   C { test_so=X; test_si=0; test_si2=0; test_so2=X; clk=0; tclk=0;
reset=1; test_se=1; }
#   Shift { W _default_WFT_;
#   V { _si=#; _so=#; CLK = P; }
# }
# }
# TMAX GENERATED POST_CLOCK_MEASURE (Closer to DFTCompiler Procedures {
# load_unload {
#   W _default_WFT_;
#   C { test_si=0; test_si2=0; clk=0; tclk=0; reset=1; test_se=1; }
#   V { _so=#; }
#   Shift { W _default_WFT_;
#   V { _si=#; _so=#; clk=P; }
# }
}
MacroDefs {
  test_setup {
    W _default_WFT_;
    V { CLK=0; RSTB=1; }
  }
}
}
```

Defining STIL Procedures

There are a variety of STIL procedures you can specify in the SPF, including the `load_unload`, `shift`, and `test_setup` procedures, and capture, system capture, generic capture, and sequential capture procedures. You can also define signal timing and signal groups, scan chains, primary input parameters, PO masks, and many other parameters. Some of these settings can be specified using TestMAX ATPG commands.

If you don't have an existing SPF, see [Creating a New STIL Procedure File](#).

The following sections describe how to define STIL procedures:

- [Defining Scan Chains](#)
- [Defining the `load_unload` Procedure](#)
- [Defining the `test_setup` Procedure](#)
- [Predefined Signal Groups in STIL](#)
- [Defining Basic Signal Timing](#)
- [Defining Capture Procedures in STIL](#)
- [Defining Constrained Primary Inputs](#)

- [Defining Equivalent Primary Inputs](#)
- [Defining PO Masks](#)
- [Defining System Capture Procedures](#)
- [Creating Generic Capture Procedures](#)
- [Defining Sequential Capture Procedures](#)
- [Defining Reflective I/O Capture Procedures](#)
- [Using the master_observe Procedure](#)
- [Using the shadow_observe Procedure](#)
- [Using the delay_capture_start Procedure](#)
- [Using the delay_capture_end Procedure](#)
- [Using the test_end Procedure](#)
- [Scan Padding Behavior](#)
- [Using the Condition Statement in STIL](#)
- [Excluding Vectors From Simulation](#)
- [Defining Internal Clocks for PLL Support](#)
- [Specifying an On-Chip Clock Controller Inserted by DFT Compiler](#)

Note that STIL keywords are case-sensitive. When you enter a keyword in an STL procedure file, 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 the following sections, text strings are sometimes enclosed in quotation marks. The general rule in STIL procedure files is that quotation marks are optional unless the text string contains parentheses “()”, braces “[]”, or spaces.

Defining Scan Chains

You define scan chains in the `ScanStructures` block of the STL procedure file. In the following example, the text in bold type illustrates four scan chains. The labels “c1”, “c2”, and so forth., are the symbolic names assigned to the scan chains. The STIL specification indicates a length, but this item is optional for TestMAX ATPG input.

The following example also represents the minimum STL procedure file needed by TestMAX ATPG as it defines the scan chains, the `load_unload` procedure, and the `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#####;
            }
        }
    }
}
    
```

Defining the load_unload Procedure

The `load_unload` procedure describes how to place a design into a state in which the scan chains can be loaded and unloaded. This typically involves asserting a `SCAN_ENABLE` port, or other control line, and placing bidirectionals into a Z state. Standard DRC rules also require that ports defined as clocks must be placed in their off states at the start of the scan chain load/unload process if they are not initialized to an off state in the `test_setup` procedure.

The `load_unload` procedure is required by TestMAX ATPG. If you define the scan enable information before you write the STIL file, TestMAX ATPG automatically creates the `load_unload` procedure.

The scan chain length is required in standard STIL syntax, but is optional for STIL input files used by TestMAX ATPG. When writing a STIL pattern file, TestMAX ATPG determines the scan chain lengths and defines the correct length of each scan chain while writing STIL output.

The following example 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 `CLOCK` and `RESETB` clocks are set to their off states and the `SCAN_ENABLE` port is driven high to enable the scan chain shift paths.

Example 1: Defining Scan Chain Loading and Unloading in the STL procedure file

```

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;
        }
    }
}
    
```

See Also

- [JTAG/TAP Controller Variations for the load_unload Procedure](#)

Controlling Bidirectional Ports

During scan chain shifting defined by the load_unload procedure, the control logic for bidirectional ports sometimes operates at random states. This condition causes Z class DRC violations. You can prevent these violations by doing the following:

- Place a Z value on the bidirectional port, which turns off the ATE tester drive
- Enable a top-level control port, applied only for test mode, to globally disable all bidirectional drivers

Example 1 illustrates a design with 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 1 Controlling Bidirectional Ports in the STL Procedure File

```
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 = zzzzzzzzzzzzzzzzzzz; }
V {PLL_RESET = 0; }
V {PLL_RESET = 1; }
}
}

```

You can use both the `load_unload` procedure and the `test_setup` procedure for bidirectional control. The control mechanisms for the `load_unload` procedure are as follows:

- You can add the following lines 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.

- You can also use an empty test cycle:

```
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:

```
V{ bidi_ports = \r4 1010 ; }
```

In this case, 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 1010101010101010.

In the `test_setup` procedure, the following line can be added to the first test cycle:

```
BIDI_DISABLE = 1; bidi_ports = zzzzzzzzzzzzzzzzzzz;
```

In this case, the `BIDI_DISABLE` port is forced high and the `bidi_ports` are set to a Z state.

Defining the Shift Procedure

The `shift` procedure specifies how to shift the scan chains. It is placed within the `load_unload` procedure.

Shift is a recognized keyword to the STIL language and is not enclosed in quotation marks. The "_si" and "_so" names are predefined symbolic names used by TestMAX ATPG to represent the list of scan inputs and scan outputs. "CLOCK" is the name of a clock port that affects scan chains. More than one clock port is often required.

The bold text shown in Example 1 defines the Shift procedure.

Example 1: STL procedure file: 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 required to shift as many bits as are in the longest scan chain.

A test cycle is 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 TestMAX ATPG. They refer to the scan inputs and scan outputs. The STIL file output generated by TestMAX ATPG 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 Procedure

The `test_setup` procedure defines all initialization sequences that a design needs for test mode or to ensure that the device is in a known state.

In Example 1, the `test_setup` procedure is highlighted in bold text. This example procedure consists of three test cycles:

- The first cycle sets the inputs `TEST_MODE`, `PLL_TEST_MODE`, and `PLL_RESET` to 1
- The second cycle changes `PLL_RESET` to 0
- The third cycle returns `PLL_RESET` to 1.

Example 1: Defining the `test_setup` Macro in the STL procedure file

```
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` procedure, use the "N" STIL assignment character. An "X" character indicates that an output measure is performed and the result is masked.

You can use the `test_setup` procedure to perform several other tasks, including:

- Place a device in ATPG test mode
- Place clocks in their off states
- Initialize constrained ports
- Initialize bidirectional ports to Z
- Initialize JTAG TAP controllers
- Implement Loop statements (see the "Loop Statements" section).

Using Loop Statements

You can use loops in test_setup procedures, however you should limit their usage. If you use too many loops:

- The size of the test_setup procedure dramatically increases
- The time to simulate the clock pulses dramatically increases

You should 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) are not appropriate or necessary in the ATPG environment when a PLL clock is a black box.

Vectors can 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.

Predefined Signal Groups in STIL

A SignalGroup is a method in STIL that describes a list of pins using a symbolic label. You can use symbolic labels to reference a large number of pins without excessive typing.

TestMAX ATPG accepts the following predefined SignalGroups:

- `_in` = input pins
- `_out` = output pins
- `_io` = bidirectional pins
- `_pi` = inputs + bidirectional pins
- `_po` = outputs + bidirectional pins
- `_si` = scan chain inputs
- `_so` = scan chain outputs

If your STIL DRC description defines a symbolic group with the same name as the predefined TestMAX ATPG groups, your definition supersedes the predefined definition.

Defining Basic Signal Timing

You can define clocks and other pulsed ports, such as asynchronous sets and resets, in the STL procedure file. This is an alternative method to using the Edit Clocks dialog box in the TestMAX ATPG GUI or the `add_clocks` command (for more information, see [Declaring Clocks](#)).

You do not need to define signal timing to perform DRC or to generate patterns. However, timing definition is necessary for writing patterns that require meaningful timing. If you do not explicitly define the signal timing, TestMAX ATPG uses a set of default values.

You should avoid editing signal timing values in ATPG-generated pattern files because it causes simulation mismatches or ATE mismatches. Make sure you define signal timing in the STL procedure file and run DRC with the same STL procedure file before generating hand-off patterns with ATPG.

Example 1: STL procedure file: 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]" + "D[5]" + "D[6]" + "D[7]" + "D[8]" + "D[9]" + "D[10]" + "D[11]" +
        "D[12]" + "D[13]" + "D[14]" + "D[15]" `;
```

```

6.         input_grp1 'SCAN_ENABLE + BIDI_DISABLE + TEST_MODE +
    PLL_TEST_MODE' ;
7.         input_grp2 'SDI1 + SDI2 + DIN + "IRQ[4]"' ;
8.         in_ports 'input_grp1 + input_grp2';
9.         out_ports 'SDO2 + D1 + YABX + XYZ';
10.    }
11. Timing {
12.     WaveformTable "BROADSIDE_TIMING" {
13.         Period '1000ns';
14.         Waveforms {
15.             CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } } // clock
16.             CLOCK { 01Z { '0ns' D/U/Z; } }
17.             RESETB { P { '0ns' U; '400ns' D; '800ns' U; } } /
                / async reset
18.             RESETB { 01Z { '0ns' D/U/Z; } }
19.             input_grp1 { 01Z { '0ns' D/U/Z; } }
20.             input_grp2 { 01Z { '10ns' D/U/Z; } }
                // outputs are to be measured at t=350
21.             out_ports { HLTX { '0ns' X; '350ns' H/L/T/X; } }
                // bidirectional ports as inputs are forced at t=20
22.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
23.             // bidirectional ports as outputs are measured at t=350
24.             bidi_ports { X { '0ns' X; } }
25.             bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
26.         }
27.     } // end BROADSIDE_TIMING
28.     WaveformTable "SHIFT_TIMING" {
29.         Period '200ns';
30.         Waveforms {
31.             CLOCK { P { '0ns' D; '100ns' U; '150ns' D; } }
32.             CLOCK { 01Z { '0ns' D/U/Z; } }
33.             RESETB { P { '0ns' U; '20ns' D; '180ns' U; } }
34.             RESETB { 01Z { '0ns' D/U/Z; } }
35.             in_ports { 01Z { '0ns' D/U/Z; } }
36.             out_ports { X { '0ns' X; } }
37.             out_ports { HLT { '0ns' X; '150ns' H/L/T; } }
38.             bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
39.             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;}
    }
}

```

Chapter 11: STIL Procedures

Defining STIL Procedures

```

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 = ZZZZZZZZZZZZZZZZZZZ; }
68.         V {PLL_RESET = 0; }
69.         V {PLL_RESET = 1; }
70.     } // end test_setup
71. } //end procedures

```

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 Pulsed Ports

You can define pulsed ports for clocks and asynchronous sets and resets using the Add Clocks dialog box in the TestMAX ATPG GUI, the `add_clocks` command, or by specifying an optional section in the STL procedure file.

The bold text in Example 1 defines two pulsed ports, `CLOCK` and `RESETB` in the STL procedure file. This specification adds a `Timing{..}` section and a `WaveformTable` definition with the special-purpose name recognized by TestMAX ATPG, `_default_WFT_`.

Example 1: STL procedure file: Defining Pulsed Ports

Chapter 11: STIL Procedures

Defining STIL Procedures

```

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 = ZZZZZZZZZZZZZZZZZ; }
    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; } }
```

Note that `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.

Selecting Strobed or Windowed Measures in STIL

Some testers and vendors prefer a windowed measure for selecting timing in STIL. For this approach, the outputs are compared continuously for a window of time against the expected values instead of at a single time.

STIL supports the definition of windowed measures by using some slightly different syntax involving lowercase Waveform Events. The first following example illustrates strobed comparisons that occur at an offset of 450ns into each cycle.

```
Timing {
  WaveformTable "STROBED_COMPARE" {
    Period '1000ns';
    Waveforms {
      clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
      input_ports { 01Z { '0ns' D/U/Z; } }
      out_ports { X { '0ns' X; '450ns' X; } }
      out_ports { HLT { '0ns' X; '450ns' H/L/T; } }
      bidi_ports { X { '0ns' X; } }
      bidi_ports { 01Z { '0ns' D/U/Z; } }
      bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
    }
  }
}
```

This second example uses a windowed comparison for the group "out_ports" that compares the outputs between the offsets of 450 nS and 490 nS into each test cycle. Notice how the standard STIL WaveformChars of "H/L/T" have been replaced by lowercase STIL Waveform Events of "h", "l", "t". This indicates to TestMAX ATPG that windowed measures are required.

```
Timing {
  WaveformTable "WINDOW_COMPARE" {
```

```

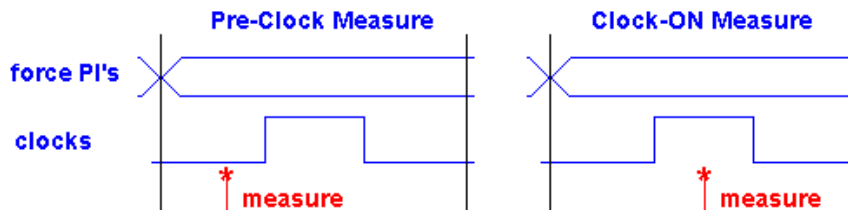
Period '1000ns';
Waveforms {
  clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
  input_ports { 01Z { '0ns' D/U/Z; } }
  out_ports { X { '0ns' X; } }
  out_ports { HLT { '0ns' X; '450ns' h/l/t; '490ns' X; } }
  bidi_ports { X { '0ns' X; } }
  bidi_ports { 01Z { '0ns' D/U/Z; } }
  bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
}
}
}

```

TestMAX ATPG supports the definition of windowed measure in the STIL timing block and if STIL or WGL output patterns are written, then this timing definition is carried into the output patterns. However, when writing Verilog or VHDL patterns, the patterns will contain a strobed measure and the call for a windowed measure is not supported and is ignored.

Supporting Clock ON Patterns in STIL

The default patterns generated by TestMAX ATPG use a preclock measure. Certain types of faults on combinational paths involving clock pins and primary outputs require a different style of pattern, called a "Clock ON" pattern, where the measure is performed during the interval in which the clock is asserted. This difference is shown graphically below and these types of faults in the design are signaled by the presence of C17 DRC violations.



TestMAX ATPG does not generate this additional style of patterns by default, because it is not supported by all testers. Your target tester must either support multiple waveform timings and dynamically switching between them on a pattern by pattern basis, or you must be willing to create patterns that contain clock-on measure and preclock measure and write them into separate pattern blocks before use (the `-type` option of the `write_patterns` command is handy for this).

- The first step necessary to support the generation of Clock-On patterns is to edit your DRC file and to create a unique timing definition to be used for the clock-on measure patterns. This is usually accomplished by copying the existing or default waveform timing and adjusting the measure time of outputs to occur within the time interval where the clock is asserted.

- After creating a unique waveform timing definition modify or create the non-clocking capture procedure named "capture" and have it reference the clock-on waveform timing.
- Next enable the generation of clock-on measure patterns by use of the `-allow_clockon_measures` option of the `set_atpg` command.
- Finally, use the modified DRC file in your `run_drc` command.

When the ATPG algorithm generates patterns, it will reference the defined waveform timing of the non-clocking capture procedure for any patterns created that require the clock-on measure. These patterns have a recognizable label when reported with the `-types` option of the `report_patterns` command.

It is also possible for the ATPG algorithm to create regular patterns that do not require a clock. If this occurs, these patterns will also reference the defined timing of the "capture" procedure. Usually only a few patterns are generated for any particular design that do not require a clock be used. These patterns should work but can increase the amount of dynamic timing switches in your tester. If this is a concern, then explore the `-clock-one_hot` option of the `set_drc` command as a way to inhibit the generation of non-clocking patterns.

The following example defines a unique timing set for use by the clock-on patterns. The timing of `CLOCK_ON` is identical to `PRE_CLOCK`, except that the measure time has been moved from 40ns (preclock) to 60ns (clock asserted).

```

:
:
Timing {
  WaveformTable "PRE_CLOCK" {
    Period '100ns';
    Waveforms {
      clocks { P { '0ns' D; '50ns' U; '80ns' D; } }
      clocks { 01Z { '0ns' D/U/Z; } }
      _in { 01ZN { '0ns' X; '40ns' L/H/T/X; } }
      _out { LHZX { '0ns' X; '40ns' L/H/T/X; } }
      _io { LHZX { '0ns' X; '40ns' L/H/T/X; } }
      _io { 01ZN { '0ns' D/U/Z/N; } }
    }
  }
  WaveformTable "CLOCK_ON" {
    Period '100ns';
    Waveforms {
      clocks { P { '0ns' D; '50ns' U; '80ns' D; } }
      clocks { 01Z { '0ns' D/U/Z; } }
      _in { 01ZN { '0ns' D/U/Z/N; } }
      _out { LHZX { '0ns' X; '60ns' L/H/T/X; } }
      _io { LHZX { '0ns' X; '60ns' L/H/T/X; } }
      _io { 01ZN { '0ns' D/U/Z/N; } }
    }
  }
}

```



```

}
:
:
capture_CLK {
    W PRE_CLOCK;
    V { _pi=\r13 # ; _po=\j \r9 X ; }
    V { _po=\r9 # ; }
    V { CLK=P; _po=\j \r9 X ; }
}

capture {
    W CLOCK_ON; // reference the alternate timing definition
    V { _pi=\r13 # ; _po=\j \r9 X ; }
    V { _po=\r9 # ; }
}
:
:\line
    
```

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 STL procedure file.

The STL procedure file in the following example illustrates the two changes that allow TestMAX ATPG 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 1: 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 Capture Procedures in STIL

Capture procedures offer the flexibility to control the timing of the force primary inputs and bidis, measure primary outputs and bidis, and, optionally, a capture operation with a functional (nonscan) clock. These three events must be in the order shown, and can be arranged in three, two, or one tester cycles (Vectors). Different capture procedures are used for each capture clock, as well as a non-clock capture procedure.

Note:

Each port can only be forced one time among all vectors in the capture procedure.

The following examples are from a design with CLK and RSTB defined as clock ports. The "capture_CLK" procedure illustrates forcing PI's, measuring PO's, and pulsing the clock in the same cycle. The "capture_RSTB" illustrates using two cycles.

```
"capture_CLK" {
  W "_default_WFT_";
  V { "_pi"="\r 12 # ; "_po"="\r 8 # ; "CLK"=P; }
}

"capture_RSTB" {
  W "_default_WFT_";
  "force_and_measure": V { "_pi"="\r 12 # ; "_po"="\r 8 # ; }
  "pulse": V { "RSTB"=P; }
}

"capture" {
  W "_default_WFT_";
  "forcePI": V { "_pi"="\r 12 # ; }
  "measurePO": V { "_po"="\r 8 # ; }
}
```

The default algorithm for combinational ATPG produces an event order of: force inputs, measure outputs, pulse clocks (optional). If you should need to produce an end-of-cycle measure or postclock measure instead of this preclock measure, you will need to use a specific 2-cycle capture procedure with the following event order: cycle 1 {force inputs, measure outputs}, cycle 2 {pulse clocks}. You will also need to adjust the defined timing on the ports. An example of this style follows:

```
"capture_CLK" {
  W "spec_timing_set";
  V { "_pi"="\r 12 # ; "_po"="\r 8 # ; }
  V { "_po"="\r 8 X ; CLK=P; }
}
```

TestMAX ATPG defaults to the first WaveformTable encountered in the file if it is not specified in the sequential_capture procedure (if 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 STL procedure file had two waveform tables, say "_first_WFT_" followed later by "_default_WFT_", and you did not list your capture clocks in the STL procedure file, then TestMAX ATPG would use "_first_WFT_" for waveform timing information.

Limiting Clock Usage

You might need 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-T> set_drc -clock TCK
```

This option restricts the ATPG algorithm to use only the specified clock for capture.

Defining Constrained Primary Inputs

You can use the STIL Procedure file to define constraints on ports. This is an alternative method to using the `add_pi_constraints` command or the Constraints menu in the TestMAX ATPG GUI.

The following example is a fragment of a STIL file in which the "F{...}" or Fixed construct is used to define a fixed port condition. The STIL specification defines that this Fixed relationship persists only within the procedure in which it occurs. A TestMAX ATPG PI constraint applies to every capture procedure. Because of this difference, you should repeat the Fixed relationship in every capture procedure. If you don't, TestMAX ATPG issues V12 warnings and continues as if the missing Fixed statements are present.

TestMAX ATPG does not support use of the "F{...}" statement in the `test_setup` or `load_unload/shift` or other procedures. You must explicitly set any ports you want held at fixed values in these procedures. In the following example, the ports `TEST_MODE` and `PLL_TEST_MODE` are explicitly set in both the `load_unload` procedure and the `test_setup` 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;
      TEST_MODE=1; PLL_TEST_MODE=0;
    }
    Shift {
      V { _si=####; _so=####; CLOCK=P; }
    }
  }
  "capture_CLOCK" {
    F { TEST_MODE=1; PLL_TEST_MODE=0; }
    8 #; "CLOCK"=P; }
    V { "_pi"=\r 12 # ; "_po"=\r
```

```

    "capture_RESETB" {
        F { TEST_MODE=1; PLL_TEST_MODE=0; }      V { "_pi"=\r 12 # ; "_po"=\r
8 #; "RESETB"=P; }
    }
    "capture" {
        F { TEST_MODE=1; PLL_TEST_MODE=0; }      V { "_pi"=\r 12 # ; "_po"=\r
8 #; }
    }
}

MacroDefs {
    test_setup {
        V { TEST_MODE=1; PLL_TEST_MODE=0; CLOCK=0; }
    }
}

```

Defining Equivalent Primary Inputs

Primary inputs that need to be held at the same values or at complementary values can be defined in the STL procedure file as an alternative to using the `add_pi_equivalences` command. Example 1 shows how to define equivalent primary inputs in the STL procedure file.

Example 1: STL procedure file: 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 PO Masks

You can use the STIL Procedure file to define masks on output port measures. The following example shows a fragment from a STIL file in which the "F{...}" or Fixed construct is used to define a masked output condition by setting the expect value to X. This Fixed

relationship definition persists only within the procedure in which it occurs, so it must be repeated in all capture procedures to properly define a PO Mask to TestMAX ATPG.

TestMAX ATPG does not support use of the "F{...}" statement in the test_setup or load_unload/shift or other procedures.

The "F{...}" statement is also used for defining PI Constraints.

```
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;
      TEST_MODE=1;
    }
    Shift {
      V { _si=####; _so=####; CLOCK=P; }
    }
  }
  "capture_CLOCK" {
    F { YOUT = X; }      V { "_pi"=\r 12 # ; "_po"=\r 8 #; "CLOCK"=P; }
  }
  "capture_RESETB" {
    F { YOUT = X; }      V { "_pi"=\r 12 # ; "_po"=\r 8 #; "RESETB"=P; }
  }
  "capture" {
    F { YOUT = X; }      V { "_pi"=\r 12 # ; "_po"=\r 8 #; }
  }
}

MacroDefs {
  test_setup {
    V { TEST_MODE=1; PLL_TEST_MODE=0; CLOCK=0; }
  }
}
```

Defining System Capture Procedures

TestMAX ATPG uses a default capture procedure that defines how a declared clock port is pulsed for a system (nonscan) test cycle. This procedure uses the naming convention `capture_clockname` (where `clockname` is the clock port name).

The default system capture procedure usually contains three test cycles that perform the following tasks:

1. Force inputs
2. Measure outputs
3. Pulse the clock/set/reset port (optional)

If you defined ports named `CLOCK` and `RESETB` to be clocks using the `write_drc_file` command, the output file contains default capture procedures similar to those shown in Example 1.

Example 1: 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 non-default timing or sequencing, copy the definitions for the capture procedures from the default output template into the `Procedures` section of your STL procedure file and edit the procedures.

TestMAX ATPG defaults to the first WaveformTable it encounters 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 named, for example, “_default_WFT_”. If your STL procedure file has two waveform tables, “_first_WFT_” and “_default_WFT_”, and you do not list your capture clocks in the STL procedure file, TestMAX ATPG uses “_first_WFT_” for waveform timing information.

The bold text in Example 2 shows some typical modifications to the capture procedure files. In this case, the three cycles are merged into a single cycle and the non-default timing is specified using the `BROADSIDE_TIMING` statement.

Example 2: Modified Capture Procedure Examples

Chapter 11: STIL Procedures

Defining STIL Procedures

```

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 = ZZZZZZZZZZZZZZZZZ; }
    V {PLL_RESET = 0; }
    V {PLL_RESET = 1; }
  }
}

```

Creating Generic Capture Procedures

This section describes how to write a set of single-cycle generic capture procedures. These procedures include: `multiclock_capture()`, `allclock_capture()`, `allclock_launch()`, and `allclock_launch_capture()`.

Generic capture procedures offer the following advantages:

- A single cycle capture procedure is efficient because it matches the event ordering (force PI, measure PO, pulse clock) in TestMAX ATPG without any manual modifications.
- Stuck-at and at-speed ATPG can 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.

The following topics describe how to create generic capture procedures:

- [Generating Generic Capture Procedures](#)
- [Controlling Multiple Clock Capture](#)
- [Using Allclock Procedures](#)
- [Using load_unload for Last Shift-Launch Transition](#)
- [Example Post-Scan Protocol](#)
- [Generic Capture Procedures Limitations](#)

See Also

- [Defining a Sequential Capture Procedure](#)
- [Defining a System Capture Procedure](#)

Generating Generic Capture Procedures

Generic capture procedures are generated by default when you specify the `write_drc_file` command (the `-generic_captures` option of the `write_drc_file` command is on by default). This command overrides the default-generated procedures (`capture_clockname` - except for an explicitly defined clocked capture procedure from a prior `run_drc` command. An unlocked capture procedure is not written. Also, the default timing is compatible with single-cycle capture procedures (a Z event is 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 is generated in the output file, and all procedures will reference that same timing. If you want to create multiple WaveformTables ("`_launch_WFT_`", "`_capture_WFT_`", and "`_launch_capture_WFT_`"), use the `set_faults -model transition` command or the `set_faults -model path_delay` command before the `write_drc_file` command. These command specify that the data should be generated to cover this mode of operation.

Note the following requirements and scenarios:

- You must use 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_` WaveformTables (`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 might 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 *TestMAX DFT User Guide*.)
- A two-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.

Generating QuickSTIL File Flows

There are three scenarios to carefully consider when generating a QuickSTIL file flows:

- Running stuck-at STIL procedure file generation with no generic captures creates a set of default generic captures which use the default WaveformTables. All the generic captures are defined — not just the multiclock WaveformTables. But the `allclock_*` WaveformTables are defined at this time.
- Running transition STIL procedure file generation with no generic captures creates generic captures using all of the transition WaveformTables. All the generic captures are defined — not just the multiclock WaveformTables. But the `allclock_*` WaveformTables are defined at this time. You are responsible to update the timing needed for the at-speed timing WaveformTables.
- Running transition STIL procedure file generation with generic captures already present (for example, from a stuck-at flow), will not change or update the generic captures or WaveformTables already present in the original STL procedure file. If you

want transition timing and full WaveformTables in your STL procedure file, you need to one of the following:

- Edit and copy the “_default_wft_” multiple times, and change them to the transition WaveformTables and timing needed for their design.
- Rerun DRC with the deletion of the `allclock_*` procedures, and regenerate default timing in transition mode for these procedures.

For an example that compares the different techniques, see the following figures. Note that the blue font follows the default WaveformTables, while the green font follows the at-speed WaveformTables.

Figure 87 Comparing Generic Captures Flows (Part 1)

SA-Fault Generic Captures Flow	TR-Fault Generic Captures Flow
<pre> 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' Z; } } "all outputs" { H { '0ns' Z; '40ns' H; } } "all outputs" { T { '0ns' Z; '40ns' T; } } "all outputs" { L { '0ns' Z; '40ns' L; } } "CK" { P { '0ns' D; '45ns' U; '55ns' D; } } } } </pre>	<pre> Timing { WaveformTable "launch_capture_wft_" { Period '100ns'; Waveforms { < same contents as in _default_wft_ below User to modify timing specifics > } } WaveformTable "launch_wft_" { Period '100ns'; Waveforms { < same contents as in _default_wft_ below User to modify timing specifics > } } WaveformTable "capture_wft_" { Period '100ns'; Waveforms { < same contents as in _default_wft_ below User to modify timing specifics > } } 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' Z; } } "all outputs" { H { '0ns' Z; '40ns' H; } } "all outputs" { T { '0ns' Z; '40ns' T; } } "all outputs" { L { '0ns' Z; '40ns' L; } } "CK" { P { '0ns' D; '45ns' U; '55ns' D; } } } } </pre>

Figure 88 Comparing Generic Captures Flows (Part 2)

SA-Fault Generic Captures Flow	TR-Fault Generic Captures Flow
<pre> Procedures { "load unload" { W ".default_WPT_"; C { "CK"-0; "SO"-X; "SI"-0; "SEN"-0; } Shift { W ".default_WPT_"; } V { "CK"-P; "SEN"-0; "SO"-#; "SI"-#; } } "multiclock capture" { W ".default_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock capture" { W ".default_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock launch" { W ".default_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock launch capture" { W ".default_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} </pre>	<pre> Procedures { "load unload" { W ".default_WPT_"; C { "CK"-0; "SO"-X; "SI"-0; "SEN"-0; } Shift { W ".default_WPT_"; } V { "CK"-P; "SEN"-0; "SO"-#; "SI"-#; } } "multiclock capture" { W ".default_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock capture" { W ".capture_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock launch" { W ".launch_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} "allclock launch capture" { W ".launch_capture_WPT_"; V { " pi"-\j \r7 #; " po"-\j \r1 #; }} </pre>

Controlling Multiple Clock Capture

You can control multiple clock capture by specifying a single general capture procedure, called `multiclock_capture`, in an STIL procedure 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:

- [Multiple Clock Capture for a Single Vector](#)
- [Multiple Clock Capture for Multiple Vectors](#)
- [Using Multiple Capture Procedures](#)

Multiple Clock Capture 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 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 TestMAX ATPG 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.

Multiple Clock Capture for Multiple Vectors

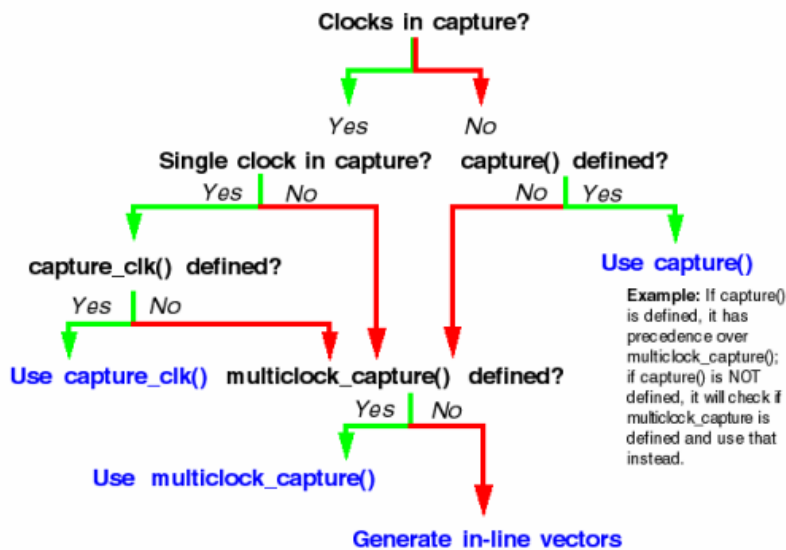
As with standard capture procedures, the `multiclock_capture` procedure can consist of multiple vectors. In this case, you need to specify an additional argument to hold the variable clock-pulse information, as shown in the following example:

```
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"= ###; }
  }
}
```

Using Multiple Capture Procedures

Figure 3 shows how the `multiclock_capture` procedure is used when other `capture()` or `capture_clk()` procedures are defined.

Figure 89 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 for use 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 use inline 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 inline vectors are used, transition and path delay timing is controlled by using fixed WaveformTable names (and not the allclock capture procedures) for full-sequential patterns.

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 apply the multiclock_capture procedure in this situation if it is present (based on the presence of other capture procedures as diagrammed in Figure 11-4), and you may define the timing of the transition capture operation from this procedure. The timing of the launch operation is defined by the last vector of the load_unload procedure for a last-shift-launch context.

TestMAX ATPG 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 following example `allclock_capture()` procedure:

```
Procedures {  
    "allclock_capture" {  
        W "TS1";  
        C { "_po"=\r9 X ; }  
        V { "_po"=\r9 # ; "_pi"=\r11 # ; }  
    }  
}
```

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 standalone 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, as shown in the following 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

The following example shows a post-scan protocol containing generic capture procedures:

```
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" = ##;
            }
        }
    }
}

```

Generic Capture Procedures Limitations

Note the following limitations related to generic capture procedures:

- WGL patterns are not supported if the multiclock_capture is multiple cycle 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, VHDL, and legacy Verilog formats do not support 3-cycle generic capture procedures.
- Using the TestMAX DFT 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.
- TestMAX ATPG 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).

Defining Sequential Capture Procedures

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. For more information on ATPG modes, see [ATPG Modes](#).

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 TestMAX 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.

The following sections describe how to define sequential capture procedures:

- [Using Default Capture Procedures](#)
- [Using a Sequential Capture Procedure](#)
- [Sequential Capture Procedure Syntax](#)

Using Default Capture Procedures

By default, all ATPG modes use the same `capture_clockname` procedures described in [Defining System Capture Procedures](#). 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. Loads scan cells, changes inputs, and measures outputs (optional)
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 to detect faults.

Using a Sequential Capture Procedure

To use a sequential capture procedure, add the `sequential_capture` procedure to the STIL file, then set the `-clock -seq_capture` option of the `set_drc` command, as shown in the following example:

```
DRC-T> 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. Sequential Capture Procedure in STIL for more information.

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 (might 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;}
```

The following example shows 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. TestMAX ATPG adds more time frames only as necessary to model discrete clock pulse events. It strobos outputs in no more than one of the existing time frames for each cycle.

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 identify which TestMAX ATPG port acts as the global bidirectional control 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 the following command:

```
DRC-T> 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 1, 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. TestMAX ATPG understands the correspondence required for proper pattern data.

Example 1: 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; }
}

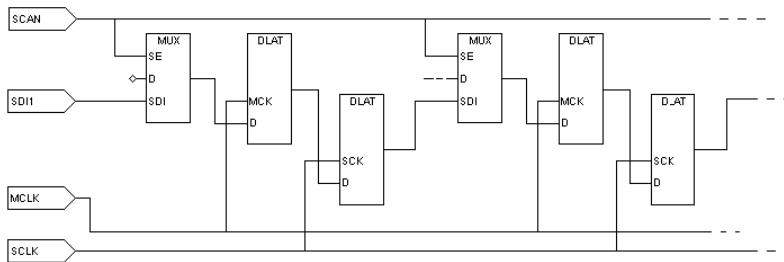
capture_RESETB {
    W "_default_WFT_"; // force PI, measure PO, BIDI_EN=1
    V { "_pi"="\r10 # ; "_po"="%%%%%%%% ; } // disable bidis, mask
PO measures
    V { BIDI_EN=0; "_po"="XXXXXX; } // reflect bidis, pulse RESETB
    V { "_io"="%%%" ; RESETB=P; }
}

capture {
    W "_default_WFT_";
    V { "_pi"="\r10 # ; "_po"="##### ; } // force PI, measure PO
    V { "_po"="XXXXXX; } // mask measures
    V { } // pad procedure to 3 cycles
}
```

Using 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 the following figure. 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 90 Master-Slave Scan Chain



The following example 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 `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";

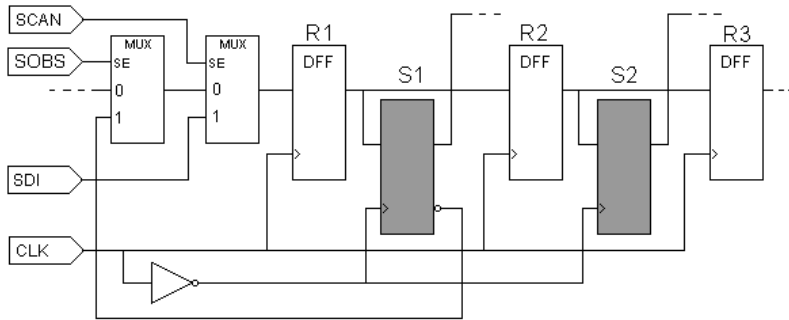
            V { _si=##; _so=##; MCLK=P; SCLK=0;}
            V { MCLK=0; SCLK=P;}
        }
        V { SCLK=0;}
    }
    master_observe {

        W "BROADSIDE_TIMING";
        V { MCLK=0; SCLK=P; RESETB=1; }
        V { SCLK=0;}
    }
}
```

Using the `shadow_observe` Procedure

You use a `shadow_observe` procedure when a design has shadow registers and each shadow register output is observable at the scan cell to which it is a shadow. The following figure 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 (R1) 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 91 A Shadow Register



The following example shows a `shadow_observe` procedure that corresponds to the preceding figure. 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 `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 the `delay_capture_start` Procedure

You can use the `delay_capture_start` procedure to specify a wait period at the start of a capture operation. TestMAX ATPG inserts calls to this procedure in the patterns at the end of each shift to establish the presence of the delay before the start of the capture operation. The first PI state present in the capture operation is asserted in this procedure, otherwise the transition time of slow-propagating signals will not occur before other capture events.

There are several different methods you can use to specify the parameters of this delay:

- Specify the number of vectors contained in the `delay_capture_start` procedure
- Control the period of the WaveformTable referenced by this procedure
- Control the number of times to insert this procedure at the end of shift using the `-use_delay_capture_start` option of the `write_patterns` command

The `delay_capture_start` procedure has several format requirements to operate as a simple `wait` statement. The default form of this procedure is as follows:

```
"delay_capture_start" {  
W "_default_timing";  
C { "_po"=\rn X; }  
V { "_pi"=\rm #; }  
}
```

Note the following:

- This procedure must contain a `Condition` statement that sets the `_po` to `X` at the start of the procedure. This ensures that any potential measure contexts at the end of the last operation are reset.
- This procedure must contain a call to the `_pi` group, with a parameter assignment of values, to ensure that the `_pi` states are applied at the start of the capture through this wait operation. No other signal assignment should be made in the `v` statement, as this will cause unpredictable results when the STIL patterns are used.
- The default form of this procedure calls the current default WaveformTable defined in the flow, and contains a single `Vector` statement. If this procedure is not defined in the incoming STIL procedure file, this default form is generated with the first use of the `-use_delay_capture_start` option of the `write_patterns` command. In this situation, this procedure is present in the `Procedure` block for all subsequent `write_patterns` or `write_drc_file` commands, although it will only be applied in the patterns if `-use_delay_capture_start` is specified.
- When the pattern set is written for transition patterns, the `set_delay -nopi_changes` command inserts one leading delay cycle in the `delay_capture_start` operation and uses the `multiclock_capture` procedure to set the PI states into all transition capture operations. Therefore, the pattern set will already have one delayed capture start event present. The `delay_capture_start` procedure calls are inserted after the number of requested delays exceed the number already present in the pattern data. If you require additional delays beyond those already present in the patterns, you need to set the `delay_capture_start` calls to a number greater than 1.

- If you define the `delay_capture_start` procedure in your STIL procedure file, it is present or defined in all subsequent STIL and WGL patterns that are written out.
- If the `delay_capture_start` procedure is not defined in the STIL procedure file, then the first time that `write_patterns -use_delay_capture_start` is specified, it is created and defined in all STIL patterns that are written out. After it is created from the `write_patterns -use_delay_capture_start` command, this procedure will function just as if it were specified in the STIL procedure file. However, it will not be called in the patterns unless the `write_patterns -use_delay_capture_start` command is specified.
- The `delay_capture_start` procedure calls are eliminated when patterns are read back into TestMAX ATPG. Each `write_patterns` command must use the `-use_delay_capture_start` option in order for this procedure present. While this procedure is not present on the internal pattern data, the presence of these function calls, and the generated vectors due to these procedures, are still counted during the pattern read-back operation. This allows cycle-based diagnostic flows to function with no changes. When the patterns are rewritten, you must set the `-use_delay_capture_start` option properly for every pattern write operation.
- If On-Chip Clock (OCC) controllers are used, they may be triggered by the transition of the scan-enable signal at the beginning of the first `delay_capture_start` procedure. In this case, the internal clocks will pulse following the OCC controller latency, so their clock pulses may occur during the `delay_capture_start` procedures.

Using the `delay_capture_end` Procedure

You can use the `delay_capture_end` procedure to specify a wait period at the end of a capture operation. You will need to insert calls to this procedure in the patterns at the end of each capture to establish the presence of the delay before the start of the next LOAD operation.

There are several ways you can specify the parameters of this delay:

- Specify the number of vectors contained in the `delay_capture_end` procedure
- Control the period of the WaveformTable referenced by this procedure
- Control the number of times to insert this procedure at the end of capture by using the `-use_delay_capture_end` option of the `write_patterns` command.

The `delay_capture_end` procedure has several format requirements that enable it to operate as a simple wait statement. The default form of this procedure is as follows:

```
"delay_capture_end" {  
  W "_default_timing";  
  C { "_po"="\r\n X; }  
}
```

```
V{ "_pi"=\rm #; }  
}
```

Note the following:

- If you define the `delay_capture_end` procedure in your STIL procedure file, it is present or defined in all subsequent STIL and WGL patterns that are written out.
- If the `delay_capture_end` procedure is not defined in the STIL procedure file, then the first time that `write_patterns -use_delay_capture_end` is specified, it is created and defined in all STIL patterns that are written out. After it is created from the `write_patterns -use_delay_capture_end` command, this procedure will function just as if it were specified in the STIL procedure file. However, it will not be called in the patterns unless the `write_patterns -use_delay_capture_end` command is specified.
- The `delay_capture_end` procedure calls are eliminated when patterns are read back into TestMAX ATPG. Each `write_patterns` command must use the `-use_delay_capture_end` option in order for this procedure present. While this procedure is not present on the internal pattern data, the PRESENCE of these function calls, and the generated vectors due to these procedures, are still counted during the pattern read-back operation. This allows cycle-based diagnostic flows to function with no changes. When the patterns are rewritten, you must set the `-use_delay_capture_end` option properly for every pattern write operation.

Using the `test_end` Procedure

You can define the `test_end` procedure or macro in the `Procedures` or `MacroDefs` sections of the STIL procedure file so that it is called at the end of every pattern block that is written out. This procedure must contain only signal drive assignments; measures are not supported.

When you define the `test_end` procedure or macro, TestMAX ATPG places it at the end of STIL and WGL-formatted patterns only. When STIL or WGL patterns are read back, this procedure is removed. It will not be included in the internal pattern data. If patterns are rewritten, then the `test_end` procedure must be present in the STIL procedure file to place (or replace) it at the end of any new patterns.

```
MacroDefs  
{ "test_setup"  
  
  { W "_default_WFT_"; V \{ "CLK"=P; }  
  V { "CLK"=0; } }  
  "test_end"  
  
  { W "_default_WFT_"; V \{ "CLK"=0; "D1"=1; }  
  V { "CLK"=0; "D1"=0; } } }
```

Scan Padding Behavior

When scan chains are of unequal lengths and shifted in parallel, the shorter scan data must be padded or extended during the time after the short chain is exhausted but the shift procedure (or pattern operation) continues to complete the shifting of the longest chain.

Some TestMAX ATPG output formats allow you to control the padding state from command-line options. For example, the combination of the `set wgl -pad` and `write patterns -pad_character` commands control padding values for WGL files.

The STIL environment defines the padding values directly from the procedure definitions. For instance, for the `load_unload` procedure, the last assigned state, even if it was not applied in a Vector (it might have only been defined in a Condition statement) before the Shift block or first statement that contains an assignment of '#' to a scan signal, is the value used to pad that signal. If the scan signals are not assigned values before the Shift block, then when the `load_unload` procedure is written out, the inputs are assigned '0' and the outputs assigned 'X'. These defaults are sufficient for most environments, but sometimes additional data is specified in the procedure that might affect the padding behavior. When this occurs, it might become necessary to assign explicit values to signals in order for the STIL file to have the expected behavior.

Several DRC messages might be generated when STIL padding issues are detected in the patterns. These messages are all warnings, because consistency of the STIL data might not be a concern if your flow uses a different format. These warnings are either V12 (unexpected item) or V14 (missing state) messages. All of these messages contain the text "STIL scan pad", to indicate they are being generated for STIL scan padding issues.

Certain design situations (for instance, reusing scan signals on multiple scanchains and making use of scan groups) limits the ability of these messages to detect all error conditions. Assigning an X to the scan outputs before the Shift block will define a correct test program, and is the most direct path to fixing padding problems.

TestMAX ATPG tri-state checks might require that bidirectional scan outputs be assigned a Z WaveformCharacter, to trace a scan chain properly. This Z value enforces that the bidirectional output values are visible during the shift operation. However, during the scan operation it is likely that an X WaveformCharacter would be preferable, especially for padding. The Z reference is not wrong, but it is a "drive" waveform being assigned to a bidirectional being used as an output. Some environments might not like to see this drive value on a scan output. One way to define an X for pad operation is to place this X in a Condition statement before the first assignment to a '#'. For example:

```
V { .... s01=Z; ... }
C { s01=X; }
Shift { V { s01=# ...
```

The DRC messages are generated only when a potential violation is detected. This will only happen for scan chains that are shorter than the longest chain in the shift operation.

These checks will not occur on the longest chain in the design, even if the values assigned to the scan signals of that chain are incorrect, because the longest chain will not be padded.

These messages are not generated during the STL procedure file checks, because at this point there is not sufficient analysis of the scan chains to accomplish this checking. Therefore, these DRC messages are generated later in the process. These messages are V warnings but are not generated with the STL procedure file read, so be aware that additional V messages might be generated after the STL procedure file read has completed.

The specific messages, and how to address each, are explained as follows:

- `V12, Scan output [name] is assigned a drive [state] before Shift; used as STIL scan pad`

In this circumstance, a signal identified as a scan output, has been assigned a value commonly associated with an input signal. The [state] value is one of 0, 1, or N.

In most circumstances this is easily fixed by adding a `C {}` statement before the Shift block, and assigning the bidirectional scan output to an X value (or other preferred state). This was shown in the previous example.

- `V14, Scan input [name] has no assignment before Shift, output [name] is [state]; missing STIL scan pad [input_state]`

This warning is generated when a scan-output is assigned a known state, either H or L, before the Shift block, but the scan-input is not specified. The fix is to either assign the scan-output to an X before the Shift block (as done above), or to assign the appropriate input drive value (0 or 1) to the scan-input. Be aware that the appropriate value is a function of the parity of the scan chain, as discussed in the next message.

Note that this warning occurs only when scan outputs have been specified, but scan-inputs have not been assigned. The reverse condition, when scan-inputs are specified but scan-outputs have not been specified, is not a problem because the default handling of scan-outputs will place an X on the outputs, making the environment insensitive to input states.

- `V14, Scan output [name] is assigned [state] before Shift, input [name] is [state]; wrong STIL scan pad`

`V14, Scan input [name] is assigned [state] before Shift, output [name] is [state]; wrong STIL scan pad`

These two warnings indicate the same condition, it just depends on the order of the signals in the design as to which you will see. These warnings are generated when both the scan-input and scan-output signals are assigned known values, but the scan data (and the parity of the scan chain) will cause this data to fail at test. In this circumstance, either the scan-input state must be changed, or the scan-output state,

(but, obviously, not both) or the scan-output can be assigned an X value before the Shift.

DRC checks STIL padding to validate special conditions around bidirectional signals used as scan outputs.

- V14, Scan output bidi (signal) has no assignment before Shift; Z added for scan padding

During pattern write (of STIL data), the Z assignment is added to these signals, correcting the situation. You can always override the correction (and eliminate the V14) by specifying an assignment to this scan signal before the first # in the load_unload operation.

Using the Condition Statement in STIL

You can use the Condition statement, C{...}, to define force or measure values for defaults, without immediately resulting in an action. The conditioned values are deferred from being applied until a vector statement, V{...}, is encountered.

The WaveformTable used to translate the conditions is the waveform in effect at the time of the next Vector statement, not the waveform in effect at the time of the Condition statement.

Multiple Condition statements can be defined between vector statements. The last state defined in a Condition or vector statement for each pin is the state applied to that pin on the vector statement.

A vector statement defining a value to be applied to a pin will override any value defined in any preceding Condition statements.

Condition statements are useful when setup information is available; however, if this setup is applied as a vector, then the subsequent data becomes difficult to align. A typical situation in which to use a Condition statement is to enable the scan clocks preceding a Shift operation. Condition statements are also useful at the end of a macro to set up information for the return. Note that Condition statements would not be useful at the end of procedures because procedures return to the state before the procedure call and any condition information would be discarded.

```
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" {
  C { CLOCK = 1; RESETB = 1; SCAN_ENABELE = 1;
  Shift {
    V { _si=####; _so=####; CLOCK=P; } // pulse shift clock
  }
}

MacroDefs {
  test_setup {
    V { TEST_MODE = 1; CLOCK = 0; RESETB = 1; }
  }
}
```

Excluding Vectors From Simulation

Passing a large number of loops through DRC negatively affects the performance of TestMAX ATPG. You can use the `DontSimulate` statement in the `test_setup` procedure of the STL procedure file to eliminate loops with large count values and reduce activity, such as clock pulses, in the vectors of the loop.

The following sections describe how to use the `DontSimulate` statement:

- [Using the DontSimulate Statement for Loops and Reference Clocks](#)
- [Syntax and Example for Excluding Vectors](#)

Using the DontSimulate Statement for Loops and Reference Clocks

The `DontSimulate` statement identifies a PLL initialization or synchronization block of vectors for exclusion from DRC simulation. Because most PLL models are black boxes in TestMAX ATPG, DRC results are not affected when clock pulses associated with these models are bypassed.

DRC procedures often require excessive memory resources when expanding the events of a loop. Also, the `test_setup` procedure causes excessive runtime when processing loop events, even though they do not affect the simulation. If the number of events in the `test_setup` procedure exceeds a 32-bit time value, the memory and runtime of all subsequent TestMAX ATPG operations are affected.

The `DontSimulate` statement prevents the insertion of loop events into TestMAX ATPG operations, while preserving the loops throughout the flow. This includes writing the loops in the patterns even though the loop events bypassed other TestMAX ATPG operations.

In addition to loops, the `DontSimulate` statement applies to the following clocks:

- Reference clocks that are not identified as free-running clocks (because they affect other logic in the design).
- Reference clocks with a large number of pulsed vectors during setup. These pulses are not necessary for DRC because they are only driving the black box PLL logic.

You must make sure that any other logic driven by a reference clock uses all the necessary pulses.

Syntax and Example for Excluding Vectors

To exclude vectors from simulation:

1. Add the `UserKeywords DontSimulate` construct before the appropriate `MacroDefs` block in the STL procedure file. The syntax for this statement is as follows:

```
UserKeywords DontSimulate;
```

2. Add the `DontSimulate ATPGDRC` construct before the appropriate `Loop` statement in the STL procedure file. The syntax for this construct is as follows:

```
DontSimulate ATPGDRC;
```

The following example from an STL procedure file shows a typical implementation for excluding vectors from simulation:

```
UserKeywords DontSimulate;  
MacroDefs {  
  "pll_setup" {  
    DontSimulate ATPGDRC;  
    Loop 1000 {V {clock1=P;}}  
  }  
  "test_setup" {  
    W "_default_WFT";  
    C {"all_inputs"= ...; "all_outputs" = ...;}  
    ...  
  }  
  Macro "pll_setup";  
}
```

See Also

- [Using Internal Clocking Procedures](#)

Defining Internal Clocks for PLL Support

TestMAX ATPG supports two methods for defining internal clocks for PLL support:

- From the command line using the `add_clocks` command
- From an optional `ClockStructures` block of a STIL procedure file .

The following is an example defining two internal clocks for PLL support by entering commands from the command line.

```
add_clocks 0 intclk3 -intclk -pll_source pllclk3 \  
  -cycle { 0 clock_chain/cell[3]/Q 1 \  
    1 clock_chain/cell[4]/Q 1 } \  
add_clocks 0 intclk1 -intclk -pll_source pllclk3 \  
  -cycle { 0 clock_chain/cell[1]/Q 1 clock_chain/cell[0]/Q 0 \  
    clock_chain/cell[2]/Q 1 clock_chain/cell[0]/Q 0 }
```

In the preceding example:

- `intclk3` as an internal clock with offstate 0; its PLL source is `pllclk3`; `intclk3` is pulsed in cycle 0 when cell 3 of chain `clock_chain` is 1, and is pulsed in cycle 1 when cell 4 of chain `clock_chain` is 1.
- `intclk1` as an internal clock with offstate 0; its PLL source is `pllclk3`; `intclk1` is pulsed in cycle 0 when cell 1 of chain `clock_chain` is 1 and cell 0 is 0, and is pulsed in cycle 1 when cell 2 of chain `clock_chain` is 1 and cell 0 is 0.

The following is an example showing the corresponding definitions in a STL procedure file `ClockStructures` block.

```
Signals { "refclk1" In; "refclk2" In; \  
  "pllclk1" Pseudo; "pllclk2" Pseudo; "pllclk3" Pseudo; \  
  "intclk1" Pseudo; "intclk2" Pseudo; "intclk3" Pseudo; \  
} \  
SignalGroups { "all_inputs" = '... + "refclk1" + "refclk2" + ... ` } \  
Timing { \  
  WaveformTable "_default_WFT_" { \  
    Period '100ns'; \  
    Waveforms { \  
      "all_inputs" { 01ZN { '0ns' D/U/Z/N; } } \  
      "refclk1" { P { '0ns' D; '45ns' U; '55ns' D; } } \  
      "refclk2" { P { '0ns' D; '45ns' U; '55ns' D; } } \  
    } \  
  } \  
} \  
UserKeywords ClockStructures; \  
ClockStructures { \  
  PLLStructures specpll { \  
    PLLCycles 2; \  
  } \  
}
```



```

Clocks {
    "refclk1" Reference; "refclk2" Reference;
    "pllclk1" PLL { Offstate 0 ; }
    "pllclk2" PLL { Offstate 0 ; }
    "pllclk3" PLL { Offstate 1 ; }
    "intclk1" Internal { Offstate 0; PLLSource "pllclk3";
        Cycle 0 "clock_chain/cell[0]/Q" 0;
        Cycle 0 "clock_chain/cell[1]/Q" 1;
        Cycle 1 "clock_chain/cell[0]/Q" 0;
        Cycle 1 "clock_chain/cell[2]/Q" 1;
    }
    "intclk3" Internal { Offstate 0; PLLSource "pllclk3";
        Cycle 0 "clock_chain/cell[3]/Q" 1;
        Cycle 1 "clock_chain/cell[4]/Q" 1;
    }
}
}
}
}

```

The following is a template for a generic STL procedure file ClockStructures block.

```

UserKeywords ClockStructures;
ClockStructures {
    (PLLStructures struct_name {
        (PLLCycles integer ;)
        (RefCycles integer ;)
        (Clocks {
            (sig_name <Reference | PLL| Internal> ;)*
            (sig_name <Reference | PLL| Internal> {
                (Offstate <0|1> ;)
                (PLLSource sig_name ;)
                (Cycle integer {AlwaysOn| AlwaysOff} ;)*
                (Cycle integer {net_or_pin_name <0|1>}+ ;)*
            })*
        })*
    })*
}

```

Where:

PLLCycles specifies the number of PLL clock cycles supported per load. This block is required if Cycle constructs are used. The PLLCycles block must precede all Cycle constructs.

RefCycles specifies the minimum number of system cycles each pattern must have.

Clocks defines the clocks in the PLLStructures block. The sig_name construct identifies the clock name and type. A type is required and must be one of the values shown. The Offstate construct is syntactically optional, but semantically required for all but reference clocks, and must be 0 or 1 (the offstate for reference clocks is derived from a

Waveformtable). The `PLLSource` construct is used for internal clocks and identifies the corresponding PLL clock source. The `Cycle` construct is used for internal clocks and identifies the corresponding control nets and their values.

Specifying an On-Chip Clock Controller Inserted by DFT Compiler

This section describes the process for specifying an OCC controller inserted by the `insert_dft` command in TestMAX DFT. For information on signal requirements, see the "On-Chip Clocking Support" chapter in the *TestMAX DFT User Guide*.

The following commands are used for specifying a default OCC controller inserted by DFT Compiler (*Note:* For user-defined OCC controllers, the commands are similar but will differ if the OCC controller is controlled differently):

- `add_scan_chains ...`

- `add_scan_enables 1 test_se`

TestMAX DFT uses the default pin name `test_se`, if a name is not provided.

- `add_pi_constraints 1 test_mode`

TestMAX DFT uses the default pin name `test_mode`, if a name is not provided.

- `add_pi_constraints 0 {test_se pll_reset pll_bypass}`

TestMAX DFT uses the default pin names `test_se`, `pll_reset`, and `pll_bypass` if the pin names are not provided.

- `set_drc -num_pll_cycles`

- `add_clocks 0 {port_names} -shift -timing {period LE TE measure_time}`

Use this command to specify external clocks that are controllable by ATPG.

- `add_clocks 0 {port_names} -shift -refclock -timing {period LE TE measure_time}`

Use this command to specify ATE and reference clocks with the same period as the shift clock.

- `add_clocks 0 {port_names} -refclock -ref_timing {period LE TE }`

Use this command to specify reference clocks with different periods than the shift clock.

- `add_clocks 0 {pin_names} -pllclock`

Use this command to specify the PLL clocks.

- `add_clocks 0 pin_name -intclock -pll_source node_name -cycle ...`

Use this command to specify the internal clock and the PLL source clock.

- `write_drc_file file_name`

In addition to using these commands, you will need to make the following changes from the output STIL procedure file created by the `write_drc_file` command 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, and specify only the ATE clocks and reference clocks with the same period:

```
V { "clkate"=P; "clkref0"=P; }
```

5. In `test_setup`, copy the `v` statement and do the following:
 - Change the polarity of the PLL reset constraint in the first `v` statement (the PLL reset is the same port identified as `pll_reset` in the previous command list).
 - Change `0` to `P` for all the ATE clocks and synchronous reference clocks in both `v` statements (these are exactly the same clocks specified in Step 4).
6. Change the timing of the WFTs as required. This can be done in an editor, or you can specify another TestMAX ATPG run and use the `update_wft` and `update_clock` commands.

Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller

You can use the `ClockTiming` block to implement synchronized internal clocks at one or multiple frequencies in an OCC Controller. The `ClockTiming` block is placed in the top level of the `ClockStructures` block that already describes other aspects of the internal clocks.

The following sections show you how to specify synchronized internal clocks at one or multiple frequencies in an OCC Controller:

- [ClockTiming Block Syntax](#)
- [Timing and Clock Pulse Overlapping](#)
- [Controlling Latency for the PLLStructures Block](#)
- [ClockTiming Block Selection](#)
- [ClockTiming Block Example](#)

For more information on this feature, see the [Using Synchronized Multi Frequency Internal Clocks](#) section.

ClockTiming Block Syntax

The syntax and location of the `ClockTiming` block is as follows, with instance-specific input in *italics*, optional input in `[squarebrackets]` and mutually-exclusive choices separated by a `|` pipe symbol:

```
ClockStructures [name] {
    PLLStructures name {
        // The contents of the PLLStructures block are unchanged,
        // except for the addition of the optional Latency statement.
    }
    [PLLStructures name2 {
        // Multiple PLLStructures blocks are possible, and have a specific
        // meaning.
        // See the section PLLStructures Block and Latency.
    }]
    ClockTiming name {
        SynchronizedClocks name {
            Clock name { Location "internal_clock_signal"; Period
                'time';
                [Waveform 'rise' 'fall'];}
            [Clock name2 { Location "internal_clock_signal2"; Period
                'time2';
                [Waveform 'rise2' 'fall2'];}]
        }
        // Multiple Clocks can be defined within a SynchronizedClocks block.
    }
}
```

```

// These Clocks are considered to be synchronized to each other.
// Note that each clock's Location value is used, not its name.
    [MultiCyclePath number [Start|End] {
        [From clocklocation;]
        [To clocklocation2;]
    }]
// As many MultiCyclePath blocks as needed might be defined.
// All clocks inside them must be in the current SynchronizedClocks group.
    }
    [SynchronizedClocks name2 {
// Multiple SynchronizedClocks blocks can be defined within a ClockTiming
    block
// These SynchronizedClocks are considered to be asynchronous to each
    other
// The Clocks defined in each SynchronizedClocks group must be different.
    }]
    }
[ClockTiming name2 {
// Multiple ClockTiming blocks can be defined, but only one is used.
// The Clocks must be defined again in each ClockTiming block.
// See the ClockTiming Block Selection section.
    }]
}

```

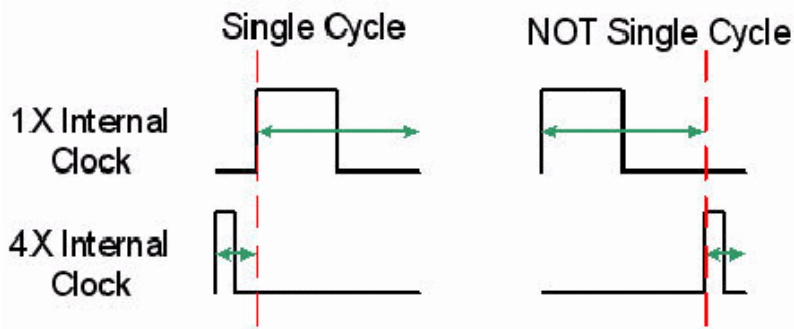
Note the following:

- The `ClockTiming` name is arbitrary and is only used by the `set_drc -internal_clock_timing` option. See [ClockTiming Block Selection](#) for details.
- The `SynchronizedClocks` name and `Clock` name are arbitrary.
- The `Location` argument must be identical to the name of the internal clock source defined in one of the `PLLStructures` blocks (see the previous example).
- The `Period` and `Waveform` times are either ns or ps. If they are defined as ps but the rest of the STL procedure file is in ns, they are converted to ns and the fractional part truncated. For example, 1900 ps is converted to 1 ns.

Timing and Clock Pulse Overlapping

The `Waveform` values and `MultiCyclePath` blocks are optional, and ATPG can use different clocks safely for launch and capture without them. However, different frequency clock pulses are not allowed to overlap if they are missing. Figure 1 illustrates synchronized clocking without overlapping pulses.

Figure 92 Non-Overlapping Synchronized Internal Clock Pulses

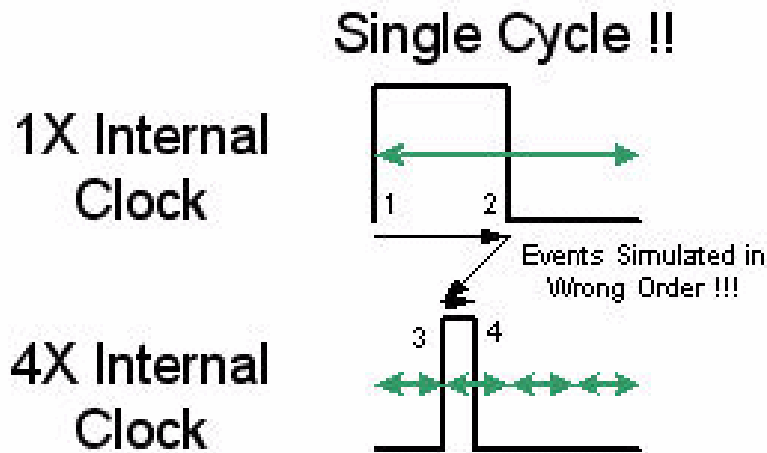


The criteria for allowing synchronized clocks of different frequencies to have overlapping pulses, which in turn allows single-cycle transition fault testing from the slower to the faster clock domain, are as follows:

- The `Waveform` values must be specified for both clocks.
- A `MultiCyclePath 1` block must be specified from the slower clock to the faster clock.
- For non-integer Period ratios, a `MultiCyclePath 1` block is needed for both directions.

Figure 2 illustrates synchronized clocking with overlapping pulses from the slower clock to the faster clock. For the other direction, from faster clock to slower clock, there is no difference between the overlapping and non-overlapping cases.

Figure 93 Overlapping Synchronized Internal Clock Pulses



TestMAX ATPG generates and simulates patterns with both edges of the first clock pulse preceding either edge of the second clock pulse. Clock pulse overlapping can change this timing relationship. If this situation also changes the behavior of the circuit when simulated on a timing simulator, then the patterns will mismatch. This will occur when trailing-edge or mixed-edge clocking is used, and paths exist from the fast clock to the trailing-edge of the slow clock. If these paths exist, TestMAX ATPG will prevent overlapping of the clock pair.

Controlling Latency for the PLLStructures Block

The number of `PLLStructures` blocks that are specified affects clock latency, which in turn affects the required length of the clock chain. Latency can be controlled using the following top-level statement in the `PLLStructures` block:

```
Latency number;
```

The default latency is 5. This number refers to the number of pulses of the `PLLClock` that must pulse before the first internal clock pulse is issued by the OCC controller. This number is important when clocks from the same `SynchronizedClocks` group are defined as internal clocks in more than one `PLLStructures` block. In this case, each `PLLStructures` block is interpreted as being a separate OCC controller with its own latency.

If there is more than one clock in a `PLLStructures` block, the latency is in terms of the fastest clock defined in that `PLLStructures` block. (Thus, the same Latency number might mean very different latency times in different `PLLStructures` blocks.) The latency time for each clock should be an integer multiple of its period. For example, if a `PLLStructures` block contains synchronized clocks of 10 ns and 20 ns periods, and the latency is allowed to default to 5, then the latency time is $5 * 10$ ns which is not a multiple of the 20 ns clock's period. The 20 ns clock gets a C40 violation and is flagged as restricted.

The latency number is not used if the clocks for each `SynchronizedClocks` group are defined in a single `PLLStructures` group. In that case, it can be set to 0.

ClockTiming Block Selection

By default, the last `ClockTiming` block to be defined in an STL procedure file is used. To use a specific block in a case where multiple `ClockTiming` blocks have been defined, use the following `set_drc` command:

```
set_drc -internal_clock_timing name
```

To ignore all `ClockTiming` blocks and return to legacy non-synchronized internal clocks behavior, use the following setting:

```
set_drc -nointernal_clock_timing
```

ClockTiming Block Example

The following `ClockStructures` block defines three synchronized clocks in one group:

```
ClockStructures Internal_scan {
  PLLStructures "TOTO" {
    PLLCycles 6;
    Latency 4;
    Clocks {
      "clkate" Reference;
      "dut/CLKX4" PLL {
        OffState 0;
      }
      "TOTO/U2/Z" Internal {
        OffState 0;
        PLLSource "dut/CLKX4";
        Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_38/q_reg/Q"
      }
1;
//Note that the rest of the clock chain goes here.
    }
    "dut/CLKX2" PLL {
      OffState 0;
    }
    "TOTO/U5/Z" Internal {
      OffState 0;
      PLLSource "dut/CLKX2";
      Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_19/q_reg/Q"
    }
1;
//The rest of the clock chain goes here.
    }
    "dut/CLKX1" PLL {
      OffState 0;
    }
    "TOTO/U8/Z" Internal {
      OffState 0;
      PLLSource "dut/CLKX1";
      Cycle 0 "snps_clk_chain_0/U_shftreg_0/ff_0/q_reg/Q"
    }
1;
//The rest of the clock chain goes here.
    }
  }
}

ClockTiming CTiming_2 {
  SynchronizedClocks group0 {
    Clock CLKX4 { Location "TOTO/U2/Z"; Period '10ns'; }
    Clock CLKX2 { Location "TOTO/U5/Z"; Period '20ns'; }
    Clock CLKX1 { Location "TOTO/U8/Z"; Period '40ns'; }
  }
}
```



```
}
ClockTiming CTiming_1 {
  SynchronizedClocks group0 {
    Clock CLKX4 { Location "TOTO/U2/Z"; Period '10ns';
      Waveform '0ns' '5ns'; }
    Clock CLKX2 { Location "TOTO/U5/Z"; Period '30ns';
      Waveform '0ns' '15ns'; }
    MultiCyclePath 1 { From "TOTO/U5/Z"; To "TOTO/U2/Z"; }
  }
  SynchronizedClocks group1 {
    Clock CLKX1 { Location "TOTO/U8/Z"; Period '40ns'; }
  }
}
}
```

This example shows two `ClockTiming` blocks. The one labeled `CTtiming_1` is the default because it is the last one to be defined.

In the first `ClockTiming` block, the three clocks can be used as a single synchronization group. However, clock pulse overlapping is not possible because there are no `Waveform` statements.

In the second `ClockTiming` block, the same three clocks are defined but in a different relationship. The first two clocks are in one `SynchronizedClocks` group and, because their `Waveforms` and a `MultiCyclePath 1` relationship is defined, clock pulse overlapping can be done. The third clock is defined separately, so it is considered to be asynchronous to the others. For this purpose, it could also have been omitted since any clock that is not assigned to a `SynchronizedClocks` group is considered to be asynchronous to all other clocks.

Note that when `ClockTiming` blocks are used, the lengths of the clock chains might be different for different internal clocks. (This is still an error when there is no `ClockTiming` block.)

Specifying Internal Clocking Procedures

Internal clocking procedures are comprised of combinations of internal clock pulses.

The following sections describe the syntax for specifying internal clocking procedures in the STIL procedures file (SPF):

- [ClockConstraints and ClockTiming Block Syntax](#)
- [Specifying the Clock Instruction Register](#)
- [Specifying External Clocks](#)

- [Example 1](#)
- [Example 2](#)

For more information on using internal clocking procedures in the ATPG flow, see the [Using Internal Clocking Procedures](#) section.

ClockConstraints and ClockTiming Block Syntax

You can use either the `ClockConstraints` block or the `ClockTiming` block in the top level of the `ClockStructures` block to specify internal clocking procedures. However, you cannot combine the `ClockConstraints` and `ClockTiming` blocks. External clocks specified in the `ClockStructures` block must be specified as separate entities in the SPF.

The following syntax is used for specifying internal clocking procedures:

```
ClockStructures (name) {
    (ClockController name { // alias of PLLStructures - either may be used
        (PLLCycles number;)
        (MinSysCycles number;) // equivalent to set_atpg
    -min_atclock_cycles
        (Clocks {
            (location <External|Internal|PLL> {
                (OffState <0|1>;)
                (Name name;)
                ...
            })*
            (name <External|Internal|PLL> {
                (OffState <0|1>;)
                (Location location (location)+;)
                ...
            })*
        })*
        ...
        (InstructionRegister name {(signature)+})*)*
    })*
    (ClockTiming (name){ ... })*
    (ClockConstraints (name){
        (UnspecifiedClockValue <Off|On|0|1>;)
        (ClockingProcedure name {
            (UnspecifiedClockValue <Off|On|0|1>;)
            (clk=<0|1|P>;)* // Clock assignments
            (clkIR=<0|1>;)* // Corresponding Clock Instruction
                               // Register assignments
        })*
    })*
}
```

Note the following when specifying internal clocking procedures:

- The `MinSysCycles` keyword specifies the number of external ATE cycles required for the clock controller to return to its initial state after being enabled. This statement works for any type of on-chip clocking. It specifies the minimum number of ATE cycles of the capture operation. Extra cycles are appended to the end of the capture sequence if necessary. This keyword can be used instead of `set_atpg -min_ateclock_cycles` command, which has the same behavior in both cases.
- You can define external clocks in the `Clocks` block. You can also define aliases between the location of the clock (which must be the pin pathname to its driving cell) and a name that can be used in the constraint definitions. A clock name can be defined with multiple locations, which allows multiple clocks to be defined with one statement in the constraint definitions.
- The `InstructionRegister` block is a construct in the `ClockController` block. It consists of a sequence of locations that must be set externally to control the specifics of a clocking sequence. The `InstructionRegister` is associated with the controller – not with individual clocks. The `InstructionRegister` construct subsumes the clock chains as specified by using the `Cycle` statements. When constraints are used, the `Cycle` statements are ignored if present.
- Only one `ClockTiming` or `ClockConstraints` block can be used at the same time. If a `ClockTiming` block exists, you must specify the `set_drc -nointernal_clock_timing` command to use internal clocking procedures.
- The `ClockConstraints` block describes a set of clock constraints. The `ClockingProcedure` block describes a set of clock pulse sequences intended to be used jointly. The `ClockingProcedure` specifications satisfies the `ClockConstraints` specifications.
- A `ClockingProcedure` block consists of two types of assignments:

- The first set of assignments correspond to the clocks, which constitute the actual clock constraints:

```
clk=<0|1|P|-+;
```

In this case, `clk` is the name of an internal clock, as defined in the `ClockController` block. The `n`th bit at the end of the line is the constrained assignment to the clock in the `n`th capture time frame of the pattern in which it is used. The clock-off value (from the `Clocks` definition) indicates no pulse; the `P` value indicates a pulse. The non clock-off value is synonymous to `P`.

- The second set of assignments are for the `InstructionRegister` contents. These assignments specify the externally assignable values required to realize the clock assignments. In this case, the `n`th bit at the end of the line is the value used to

set the `nth` bit of the `InstructionRegister` in the same order that the bits were defined in the `ClockController` block.

The components of the clock controller hardware must be apparent to validate that the specified assignments to the `InstructionRegister` contents actually cause the expected clock pulses. This typically requires functional validation techniques and is beyond the scope of test DRC. Therefore, functional validation is your responsibility. As a last resort, a full timing simulation of the generated patterns should detect any issue.

- The `UnspecifiedClockValue` statement defines the behavior of clocks that are not defined in a particular `ClockingProcedure` block. An `UnspecifiedClockValue` statement outside of all of the `ClockingProcedure` blocks globally specifies the behavior of all unspecified clocks. However, an `UnspecifiedClockValue` statement inside a `ClockingProcedure` block overrides the global value for that block only. The values that can be specified are `Off` (the clocks do not pulse), `On` (the clocks do pulse), `0` or `1`. The default is that the clocks are unspecified. Incomplete clocking procedures are not recognized, so either the `On` or `Off` value should be used or all clock values should be specified.

Specifying the Clock Instruction Register

The `InstructionRegister` block can comprise any of the following defined signals:

- Outputs of scan cells
- Primary inputs
- Outputs of nonscan cells

You can define a combination of any of these signals. Nonscan cells in the clock instruction register must be constant-value C0 or C1 cells and are not allowed to change during the test.

You can apply the `add_cell_constraints` or `add_pi_constraints` command to the clock instruction register members if you want to limit the number of usable clocking procedures.

Specifying External Clocks

External clocks are clocks that are controlled from top-level design ports. They are generally incompatible with internal clocking. When internal clocking procedures are used, unspecified external clocks should be disabled using the `add_pi_constraints` command.

External clocks can be specified inside internal clocking procedures. Although they are already defined as clocks elsewhere in the STL procedure file, they must be

redefined in the `Clocks` block of the `ClockController` block if they are specified in the `ClockConstraints` block.

The external clocks are defined just like other clocks in the `ClockingProcedure` blocks. The external clock pulses can be used to control the pulsing of internal clock pulse and are considered as part of the clock instruction register.

You can define the external clocks in a way that they do not affect the internal clocking and are allowed to pulse. In this case, you should define multiple `ClockingProcedure` blocks. These blocks are identical except for the external clock definitions. As a result, the external clocks are not part of the conditioning to specify the pulses of the internal clocks.

Example 1

```
ClockStructures {
    ClockController controller1 {
        PLLCycles 2;
        Clocks {
            // All clocks are defined by their instance/pin names as usual.
            // The Cycle statements are not needed, so they can be omitted.
            "U1/U2/U_CLK_A_CNTL/Y" Internal {OffState 0;}
            "U1/U2/U_CLK_B_CNTL/Y" Internal {OffState 0;}
            "U1/U2/U_CLK_C_CNTL/Y" Internal {OffState 0;}
            // The next two clocks are equivalent inside the clocking procedures.
            "ClkDE" Internal {
                Offstate 0;
                Location "U1/U2/U_CLK_D_CNTL/Y" "U1/U2/U_CLK_E_CNTL/Y";
            }
        }
        InstructionRegister CLKIR {
            "U1/U3/U_CLK_REG/clock_reg_2/Q";
            "U1/U3/U_CLK_REG/clock_reg_1/Q";
            "U1/U3/U_CLK_REG/clock_reg_0/Q";
        }
    }
    ClockConstraints constraints1 {
        UnspecifiedClockValue Off;
        ClockingProcedure one { // A launches, B captures, C & DE default
(off)
            // Clocks are still defined by their instance/pin names.
            "U1/U2/U_CLK_A_CNTL/Y"=P0;
            "U1/U2/U_CLK_B_CNTL/Y"=0P;
            CLKIR=001;
        }
        ClockingProcedure three { // B & C both launch & capture, A & DE
are off
            UnspecifiedClockValue On;
            "U1/U2/U_CLK_A_CNTL/Y"=00;
            "ClkDE"=00;
            CLKIR=010;
        }
    }
}
```

Chapter 11: STIL Procedures

Specifying Internal Clocking Procedures

```

    }
    ClockingProcedure four { // DE launches & captures, C also captures
        `ClkDE`=PP;
        "U1/U2/U_CLK_C_CNTRL/Y`=0P;
        CLKIR=100;
    }
    ClockingProcedure ClockOff { // All clocks off to prevent a C37
error
        "U1/U2/U_CLK_A_CNTRL/Y`=00;
        "U1/U2/U_CLK_B_CNTRL/Y`=00;
        "U1/U2/U_CLK_C_CNTRL/Y`=00;
        CLKIR=011;
    }
    // These are all that are defined, so no other clock pulse combinations
    // or CLKIR values are allowed in the ATPG patterns.
    }
}

```

Example 2

```

ClockStructures Internal_scan {
    ClockController "PLL_STRUCT_0" {
        PLLCycles 2;
        Clocks {
            "dutm/clk1" Internal { OffState 0; }
            "dutm/clk2" Internal { OffState 0; }
            "clkref" Reference;
        }
        // "clkext0" is used to control clocking
        "clkext0" External;
        // "clkext1" is allowed to pulse in some procedures
        "clkext1" External;
    }
    InstructionRegister CLKIR {
        "dutcl/FF_0_reg/Q";
        "dutcl/FF_1_reg/Q";
    }
}
ClockConstraints constraints1 {
// Force external clocks off when they're unspecified
    UnspecifiedClockValue Off;
    ClockingProcedure intraU1 {
        "dutm/clk1`=PP;
        "dutm/clk2`=00;
        CLKIR=11;
        "clkext0`=00;
    }
    ClockingProcedure extraU1 {
        "dutm/clk1`=PP;
        "dutm/clk2`=P0;
        CLKIR=11;
        "clkext0`=P0;
    }
}

```

```
    }  
    ClockingProcedure intraU0 {  
        "dutm/clk1"=00;  
        "dutm/clk2"=PP;  
        CLKIR=01;  
        "clkext1"=PP;  
    }  
    ClockingProcedure ClockOff {  
        "dutm/clk1"=00;  
        "dutm/clk2"=00;  
        CLKIR=00;  
    }  
} }  
}
```

See Also

- [Using Internal Clocking Procedures](#)

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.

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 the following example shows one additional scan chain force and measure placed outside of the `Shift` procedure. For a scan chain length of N , TestMAX ATPG performs $N-1$ shifts using the vector inside the `Shift` procedure, and the final shift using the vector which follows, where $TMS=1$.

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; }  
    }  
}
```

Multiple Scan Groups

You should set up TestMAX ATPG 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). TestMAX ATPG 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 the following example, the `ScanStructure` labels are `g1`, `g2`, `g3`, and `g4`. TestMAX ATPG 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.

TestMAX ATPG 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 are written out from this data, then each scan signal in each Shift Vector needs a unique symbolic label. A V14 warning is 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=##; ... }}`.

If you are using named `ScanStructure` blocks, they must to be specified as part of the `PatternBurst` block. However, if STIL patterns are written out, then each scan signal used across more than one scan group will require a separate symbolic label to associate scan data with this specific scan block. The example shown in the [JTAG/TAP Controller Variations for the load_unload Procedure](#) section does not follow this constraint (and would generate V14 warnings which can be ignored if STIL patterns are not generated), however, the following example (with only one scan chain per Shift) does. The following example 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. The following example identifies the necessary `SignalGroup` definitions that must be part of this context.

Four Scan Groups Structured for STIL Pattern Generation

```
STIL 1.0;
SignalGroups {
  _si11="SDI[1]" {ScanIn;}  _si12="SDI[2]" {ScanIn;}
  _si13="SDI[3]" {ScanIn;}
  _so11="SDO[1]" {ScanOut;} _so12="SDO[2]" {ScanOut;}
  _so13="SDO[3]" {ScanOut;}
  _si21="SDI[1]" {ScanIn;}  _si22="SDI[2]" {ScanIn;}
  _si23="SDI[3]" {ScanIn;}
  _so21="SDO[1]" {ScanOut;} _so22="SDO[2]" {ScanOut;}
  _so23="SDO[3]" {ScanOut;}
  _si31="SDI[1]" {ScanIn;}  _si32="SDI[2]" {ScanIn;}
  _si33="SDI[3]" {ScanIn;}
  _so31="SDO[1]" {ScanOut;} _so32="SDO[2]" {ScanOut;}
  _so33="SDO[3]" {ScanOut;}
  _si41="SDI[1]" {ScanIn;}  _si42="SDI[2]" {ScanIn;}
  _si43="SDI[3]" {ScanIn;}
  _so41="SDO[1]" {ScanOut;} _so42="SDO[2]" {ScanOut;}
  _so43="SDO[3]" {ScanOut;}
}

PatternBurst "_burst"{
  ScanStructures g1;ScanStructures g2;ScanStructures g3; ScanStructures
  g4;
  PatList {"_pattern_"{
}}

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 {
```

Chapter 11: STIL Procedures

Multiple Scan Groups

```

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 { _si11=#; _si12=#; _si13=#; _so11=#; _so12=#;
    _so13=#; clk=P; mclk=0; }
    Shift { ScanStructures g1; V { _si11=#; _si12=#; _si13=#; _so11=#;
    _so12=#; _so13=#; clk=P; } }

    ScanStructures g4;
    Shift { V { _si41=#; _si42=#; _si43=#; _so41=#; _so42=#; _so43=#;
    clk=P; mclk=0; } }
    V { clk=0; mclk=0; mode=1; }
    "single_shift1:" V { _si11=#; _si12=#; _si13=#; _so11=#; _so12=#;
    _so13=#; clk=P; }
    V { chain_sel = 0; mclk=P; clk=0; }
    V { chain_sel = 1; }
    V { mclk=0; }

    ScanStructures g2;
    Shift { V { _si21=#; _si22=#; _si23=#; _so21=#; _so22=#; _so23=#;
    clk=P; } }
    "single_shift2:" V { _si21=#; _si22=#; _si23=#; _so21=#; _so22=#;
    _so23=#; clk=P; }
    V { chain_sel = 1; mclk=P; clk=0; }

    ScanStructures g3;
    V { chain_sel = 0; mclk=P; }
    Shift { V { _si31=#; _si32=#; _si33=#; _so31=#; _so32=#; _so33=#;
    clk=P; mclk=0; } }
    V { clk=0; }
    V { chain_sel = 1; mclk=P; }
    V { }

  }
}
MacroDefs {
"test_setup" {

  V { "mclk"=0; "clk"=0; "rst"=1; scan_en=0; inc=0; mode=1; }
  }
}
}

```

The preceding example 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 preceding example.

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.

The following example illustrates syntax for a design with four different groups of scan chains that must be accessed serially during the `load_unload` process.

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]"; }
}
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; }
        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; }
        }
    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 the preceding example 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 the preceding example, 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.

The following example 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.

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 previous 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=11xxxx
    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 { _sil=# ; _sol=# ; 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
} }

```

DFTMAX Compression with Serializer

The DFTMAX compression scan architecture creates by default a combinational connection between the input and output of the compressor/decompressor ("CODEC") to the top-level ports or pins. To improve the ATPG quality of results (QOR) for designs or blocks with a limited number of top-level ports, DFTMAX compression also supports an optional serial connection between the CODEC and the top-level ports, called "serializer."

You should refer to the "DFTMAX with Serializer" chapter in the *DFTMAX Compression User Guide* to see an example STIL procedure file specifically used with serializer. This chapter includes a description of the `SerializerStructures` statement, which is specific to serializer.

Also note that the `report_serializers` command in TestMAX ATPG generates a report containing data for the specified serializers.

12

Design Rule Checking

The DRC process verifies that the physical layout of a design satisfies a series of parameters or rules required by semiconductor manufacturers. By performing DRC, you can verify that a design will function properly when it is fabricated.

You can refer to [Performing Test Design Rule Checking](#) for a basic guide on how to specify basic DRC settings, run DRC, and review DRC results.

The following sections describe the various settings you make when performing DRC:

- [Understanding the DRC Process](#)
- [Contention Analysis](#)
- [Scan Chain Tracing](#)
- [Clock Grouping](#)
- [Declaring Equivalent and Differential Input Ports](#)
- [Cells With Asynchronous Set/Reset Inputs](#)
- [Masking Input and Output Ports](#)
- [Masking Scan Cell Inputs and Outputs](#)
- [Previewing Potential Scan Cells](#)
- [Transparent Latches](#)
- [Shadow Register Analysis](#)
- [Feedback Paths Analysis](#)
- [Procedure Simulation](#)
- [Changing the Design Rule Severity](#)
- [Understanding the DRC Summary Report](#)
- [Binary Image Files](#)
- [Save/Restore in TEST Mode](#)

Understanding the DRC Process

When performing design rule checking, TestMAX ATPG takes the following actions:

1. Reads the STIL procedures to gather information and to check for syntax and consistency errors. For more information, see [STIL Procedures](#).
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. For more information, see [Contention Analysis](#).
3. Simulates the test procedures in the STL procedure file 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. For more information, see [Scan Chain Tracing](#).
5. Analyzes all clocks and clocked devices against the ATPG rules for clock usage. For more information, see 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 multi driver 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, TestMAX ATPG 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. For more information, see [Understanding the DRC Summary Report](#).

Contention Analysis

Three-state circuitry is characterized by its ability to use the high impedance state (Z state). The supported gate types that model the logical behavior of three-state circuitry and use the Z state include the BUS, BUSK, TSD, SW, PI, PO, PIO, and TIEZ gates.

Most three-state activity occurs on a BUS gate, which is primarily used to resolve the net value from a net with multiple drivers. A BUS gate can have bidirectional connections to

external pins (PIO) or bus keepers (BUSK). All inputs and bidirectional connections are either strong or weak.

A *contention condition* occurs when a BUS gate has two strong drivers of opposing values. This condition can damage the chip, so extensive contention checking is required to prevent its occurrence.

The following sections describe contention checking:

- [BUS Contention Ability Checking](#)
- [BUS Z State Ability Checking](#)
- [Contention Prevention Checking](#)
- [Simulation Contention Detection](#)
- [ATPG Contention Prevention](#)
- [Post-Capture Contention Checking](#)

BUS Contention Ability Checking

During DRC, the Z1 rule checks BUS gates with circuitry that could potentially cause BUS contention. This check eliminates false contention reporting when multiple inputs to a BUS are at X. The BUS contention ability analysis searches for two strong three-state drivers on a BUS gate that can simultaneously have their enable lines active. Unless the `-nomultiple_on` option of the `set_contention` command is set for contention checking, the data lines must be at different values to fail the check. After BUS contention ability checking is performed, a summary message shows the number of buses falling into each of the following contention ability categories:

- **Pass** - The BUS gate cannot satisfy contention conditions and can be ignored for contention checking.
- **Bidi** - The BUS has an external bidirectional connection. Except for this connection, it passes contention ability checking. To control contention, it need only be controlled by the value placed on the bidirectional port.
- **Fail** - The BUS is capable of contention and must be checked and controlled.
- **Abort** - Contention ability checking of the BUS was aborted. It is uncertain whether the BUS is capable of contention, so it must be checked and controlled.

The BUS contention ability analysis is performed only when required. If the analysis was previously performed and nothing has changed that could affect the results for a BUS, it is not checked again.

BUS Z State Ability Checking

During DRC, the Z2 rule identifies BUSes with circuitry that could potentially cause a Z state. This check attempts to satisfy the conditions necessary to justify a Z state on a BUS gate. After this check is performed, a summary message shows the number of BUSes falling into each Z state ability category:

- Pass - The BUS gate cannot satisfy Z-state conditions.
- Bidi - The BUS has an external bidirectional connection. Except for this connection, it passes Z-state checking.
- Fail - The BUS is capable of holding a Z state.
- Abort - Z-state ability checking of the BUS was aborted. It is uncertain whether the BUS is capable of holding a Z state.

The BUS Z state ability analysis is performed only when required. If the analysis was previously performed and nothing has changed that could affect the results for a BUS, it is not checked again.

Contention Prevention Checking

For BUSes that fail or abort the Z1 rule, an ATPG analysis is performed by the Z7 rule to determine if it is possible to simultaneously satisfy the conditions necessary to prevent contention on these buses. A Z7 failure indicates that ATPG is unlikely to be successful in avoiding bus contention. See the description of the Z7 rule in TestMAX ATPG Help for a complete description of how to properly analyze a Z7 failure.

Simulation Contention Detection

BUS gates that fail contention ability checking during simulation are checked to determine if there are in a potential contention condition. A violation of BUS contention during fault simulation causes the pattern to be rejected and disallows any detection credit. A message is issued for each simulation pass indicating the number of patterns rejected due to contention and the site of the first contention. You can turn off contention checking during simulation using the `nobus` option of the `set_contention` command.

ATPG Contention Prevention

BUS gates that fail contention ability checking during test generation are forced to satisfy a contention-free state. If the process of satisfying contention prevention causes an abort condition, a special message reports the number of faults per simulation pass (32 patterns) that were aborted due to this condition. You can turn off ATPG contention prevention using the `-noatpg` switch of the `set_contention` command.

Post-Capture Contention Checking

Normal scan-based simulation only considers the effect of values loaded into scan cells and not the effect of values that can be captured. If the enable lines of three-state drivers of bus gates that are not contention-free are connected to scan cells, it is possible for these BUS gates to go into contention after the capture clock, even if they were contention-free before the capture clock. These conditions are checked by the Z9 and Z10 rules.

You can configure the simulation process to simulate the captured values using the `-capture` option of the `set_contention` command. As a result, the simulation checks for contention that could occur at capture time, and rejects and reports patterns that fail the contention check.

Scan Chain Tracing

When performing scan chain tracing, TestMAX ATPG takes the following actions:

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-T> set_drc -trace
```

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

See Also

- [Performing Scan Chain Diagnosis](#)

Clock Grouping

TestMAX ATPG applies dynamic clocking grouping by default. This enables basic scan ATPG to simultaneously pulse clocks and detect clocks that can be serially pulsed during the same capture cycle. Clocks with a small amount of sequential effects can also be detected and grouped. In this case, TestMAX ATPG sets up the pattern generation environment to avoid generating vectors that would fail simulation.

Clock grouping can potentially reduce pattern count since ungrouped clocks require separate scan loads and patterns to test faults in each clock domain for basic-scan patterns. Grouped clocks can be pulsed in a single pattern. The clocks pulsed for a given vector are selected dynamically during pattern generation, maximizing the fault detection and minimizing the pattern count.

In addition to dynamic clocking, TestMAX ATPG can use disturbed clocking to group some clocks with a limited number of cells containing sequential effects. In this case, even if there are sequential effects, grouping these clock can further reduce pattern count. TestMAX ATPG then masks any disturbed cells to avoid sequential effects. Potential disturbed grouping is done during DRC analysis.

During the DRC process, TestMAX ATPG automatically performs clock grouping analysis and reports the results in the transcript. All PI equivalences are removed, except for differential inputs.

The following sections describe how to work with clock groups:

- [Reducing the Pattern Count Through Clock Grouping](#)
- [Clock Grouping Analysis](#)
- [Generating a Clock Group Report](#)
- [Clock Grouping Limitations](#)

Reducing the Pattern Count Through Clock Grouping

When you generate combinational vectors in basic-scan ATPG, TestMAX ATPG normally uses only one clock pulse per pattern. However, it is sometimes possible to pulse several clocks in the same vector, which enables you to observe more logic and reduces the need for additional patterns.

If your design has two independent clocks (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, which saves 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 are pulsed alone.

Dynamic clock grouping selects the clocks pulsed for a given vector during pattern generation, which maximizes the fault detection and minimizes the pattern count.

The disturbed clocking scheme allows TestMAX ATPG 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. TestMAX ATPG cannot use the disturbed cells. To manually group clocks, use the `add_pi_equivalences`

command. After you have defined a group, any clock that belongs to this group cannot be pulsed alone.

To use clock grouping to reduce the pattern count:

1. Read your netlist and library model files, and build your design in TestMAX ATPG. For details, see [Setting Up and Building the ATPG Model](#).
2. Choose your criteria for clock grouping. The `set_drc` command has several options that affect clock grouping, including the `-allow_unstable_set_resets`, the `-blockage_aware_clock_grouping`, the `-clock -dynamic`, the `-disturb_clock_grouping`, and the `-dynamic_clock_equivalencing` options.
3. Run DRC. For details, see [Performing Test Design Rule Checking](#).

DRC performs an analysis for clock grouping. For details, see [Clock Grouping Analysis](#).

4. Generate the basic-scan test vectors, for example:

```
run_atpg -auto_compression
```

Clock Grouping Analysis

During the DRC process, clock grouping analysis is automatically performed and the results are reported in the transcript, as shown in the following example:

```
Clocks C1 (8) and C2 (13) were identified as potentially
groupable.
Clocks C1 (8) and C3 (17) were identified as potentially
groupable.
Clocks C1 (8) and C4 (19) were identified as potentially
groupable.
Clocks C1 (8) and W4 (20) were identified as potentially groupable.
Clocks C2 (13) and C3 (17) were identified as potentially groupable.
Clocks C2 (13) and C4 (19) were identified as potentially groupable.
Clocks C2 (13) and W4 (20) were identified as potentially groupable.
Clocks C3 (17) and C4 (19) were identified as potentially groupable.
Clocks C3 (17) and W4 (20) were identified as potentially groupable.
Clock grouping analysis completed, #clock_groups_identified=9
```

The lines in the example indicate that clock 'C1', with gate ID 8, can be grouped with clocks 'C2', 'C3', 'C4', and 'W4'.

In addition, clock 'C2' is groupable with {C3,C4,W4} and 'C3' is groupable with {C4,W4}.

Clock grouping might be affected by the order in which the clock list is processed. It is suggested that if you define clocks using `add clocks` commands, that you define the clock with the highest fanout first, and all asynchronous set/resets last.

The clock grouping algorithm considers clocks as groupable if all of the following conditions are true:

- The clocks do not connect to a common clock-off stable state element.
- There are no level sensitive (LS) or trailing edge (TE) ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port data input with any of the following conditions:
 - LS/LE connection
 - TE connectionwhere the off-time of the first clock occurs later than the off-time of the other clock.
- There are no LE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port data input with any of the following condition:
 - LS/LE connectionwhere the on-time of the first clock occurs later than the on-time of the other clock.

There are no LS or TE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port clock/write input with any of the following conditions:

- LS/LE connection
 - TE connection
- where the off-time of the first clock occurs later than the off-time of the other clock.

There are no LE ports where one clock is connected to the clock or write port input and the other clock has a clock-effect connection to the port clock or write input with any of the following condition:

- LS/LE connection
- where the on-time of the first clock occurs later than the on-time of the other clock.

If the design has two state elements A and B such that:

- The output of A is connected to the input of B.
- A and B are clocked by different clocks that have nearly identical timing

Then, the two clocks have a parallel grouping relationship only if the capture edge of clock B occurs at or before the capture edge of clock A minus the skew value. Otherwise, the clocks are ungrouped, or have a disturbed grouping. The default for the skew is 1 time unit, which eliminates clocks with exactly the same timing from being grouped in this type of design connectivity.

Note that the unstable state elements (including transparent latches) are ignored for this clock grouping analysis.

The clock grouping analysis is always performed at the end of the clock rules checking during DRC with all grouped clocks reported in the transcript.

Generating a Clock Group Report

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

```
report_clocks -matrix -verbose
```

The `-matrix` option of the `report_clocks` command displays 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
```


Chapter 12: Design Rule Checking

Declaring Equivalent and Differential Input Ports

```

--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

```

Clock Grouping Limitations

Clock grouping has the following limitations:

- Dynamic and disturbed clocking are not used by Full-Sequential ATPG.
- Disturbed clocking can result in a slightly lower test coverage because of disturbed cell masking.

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.

The following sections describe how to declare equivalent and differential input ports:

- [Using the Add PI Equivalences Dialog Box](#)
- [Using the `add_pi_equivalences` Command](#)

Using the Add PI Equivalences Dialog Box

The following steps describe how 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 the description of the `add_pi_equivalences` command in TestMAX ATPG Help.

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, as shown in the following example:

```
DRC-T> add_pi_equivalences ENA_P -inv ENA_N
```

For the complete syntax and option descriptions, see the description of the `add_pi_equivalences` command in TestMAX ATPG Help..

In the following example, the first line defines the two input ports `spec_port1` and `spec_port2` as equivalent; the second line defines that the following ports should be constrained to be at an inverted value relative to the first port in the list.

```
DRC-T> add_pi_equivalences {spec_port1 spec_port2}  
DRC-T> add_pi_equivalences spec_port1 -invert spec_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-T> add_clocks 0 clock_pos  
DRC-T> add_clocks 1 clock_neg  
DRC-T> add_pi_equivalences 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 is sometimes ignored by clock grouping. Pins declared as having a differential relationship are driven to opposite values by generated patterns.

See Also

- [Understanding Flattening Optimization](#)

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.

See Also

- [Declaring Clocks](#)
- [Power Aware Testing with Asynchronous Primary Inputs](#)

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 standalone 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 the “Declaring Primary Input Constraints” section) or by using the `add_pi_constraints` command:

```
DRC-T> 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-T> add_po_masks port_name
```

Masking Scan Cell Inputs and Outputs

TestMAX ATPG 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.

The following sections describe how to mask scan cell inputs and outputs:

- [Specifying Cell Constraints Locations and Scan Cell Controls](#)
- [Using the Add Cell Constraints Dialog Box](#)
- [Using the `add_cell_constraints` Command](#)

Specifying Cell Constraints Locations and Scan Cell Controls

You specify the location of the cell constraint using either of the following techniques:

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

You can use any of the following five scan cell controls:

- 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.

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

The following steps describe how 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.

For additional information about the available options, see description of the `add_cell_constraints` command in TestMAX ATPG Help.

4. Click OK.

Using the `add_cell_constraints` Command

You can also define scan cell controls using the `add_cell_constraints` command, as shown in the following example:

```
DRC-T> add_cell_constraints 0 /TOP/U1/sifter/reg42
```

For the complete syntax and option descriptions, see the description of the `add_cell_constraints` command in TestMAX ATPG Help.

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. TestMAX ATPG 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. (TestMAX ATPG converts the devices into a scan chain but does not change set/reset pins.)

The following sections describe how to preview potential scan cells:

- [Scan Cell Types](#)
- [Using the set_scan_ability Command](#)
- [Using the Set Scan Ability Dialog Box](#)

Scan Cell Types

Scan cells are the independent units that can be used as control and observe points in scan-based fault simulation and test generation. Each scan cell contains one or more state gates (latches or flip-flops). Each scan cell gate is assigned a "type" according to its behavior in the scan cell. All combinations of inversion parity between scan cell gates of a scan cell are supported.

All scan cell gates of a scan cell are controllable, but they have a fixed relationship to one another that depends on the inversion parity. They must have values that are consistent with this relationship. The loading of the scan chain that contains the scan cells provides the control ability.

Similarly, the unloading of the scan chain that contains the scan cells provides observe ability. However, only the scan cell gate at the output of the scan cell can be observed by unloading the scan chain. To observe any other scan cell gate requires an additional step to transfer its value to the scan cell output gate before unload. Test procedures are predefined in the test protocol file that perform this process. Each procedure is called either the master_observe procedure or the shadow_observe procedure.

Note the following scan cell types:

- *Master* - The master gate is an independently clocked state gate that captures shift data from outside the scan cell on the active edge of the shift clock. Every scan cell must contain one and only one master gate, which is considered the primary gate of the scan cell.
- *Slave* - A slave gate is an independently clocked state gate that captures its shift data from another member of the scan cell. This is an optional gate of the scan cell and is used most often in the LSSD (Level-Sensitive Scan Design) architecture. When a slave exists in a scan cell, it is the output gate and is the default observable gate of the scan cell. The observation of the master gate requires a master_observe procedure. There is no requirement that the slave gate be in the same parent library cell as the master

gate. This relationship is determined by physical connectivity and clock event ordering during shift, and not placement in any particular library cell.

- *Shadow* - A shadow gate is not in the scan chain path, but is capable of attaining its associated scan cell value during the scan chain load process. This can be accomplished during the shift procedure or from a separate clocking that occurs after the shift process in the load operation. Normally, shadow gates are used only as control points. The identification of shadow gates can be suppressed by selecting the `-noshadows` option of the `set_drc` command.
- *Parallel Shadow* - Obtains its value in parallel with the master scan cells during the Shift procedure. Its input and the input of the master gate have a common source. The identification of parallel shadows can be suppressed by selecting either the `-noshadows` or the `-serial_shadows_only` option of the `set_drc` command.
- *Serial Shadow* - Obtains its value during the post-amble sequence after the Shift procedure. The identification of parallel shadows can be suppressed by selecting the `-noshadows` option of the `set_drc` command.
- *Dslave* - A dslave (dependent slave) gate is a dependently clocked state gate that captures its shift data from another member of the scan cell (source). Dependent clocking means that its captured shift value is the same value as its source for the capturing clock pulse. It is distinguished from a slave in that it can never hold a value different from its source after the capturing clock is applied. A dslave is not considered an observe point. Lockup Latches in scan chains are identified as dslave devices.
- *Observable Shadow* - An observable shadow is a shadow gate which has an ability to also transfer its captured value to the output of its associated scan cell. This requires using a `shadow_observe` procedure that has been defined in the test protocol file.
- *Scan TLA* - A scan TLA (scan transparent latch) is a state element in the scan chain path that participates in the shift process but does not have an ability to hold its scan cell value when all clocks are off. It is treated as a transparent latch for scan-based simulation and test generation and gets no control or observe credit.

Identifying Scan Cells

Scan cells are identified and assigned as scan cell gates during the DRC process. Starting from the scan chain output ports, a backtrace is performed through a sensitized path, considering the simulated values that result from the application of the shift procedure. State gates in the traced path are placed into scan cells. The amount of circuitry traced through a single application of the shift procedure determines the boundaries of the scan cells.

After tracing is complete, the remaining nonscan state gates are analyzed to determine whether they have attained a value that is a direct function of a single scan cell. Those gates that have this condition are added to the scan cell gate list of the associated scan cell and are called shadows.

Reporting Scan Cells

To get a report on the scan cells in the design, use the `report_scan_cells` command. The `report_primitives` command also displays scan cell information for all reported scan cell gates.

Scan Cell Inversion Data

When you obtain scan cell data with the `report_scan_cells` command, the inversion information for scan cell gates is presented as a set of two characters, where "N" indicates no inversion and "I" indicates inversion. The first character shows the inversion relationship of the scan cell gate to the scan chain input, and the second character shows the inversion relationship of the scan cell gate to the scan chain output. The possible combinations are:

- `NN` - No inversion.
- `NI` - The scan cell gate is inverted relative to the scan chain output.
- `IN` - The scan cell gate is inverted relative to the scan chain input.
- `II` - The scan cell gate is inverted relative to both scan chain output and input.

If you use the `-pin` option of `report_scan_cells` to display information on scan cell pins, the inversion information for each scan cell input and output is presented as a single character. For an input port, this character indicates the inversion relationship between the scan cell input pin and the scan chain input. For an output port, this character indicates the inversion relationship between the scan cell output pin and the scan chain output.

Using the `set_scan_ability` Command

You can also place nonscan sequential devices in a virtual scan chain using the

`set_scan_ability` command, as shown in the following example:

```
DRC-T> set_scan_ability on core/host/status
```

For the complete syntax and option descriptions, see the description of the `set_scan_ability` command in TestMAX ATPG Help.

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

```
DRC-T> set_scan_ability on /top/U1/U2/reg1
DRC-T> set_scan_ability on /top/U1/U2/reg2
DRC-T> set_scan_ability on /top/U1/U2/reg3
DRC-T> set_scan_ability on /top/U1/U2/reg4
```

When you use a list format in the `set_scan_ability` command, 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.

Note that the `set_scan_ability` command is not compatible with any type of scan compression. DRC will fail if the STIL procedure file contains a CompressorStructures block.

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. The following steps describe how 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/DFE 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.

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, TestMAX ATPG 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 TestMAX ATPG 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.

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, TestMAX ATPG searches for nonscan cells that can be considered shadow registers.

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

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

```
BUILD-T> set_drc -noshadow
```

Feedback Paths Analysis

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

During DRC, TestMAX ATPG 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, TestMAX ATPG issues an X1 rule violation.

See Also

- [Analyzing a Feedback Path](#)

Procedure Simulation

In addition to the test_setup, load_unload, and Shift procedures, there are other procedures in the STL procedure file or implied by the definition of clock ports. TestMAX ATPG simulates all of these procedures as part of the design rule checking process to guarantee that they accomplish their intended purposes. For details on the running the various procedures, see [STIL Procedure Files](#).

Changing the Design Rule Severity

Each design rule is assigned a severity level that determines the action taken if a rule violation occurs. A design rule violation has possible 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 terminated. Before continuing, you must either correct the problem or change the rule severity level.
- Fatal - Violation of the rule produces an error message, and the current processing step is terminated. the severity level cannot be changed. Before continuing, you must correct the problem.

You can change the rule severity level by using the Set Rules dialog box, or by running the `set_rules` command from the command line.

You can determine the severity level setting of a particular rule and the number of violations that have occurred by selecting Rules > Report Rules in the TestMAX ATPG GUI or by running the `report_rules` command.

Using the Set Rules Dialog Box

To change the rule severity by using the Set Rules dialog box:

1. From the menu bar in the TestMAX ATPG GUI, 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 the description of the `set_rules` command in TestMAX ATPG Help.

3. Click OK.

Using the `set_rules` Command

You can change the rule severity level of any rule (except those that are Fatal) by using the `set_rules` command, as shown in the following example:

```
BUILD-T> set_rules B5 warning
```

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

```
set_rules R22 warning
```

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

Understanding the DRC Summary Report

The `run_drc` command performs design rule checking (DRC), and produces a DRC summary report, as shown in the following example:

```
DRC> run_drc top.spf
-----
Begin scan design rule checking...
-----
Begin reading test protocol file top.spf...
End parsing STIL file slo_gin.spf with 0 errors.
Test protocol file reading completed, CPU time=0.08 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) was violated 12
times.
Test protocol simulation completed, CPU time=0.15 sec.
-----
Begin scan chain operation checking...
Chain c1 successfully traced with 31 scan_cells.
Chain c2 successfully traced with 31 scan_cells.
Chain c3 successfully traced with 31 scan_cells.
Chain c4 successfully traced with 31 scan_cells.
Chain c5 successfully traced with 31 scan_cells.
  : : : : :
Chain c44 successfully traced with 30 scan_cells.
Chain c45 successfully traced with 30 scan_cells.
Chain c46 successfully traced with 30 scan_cells.
Scan chain operation checking completed, CPU time=0.47 sec.
-----
Begin clock rules checking...
Warning: Rule C17 (clock connected to PO) was violated 16 times.
Warning: Rule C19 (clock connected to non-contention-free BUS) was
violated 1 times.
Clock rules checking completed, CPU time=0.15 sec.
-----
Begin nonscan rules checking...
Nonscan cell summary: #DFE=201 #DLAT=0 tla_usage_type=none
Nonscan behavior: #C0=4 #C1=7 #LE=11 #TE=179
Nonscan rules checking completed, CPU time=0.04 sec.
-----
Begin DRC dependent learning...
```

```
DRC dependent learning completed, CPU time=1.01 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) was
violated 24 times.
Contention prevention checking completed, CPU time=0.03 sec.
-----
DRC Summary Report
-----
Warning: Rule C17 (clock connected to PO) was violated 16 times.
Warning: Rule C19 (clock connected to non-contention-free BUS) was
violated 1 times.
Warning: Rule Z4 (bus contention in test procedure) was violated 12
times.
Warning: Rule Z9 (bidi bus driver enable affected by scan cell) was
violated 24 times.
There were 54 violations that occurred during DRC process.
Design rules checking was successful, total CPU time=2.27 sec.
-----
```

scan design rule checking

This indicates the beginning of the scan design rule checking process.

reading test protocol file

The first message indicates the beginning of the reading of the test protocol file. The second message indicates the parsing of the file was successful with no errors. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

Bus/Wire contention ability checking

The first message indicates the beginning of the bus and wire contention checking rules.

The second message summarizes the types of bus gates that are used in the circuit. This includes the total number of bus gates, the number of bidirectional bus gates, the number of weak bus gates (only weak drivers), the number of pull bus gates (a mixture of strong and weak drivers), and the number of bus gates which have a bus keeper.

The next message gives a summary of contention ability status of the bus gates after the analysis is completed. This includes the number of buses which pass (proven contention free), are bidirectional (contention free except for the bidi input), fail (proven contention sensitive), and abort (aborted during analysis).

The next message gives a summary of Z-state ability status of the bus gates after the analysis is completed. This includes the number of buses which pass

(proven incapable of attaining a Z state), are bidirectional, fail (proven capable of attaining a Z state), and abort (aborted during analysis).

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

simulating test protocol procedures

The first message indicates the beginning of the simulation of the test protocol procedures. The results of the simulation is used to first determine state elements that have a constant state behavior and those that attain a set value after the scan chain load.

The second message in the example indicate 4 state elements have a constant 0 behavior and 7 state elements have a constant 1 behavior.

The third message indicate 4 state elements are set to 0 and 7 state elements are set to 1 at the end of the scan chain load. During simulation, certain rules are checked. In this case, the rule checking for bus contention during the test procedures was violated 12 times and a warning message is given.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

scan chain operation checking

The first message indicates the beginning of the scan chain operation checking. The results of the previous simulation are used to verify the operation of the scan chains and identify the associated scan cells. As each scan chain is successfully verified, a message is given indicating its completion with its name and length. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

clock rules checking

The first message indicates the beginning of the clock rules checking. During this process many clock rules are checked and messages are given when violations occur. In this case, two messages are given indicating there were 16 violations of rule C16 and 1 violation of rule C19. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

nonscan rules checking

The first message indicates the beginning of the nonscan rules checking. The objective of this checking is to determine the appropriate behavior for all non scan state elements.

The second message gives a summary of the nonscan state elements. This includes the nonscan DFFs, nonscan DLATs, and the transparent latch usage. In this case, there are no transparent latches.

The next message gives a summary of the calculated nonscan behaviors. This includes C0 (constant 0), C1 (constant 1), LE (edge sensitive state elements that capture on the leading edge of a pulse on the clock pin), and TE (edge sensitive state elements that capture on the trailing edge of a pulse on the clock pin).

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

DRC dependent learning

The first message indicates the beginning of the DRC dependent learning process. Using the behaviors learned during DRC, analyses are performed to determine control ability, observe ability, constraint effects, and blockages due to constraint effects for all gates in the circuit. The last message indicates the process is completed and the CPU time in seconds that was used for the process.

contention prevention rules checking

The first message indicates the beginning of the contention prevention rules checking.

The second message indicates that there were 26 scan cells which had connectivity to bidirectional bus gates. This normally indicates a potential problem that will cause some rule violations.

The next message indicates the rule violations that was the result of that connectivity.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

DRC Summary Report

The first message indicates the beginning of the summary report. For each rule that had at least one violation, a summary message for that rule is given indicating the number of times it was violated.

The next message indicates the total number of rule violations that occurred during the DRC process.

The last message indicates the process is completed and the CPU time in seconds that was used for the process.

Binary Image Files

A binary image file is a data file that stores design information in an efficient and proprietary format for reading by TestMAX ATPG. It contains a flattened version of the design, along with some selected TestMAX ATPG settings.

Using an image file provides several key benefits:

- *Simplifies file management*

Because an image file stores all netlist, library, and STL procedure file details in a single file, it is easy to archive and share design data.

- *Avoids repetitive tasks*

When TestMAX ATPG reads an image file, you do not need to repeat the entire build and DRC phases, since this data is already stored in the file. This results in significant time savings when using large designs.

- *Restricts command usage*

You can create secure image files that allow only a restricted set of commands. These commands are stored in the encrypted image file. You can also control whether schematic viewing is allowed.

When a secure image file is read, the TestMAX ATPG session switches to a secure state in which only the allowed commands can be executed. If you specify a disallowed command, TestMAX ATPG does not execute it and issues a warning message.

- *Provides intellectual property protection*

TestMAX ATPG can obfuscate instance, net, and module names when creating a binary image. This provides an additional level of security by hiding design context.

- *Stores context-sensitive design data*

An image file stores different types of design information, depending on what mode is active when you create it:

- When the Test mode is active, both build and DRC data is stored in the image file.
- When the DRC mode is active, only the build data is stored in the image file.

Creating and Reading Image Files

You use the `write_image` command to create an image file and the `read_image` command to read it.

You can also create and read secure image files. This functionality is implemented through the following 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]`

Note that TestMAX ATPG can obfuscate instance, net and module names. This provides an additional level of security by hiding design context. The names are changed to the following format (where "###" is an integer number of any length):

- Instance names use the format `u###`
- Net names use the format `n###`
- Module names use the format `m###`

The `-garble` option of the `write_image` command modifies the names in the output image. You can send this secure image to a third party with controlled data access. You can also translate the modified instance and net names back to the original names using the `-ungarble` option of the `report_nets` command and the `report_instances` command, if the original design database or the unmodified image file is accessible.

After TestMAX ATPG reads the image, it remains in the same mode in which the image was created (DRC or Test). An image file does not create an identical session as when it was originally created. Some settings and data, such as net names and intermediate levels of hierarchy, are not in the image file. Thus, TestMAX ATPG can only operate in primitive view and not design view

You can use the `report_settings -all -command_report` command to view the stored settings.

For details on how create non-secure and secure image files, see the following sections:

- [Creating a Non-Secure Image File](#)
- [Creating a Secure Image File](#)

Creating a Non-Secure Image File

To create a non-secure image file:

1. Read the netlist file, as shown in the following example:

```
read_netlist top.v
```

2. Read the library models.

```
read_netlist spec_lib.v -library
```

3. Create the design model.

```
run_build_model spec_chip
```

4. Perform design rule checking.

```
run_drc spec_chip.spf
```

5. Write the image file.

```
write_image spec_chip_post_drc.img -violations -replace
```

To read the image during a subsequent run, use the `read_image` command, as shown in the following example:

```
read_image spec_chip_post_drc.img
```

Creating a Secure Image File

You can use a combination of `set_commands` and `write_image` commands to create a secure image file.

When using a secure image file, the following neutral commands are always allowed: `exit`, `alias`, `unalias`, `help`, `source`, `c`, and `pwd`.

To create a secure image file:

1. Read the netlist file, as shown in the following example:

```
read_netlist top.v
```

2. Read the library models.

```
read_netlist spec_lib.v -library
```

3. Create the design model.

```
run_build_model spec_chip
```

4. Perform design rule checking.

```
run_drc spec_chip.spf
```

5. Use the `set_commands` command to specify all commands you want to allow in the secure image. For example:

```
set_commands -secure add_equivalent_nofaults set_commands -secure
add_nofaults set_commands -secure source set_commands -secure
help set_commands -secure read_faults set_commands -secure
read_nofaults set_commands -secure remove_nofaults set_commands
-secure report_licenses set_commands -secure report_nofaults
set_commands -secure report_patterns set_commands -secure
report_version set_commands -secure set_simulation set_commands
-secure run_diagnosis set_commands -secure run_simulation
set_commands -secure set_patterns set_commands -secure set_diagnosis
```

6. Use the `write_image` command to create the secure image. For example:

```
write_image image_enc.gz -password top_secret \ -schematic_view
-replace -garble
```

7. Generate the ATPG patterns.

```
run_atpg -auto
```

8. Write the ATPG patterns to a binary file.

```
write_patterns pat.bin -format binary
```

To read the secure image:

1. Force TestMAX ATPG to BUILD mode, as shown in the following example:

```
build -force
```

2. Read the image.

```
read_image image_enc.gz -password top_secret
```

3. Read the binary pattern format.

```
set_patterns -external pat.bin
```

4. Create the STIL/WGL patterns to be used with garbled image.

```
write_patterns pat_garbled.wgl -format wgl -external write_patterns
pat_garbled.stil -format stil -external
```

To translate a garbled name back to the original name:

1. Read the original design database, as shown in the following example:

```
read_netlist specnetlist.v
```

2. Create the design mode.

```
run_build_model ...
```

3. Perform design rule checking.

```
run_drc ...
```

4. Supply the garbled name as argument to get ungarbled name in output.

```
report_instances u43259 -ungarble
```

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 TestMAX ATPG 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.

13

Optimizing ATPG

TestMAX ATPG enables you to set some of the basic ATPG parameters, as described in [Running ATPG](#). You can further optimize the ATPG process by specifying other settings, such as ATPG constraints and test points, limiting the number of patterns and aborted decisions, applying pattern masking, and running multicore ATPG.

The following sections describe the various settings you can make to optimize ATPG:

- [Optimizing Basic Scan Patterns](#)
- [Using ATPG Constraints](#)
- [Using the Random Decision Option](#)
- [Obtaining Target Test Coverage Using Fewer Patterns](#)
- [Maximizing Test Coverage Using Fewer Patterns](#)
- [Improving Test Coverage With Test Points](#)
- [Limiting the Number of Patterns](#)
- [Limiting the Number of Aborted Decisions](#)
- [Using ATPG Checkpoint Files](#)
- [Creating Test Patterns for Diagnosing Scan Chain Failures](#)
- [Performing Scan Chain Diagnosis](#)
- [Creating End-of-Cycle Measures in ATPG Patterns](#)
- [Deleting Top-Level Ports From Output Patterns](#)
- [Detecting Faults Multiple Times Using N-Detect](#)
- [WGL Pattern Generation Options](#)
- [Running Multicore ATPG](#)
- [Running Logic Simulation](#)

- [Data Volume and Test Application Time Reduction Calculations](#)
- [Pattern Porting](#)

Optimizing Basic Scan Patterns

You can use the `-optimize_patterns` option of the `run_atpg` command to produce a compact set of patterns with high test coverage. This option enables you to use a single `run_atpg` command instead of iterating multiple `run_atpg` commands and manually adjusting various parameters.

When the `-optimize_patterns` option is set, TestMAX ATPG monitors the ATPG process and dynamically adjusts the internal algorithms to generate a compact pattern set. The trade-off is a longer runtime. All manually specified `run_atpg` settings, such as abort limits, minimum detects, and merge limits, are ignored during this operation. However, these settings are restored after pattern optimization is completed.

Note that the `-optimize_patterns` option generates two-clock ATPG patterns as basic scan patterns. But they are stored, read, and simulated as fast-sequential patterns. As a result, a fault simulation that uses two-clock ATPG patterns usually takes longer than the original ATPG run.

The `-optimize_patterns` option of the `run_atpg` command will work with the `-chain_test`, `-coverage`, and `-patterns` options of the `set_atpg` command. This option also works with all power aware options of the `set_atpg` command. However, the power aware options might impact the effectiveness of the pattern optimization process.

The `-optimize_patterns` option is useful during a final TestMAX ATPG run when you want to optimize the pattern count. It generates a lower number of patterns and produces similar test coverage compared to a single `run_atpg -auto_compression` command. You cannot use the `-optimize_patterns` option with any additional `run_atpg` options.

You should use the `run_atpg -auto_compression` command for general pattern generation purposes, such as initial test coverage estimates, writing patterns for verification, analyzing the effects of various options, and obtaining good test coverage and pattern count without increased runtimes. For details on using the `-auto_compression` option, see [Using Automatic Mode to Generate Optimized Patterns](#).

Note the following limitations when using the `-optimize_patterns` option:

- Multiple `run_atpg` commands are supported, but pattern optimization can only be specified one time.
- A learned recipe is not saved.
- Fast-Sequential and Full-Sequential ATPG modes are not supported.

- Be aware that unlike the `run_atpg -auto_compression` command, specifying `set_atpg -capture_cycle number` will not enable Fast-Sequential ATPG during the pattern optimization process. To run Fast-Sequential top-off ATPG, it must be done as an extra step. For example:

```
run_atpg -optimize_patterns
set_atpg -capture 4
run_atpg -auto fast_sequential
```

- Only stuck-at and transition fault models are supported.
- Distributed ATPG is not supported.

Using ATPG Constraints

You can use ATPG constraints to define internal constraints that must be satisfied during ATPG pattern generation and DRC.

The following sections show several examples of how to apply ATPG constraints:

- [Adding ATPG Constraints to Block a Timing-Sensitive Path](#)
- [Defining, Reporting, and Removing No Detection Credit Cells](#)
- [Using ATPG Constraints to Control ATPG Assertions](#)

Adding ATPG Constraints to Block a Timing-Sensitive Path

In this example, a combinational gate is buried within the design hierarchy. Under random conditions, a timing-sensitive path causes generated ATPG patterns to fail simulation. Your analysis concludes that if you could hold two of the pins of a four-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.

You can add the required constraints using either the TestMAX ATPG GUI or the `add_atpg_constraints` command.

To add constraints using the TestMAX ATPG GUI:

1. Select Constraints > ATPG Constraints > Add ATPG Constraints.

The Add ATPG Constraints dialog box appears.

2. For each constraint, 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.

Defining, Reporting, and Removing No Detection Credit Cells

You can use the `-no_detection_credit` option of the `add_atpg_constraints` command to define cells that you don't want credited as faults but can still be used for good machine values. Note that this feature is available only in TestMAX ATPG.

You can also report these cells and remove them, as shown in the following examples.

The following example shows a file, `round23q.list`, containing a list of no detection credit cells:

```
// initial test
de_encrypt/round_reg_2_/Q
de_encrypt/round_reg_3_/Q
```

This file is read by the `-no_detection_credit` option of the `add_atpg_constraints` command, as shown in the following example:

```
TEST-T> add_atpg_constraints -no_detection_credit round23q.list
2 no_detection_credit_cells were successfully read in.
```

You can also report these cells using the `-no_detection` option of the `report_atpg_constraints` command, as shown in the following example:

```
TEST-T> report_atpg_constraints -no_detection_credit
No_detection_credit_cells list: #no_detection_credit_cells=2
1: gate_id=8564, instance=de_encrypt/round_reg_3_ (chain=120, position=3)
2: gate_id=8570, instance=de_encrypt/round_reg_2_ (chain=120, position=4)
```

To remove no detection credit cells, use the `-no_detection_credit` option of the `remove_atpg_constraints` command, as shown in the following example:

```
TEST-T> remove_atpg_constraints -no_detection_credit
```

The `-no_detection_credit` option changes the fault status of the no detection credit cells from AU (ATPG Untestable) to NC (Not Controlled), and can be used in the DRC and TEST modes.

Using ATPG Constraints to Control ATPG Assertions

In this example, a library module, FIFO, has two control inputs: push and pop. Under normal operation, the control logic for push and pop ensures that both inputs 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 adding a temporary gate to facilitate the ATPG constraint. You can then define the actual constraint.

In this case, you want a logic function with a single output that can be monitored to assure that the push and pop pins are at the required logic states.

You can use the TestMAX ATPG GUI or the `add_atpg_primitives` command to define an ATPG primitive to implement this logic function and apply the ATPG constraints to the push and pop inputs. You can then use the `add_atpg_constraints` command to apply an ATPG constraint to the newly created primitive.

The following example flow uses the TestMAX ATPG GUI to define an ATPG primitive and apply the ATPG constraints to the applicable inputs:

1. Select Constraints > ATPG Primitives > Add ATPG Primitives.

The Add ATPG Primitives dialog box appears.

2. In the Type list, select the `SEL01` ATPG primitive. (For a list of all available ATPG primitives, see the description of 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 other inputs are 0. For the example two-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 for the primitive.
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.

The following example shows how to add the primitive using the `add_atpg_primitives` command:

```
DRC-T> add_atpg_primitives 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.

To apply a constraint to the output of the newly added primitive, use the `add_atpg_constraints` command, as shown in the following example:

```
DRC-T> add_atpg_constraints spec_LABEL 1 -module FIFO FIFO_CTRL
```

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

Using the Random Decision Option

You can use the Random Decision check box in the Run ATPG dialog box to specify how TestMAX ATPG 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 `set_atpg` command is equivalent to checking the Random Decision check box:

```
TEST-T> set_atpg -decision random
```

See Also

- [Specifying General ATPG Settings](#)

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.

TestMAX ATPG 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_faults -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 is higher than your target.

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. For details, see [Quickly Estimating Test Coverage](#). 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.

Improving Test Coverage With Test Points

You can improve TestMAX ATPG test coverage by adding control and observation points to specific areas with known low controllability and observability. TestMAX ATPG 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 99 percent range.

You can use TestMAX ATPG to further improve test coverage by performing an analysis to determine the optimal placement of test points.

The following sections describe how to improve test coverage with test points:

- [Test Points Analysis Options](#)
- [Running the Test Points Analysis Flow](#)
- [Limitation](#)

Test Points Analysis Options

You can use the `analyze_test_points` command to select a particular type of analysis:

```
analyze_test_points -target <pattern_reduction | testability |  
fault_class>
```

The analysis options are described as follows:

- `pattern_reduction` — Uses static analysis with SCOAP (Sandia Controllability and Observability Analysis Program) numbers to target reduced pattern size with observe points (does not require prior ATPG).
- `testability` — Uses iterative static analysis with random patterns to target improved test coverage with control and observe points (does not require prior ATPG).
- `fault_class` — Uses dynamic analysis with fault cone topology to target improved test coverage with observe points for fault classes (requires initial ATPG for analysis of fault cones).

Running the Test Points Analysis Flow

The following steps describe the flow for running test-point insertion:

1. Run the `run_atpg -auto` command or use any other method for generating patterns.

If you do not perform ATPG before running the `analyze_test_points` command, all undetected faults are analyzed, which might result in very long runtimes.

2. Run the `analyze_test_points` command to generate a list of test points. For example:

```
analyze_test_points -target fault_class -test_points_file tp_file_a
```

You can run the `analyze_test_points -target testability` or the `analyze_test_points -target pattern_reduction` commands before or after ATPG to obtain a list of test points. A previous ATPG run is only required when you use the `-target fault_class` option with the `analyze_test_points` command.

3. Use the `run_atpg -auto` command to launch another ATPG run. TestMAX ATPG estimates the test coverage improvement by reading in the generated test points file. For example:

```
run_atpg -auto -observe_file tp_file_a
```

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

4. Use TestMAX DFT to insert the test points by reading in the file generated by the `analyze_test_points` command, then rerun TestMAX ATPG on the new netlist to generate the final ATPG patterns and coverage.

Limitation

Note the following limitation associated with test points analysis:

- If you have a LSSD design, you can use the `analyze_test_points` command in TestMAX ATPG. However, TestMAX DFT does not support the insertion of observe and control test points on this style of scan. In this case, a TESTXG-61 message is issued in TestMAX DFT.

Limiting the Number of Patterns

By default, the number of ATPG patterns TestMAX ATPG 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 example:

```
TEST-T> set_atpg -patterns 1234
```

If there is a pattern limit in effect, you can turn it off by running 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 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](#)).
- You find that after performing pattern generation, you have ND (not detected) faults remaining. See [Analyzing the Cause of Low Test Coverage](#).
- You have aborted buses reported during design rule checking (DRC). See [Analyzing Buses](#).
- 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 ATPG Checkpoint Files

You can use ATPG checkpoint files to retain generated fault lists and patterns in case of a crash.

If you specify the `-checkpoint` option of the `set_atpg` command, fault lists and patterns are periodically saved to files at a specified checkpoint interval during ATPG pattern generation.

The following example saves faults to the `chkp_fault` checkpoint file and patterns to the `chkp_patt` file at intervals of 3600 CPU seconds:

```
set_atpg -checkpoint { 3600.00 chkp_fault chkp_patt }
```

The following steps describe how to recover faults and patterns from checkpoint files after a crash:

1. Invoke TestMAX ATPG or TestMAX ATPG again.
2. Run the `build` command and DRC, and specify the ATPG settings to the same state before you started the `run_atpg` command in the original run.
3. Use the `read_faults` command to read the faults saved in the fault checkpoint file. These faults replace all faults added from the `add_faults` commands and any other commands associated with adding faults in the original run. The following example reads in the `chkp_fault` file and retains the fault codes:

```
read_faults chkp_fault -force_retain_code
```

4. Read the patterns saved in the pattern checkpoint file and transfer them to the internal pattern buffer, as shown in the following example:

```
set_patterns -external chkp_patt run_simulation -store set_patterns  
-internal
```

5. Start the `run_atpg` command using the same command options that you used in the original run.
6. Include all post-ATPG commands in the run script so the ATPG process can finish as intended.

Creating Test Patterns for Diagnosing Scan Chain Failures

By default, the `run_atpg` command creates an initial pattern, called a chain test, to test scan cells, scan clocking, and scan enable signals. This pattern does not pulse capture clocks or asynchronous set and reset signals. It only loads and unloads the repeating pattern of 0 and 1 signals. If the chain test pattern fails, TestMAX ATPG assumes that all failures are caused by scan chain defects.

If your design uses DFTMAX compression with high X-tolerance, you can use the

`-xtol_chain_diagnosis` option of the `set_atpg` command to create additional patterns that improve the identification of failing scan chains, failing scan cells, and multiple chain defects. When you specify the `-xtol_chain_diagnosis` option, the `run_atpg` command creates two additional sets of patterns:

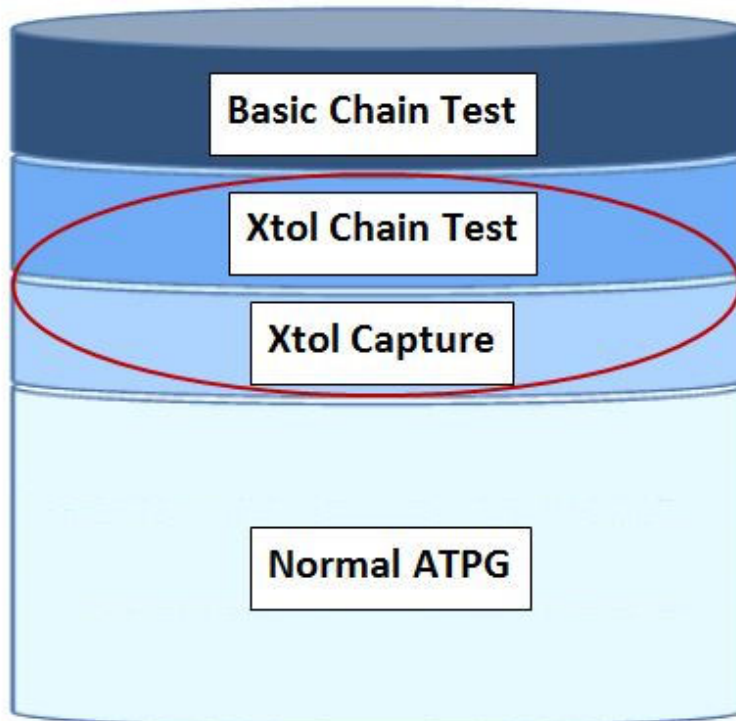
- X-tolerant chain tests (also known as augmented chain test patterns)
- X-tolerant capture patterns

After generating these additional pattern sets, the `run_atpg` command continues generating normal capture patterns and targets the remaining undetected faults.

Figure 1 shows the components of the final generated pattern set when the

`-xtol_chain_diganosis` option is enabled.

Figure 94 Pattern Set Generated Using the `-xtol_chain_diganosis` Option



The following sections explain the process for generating test patterns for diagnosing scan chain failures:

- [Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns](#)
- [Generating Pattern Sets](#)

For more information on DFTMAX high X-tolerance scan compression, see the "Managing X Values in Scan Compression" chapter in the *DFTMAX User Guide*.

Understanding DFTMAX Unload Modes and Chain Diagnosis Patterns

The X-tolerant chain tests and X-tolerant capture patterns use additional unload modes from the high X-tolerance DFTMAX architecture. These modes dynamically configure the compressor so each internal scan chain is observed on no more than one scan output pin of the device under test. These modes are classified as either N:1 or 1:1 modes.

The N:1 modes observe multiple internal scan chains on each scan output pin. The 1:1 modes observe a single internal scan chain on each scan output pin, which is optimal for mapping tester failures back to a failing scan cell. Each 1:1 mode can observe only a subset of all internal scan chains, inversely proportional to the compression ratio. As a

result, TestMAX ATPG implements several 1:1 modes so it can directly observe all internal scan chains. You can use the

```
report_compressors -unload
```

 command to report the number of unload modes and list the 1:1 modes.

The X-tolerant chain tests use the 1:1 X-tolerant modes to observe individual chains. Each 1:1 mode is enabled for 20 shift cycles, and each X-tolerant chain test pattern has an additional padding pattern. To calculate the number of X tolerant chain test patterns, multiply the compression ratio by 20 and divide by the number of shift cycles. You should double this number to account for padding patterns. For example, a design with 32 scan I/Os, 1600 internal chains, and a maximum chain length of 250 requires 8 ($=2*1600*20/(32*250)$) additional patterns.

The X-tolerant capture patterns use all available N:1 or 1:1 X-tolerant modes. These patterns pulse capture clocks and target primary and secondary faults similar to patterns produced from standard ATPG. However, X-tolerant capture patterns are optimized to improve the diagnosis of failing scan cells, while standard ATPG maximizes the number of faults detected per pattern. When you specify the `-xtol_chain_diagnosis low` option of the `set_atpg` command, 32 capture patterns and 32 padding patterns are generated that use only N:1 X-tolerant modes. These modes provide a limited improvement for identifying failing scan cells. When the `high` option is specified, TestMAX ATPG generates 10 capture patterns for each available 1:1 X-tolerant mode. Specifying the `high` option creates additional patterns which provide a significant improvement for diagnosing chain defects.

Generating Pattern Sets

To generate pattern sets for diagnosing scan chain failures, specify the

```
-xtol_chain_diagnosis
```

 option of the `set_atpg` command, followed by the `run_atpg` command.

The following example creates a single pattern set for both accurate diagnosis of scan chain defects and high manufacturing test coverage:

```
TEST-T> set_atpg -xtol_chain_diagnosis high
TEST-T> run_atpg -auto
```

You might need two separate patterns sets: one for accurate diagnosis of scan chain defects and another for high manufacturing test coverage. You can use the

```
run_atpg -only_chain_diagnosis
```

 command to terminate ATPG after generating the X-tolerant chain tests and capture patterns. The following example shows how to generate a separate pattern set for diagnosis of chain defects. This pattern set includes an additional 100 standard ATPG patterns to further improve diagnosis resolution.

```
TEST-T> set_atpg -xtol_chain_diagnosis high
TEST-T> run_atpg -only_chain_diagnosis -auto
TEST-T> set_atpg -xtol_chain_diagnosis off
TEST-T> set_atpg -patterns [expr [sizeof_collection \
    [get_patterns -all]] + 100]
TEST-T> run_atpg -auto
```

See Also

- [Performing Scan Chain Diagnosis](#)
- [Preparing for ATPG](#)

Performing Scan Chain Diagnosis

Functional logic diagnostics assumes that scan data is properly loaded and unloaded. If patterns show failures during the chain test, a chain defect is interfering with the loading and unloading processes. TestMAX ATPG scan chain diagnostics isolates the defects that affect scan chain shifting.

You can use both standard scan patterns and DFTMAX patterns for scan chain diagnostics. If you are testing an X-tolerant design, TestMAX ATPG can generate additional chain test patterns that use the X-tolerant modes to directly observe a group of chains at the scan outputs. For more information on this process, see the [Creating Test Patterns for Diagnosing Scan Chain Failures](#) section.

The following sections explain how to perform scan chain diagnostics:

- [Running Scan Chain Diagnosis](#)
- [Understanding the Scan Chain Diagnosis Report](#)
- [Diagnosing Defects Related to Power Issues](#)

Running Scan Chain Diagnosis

Chain diagnostics are enabled by default, and can be disabled using the `set_diagnosis -noauto` command. For optimal accuracy, you should use failure data from ten or more patterns. Always provide TestMAX ATPG with as many failures as possible, including failures that occur when running the chain test pattern. The following example shows how to set up and run scan chain diagnosis:

```
set_patterns -external pat.stil
set_diagnosis -auto
run_diagnosis fail.log
```

Understanding the Scan Chain Diagnosis Report

Scan chain diagnosis identifies several types of defects that affect shifting, including slow clock signals that cause a hold time violation and reset lines stuck at an active value.

The output diagnosis report identifies the location of stuck-at, slow-to-rise, slow-to-fall, fast-to-rise or fast-to-fall faults. The latter two fault types address hold time problems that affect the scan chain shift operation.

To isolate the location of the defect, TestMAX ATPG scan chain diagnosis analyzes the control and observability of scan cells. For example, assume that a stuck-at fault prevents scan cell A from shifting to scan cell B. In this case, scan cell A, and all cells located before it, drive valid values to functional logic. These cells appear as tied cells when they are unloaded, and are therefore unobservable. Scan cell B, and all cells that follow it, drive invalid values to functional logic, but they might capture observable valid values.

The diagnosis report includes a set of possible defect locations (chain, cell position, and instance name). It also includes a match percentage score that indicates the confidence of each location. This score is a percentage that measures the degree to which a failure on the tester matches a simulated chain defect at that location. The predicted type of defect is also included in the diagnosis report. 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 likely the cause of the failures. It also identifies the three scan cell locations that have an output with the physical defect. Some chain test patterns do not fail on the tester, even though the failures appear to be related to a chain defect. Also, the tester might not collect all failures for the chain test patterns. In both cases, scan chain diagnostics cannot analyze or locate the defect location. To address these situations, use the `-assume_chain_defect` option of the `run_diagnosis` command to specify a defect location and force TestMAX ATPG to obtain the scores.

Diagnosing Defects Related to Power Issues

TestMAX ATPG can also improve the characterization and diagnosis of chain defects related to power issues. To screen failures based on switching activity, TestMAX ATPG uses the quiet chain test patterns instead of regular chain test patterns. Specify the

`-quiet_chain_test` option of the `set_atpg` command to enable the `run_atpg` command to generate quiet chain test patterns.

For more information, see the [Applying Quiet Test Patterns](#) section.

Creating End-of-Cycle Measures in ATPG Patterns

The TestMAX ATPG combinational ATPG algorithm is based on a `preclock` measure of scan outputs and regular design outputs. This preclock measure requires a fundamental event order within a tester cycle of:

- Force inputs
- Measure outputs
- Pulse capture clocks (optional)

This preclock measure has been chosen because it enables superior ATPG pattern generation performance without compromising on pattern count or tester cycle count.

Many ASIC vendors and users prefer to have patterns with an event order using `postclock` or End-of-Cycle measures. A `postclock` measure seems to be a more comfortable form because it matches the event order of most functional patterns and is perhaps easier to debug.

Many ASIC vendors claim that they can only accept `postclock` measure format. It is rare to find an ASIC tester which does not support the `preclock` measure. More often than not it is a software translation limitation rather than a tester limitation. The fundamental event order for a `postclock` measure cycle is:

1. Force inputs
2. Pulse capture clocks (optional)
3. Measure outputs

The TestMAX ATPG combinational ATPG algorithm will not produce this `postclock` form of patterns. However, the `postclock` style of patterns can be created using some post processing techniques applied during the `write_patterns` command.

Drawbacks of Using End-of-Cycle Measures

Here are some drawbacks of creating End-of-Cycle style ATPG patterns:

- The internal pattern format is in preclock format and attempting to compare internal patterns to an external form in STIL, Verilog, VHDL, and so forth. is more difficult.
- At least one additional tester cycle is needed for every ATPG pattern. This additional cycle is placed in the load_unload procedure and performs a scan chain pre-measure before the Shift procedure.
- Capture Clock procedures cannot be condensed into a single tester cycle and must be defined with a minimum of 2 tester cycles. The first cycle performs a force PI, measure PO, and the second cycle performs an optional clock pulse.

In general terms, the cost of implementing the End-of-Cycle measure is two additional tester cycles for every ATPG pattern generated. There is no increase or decrease to overall test coverage or the number of ATPG patterns produced by choosing End-of-Cycle measures over preclock measure. This can or cannot be significant, depending upon your budget for test cycles or tester time.

Requirements Needed to Produce End-of-Cycle Measures

To create End-of-Cycle style ATPG patterns with the `write_patterns` command the following setup steps are required:

1. The DRC procedure file must contain a timing definition block and the time at which outputs and scan outputs are measured must be defined to occur at the end of a test cycle, after any potential clock pulses.
2. All capture procedures must be defined using two or more test cycles and the event order must be:

```
cycle 1: force PI's, measure PO's cycle 2: mask PO's, pulse clocks
```
3. The load_unload procedure must pre-measure the first scan chain output before the first scan shift is performed.

In this case, you are still measuring outputs before a clock. You do not change the fundamental event order which must continue to be: 1) force PI's, 2) measure PO's, 3) pulse clocks; make sure that relative to a single tester cycle timing, the measures occur after any clock pulses. For example, if you define tester timing for a 100nS period in which PI's are forced at offset zero, a clock is pulsed from 50 to 70ns and outputs are measured at 99ns, then your "capture_XXX" procedures produce a 2-cycle timing of:

```
time action                               cycle
-----
000 force PI's                             1
```

Chapter 13: Optimizing ATPG

Deleting Top-Level Ports From Output Patterns

```

050 assert clock (but inhibited)      1
070 remove clock                     1
099 measure PO's                     1
100 force PI's (no change needed)    2
150 assert clock                     2
170 remove clock                     2
199 measure PO's (masked)            2

```

The fundamental event order is still that of the preclock timing under which the ATPG patterns are generated but the per cycle timing is such that measures are performed at the end of a tester cycle.

See Also

- [End-of-Cycle Measures and Load_Unload](#)
- [End-of-Cycle Measures and Timing](#)
- [End-of-Cycle Measures and Capture Procedures](#)

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-T> add_net_connections pwr1 pwr2 pwr3 -remove
```

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

Detecting Faults Multiple Times Using N-Detect

The N-detect functionality detects faults a specified number (n) of times during ATPG. The default is one fault detection. During fault simulation, the fault is kept in the active list until it is detected n times. Detecting faults with multiple patterns identifies defects that cannot be modeled with standard fault models. Examples include transistor stuck-open or cell-level faults.

The N-detect capability is implemented with `-ndetects` option of the `run_atpg`, `run_fault_sim`, and `report_faults` commands.

Note the following:

- The N-detect capability increases the pattern size, memory consumption, and runtime.
- All fault models are supported, except the IDDQ and path delay fault models.
- Multicore ATPG, distributed processing, full-sequential ATPG, and fault simulation are not supported.
- N-detect ATPG should be used with the `set_atpg -decision random` command to increase the probability of detecting the faults in different ways. TestMAX ATPG does not guarantee that each fault is detected in different ways.

See Also

- [Distributed ATPG Limitations](#)

WGL Pattern Generation Options

The following sections explain the various WGL pattern generation options:

- [Creating LSI-Compatible WGL Patterns](#)
- [Creating NEC-Compatible WGL Patterns](#)
- [WGL Scan Chain Padding](#)
- [WGL Scan Chain Definitions](#)
- [Macro Usage in WGL](#)
- [Grouping Bidirectional Port Data in WGL](#)
- [Controlling Port Data Order in WGL](#)
- [Specifying Windowed Measures in WGL](#)
- [Delayed Input Force Timing and Force Prior in WGL](#)
- [Balancing Vector and Scan Statements in WGL](#)
- [Mapping Bidirectional Ports Within Vector Statements in WGL](#)
- [Mapping Bidirectional Ports Within Scan Statements in WGL](#)
- [Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL](#)
- [Selecting Scan Chain Inversion Reference in WGL](#)

- [Effect of CELLDEFINE in WGL](#)
- [Ambiguity of the Master Cell in WGL](#)

Creating LSI-Compatible WGL Patterns

To produce LSI-compatible WGL output you need to use the `set_drc`, `set_buses`, `set_simulation`, and `set_wgl` commands, as shown in the following example:

```
set_drc -nomulti_captures_per_load
set_buses -external_z x

set_simulation -xclock_gives_xout

set_rules c13 error

set_rules z4 error

set_wgl -nolast_scan

set_wgl -scan_map keep

set_wgl -pre_measured

set_wgl -inversion_reference master

set_wgl -chain_list shift

set_wgl -nomacro -nopad -nogroup_bidis

set_wgl -bidi_map { 0x 0-
1x 1- xx x- z0 -0 z1 -1 zx -x zz -z }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see "[Defining the Shift Procedure](#)."
- Scan Chain names defined in the STIL procedure file must not contain spaces or other white space. For example, use "chain_1" instead of "chain 1".
- You must define the end-of-cycle timing, as follows:
 1. The timing block must define the end-of-cycle measure. For more information, see "[Creating End-of-Cycle Measures in ATPG Patterns](#)."
 2. The load_unload procedure must use pre-measure scan outputs. For more information, see "[Defining the load_unload Procedure](#)."

- You can use the ReflectIO protocol. However, unless all bidirectional pins are fully controlled, you should avoid this protocol since it can create patterns which fail in simulation and might contain contention when all BIDI pins are not controlled.

For a design with bidirectional ports, the ReflectIO protocol causes each `capture_XXX` procedure to use the `reflectIO` style of syntax. For example, you can define all clocks and then issue the `set_drc -bidi_control_pin` command followed by a `write_drc` command to create a template STIL procedure file. Then modify the `capture_XXX` procedures to appear similar to the following 3-cycle protocol:

```
capture_CLK { W _default_WFT_; V { _pi=\r15 # ; _po=\j \r44 % ; } #  
  force PI, TN=1 V { TN=0; _io=\r32 Z ; _po=\j \r44 X ; } # disable  
  bidis V { _io=\m \r32 % ; CLK=P; } # reflect bidis, pulse CLK }
```

- All `capture_XXX` procedures for clocks must have the same number of tester cycles, `V{...}` constructs. If you use a three cycle capture for 'CLK', then you must also use a three-cycle capture for 'RST', 'CLK2', and so forth. This includes the non-clocking capture procedure named `capture`.
- Use a [test_setup procedure](#) to initialize all input pins to a known value in the first test cycle. Initialize bidirectional pins to Z.
- If inputs are applied with a delay on the tester, then the Timing block of the STIL DRC procedure file should include a "ForcePrior" or "P" character at time offset zero of each cycle before applying the required value within that cycle. This generates a V6 warning during DRC which will have to be ignored. There is an example of ForcePrior at the end of topic: Controlling Pin Timing in STIL
- You can use only one timing block.
- Use the `-order_pins` option of the `write_patterns` command when writing WGL patterns.
- Do not use the `-measure_forced_bidis` option of the `write_patterns` command when writing WGL patterns
- Contact LSI for the latest advice and application notes concerning the use of TestMAX ATPG.

See Also

- [End-of-Cycle Measures and Load_Unload](#)
- [End-of-Cycle Measures and Timing](#)
- [End-of-Cycle Measures and Capture Procedures](#)

Creating NEC-Compatible WGL Patterns

To produce NEC-compatible WGL output, you need to use both the `set_simulation` and `set_wgl` commands, as shown in the following example:

```
set_simulation -strong_bidi_fill
set_wgl -nomacro
set_wgl -nopad
set_wgl -notester_ready
set_wgl -inversion_reference master

set_wgl -scan_map dash

set_wgl -bidi_map { 0x 0-
  1x 1- xx x- z0 -0
  z1 -1
  zx -x zz -z -x -- z- -- }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see [Defining the Shift Procedure](#).
- You must define the end-of-cycle timing, as follows:
 1. The timing block must define the end-of-cycle measure. For more information, see [Creating End-of-Cycle Measures in ATPG Patterns](#).
 2. The `load_unload` procedure must use pre-measure scan outputs. For more information, see [Defining the load_unload Procedure](#).
 3. The clock capture procedures must use the two-cycle end-of-cycle measure format. For more information, see [Defining Capture Procedures in STIL](#).
- You must explicitly initialize bidirectional ports to non-Z values in the `load_unload` procedure.

Use the `test_setup` procedure to eliminate uninitialized ports at T=0. For more information, see [Defining the test_setup Procedure](#).

Use the `test_setup` procedure to eliminate floating ports at T=0.

Do not use the `-measure_forced_bidis` option of the `write_patterns` command when writing WGL patterns.

Use the WGL to ALB to Verilog translation path. Other paths, such as WGL to ALB to CPT, have not been validated to work.

See Also

- [End-of-Cycle Measures and Load_Unload](#)
- [End-of-Cycle Measures and Timing](#)
- [End-of-Cycle Measures and Capture Procedures](#)

WGL Scan Chain Padding

When a design has more than one scan chain and the scan chains are not all the same length then you have the option of causing the WGL patterns to be written so that all scan load and unload data is the same length (`set_wgl -pad`) or is only the length of the scan chain (`set_wgl -nopad`). The default is not to pad, and this is preferred by most vendors.

When padding is enabled, the pad value can be any one of 0, 1, or X and you select which by the `-pad_character` option of the `write_patterns` command when the WGL patterns are written. The default when padding is enabled, is to pad with a zero. *Note*, however, that when padding is enabled and a particular pad character is chosen that this will have no effect on the padding used for the chain test patterns. The padding for chain test patterns is always the continuation of the repeating string 0011.

The first example shows a portion of the WGL `SCANSTATE` block for a design with three scan chains of length 2, 3, and 8 bits where padding is disabled.

```
# scan chain padding disabled
scanstate
  c1L0 := c1G(11);
  c2L1 := c2G(011);
  c3L2 := c3G(00110011);
  c1E3 := c1G(00);
  c2E4 := c2G(100);
  c3E5 := c3G(11001100);
```

The second example shows the same data with scan chain padding enabled and a pad character of X used so that it is easier to see where the padding occurs. For scan load strings the padding occurs on the left (first shifted in) for all shorter chains. For scan unload strings the padding occurs on the right (last shifted out).

```
# scan chain padding enabled with pad = X
scanstate
  c1L0 := c1G(XXXXXX11);
  c2L1 := c2G(XXXXX011);
  c3L2 := c3G(00110011);
  c1E3 := c1G(00XXXXXX);
  c2E4 := c2G(100XXXXX);
  c3E5 := c3G(11001100);
```

WGL Scan Chain Definitions

By convention, the `scanchain` block in WGL defines the instances in the physical sequence of each scan chain, starting at the scan input, and traversing to the scan output. The number of instances in the scan chain matches the number of bits called for in the `scanstate` block for loading or observing from the scan chain.

On some designs, generally those with JTAG used during ATPG, the final scan chain shift is done outside of the scan loop. This translates into the "scan()" vector being shortened by one bit and an additional `vector()` or more being added to the procedure to handle the final shift outside of the scan statement. Now most WGL translators require that the number of bits defined in the `scanchain` block match the physical length of the scan chain. However, a few require that the number of bits match the length of data to be loaded by the "scan()" statements. The `-chain_list` option controls how the scan chain is listed in the `scanchain` block. The default is `all` which causes all instances in the scan chain to be included in the defining list. Optionally specifying `shift` causes the list to match only those bits loaded by the "scan()" statements.

The first examples shows the default `scanchain` block for a design with two scan chains of 5 and 4 bits.

```
# set_wgl -chain_list all
scanchain
  chain1 ["si1", "A4", !, "A3", "A2", "A1", "A0", "so1" ];
  chain2 ["si2", "B3", "B2", "B1", "B0", !, "so2" ];
end
```

The second example shows the same `scanchain` block when the final shift of the scan chain is done outside of the Shift procedure and a selection of `-chain_list shift` is used. The final instance in each scan chain "A1", and "B1" have been omitted from the scan chain definitions.

```
# set_wgl -chain_list shift
scanchain
  chain1 ["si1", "A4", !, "A3", "A2", "so1" ];
  chain2 ["si2", "B3", "B2", !, "so2" ];
end
```

Macro Usage in WGL

WGL supports the definition of macros. Macros can be used to represent commonly repeated sequences and the use of macros can lead to more compact WGL pattern files. TestMAX ATPG will write WGL using macros if the `set_wgl -macro` option has been used. Most vendors do not support macros as this requires a more complex WGL reader and so the TestMAX ATPG default is not to use macros.

When macros are enabled, TestMAX ATPG adds various macro definitions to the WGL pattern file. The following example is a macro for a capture procedure for the port CLK. There will generally be a macro for each procedure in the DRC file.

```
# an example macro definition
macro capture_CLK (SDI3_I, SDO1_I, D0_I, D2_I, CLK, RSTB, SDI,
    INC, SCAN_9, SDI3_O, SDO1_O, D0_O, D2_O, P, SDO, CO)
    vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
        @INC @SCAN_9 X X X X XX XX X ];
    vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
        @INC @SCAN_9 @SDI3_O @SDO1_O @D0_O @D2_O @P
        @SDO @CO ];
    vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I 1 @RSTB @SDI
        @INC @SCAN_9 X X X X XX XX X ];
endmacro
```

The first following example shows a segment from a WGL *PATTERN* block which does not use macros and the second example is the same information using macros.

```
# example patterns without macros
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
    "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
    "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];

{ scan_test }
{ pattern 0 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
    scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
    output [c1:c1U0], output [c2:c2U1], output [c3:c3U2],
    input [c1:c1L0], input [c2:c2L1], input [c3:c3L2];
{ capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 1 X X X X X X X X X ];
vector(tp1) := [ - Z - - 0 0 0 0 0 1 Z 0 Z Z Z 0 1 0 0 ];

{ pattern 1 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
    scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
    output [c1:c1U3], output [c2:c2U4], output [c3:c3U5],
    input [c1:c1L3], input
[c2:c2L4], input [c3:c3L5];
{ capture_CLK }
vector(tp1) := [ Z Z 0 Z 0 1 1 1 0 0 X X X X X X X X X ];
vector(tp1) := [ - - 0 - 0 1 1 1 0 0 Z Z X Z Z 0 1 0 1 ];
```

```

vector(tp1) := [ Z Z 0 Z 1 1 1 1 0 0 X X X X X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
  scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
  output [c1:c1U6], output [c2:c2U7], output [c3:c3U8],
  input [c1:c1L6], input
[c2:c2L7], input [c3:c3L8];
  capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 1 1 1 1 X X X X X X X X X ];
vector(tp1) := [ - Z Z Z 0 0 1 1 1 1 Z 0 1 0 Z 0 0 0 0 ];

# example patterns using macros
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
"RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
"D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
test_setup

{ scan_test }
{ pattern 0 }
load_unload(c1U0, c2U1, c3U2, c1L0, c2L1, c3L2)
capture_RSTB(-, Z, -, -, 0, 1, 00, 0, 1, Z, 0, Z, Z, Z0, 10, 0)

{ pattern 1 }
load_unload(c1U3, c2U4, c3U5, c1L3, c2L4, c3L5)
capture_CLK(-, -, 0, -, 0, 1, 11, 0, 0, Z, Z, X, Z, Z0, 10, 1)

{ pattern 2 }
load_unload(c1U6, c2U7, c3U8, c1L6, c2L7, c3L8)
capture_RSTB(-, Z, Z, Z, 0, 1, 11, 1, 1, Z, 0, 1, 0, Z0, 00, 0)

```

Grouping Bidirectional Port Data in WGL

In WGL patterns a bidirectional port appears as two characters, one for the force input value and another for the measure output value. These two characters can appear side by side (grouped), or in independent locations within the data (split columns). The `set_wgl -group_bidis` command causes the two characters to appear as a single column of two characters, with the first representing the input action and the second representing the output action. The default is to present the bidirectional port data as two separate columns.

The first following example uses grouped bidis and in this example there are four bidirectional ports which appear as the first four columns of each `vector()` statement. The characters "ZX" indicate a force of Z (no force) and a measure of X (mask measure).

```
# example patterns using grouped bidis
pattern_group_ALL ("SDI3", "SDO1", "D0", "D2", "CLK",
  "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "P[0]",
  "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 0 0 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 0 X X X X X ];
```

In the second following example split bidis are used. Notice that the pattern data no longer has any two character columns. The port order list now lists each bidirectional port twice and follows each by either :I or :O to indicate direction. The two parts of the bidirectional port data do not appear as adjacent data in the vector, they can appear at any position.

```
#example patterns using split bidis
pattern_group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I,
  "CLK", "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O,
  "SDO1":O, "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]",
  "CO")
{ test_setup }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 0 X X X X X X X X X ];
```

Controlling Port Data Order in WGL

The default pin data order of the WGL pattern data follows the order in which the ports are defined in the design's top module. By changing the order of the ports in the top module you can affect the order of the WGL data.

There is also the `-order_pins` option of the `write_patterns` command. Use of this option causes the ports to occur in the order: inputs, bidis, and outputs. Within each grouping the port data order matches the order the ports are defined in the design's top module.

For a top-level design with port order:

```
module TOP (I1,B1,O1,O2,O4,O3,B3,B2,I3,I2);
```

the following two examples illustrate the difference in data order.

```
# default port order using grouped bidis
pattern_group_ALL ("I1", "B1", "O1", "O2", "O4", "O3", "B3",
  "B2", "I3", "I2")
{ test_setup }
vector(tp1) := [ 0 ZX X X X X ZX ZX 0 0 ];
vector(tp1) := [ 0 ZX 1 1 1 1 ZX ZX 0 0 ];
vector(tp1) := [ 1 0X 1 1 1 1 0X 0X 1 1 ];
```

```
# port order using ORDER_PINS option
pattern group_ALL ("I1", "I3", "I2", "B1", "B3", "B2", "O1",
  "O2", "O4", "O3")
{ test_setup }
vector(tp1) := [ 0 0 0 ZX ZX ZX X X X X ];
vector(tp1) := [ 0 0 0 ZX ZX ZX 1 1 1 1 ];
vector(tp1) := [ 1 1 1 0X 0X 0X 1 1 1 1 ];
```

Specifying Windowed Measures in WGL

The default WGL patterns written will define timing which performs a strobed measure (single time measure) when outputs are to be measured. If your tester supports window measure (measure over a continuous range of time) and you would like to have a windowed measure, this type of measure can be created. This time you do not use any `set_wgl` options, but instead make edits to the `Timing` block of the DRC procedure file. Note that these edits must be made before performing the `run_drc` command and before generating ATPG patterns.

The following example illustrates a window measure for the symbolic group `out_ports` defined elsewhere in the DRC file. The STIL language specifies that the uppercase {H,L,T,X} characters indicate a strobed measure, and the lowercase characters {h,l,t,x} call for a window measure. In this specific example the ports associated with the symbolic group `out_ports` is continuously measured for high/low/tristate values between an offset of 450 nS and 490 nS from the beginning of the tester cycle. The `'490ns' x;` text specifies the window measure is turned off at this time and is text which is not needed for a strobed measure.

```
Timing {
  WaveformTable "WINDOW_COMPARE" {
    Period '1000ns';
    Waveforms {
      clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
      input_ports { 01Z { '0ns' D/U/Z; } }
      out_ports { X { '0ns' X; } }
      out_ports { HLT { '0ns' X; '450ns' h/l/t; '490ns' X; } }
      bidi_ports { X { '0ns' X; } }
      bidi_ports { 01Z { '0ns' D/U/Z; } }
      bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
    }
  }
}
```

Delayed Input Force Timing and Force Prior in WGL

It is a common requirement when running the pattern timing to require that one or more pins have their inputs applied at some delayed offset from the beginning of the tester

cycle. This is another adjustment that is made in the `Timing` block of the DRC file rather than with a `set_wgl` command. In the following example the symbolic pin group `input_grp2` has its pattern data applied at an offset of 5ns into the tester cycle.

What is the value on the pins of group `input_grp2` from the start of the cycle to offset 5ns? The answer is that the value is undefined unless you specify some value in the timing block such as 0, 1, X, or perhaps Z. What if you just want the port to continue the value from the previous tester cycle? In WGL as well as STIL there is a "Force Prior" concept which indicates the value is to be whatever was previously assigned.

To cause the WGL output to call for a Force Prior, edit the `Timing` block of the DRC file before performing a `run_drc` command and before generating any ATPG patterns and add the "P" character to the beginning of the timing definition for those inputs which are applied after a delay. Note that this use of the "P" waveform character will produce a V6 warning which you can ignore. In the following example, the symbolic pin group `input_grp2` calls for the Force Prior value.

```
WaveformTable "FORCE_PRIOR_EXAMPLE" {
  Period '1000ns';
  Waveforms {
    CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
    CLOCK { 01ZN { '0ns' D/U/Z/X; } }
    RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
    RESETB { 01ZN { '0ns' D/U/Z/X; } }
    input_grp1 { 01ZN { '0ns' D/U/Z/X; } }
    input_grp2 { 0 { '0ns' P; '5ns' D; } }
    input_grp2 { 1 { '0ns' P; '5ns' U; } }
    input_grp2 { Z { '0ns' P; '5ns' Z; } }
    out_ports { HLTX { '0ns' X; '490ns' H/L/T/X; } }
    bidi_ports { 01ZN { '0ns' Z; '20ns' D/U/Z/X; } }
    bidi_ports { X { '0ns' X; } }
    bidi_ports { HLT { '0ns' X; '490ns' H/L/T; } }
  }
} # end FORCE_PRIOR_EXAMPLE
```

Balancing Vector and Scan Statements in WGL

By default, the last event in the WGL pattern file is a scan chain unload to observe the measure values of the final capture clock. This corresponds to a `scan()` statement in the WGL file. Some vendors require that the final event in the WGL pattern file be a `vector()` statement to ensure that clocks are off and to provide a symmetric order where the scan statements are always followed by an identical number of vector statements. You can cause the final events in the WGL file to be vector statements by using the `set_wgl -nolast_scan` option to change the default behavior.

The first following example shows the default final pattern where the last event is a `scan()` statement. The second example shows the effect of using `-nolast_scan`.

```
#example made with -last_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
  output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
  input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture
vector(tp1) := [ -Z -0 -0 -1 0 1 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
  output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
  input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
end
```

```
#example made with -nolast_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
  output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
  input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture_CLK }
vector(tp1) := [ -Z -0 -0 -1 0 1 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
  output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
  input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
{ nocapture }
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
end
```

Mapping Bidirectional Ports Within Vector Statements in WGL

You've seen an example earlier of how TestMAX ATPG supports creating WGL patterns with bidirectional port data represented as either a single column of two characters (grouped) or as two columns of single characters (non-grouped or split). In addition to this choice in grouping there is also the ability to change or map the characters used. Not every vendor agrees on what the WGL character representation should be for bidirectional

port data so TestMAX ATPG has been designed to provide flexibility by use of the `set_wgl -bidi_map` option.

The syntax for this option is: `set_wgl -bidi_map <from> <to>`

There are 9 mappings that can be adjusted: 3 for which the bidirectional port is an input, 4 for which the bidirectional port is an output, and 2 for when the bidirectional port is a scan input or scan output. This argument can be repeated on the same command line or across multiple commands to specify more than one mapping. If the same from designator is repeated then the later one will replace the earlier ones.

The from designator is a two-character string that represents the TestMAX ATPG internal data. The to designator is a two-character string that specifies the characters which will appear in the WGL pattern output in place of this internal representation.

Definition of TestMAX ATPG Internal Representation = "from"

```
from
====
0x : force 0, no measure
1x : force 1, no measure
xx : force unknown, no measure

z0 : no force, measure 0
z1 : no force, measure 1
zx : no force, no measure
zz : no force, measure Z

-x : bidi is in scan input mode
z- : bidi is in scan output mode
```

The preceding table defines all the legal combinations available for the `from` portion of the mapping option. Any other combination is illegal. The `to` designator is also made up of characters `0/1/x/z/-` but the mapping is checked to ensure that you are not destroying the intent of the data or masking measures that would affect the test coverage reported. As an example of a mapping the following table represents a commonly requested map in which one of the bidirectional characters is always a dash:

A common mapping

```
from : to
==== : ==
0x : 0- # force 0, no measure
1x : 1- # force 1, no measure
xx : x- # force unknown, no measure
z0 : -0 # force Z, measure 0
z1 : -1 # force Z, measure 1
zx : -x # force Z, no measure
zz : -z # force Z, measure Z
-x : -- # bidi is a scan input
```

```
z- : -- # bidi is a scan output
```

With the exception of the {zz, -z} mapping above, this table represents the default mapping.

The `set_wgl` command which would implement the previous table is:

```
BUILD> set_wgl -bidi_map { 0x 0-      1x 1-      xx x-      z0 -0  \
                          z1 -1      zx -x      zz -z      x  --      z-
  -- }
```

Note in the previous example that you can specify the `-bidi_map` option only one time, and the parameters must be in a list structure. Alternatively, you can repeat the entire command line for each entry, as shown in the following example:

```
set_wgl -bidi_map {0x 0-}
set_wgl -bidi_map {1x 1-}
set_wgl -bidi_map {xx x- }
set_wgl -bidi_map {z0 -0 }
set_wgl -bidi_map {z1 -1}
set_wgl -bidi_map {zx -x}
set_wgl -bidi_map {zz -z}
set_wgl -bidi_map {x  --}
set_wgl -bidi_map {z- --}
```

Note: Not all mappings are allowed. For example, you cannot map the dash for scan input or scan output to any other character. Also, you can map "zz" to "-z", but you cannot map "zz" to "z-". because of the loss of measure and to unambiguously read back in the WGL which is written out. The "zz"->"z-" mapping still indicates a measure must be performed but a "zz"->"z-" mapping could be confused with a "zx"->"z-" mapping which generally is interpreted to mean there is no force and no measure.

Note: The ability to use some bidi mappings is affected by whether the tester can measure Z values or not. If the tester can measure Z values then the default setting of `set_buses -external_z Z` should be used and the WGL patterns can contain both ZZ and ZX data (no force, measure Z and no force, no measure). If the tester cannot measure Z values or you want to generate patterns for which no Z-measure is needed you would set the `set_buses -external_z X` option before generating patterns. This would result in WGL patterns with "ZX" data for bidirectional pins but no "ZZ". If "ZZ" does not appear in the WGL you can define a bidi map of "ZX"->"Z-" or "ZX"->"Z-" which you could not do if the Z measure were enabled and "ZZ" were possibly present.

Note: Most vendors do not support a simultaneous force and measure on the same port in the same cycle. With that in mind you should not use the `-measure_forced_bidis` option of the `write_patterns` command as this allows for a simultaneous force and measure whenever possible.

To report the current bidirectional map settings use the `report_settings wgl` command. The output is similar to the following example and the mapping will appear as a series of (from,to) settings.

```
wgl = macro_usage=off, nopad=on, scan_map=dash
      group_bidis=off, inversion_reference=master, tester_ready=on
      bidi_map=(Z0,-0) (Z1,-1) (0X,0-) (1X,1-) (XX,X-) (ZX,-X) (ZZ,-Z) (Z-,--)
```

As an example of how the `vector()` statement data changes for bidirectional ports the first following example shows some pattern data with four bidirectional pins (grouped as single column of two characters each) where the mapping is identical to the TestMAX ATPG internal representation. The second example uses a common mapping in which the bidirectional character pair always has one character represented as a dash.

An example where the mapping matches TestMAX ATPG internal representation.

```
{ pattern 1 }
{ load_unload }
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0X 1X XX ZX 1 1 - X 0 1 X X X - X ],
output [c1:c1U0], input [c1:c1L1]];
{ capture_CLK }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ Z0 Z1 ZX ZZ 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ ZX ZX ZX ZX 1 1 0 1 0 1 X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ ZX ZX ZX 0X 1 1 - X 0 1 X X X - X ],
output [c1:c1U1], input [c1:c1L2]];
{ capture_CLK }
vector(tp1) := [ 0X 0X 0X 0X 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0X 1X Z0 ZZ 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0X 1X ZX ZX 1 1 1 1 1 0 X X X X X ];
```

The same patterns after defining a mapping of:
(0x,0-) (1x,1-), (xx,x-), (z0,-0), (z1,-1), (zx,-x), (zz,-z)

```
{ pattern 1 }
{ load_unload }
vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 0 X X X X X ];
```

```

vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0- 1- X- Z- 1 1 - X 0 1 X X X - X ],
output [c1:c1U0], input [c1:c1L1]];
{ capture_CLK }
vector(tp1) := [ -X -X -X -X 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ -0 -1 -X -Z 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ -X -X -X -X 1 1 0 1 0 1 X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -X -X -X 0- 1 1 - X 0 1 X X X - X ],
output [c1:c1U1], input [c1:c1L2]];
{ capture_CLK }
vector(tp1) := [ 0- 0- 0- 0- 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0- 1- -0 -Z 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0- 1- -X -X 1 1 1 1 1 0 X X X X X ];

```

Mapping Bidirectional Ports Within Scan Statements in WGL

The `vector()` statements in WGL correspond to the application of tester cycles. The `scan()` statements correspond to the serial loading and unloading of scan chains. The various vendor rules for character mapping of the `vector()` statements cannot be the same as for the `scan()` statement and so TestMAX ATPG supports the `set_wgl -scan_map` option to allow somewhat independent control of characters in the `scan()` statement. The available choices for scan mapping are: dash, bidi, keep, and none. The default is dash.

The following examples show some of the variations of `-scan_map`. The patterns are for a design with three scan chains and the first bidirectional port is a scan input and the second bidirectional port is a scan output.

For a setting of `dash`, every scan input and output position in the `scan()` statement contains a dash, and all bidirectional ports acting as a scan input or output contain a double dash.

```

# set_wgl -scan_map dash

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],

```

For a setting of `bidi`, every scan input and output position in the `scan()` statement contains a dash, and any bidirectional port acting as a scan input or output follows the mapping defined by the `-bidi_map` options. For the following example, assume a BIDI mapping of `(-x,--)` for scan inputs, and `(z-,z-)` for scan outputs.

```

# set_wgl -scan_map bidi -bidi_map {-x --} -bidi_map {z- z- }

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];

```

```
scan(tp1) := [ -- Z- X- X- 1 1 - - 0 1 X X - - X ],
```

For a setting of `keep`, every scan input and output position in the `scan()` statement keeps the same characters as from the previous `vector()` statement in the `load_unload` procedure, including any scan inputs or outputs on bidirectional ports. It is important that the `load_unload` procedure have at least one `vector()` statement before the Shift procedure in order for a selection of `keep` to work properly.

```
# set_wgl -scan_map keep

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0- -X X- X- 1 1 X X 0 1 X X X X X ],
```

For a setting of `none`, every scan input and output position in the `scan()` statement contains a dash, and any bidirectional port acting as a scan input or output uses the TestMAX ATPG internal representation of "-X" for input and "Z-" for output.

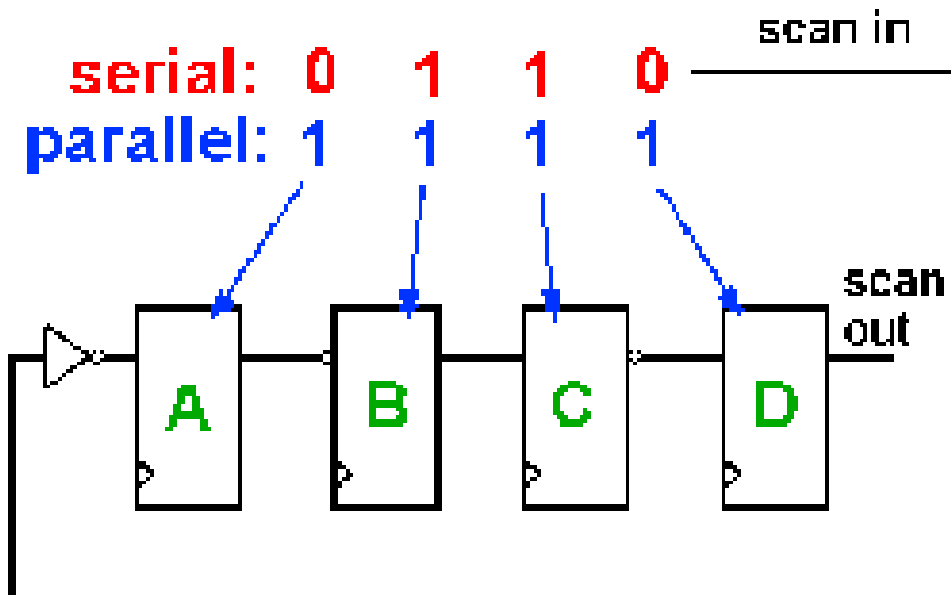
```
# set_wgl -scan_map none

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -X Z- X- X- 1 1 - - 0 1 X X - - X ],
```

Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL

The scan load data in the WGL patterns can be represented in two different ways, depending upon the reference point required by your WGL pattern translation tool. The `set_wgl -tester_ready` setting selects a data format that is ready to serially shift into the device without further processing for scan cell inversions. The `-set_wgl -notester_ready` option selects a data format that is ready to parallel load directly into the scan cells without further processing for inversions.

In the following figure, if you desire to have all devices A,B,C, and D loaded with 1's after a scan load, and your WGL translation application expects the data in parallel (`-notester_ready`) format, then the WGL scan data must be written as all 1's. However, if your WGL translation application expects the data in serial format (`-tester_ready`), then the WGL scan data must be adjusted for internal inversions that it passes through before being shifted into place. As you can see, the data is not the same: "1111" vs. "0110". So it is very important to know which data format your WGL translation application is expecting. The parallel format is the more popular, so if you do not know you should try the `-notester_ready` option first.



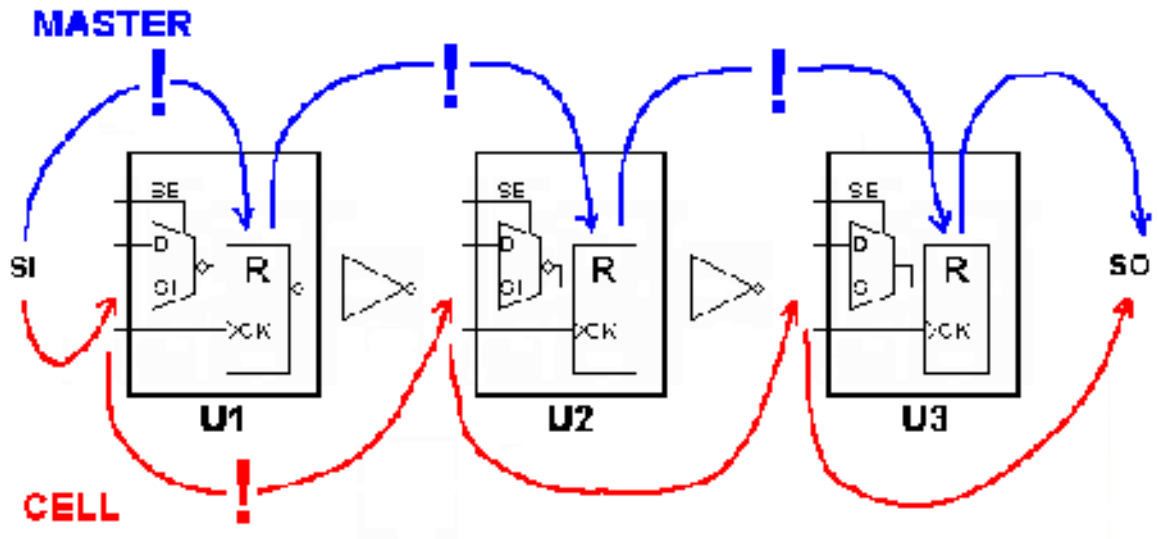
Note that both the serial and parallel load formats are sensitive to the referencing scheme for determining inversion if the final WGL translator is doing a parallel-form to serial form translation or a serial-form to parallel-form translation.

One additional variant of WGL output is needed if the WGL is to be interpreted for a parallel simulation and the end-of-cycle protocol is used. This end of cycle protocol results in a scan output pre-measure before beginning the "scan()" statement for the balance of the scan load/unload. The expected scan output vector needs to be shifted by the single bit of the pre-measure. To accomplish this, use the `-pre_measured` option instead of the `-notester_ready` option.

Selecting Scan Chain Inversion Reference in WGL

The `scanchain` block of the WGL pattern file defines each scan chain in physical order from input port to output port. When an inversion exists between positions in the scan order, and exclamation mark "!" is inserted to indicate an inversion of the data has occurred between the two positions. This inversion information is crucial for the correct translation of the scan chain load and unload data by the WGL-to-simulator or WGL-to-tester tools supported by your vendor.

More than one interpretation of the reference scheme for calculating inversions exists and so TestMAX ATPG offers options of `master`, `cell`, and `omit` for the `set_wgl_inversion_reference` command.



The previous diagram can provide some insight into the two different referencing schemes for inversion markers. When TestMAX ATPG calculates the inversion markers for a setting of `master` the reference point begins at the scan input pin, and then looks at whether the data is inverted from that point to the actual sequential simulation primitive functioning as the "master cell" where the value is stored. This is often a Verilog UDP level underneath the vendor's library cell. For a library cell with only one sequential element there is only one answer but for a library cell with two or more sequential elements, the answer might be ambiguous. As shown in the diagram, for an inversion reference of `master` there are inversions between the scan input and U1, between U1 and U2, and between U2 and U3. The corresponding WGL scanchain definition is shown in the following example.

```
# set_wgl -inversion_reference master

scanchain
  cl ["si", !, "U1/R", !, "U2/R", !, "U3/R", "so" ];
end
```

When TestMAX ATPG calculates the inversion markers for a setting of `cell` it begins at the scan in pin and then determines whether an inversion of the data occurs relative to the scan input pin of each library cell. This reference is used by some WGL translators in forming the FORCE/RELEASE statements needed for a parallel Verilog simulation. The location of the inversion markers is unambiguous and not affected by which cell is classified as the "master" cell by TestMAX ATPG during DRC. Using an inversion reference of `cell` and the preceding diagram, there is an inversion only between the scan input of cell U1 and the scan input of U2. The corresponding WGL scanchain definition is shown in the following example:

```
# set_wgl -inversion_reference cell
```

```
scanchain
  c1 ["si", "U1/R", !, "U2/R", "U3/R", "so" ];
end
```

Sometimes, no matter which inversion reference you select the external WGL translator seems to come up with patterns that mismatch in simulation. If the simulation environment serially processes scan load information, then there is one more inversion reference that might be of use and that is the `omit` option. This option leaves out all inversion markers. By combing both the `-inversion_reference omit` and `-tester_ready` options, TestMAX ATPG produces scan load/unload data that is preprocessed for inversions and is ready to shift into the device unchanged, and omits the inversion markers so the external WGL translator is mydesigned into thinking that no data adjustments for inversion are needed. The corresponding WGL scanchain data when `omit` is used is shown in the following example:

```
# set_wgl -inversion_reference omit

scanchain
  c1 ["si", "U1/R", "U2/R", "U3/R", "so" ];
end
```

Effect of CELLDEFINE in WGL

The previous examples showed the effect of different choices of inversion reference on the placement of the inversion markers "!" in the scanchain definition block. Another item which affects the scanchain block is the presence or absence of the ``celldefine` compiler directive in the definition of the library model. Consider the following two examples:

```
# Verilog library module without celldefine
module SDFF (Q, CLK, SE, D, SI);
  input CLK, SE, D, SI;
  output Q;
  uMUX M (di, SE, D, SI);
  uDFFQ R (Q , CLK, di);
endmodule

# WGL scanchain shows instance "R"
scanchain
  c1 ["si", "U1/R", !, "U2/R", "U3/R", "so" ];
end

# Verilog library module with celldefine
`celldefine
module SDFF (Q, CLK, SE, D, SI);
  input CLK, SE, D, SI;
  output Q;
  uMUX M (di, SE, D, SI);
```

```

    uDFFQ R (Q, CLK, di);
endmodule
`endcelldefine

# scanchain instances have no "R"
scanchain
  c1 ["si", "U1", !, "U2", "U3", "so" ];
end

```

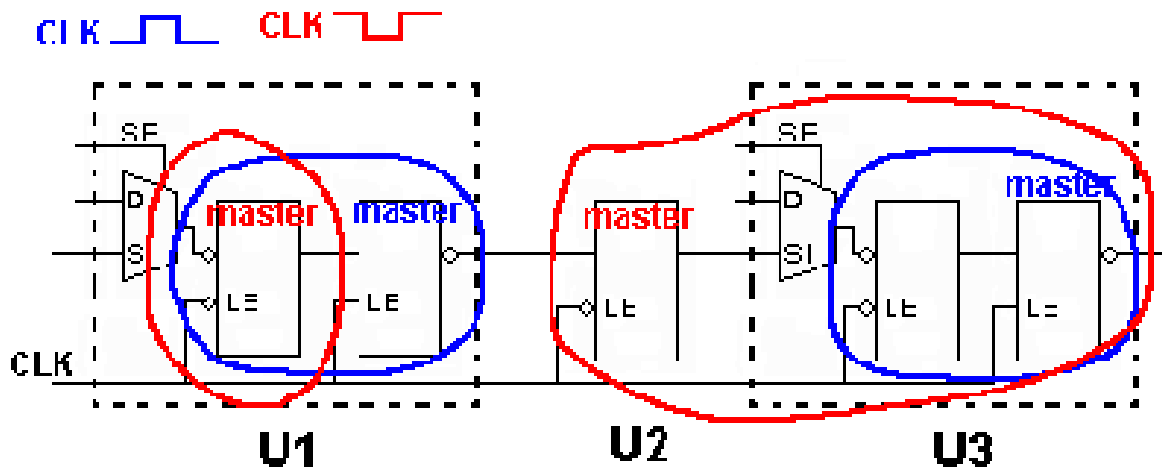
In the first example the Verilog module definition was not defined inside a ``celldefine/`endcelldefine` pair. The resulting WGL scanchain definition shows instance pathnames that include the `R` of the `uDFFQ` device.

In the second example the Verilog module definition was within a ``celldefine/`endcelldefine` pair. The resulting WGL scanchain definition does not include the instance references beneath the `SDF` module.

Note: Reading a netlist with the `-library` option has the same effect as enclosing the module with ``celldefine/`endcelldefine` pair and is yet another way to affect the WGL output.

Ambiguity of the Master Cell in WGL

The diagram below provides one simple example of the potential for ambiguity when using an inversion reference of master. In this example some DFF functions are created with a library cell using two latches. TestMAX ATPG defines the "master" based on which sequential device in a scan chain shifts first due to the leading edge of the defined shift clocks. So with the CLK port defined as active high, the "master" becomes the second LATCH in U1 and U3, with U2 acting as a lockup latch. If the polarity of CLK is reversed, then the first latch in U1 is classified as the master and the lockup latch is classified as the master for cell U3! Both polarities of CLK generates ATPG patterns but most likely only one resulting WGL inversion set is correct.



Running Multicore ATPG

Multicore ATPG is used to parallelize and improve ATPG runtime by leveraging the resources provided by multicore machines. Multicore ATPG launches multiple slaves to parallelize ATPG work on a single host. You can specify the number of processes to launch, based on the number of CPUs and the available memory on the machine.

The following sections describe multicore ATPG:

- [Comparing Multicore ATPG and Distributed ATPG](#)
- [Invoking Multicore ATPG](#)
- [Typical Multicore ATPG Run](#)
- [Multicore Interrupt Handling](#)
- [Understanding the Processes Summary Report](#)
- [Multicore Limitations](#)

For more information, see [Running Distributed ATPG](#).

Comparing Multicore ATPG and Distributed ATPG

Multicore ATPG is different than distributed ATPG, which is described in [Running Distributed ATPG](#). Distributed ATPG launches multiple slave processes on a computer farm or on several standalone hosts. The slave processes read an image file and execute a fixed command script. ATPG distributed technology does not differentiate between multiple CPUs and separate workstations.

When compared to ATPG distributed technology, multicore ATPG has several advantages:

- It is easier to use. You simply need to specify the number of slaves to use. There is no need to set up the environment for slaves, and no need to debug network or computer farm issues. Also, it is not necessary for the master to write a binary image file or for the slaves to read it.
- It is more efficient in using compute resources. Multicore ATPG shares netlist information among slaves and the master. This means the overall memory usage is much lower than the total memory usage of distributed ATPG.
- It reduces communication overhead by running all involved processes on one machine. It also improves the efficiency of parallelism by sharing more information among processes. This often results in better QoR compared to distributed ATPG.

Although multicore ATPG offers better memory utilization (<50 percent increase per core) compared to distributed ATPG (~100 percent increase per slave), the entire memory must reside on a single machine.

Multicore ATPG and distributed ATPG provide similar efficiency in reducing ATPG runtime. The runtime improvement from multicore processing is limited by the number of cores and CPUs on a single host. With distributed ATPG, however, runtime continues to improve as more hosts are added across the network.

Invoking Multicore ATPG

Multicore ATPG is activated using the following `set_atpg` command:

```
set_atpg -num_processes < number | max >
```

The *number* specification refers to the number of slave processes that are used in ATPG. If `max` is specified, then TestMAX ATPG computes the maximum number of processes available in the host, based on number of CPUs. If TestMAX ATPG detects that the host has only one CPU, then single-process ATPG is performed instead of multicore ATPG with only one slave.

To turn off multicore ATPG, specify `set_atpg -num_processes 0`.

Do not specify more processes than the number of CPUs available on the host. You should also consider whether there is other CPU-intensive processes running simultaneously on the host when running the number of processes. If too many processes are specified, performance will degrade and might be worse than single-process ATPG. On some platforms, TestMAX ATPG cannot compute the number of CPUs available and will issue an error if `max` is specified.

Typical Multicore ATPG Run

```
# Perform DRC and enter TEST mode
run_drc top_level.spf
# select the fault model and create the
fault list
set_faults -model transition
# Other ATPG settings here
...
# Use two cores during run_atpg
set_atpg -num_processes 2
run_atpg -auto
# Continue with other commands after run_atpg
...
// Multicore Usage - Farm Multi-Host (LSF)
bsub -R "span[hosts=1]" -n 4 \
tmax_multicore_batch.csh
// Ensure all slots are reserved on a single host
// 4 slots are allocated for ATPG run
read_netlist Libs/*.v -delete -library -noabort
run_build_model top_level
run_drc top_level.spf
set_faults -model stuck
...
add_faults -all
set_atpg -num_processes 4
// Multicore Usage - Farm Multi-Host (GRID)
qsub -l cputype=amd64,\
mem_free=16G,mem_avail=16G,\
cpus_used=4,model=AMD2800 \
tmax_multicore_batch.csh
// 4 slots are allocated for ATPG run
read_netlist Libs/*.v -delete -library -noabort
run_build_model top_level
run_drc top_level.spf
set_faults -model stuck...
add_faults -all
set_atpg -num_processes 4
```

Multicore Interrupt Handling

To interrupt the multicore ATPG process, use “Control-c” in the same manner as halting a single process. If a slave crashes or is killed, the master and remaining slaves will continue to run. This behavior is consistent with the default behavior of distributed ATPG.

If the master crashes or is killed, the slaves will also halt. In this case, there are no ongoing processes, dangling files, or memory leakage.

You can also interrupt the multicore ATPG process from the TestMAX ATPG GUI by clicking the Stop button. At this point, the master process sends an abort signal to the

slave processes and waits for the slaves to finish any ongoing interval tasks. If this takes an extended period of time, you can click the Stop button twice. This action causes the master process to send a kill signal to the slaves, and the prompt immediately returns. Note that clicking the Stop button twice terminates all slave processes without saving any data gathered since the last communication with the master. For more information on the Stop button, see [Command Entry](#).

Understanding the Processes Summary Report

Memory consumption needs to be measured to tune the global data structure to improve the scalability of multicore architecture. Legacy memory reports are not sufficient because they do not deal with issues related to “copy-on-modification.” To facilitate collecting performance data, a summary report of multicore ATPG is printed at the end of ATPG when the `-level` expert option is specified with the `set_messages` command. The summary report appears as shown in the following example.

Processes Summary Report

Processes Summary Report

```

-----
Process          Patterns          Time (s)          Memory (MB)
-----
ID      pid      Internal      CPU      Wall      Shared      Private
Total      Pattern
-----
0      7611      1231          0.53      35.00      67.78      30.54
98.32      5.27
1      7612      626          35.68      35.00      64.87      22.31
87.18      0.00
2      7613      605          35.50      35.00      64.71      22.47
87.18      0.00
Total          1231          71.71      35.00      67.78      75.32
143.10      5.27
-----

```

The report in the preceding example contains one row for each process. The first process with an ID of “0” is the master process. The child processes have IDs of 1, 2, 3, and so forth. The last row is the sum for each measurement across all processes.

The “pid” is the process ID of that process. The “Patterns” are the total number of patterns stored by the master or the number of patterns generated by the slave in this particular ATPG session. The “Time(s)” includes CPU time and wall time.

The “Memory” measurements are obtained by parsing the system-generated file `/proc/pid/smmaps`. The file contains memory mapping information created by the OS while the process still exists. The `/proc/pid/` directory cannot be found after the process terminates.

The tool parses this file at the proper time to gather memory information for the reporting at the end of parallel ATPG.

The “Memory” measurement includes “Shared”, “Private”, “Total” and “Pattern”. “Shared” means all processes share the same copy of the memory. “Private” means the process stores local changes in the memory. The “Total” is the sum of “Shared” and “Private”. The “Pattern” refers to memories allocated for storing patterns. The total memory consumption of the entire system is the “Total” item in the row “Total,” which is the sum of total shared memory (the maximum of shared memories for each process) and the total private memory (the sum of all private memory for all processes). Although the memory for patterns is listed separately, it is part of the master private memory.

The memory section of the summary report is only available on Linux and AMD64 platforms. No other platform gives “shared” or “private” memory information in a copy-on-write context. On other formats, the memory reports gives all “0s” for items other than the pattern memory.

Multicore Limitations

The following ATPG features are not supported by multicore ATPG:

- Streaming Pattern Validation
- Distributed ATPG
- The `-per_cycle` option of the `report_power` command is not recognized.

Running Logic Simulation

Using TestMAX ATPG, 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 the following tasks:

- 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.

The following sections describe how to run logic simulation:

- [Comparing Simulated and Expected Values](#)
- [Patterns in the Simulation Buffer](#)
- [Sequential Simulation Data](#)
- [Single-Point Failure Simulation](#)
- [GSV Display of a Single-Point Failure](#)

In addition to standard logic simulation, you can also improve simulation runtime by launching multiple slaves to parallelize fault and logic simulation to work on a single host. This process is described in [Running Multicore Simulation](#).

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 of the Run Simulation dialog box, or the `-nocompare` option of the `run_simulation` command. For more information, see [“Performing Good Machine Simulation”](#).

The following example shows a transcript of a simulation run that had no comparison errors; 139 patterns were simulated with zero failures.

Example 1 Simulation With No Comparison Errors

```
TEST-T> 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
```

The following example 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 2 Simulation With Comparison Errors

```
TEST-T> 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, TestMAX ATPG 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 the following commands:

```
TEST-T> set_pindata pattern NW
TEST-T> refresh schematic
```

For an example, see [“Displaying Pattern Data”](#).

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.”

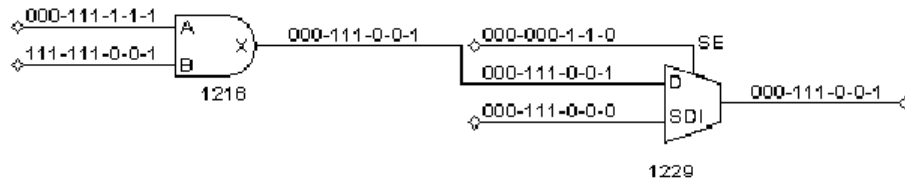
For example, with gates drawn in the schematic window, execution of the following commands generates the display shown in the following example.

```
TEST-T> set_simulation -data 85 89
```

```
TEST-T> 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 95 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 running the error site and stuck-at value, or by using a command such as the following:

```
TEST-T> run_simulation -max_fails 0 amd2910/ incr/U42/A 1
```

The following example shows the result of executing this command. TestMAX ATPG 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 3 Signature of a Simulated Failure

```
TEST-T> run_simulation -max_fails 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

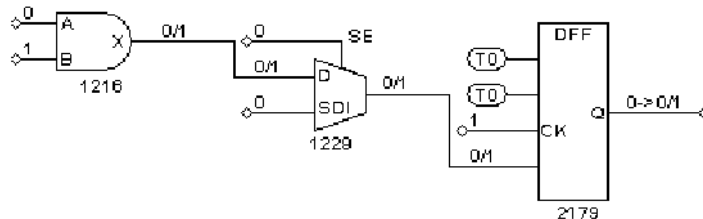
You can display simulation results for a single-point failure in the GSV. To do so, click the SETUP button on the GSV toolbar to display the Setup dialog box.

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 the following figure.

Figure 96 Fault Simulation Results Displayed Graphically



In the preceding figure, 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.

Data Volume and Test Application Time Reduction Calculations

The equations for calculating data volume and test application time reduction for running TestMAX ATPG on compressed scan designs are as follows:

$$\begin{aligned} \text{Test Data Volume Reduction} &= \\ &(\text{Scan Test Data Volume}) / \\ &(\text{Scan Compression Test Data Volume}) \\ \text{Test Application Time Reduction} &= \end{aligned}$$

```
(scan mode test application time)/  
(ScanCompression_mode test application time)
```

These calculations are explained in the following sections:

- [Test Data Volume Calculations](#)
- [Test Application Time Calculations](#)

Test Data Volume Calculations

The following information is stored on the tester in each test cycle:

- Forced value on input (signal or clock waveform)
- Value expected on output (strobed output)
- Whether output value should be strobed or not (output mask)

On every output, there are two bits of information per cycle. In the following two equations, this accounts for the factor of three in the scan-test-data-volume equation and the factor of two in the scan-compression-test-data-volume equation. The compression calculation is written differently because it accounts for the number of inputs and outputs to the compression logic.

You can use the following formulas to expand the test data volume reduction equation:

```
Scan Test Data Volume =  
3*(length of the longest Scan mode scan chain)*  
(number of scan chains in Scan mode)*  
(number of Scan mode patterns)  
Scan Compression Test Data Volume =  
(length of longest ScanCompression_mode scan chain)*  
(number of scan_in + 2*(number of scan_out))*  
(number of patterns)
```

The test data volume might not match the memory used by the tester because each ATE uses the test data volume differently. However, the tester can optimize the memory content. It can allocate memory differently, depending on the brand or version of the tester and the channels and cycles used. In such cases, the factors 2 and 3 in the scan-test-data-volume and scan-compression-test-data volume formulas, respectively, might not match the data in the tester memory.

The following ratio indicates the test data volume reduction that can be achieved:

```
Test Data Volume Reduction =  
(Scan Test Data Volume)/  
(Scan Compression Test Data Volume)
```

The test data volume reduction value calculated with this formula is just an estimate of the improvements you can get by using compression.

Test Application Time Calculations

The test application time reduction is an estimate for the improvements you can achieve by using compression. You can determine this reduction by taking the ratio of scan versus scan-compression test-application time.

The test-application-time-reduction equation can be expanded by using the following formulas:

```
Scan Test Application Time =  
(longest chain in Scan mode) *  
(number of patterns in Scan mode)  
Scan Compression Test Application Time =  
(longest scan chain in ScanCompression_mode) *  
(number of patterns in ScanCompression_mode)
```

The test application time reduction that can be achieved as follows:

```
Test Application Time Reduction =  
(scan mode test application time) /  
(scanCompression_mode test application time)
```

If you expand this equation, using the previous test application time equations for scan and scan compression, you get the following:

```
Test Application Time Reduction =  
((longest chain in Scan mode) *  
(number of patterns in Scan mode)) /  
((longest scan chain in ScanCompression_mode) *  
(number of patterns in ScanCompression_mode))
```

See Also

- [Distributed ATPG Limitations](#)

Pattern Porting

You can use the `stilgen` utility to port patterns for code at the top level of a design. To use pattern porting, you must make several adjustments when performing core-level DFTMAX insertion (as described in this topic).

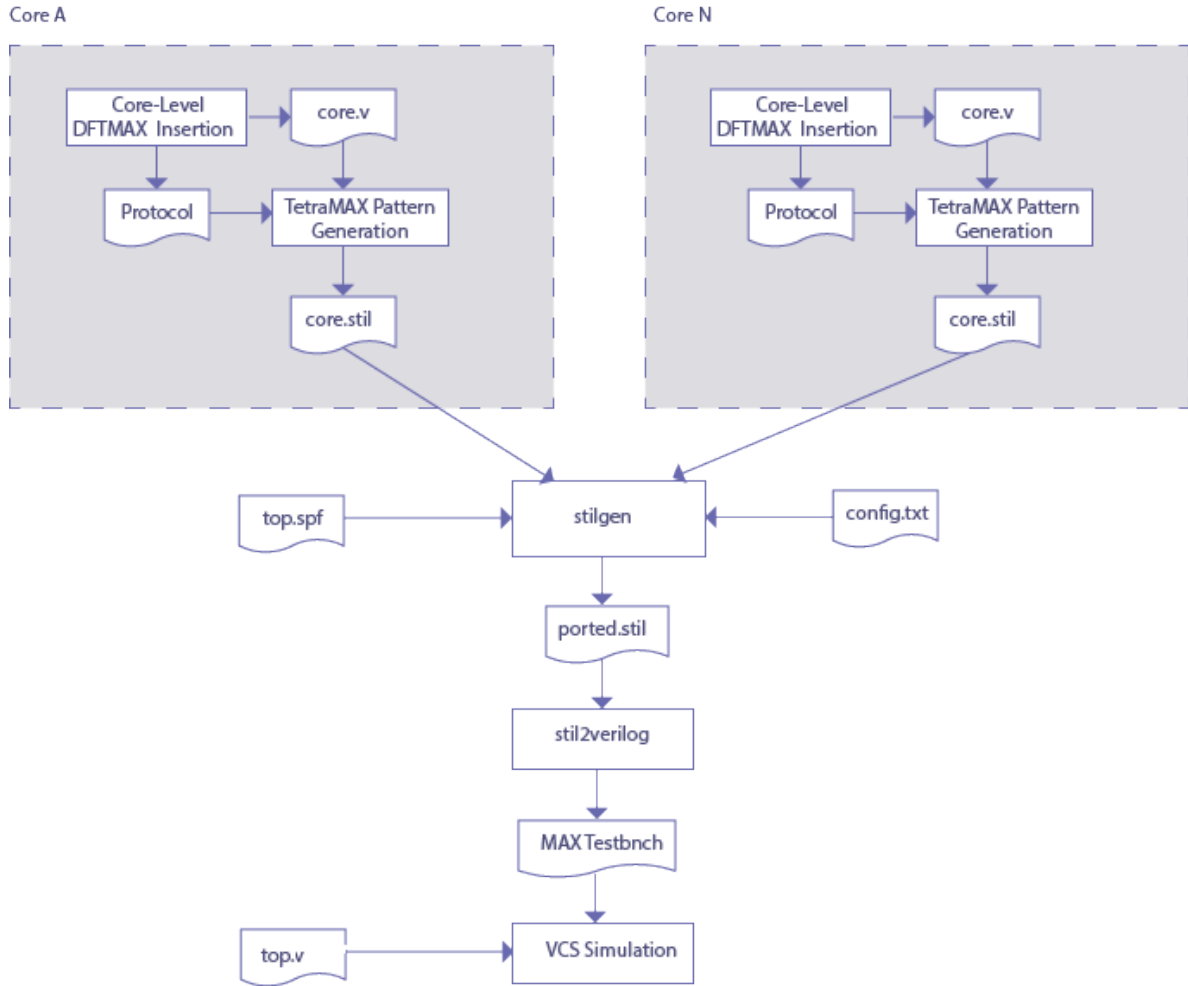
For a description of the syntax for the `stilgen` utility and its configuration files, see [stilgen Utility and Configuration Files](#).

The following sections describes the flow and requirements for porting patterns:

- [Pattern Porting Flow](#)
- [Core-Level DFTMAX Insertion](#)
- [Core-Level TestMAX ATPG Generation](#)
- [Top-Level Requirements](#)
- [Pattern Generation Requirements at Core Level](#)
- [Top-Level Pattern Simulation](#)

Pattern Porting Flow

The following diagram describes the flow used for porting patterns, including top-level pattern validation.



Core-Level DFTMAX Insertion

Note the following:

- Pattern porting of cores with DFTMAX and DFTMAX Ultra is supported.
- OCC controllers and scan-data pipelining are supported at the core-level.

Core-Level TestMAX ATPG Generation

ScanCompression mode and Internal_scan patterns can be ported to the top level during TestMAX ATPG generation. Before generating the patterns, make sure that all inputs

(except for clocks, resets, test-modes, scan-enable, and OCC signals) are constrained to X and outputs are masked using the following commands:

```
add_pi_constraints X {list of all inputs}
add_po_masks -all
```

To write patterns in STIL format for pattern merging, make sure you use the `write_patterns` command options shown in the following example:

```
write_patterns my_patterns_serial.stil -format stil -replace
```

If the top-level scan-enable `wrp_shift` signal is shared by the core-level scan-enable `wrp_shift` pin, then the scan-enable `wrp_shift` pin should be constrained to its off-state in all core-level TestMAX ATPG scripts for pattern generation.

Core-level patterns produced by TestMAX ATPG should pass VCS simulation.

Top-Level Requirements

Make sure the following top-level requirements are met:

- All the core-level test ports, clocks, resets, PLL/OCC signals, and test setup signals must be fully controllable from the top-level ports and have correspondence between core and top-level test ports.
- All the core-level scan clocks must be dedicated at the top level and cannot be shared. Only the OCC `ate_clock` can be shared.
- OCC insertion and pipeline insertion are not supported at the top level.

Pattern Generation Requirements at Core Level

- For DFTMAX and DFTMAX Ultra-wrapped cores, the longest length of a compressed scan chain (compressor length) should be uniform across all cores. For DFTMAX Ultra-wrapped cores, if the compressor length is not uniform across cores, use the following commands for the cores with a shorter compressor length:

```
set_patterns -external core.stil update_streaming_patterns -max_shifts
$MAX_LENGTH write_patterns core_up.stil -format stil -external
```

Make sure that `$MAX_LENGTH` is the maximum compressor length.

Top-Level Pattern Simulation

At the top level, only MAXTestbench patterns can be used to validate both `ScanCompression_mode` and `Internal_scan` mode patterns. The ported patterns can also be in read in TestMAX ATPG and validated using the `run_simulation` command.

14

Fault Lists and Faults

TestMAX ATPG puts faults into various fault classes, each of which are organized into categories.

The following topics describe the fault classes and explain how TestMAX ATPG calculates test coverage statistics:

- [Working with Fault Lists](#)
- [Fault Categories and Classes](#)
- [Fault Summary Reports](#)
- [Using Clock Domain-Based Faults](#)

Working with Fault Lists

TestMAX ATPG maintains a list of potential faults for a design, along with the categorization of each fault. A fault list is contained in an ASCII file that can be read and written using the `read_faults` and `write_faults` commands.

The following topics describe how to work with fault lists:

- [Using Fault List Files](#)
- [Collapsed and Uncollapsed Fault Lists](#)
- [Random Fault Sampling](#)
- [Fault Dictionary](#)

As shown in the following example, a fault list contains one fault entry per line. 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 "--".

Example 1: Typical Fault List 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
```

Note the following:

- TestMAX ATPG ignores blank lines and lines that start with a double slash and a space (//).
- 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.

See Also

- [Persistent Fault Model Operations](#)

Using Fault List Files

You can use fault list files to manipulate your fault list in the following ways:

- Add faults from a file, while ignoring any fault classes specified
- Add faults from a file, while 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:

- *New Faults*

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, TestMAX ATPG 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*

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. TestMAX ATPG 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 1: Collapsed and Uncollapsed Fault Summary Reports

```
TEST-T> set_faults -report collapsed
```

```
TEST-T> report_faults -summary
```

```
Collapsed Fault Summary Report
```

```
-----  
  
fault class                code    #faults  
  
-----
```

Chapter 14: Fault Lists and Faults
Working with Fault Lists

```

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-T> set_faults -report uncollapsed
TEST-T> 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, TestMAX ATPG 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 is 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-T> add_faults /spec_asic/block_A
TEST-T> add_faults /spec_asic/block_B
TEST-T> remove_faults -retain_sample 50
TEST-T> add_faults /spec_asic/block_C
TEST-T> 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.

TestMAX 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 [Diagnosing Manufacturing Test Failures](#).

Fault Categories and Classes

Faults are assigned to classes corresponding to their current fault detection or detectability status. A two-character code is used to specify a fault class. Fault classes are hierarchically defined: low-level fault classes can be grouped together to form a higher level fault classes. Faults are only assigned the low fault classes but the high level fault classes are used for reporting. The fault class hierarchy for all fault classes is as follows:

Fault Class Hierarchy

DT - Detected

DR - Detected Robustly

DS - Detected by Simulation

DI - Detected by Implication

D2 - Detected clock fault with loadable nonscan cell faulty value of 0 and 1

TP - Transition partially detected

PT - Possibly Detected

AP - ATPG Untestable Possibly Detected

NP - Not analyzed, Possibly Detected

P0 - Detected clock fault and loadable nonscan cell faulty value is 0

P1 - Detected clock fault and loadable nonscan cell faulty value is 1

UD - Undetectable

UU - Undetectable Unused

UO - Undetectable Unobservable

UT - Undetectable Tied

UB - Undetectable Blocked

UR - Undetectable Redundant

AU - ATPG Untestable

AN - ATPG Untestable Not-Detected

AX - ATPG Untestable Timing Exceptions

AE - ATPG Untestable

ND - Not Detected

NC - Not Controlled

NO - Not Observed

DT (Detected) = DR + DS + DI + D2 + TP

The "detected" fault class is comprised of faults which have been identified as "hard" detected. A hard detection guarantees a detectable difference between the expected value and the fault effect value. The detection identification can be performed by simulation or implication analysis.

- DR (Detected Robustly)

DR faults are hard detected by the fault simulator using weak non-robust (WNR), robust (ROB), or hazard-free robust (HFR) testing criteria to mark path delay faults. During ATPG, at least one pattern that caused the fault to be placed in this class is retained. This classification applies only to Path Delay ATPG.

- DS (Detected by Simulation)

DS faults are hard detected by explicit simulation of patterns. During ATPG, at least one pattern that caused the fault to be placed in this class is retained.

- DI (Detected by Implication)

DI faults are detected by an implication analysis. Faults are immediately placed into this fault class when they are added to the fault list. These faults include the following:

- Faults on pins in the scan chain path are detected due to the application of a scan chain functional test
- Faults on ungated circuitry that connect to the shift clock line of scan cells

- Control circuitry of clock-gating cells that connect to the shift clock line of scan cells
- Faults on ungated circuitry that connect to the set/reset lines of scan cells and cause the set/reset to be active

Note: A scan chain path that is multiply sensitized receives no credit.

Note: Nonscan DLAT/DFF Enable/Clock pin stuck-at faults are marked as DI at the end of `run_atpg` if they have not been marked already as DS. This is done only if data pins are marked as DS for both `sa0` & `sa1`. This is because if the data input pins `sa0` & `sa1` faults are both detected by simulation, ATPG can assume that the enable/clock pins must function correctly, and thus `sa` faults that force clock off should be detected by implication if not detected by simulation. For latch, `s-a-(off value)` should be DI; for DFF, both `sa0` & `sa1` on clock should be DI.

- D2

A fault is classified as D2 if a clock fault is detected and the loadable nonscan cell faulty value is set to both 0 and 1. Note that the loadable nonscan cells feature must be active.

- TP (Transition Partially-Detected)

TP faults are detected with a slack that exceeds the minimum slack by more than value specified by the `-max_delta_per_fault` option of the `set_delay` command. A TP fault can continue to be simulated with the intention of getting a better test for the fault.

PT (Possibly Detected) = AP + NP + P0 + P1

- AP (ATPG Untestable, Possibly Detected)

AP faults are possibly detected faults. A faulty machine response will simulate an "X" rather than a 1 or 0. Analysis has determined that the fault cannot be detected with the current ATPG constraints and restrictions so the fault is removed from the active fault list and no further patterns for detecting this fault is attempted.

- NP (Not Analyzed, Possibly Detected)

NP faults are identical to AP faults except that either analysis was not completed or could not prove that the fault would always simulate as an X. It is still possible that a different pattern could detect the fault and it's classification could become DS, until then it's classification remains NP and it remains in the active fault list

- P0

A fault is classified as P0 fault if a clock fault is detected and the loadable nonscan cell faulty value is set to 0. This classification applies only if the loadable nonscan cells feature is active.

P1

A clock fault is classified as P1 if a clock fault is detected and the loadable nonscan cell faulty value is set to 1. Note that the loadable nonscan cells feature must be active.

UD (Undetectable) = UU + UO + UT + UB + UR

The "undetectable" fault classes include faults which cannot be detected (either hard or possible) under any conditions. When calculating test coverage, these faults are not considered because they have no logical effect on the circuit behavior and cannot cause failures.

- UU (Undetectable Unused)

UU faults are located on circuitry with no connectivity to an externally observable point. During the creation of the simulation model, the default is to remove this unused circuitry which results in these faults not existing. To expose these faults, you need to select the `-nodelete_unused_gate` option of the `set build` command. Faults are immediately placed into this fault class when they are added to the fault list.

- UO (Undetectable Unobservable)

UO faults are similar to UU faults, except they are located on unused gates *with* fanout (that is, gates connected to other unused gates). Faults on unused gates *without* fanout are identified as UU faults.

- UT (Undetectable Tied)

A UT fault is located on a pin that is tied to a value that is the same as the fault value. Faults are immediately placed into this fault class when they are added to the fault list.

- UB (Undetectable Blocked)

A UB fault is located on circuitry that is blocked from propagating to an observable point due to tied logic. Faults are immediately placed into this fault class when they are added to the fault list.

- UR (Undetectable Redundant)

URs fault are undetectable (using both hard detection and possible detection). Test generation fault analysis is performed when adding faults, during pattern-by-pattern test generation (as a result of the `run_atpg` command), and as a dedicated analysis of local or global redundancies (also as a result of `run_atpg`). When adding faults (using the `add_faults` command), an analysis is performed to identify and remove from the active list those faults which can easily be shown to be AU or UR. A simple form of ATPG is used during this analysis. Fault grading can never place a fault in this class.

AU (ATPG Untestable) = AN

"ATPG Untestable" faults include faults which can neither be hard detected under the current ATPG conditions nor proved redundant. When calculating test coverage, these faults are considered the same as untested faults because they have the potential to cause failures.

- AN (ATPG Untestable, Not Detected)

AN faults have not been possibly detected and an analysis was performed to prove it cannot be detected under current ATPG conditions. The analysis also failed the redundancy check. Faults can immediately be placed in this class if they are inconsistent with the pre-calculated constrained value information. Others can require test generation analysis. After they are placed in this class, they are removed from the active fault list and not given any further opportunity to become possible detected. Primary reasons for faults in this classification include:

- Fault untestable due to a constraint which is in effect.
- Fault requires sequential patterns for detection.
- Fault can only be possible detected.
- Fault requires using an unresolvable Z state for detection.

- AX (ATPG Untestable, Timing Exceptions)

For each fault affected by SDC (Synopsys Design Constraints) timing exceptions, if all the gates in both the backward and forward logic cones are part of the same timing exception simulation path, then the fault is marked AU and is assigned an AX subclass. This analysis finds the effects of setup exceptions, so it does not affect exceptions that are applied only to hold time.

To enable this type of analysis, use the `set_atpg -timing_exceptions_au_analysis` command. To configure separate reporting of these faults, use the `set_faults -summary verbose` command.

Note that AX analysis is applied only for transition delay faults. The commands used for AX analysis are accepted for other fault models, but the results will not show any AX faults.

AE (ATPG Untestable) = AE

- AE stands for ATPG Untestable, Low Power. This category is a sub-category of AU (ATPG Untestable) faults.
- A fault is classified as AE, in the presence of one or more sequential compressors, when the number of reseeds or the number of shifts that require care bits do not

confirm to the ATPG budget provided by the user. A fault is marked as an AE when it is tried as a primary fault and the tool cannot obtain a test with a shift power that is lesser than the ATPG budget provided.

ND (Not Detected) = NC + NO

An ND fault indicates that test generation has not yet been able to create a pattern that controls or observes the fault. For these faults, it is possible that increasing the ATPG effort with the `set_atpg -abort_limit` command will result in these faults becoming some other classification.

- NC (Not Controlled)

The NC fault class indicates that no pattern was yet found that would control the fault site to the state necessary for fault detection. This is the initial default class for all faults.

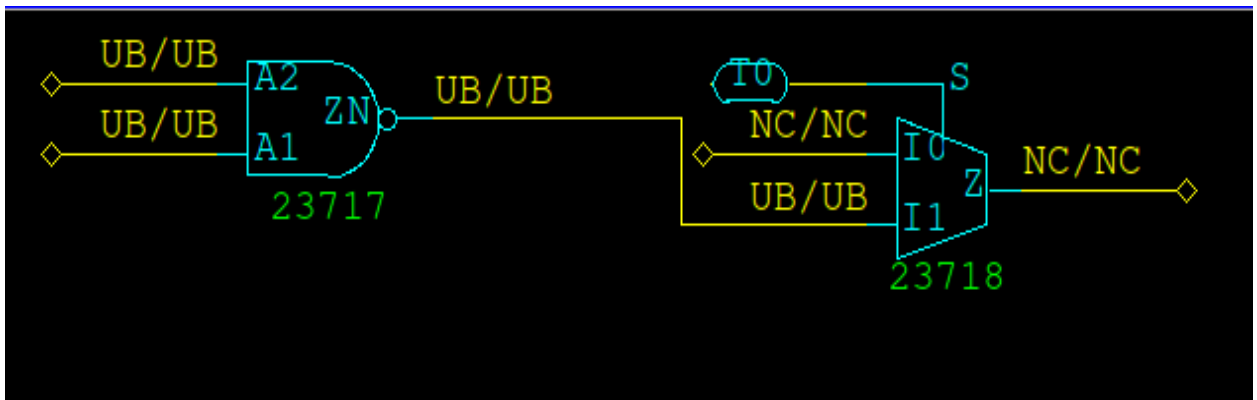
- NO (Not Observed)

The NO fault class indicates that, although the fault site is controllable, that no pattern has yet been found to observe the fault so that credit can be given for detection.

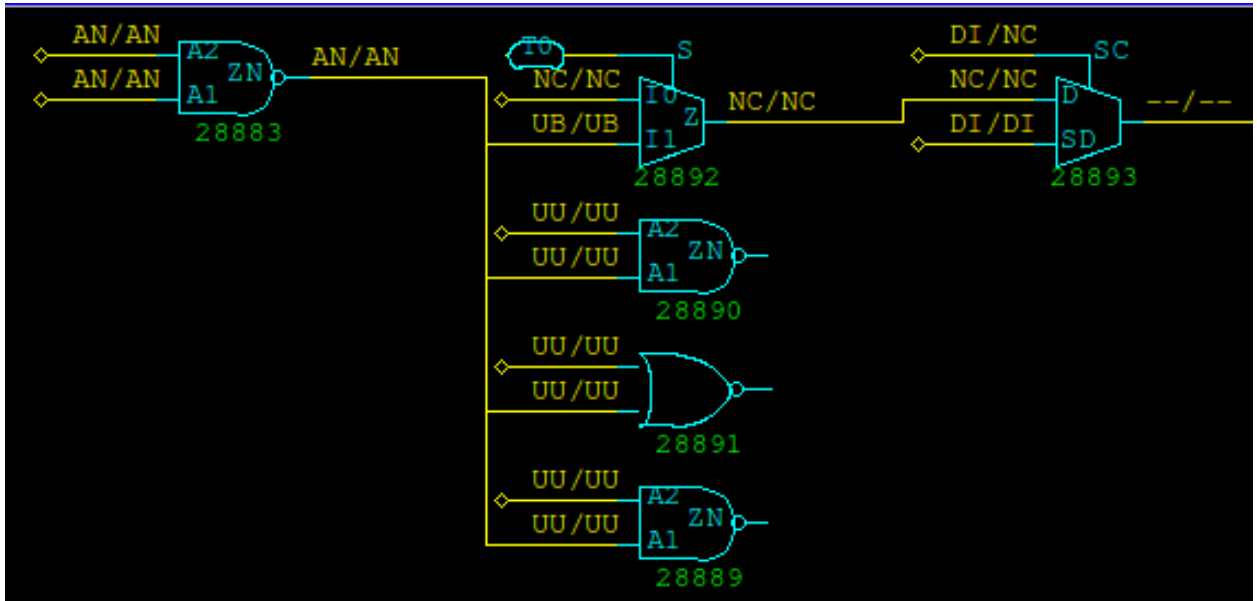
Note:

In case of using the `set_build -delete_unused_gates` command, faults that are blocked for observe are classified as UB faults. On the other hand, when using the `set_build -nodelete_unused_gates` command, the same faults are classified as AN faults.

The following image shows a simple example of a NAND gate that connects to a MUX with tied select line that prevents the observe of the output of the NAND gate:



The following image shows the usage of the `set_build -nodelete_unused_gates` command:



Fault Summary Reports

The following sections describe the various types of summary reports:

- [Fault Summary Report Examples](#)
- [Test Coverage](#)
- [Fault Coverage](#)
- [ATPG Effectiveness](#)

Fault Summary Report Examples

By default, TestMAX ATPG displays fault summary reports using the five categories of fault classes, as shown in the following example.

Fault Summary Report: Test Coverage

Uncollapsed Fault Summary Report

Chapter 14: Fault Lists and Faults

Fault Summary Reports

```

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-T> set_faults -summary verbose
```

The following example shows a verbose fault summary report, which includes the fault classes in addition to the fault categories.

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 [Fault Summary Report: Fault Coverage](#) and [Fault Summary Report: ATPG Effectiveness](#).

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.

The following example shows a fault summary report that includes the fault coverage.

Fault Summary Report: Fault Coverage

```
TEST-T> set_faults -fault_coverage
TEST-T> 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 ATPG effectiveness with the fault summary report, use the `-atpg_effectiveness` option of the `set_faults` command. The following example shows a fault summary report that includes the ATPG effectiveness.

Fault Summary Report: ATPG Effectiveness

```
TEST-T> set_faults -atpg_effectiveness
TEST-T> 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%

See Also

- [Direct Fault Crediting](#)

Using Clock Domain-Based Faults

TestMAX ATPG 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:

- TestMAX ATPG 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.
- Faults that can be launched by one clock and PI/PIO, or that can be captured by one clock and PO/PIO, are not considered MULTIPLE faults. These faults are added,

removed, or reported when only the one clock is specified as the launch or capture clock, and they are considered exclusive faults.

- When the special domains PI, PO or NO_CLOCK are specified, the only faults added are those launched or captured exclusively by the specified domain, unless shared faults are also specified. These domains are treated in a more restricted way because they generally cannot be used to test transition delay faults, so the ability to add them is included mainly for the sake of completeness.

When adding faults launched and captured by specific clocks and also other clocks, as many as four commands might be required. For example:

- The `add_faults -launch A -capture B` command adds faults launched exclusively by A and captured exclusively by B.
- The `add_faults -launch A -capture B -shared` command adds faults launched by A and another clock and captured by B and another clock.
- The `add_faults -launch A -capture B -shared_launch` command adds faults launched by A and another clock and captured exclusively by B.
- The `add_faults -launch A -capture B -shared_capture` command adds faults launched exclusively by A and captured by B and another clock.

The following table list all the commands and command options associated with reporting clock domain-based faults.

Table 4 *Commands and Options Used for 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.
<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 .
<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.
<code>add_faults -shared</code>	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 <code>add_faults</code> options described previously.

Table 4 *Commands and Options Used for Reporting Clock Domain-Based Faults (Continued)*

Command	Description
<code>add_faults</code> <code>-shared_launch</code>	Specifies that faults launched by the specified clock and other clocks are added. An error is reported if you specify this switch without also using the <code>-launch</code> option.
<code>add_faults</code> <code>-shared_capture</code>	Specifies that faults captured by the specified clock and other clocks are added. An error is reported if you use this switch without also using the <code>-capture</code> option.
<code>add_faults</code> <code>-inter_clock_domain</code>	Adds only exclusive faults that are driven and captured by different clock domains.
<code>add_faults</code> <code>-intra_clock_domain</code>	Adds only exclusive faults that are driven and captured by the same clock domains.
<code>remove_faults</code> <code>-launch</code> <code>clock_name</code>	Specifies the launch clock of the faults to be removed. You can use this switch independently, or in conjunction with the <code>-capture</code> switch (described later).
<code>remove_faults</code> <code>-capture</code> <code>clock_name</code>	Specifies the capture clock of the faults to be removed. You can use this switch independently, or in conjunction with the <code>-launch</code> switch (described previously).
<code>remove_faults</code> <code>-exclusive</code>	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.
<code>remove_faults</code> <code>-shared</code>	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 <code>remove_faults</code> options previously).
<code>remove_faults</code> <code>-</code> <code>inter_clock_domain</code>	Removes only exclusive faults that are driven and captured by different clock domains.
<code>remove_faults</code> <code>-</code> <code>intra_clock_domain</code>	Removes only exclusive faults that are driven and captured by the same clock domains.
<code>report_faults</code> <code>-per_clock_domain</code>	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.

Table 4 *Commands and Options Used for Reporting Clock Domain-Based Faults (Continued)*

Command	Description
<code>report_summaries faults -per_clock_domain</code>	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.
<code>report_summaries faults -launch clock_name</code>	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.
<code>report_summaries faults -capture clock_name</code>	Specifies the capture clock of the faults to be reported on. This switch can be used independently or in conjunction with the <code>-launch</code> switch (described previously).
<code>report_summaries faults -exclusive</code>	Excludes the multiple launch and capture section from the report.
<code>report_summaries faults -shared</code>	Reports only the section relating to multiple launch and capture clocks.
<code>report_summaries faults -inter_clock_domain</code>	Reports on only the exclusive faults that are driven and captured by different clock domains.
<code>report_summaries faults -intra_clock_domain</code>	Reports on only the exclusive faults that are driven and captured by the same clock domains.

Using Signals That Conflict With Reserved Keywords

The `MULTIPLE`, `NO_CLOCK`, `PI`, and `PO` names 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_faults -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 `report_summaries faults -per_clock` command, TestMAX ATPG provides only aggregate results. To find individual faults, specify the `report_faults -per_clock_domain` command, then use UNIX editing commands to manipulate the faults of interest.

15

Fault Simulation

Fault simulation determines the test coverage obtained by an externally generated test pattern. To perform fault simulation, you use functional test patterns that were 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 TestMAX ATPG when and what to measure.

The following topics describe fault simulation:

- [Supported Fault Models](#)
- [Fault Simulation Design Flow](#)
- [Preparing Functional Test Patterns for Fault Simulation](#)
- [Preparing Your Design for Fault Simulation](#)
- [Reading Functional Test Patterns](#)
- [Initializing the Fault List](#)
- [Performing Good Machine Simulation](#)
- [Performing Fault Simulation](#)
- [Combining ATPG and Functional Test Patterns](#)
- [Running Multicore Simulation](#)
- [Per-Cycle Pattern Masking](#)

Supported Fault Models

TestMAX ATPG supports test pattern generation for the following fault models:

- *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. For more information on stuck-at faults, see "Understanding Fault Models"

- *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, TestMAX ATPG 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 "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 "Path Delay Fault and Hold Time Testing."

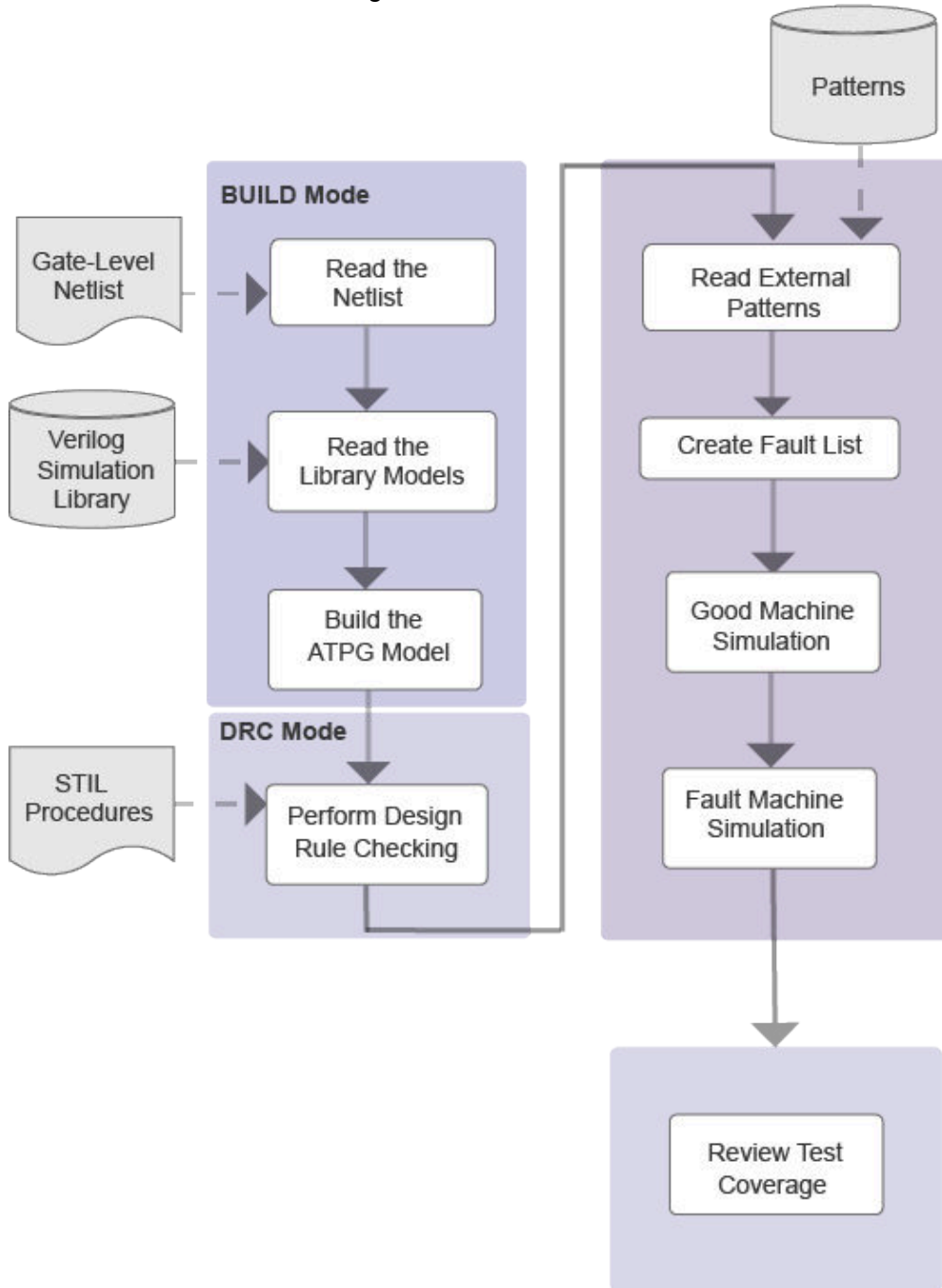
- *IDDQ*

The IDDQ fault model assumes that a circuit defect causes excessive current drain due to an internal short circuit from a node to ground or to a power supply. For this model, TestMAX ATPG 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. For more information, see "Quiescence Test Pattern Generation."

Fault Simulation Design Flow

The fault simulation design flow prepares functional test patterns for fault simulation, reads the test patterns, initializes the fault list, performs good machine simulation, performs fault simulation, and reviews the test coverage.

Figure 97 Fault Simulation Design Flow



Preparing Functional Test Patterns for Fault Simulation

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 requirements for test patterns are described in the following sections:

- [Pattern Compliance with ATE](#)
- [Checking Patterns for Timing Insensitivity](#)

Pattern Compliance with ATE

Because the functional test patterns are 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.

Every ATE has its own set of rules for timing restrictions, including the following examples:

- 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 Patterns 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.

Timing Sensitivity

The following examples 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, TestMAX 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 TestMAX 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.

- 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.

Preparing Your Design for Fault Simulation

The process for preparing your design for fault simulation is generally the same as preparing for the ATPG design flow:

- [Preprocessing the Netlist](#)
- [Reading the Design and Libraries](#)
- [Building the ATPG Design Model](#)
- [Declaring Clocks](#)
- [Running DRC](#)

Preprocessing the Netlist

If necessary, preprocess the netlist for compatibility with TestMAX ATPG. For more information, see [Netlist Requirements](#).

Reading the Design and Libraries

As with ATPG, for TestMAX ATPG fault simulation you first invoke TestMAX ATPG, read in the design netlist, and read in the library models. For details, see [Reading the Netlist and Reading Library Models](#).

Note the following example command sequence:

```
% tmax  
  
BUILD-T> read_netlist spec_asic.v  
  
BUILD-T> read_netlist spec_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 command:

```
BUILD-T> run_build_model top_module_name
```

Example 1 run_build_model Transcript

```
BUILD-T> run_build_model spec_asic
-----

Begin build model for topcut = spec_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 might want to declare clocks so that TestMAX 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 might be necessary to remove `add_clocks` commands.

If certain ports in the functional stimuli are operated in pulsed fashion within a cycle, you might want to provide this information to TestMAX ATPG by declaring these ports to be clocks.

A typical command sequence for declaring a clock is shown in the following example:

```
DRC-T> add_clocks 0 CLK
DRC-T> add_clocks 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 Operation

For nonscan operation, if you have defined a clock, you do not need to specify an STL procedure file unless it is necessary for defining port timing. To run DRC without a file, enter the following commands:

```
DRC-T> set_drc -nofile
DRC-T> run_drc
```

To run DRC with a file, enter the following command:

```
DRC-T> 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 TestMAX ATPG 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 2 shows a transcript of `run_drc` for a nonscan operation.

Example 2 Running DRC for Nonscan Operation

```
DRC-T> set_drc -nofile
DRC-T> run_drc
```

```
-----
Begin scan design rule 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
```

Chapter 15: Fault Simulation

Preparing Your Design for Fault Simulation

```
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 Operation

For scan operation, the STL procedure file 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 STL procedure file, or you can use a combination of both.

To run DRC with a STIL procedure file, enter the following command:

```
DRC-T> run_drc filename
```

Reading Functional Test Patterns

You can read functional test patterns using the Set Patterns dialog box, or by running the `set_patterns` command at 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` option.

The following sections describe how to read functional test patterns:

- [Using the Set Patterns Dialog Box](#)
- [Using the set_patterns Command](#)
- [Specifying Strobes for VCDE Pattern Input](#)

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.
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

The following example shows how to read functional test patterns using the `set_patterns` command:

```
TEST-T> set_patterns -external data.vcde -strobe rising CLK \  
-strobe offset 50 ns
```

TestMAX ATPG automatically determines the type of patterns being read and whether they are in standard or GZIP format, and handles all variations automatically.

The following example transcript show output from the `set_patterns external` command:

```
TEST-T> set_patterns 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.

Specifying Strokes for VCDE Pattern Input

Functional patterns in VCDE format do not contain 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 states:

- *None*: This option is not supported for VCDE input.
- *Period*: Strokes 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.

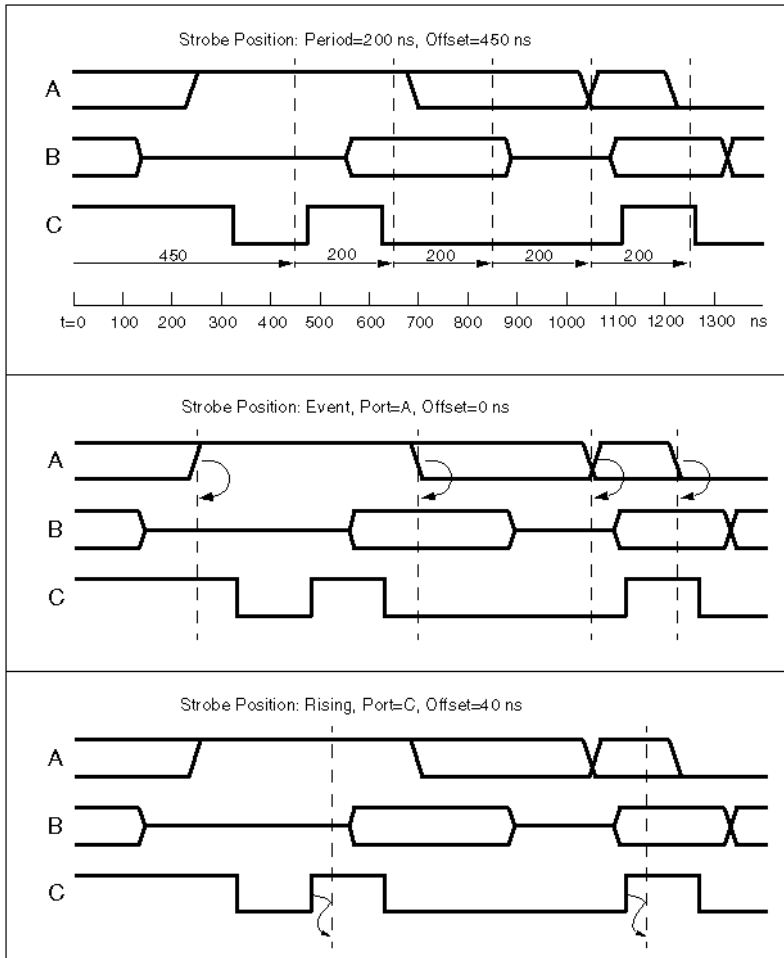
To make the strobe occur at a specific time after the trigger event, specify a positive offset value. Negative offsets for strobes are not supported.

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 of the `set_patterns` command. For details on the command syntax, see the online help for the `set_patterns` command.

The following figure shows some timing diagrams with the strobe points resulting from various strobe specification settings.

Figure 98 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

The following sections show you how to initialize a fault list:

- [Using the Add Faults Dialog Box](#)
- [Using the add_faults Command](#)

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 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:

```
TEST-T> add_faults -all
```

You can also define fault lists by reading faults or nofaults from a file or by defining specific hierarchical blocks for adding or removing faults:

```
TEST-T> read_faults saved_faults_file
TEST-T> 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 TestMAX 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 and Combining ATPG and Functional Test Patterns](#).

Performing Good Machine Simulation

You should perform a good machine simulation using the functional patterns before running a fault simulation, to compare the TestMAX ATPG 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.

As part of setting up the good machine simulation, refer to contention checking as described in [Choosing Settings for Contention Checking](#).

The following sections show you how to set up other good machine simulation parameters:

- [Using the Run Simulation Dialog Box](#)
- [Using the `set_simulation` and `run_simulation` Commands](#)

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 TestMAX ATPG main window. The Run Simulation dialog box appears.

For descriptions of these controls, see the online help for the `run_simulation` command.

2. Select required 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_simulation` and `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:

```
DRC-T> set_simulation -measure pat -oscillation 20 2 -verbose
TEST-T> run_simulation -sequential
```

For the complete syntax and option descriptions, see Online Help for each command.

The following example 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.

Example 1 Good Machine Simulation Transcript

```
TEST-T> 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.

The following sections describe how to run fault simulation:

- [Using the Run Fault Simulation Dialog Box](#)
- [Using the run_fault_sim Command](#)
- [Writing the Fault List](#)

The `set_simulation` command described in the [Performing Good Machine Simulation](#) section 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 TestMAX ATPG main window. The Run Fault Simulation dialog box appears.

For descriptions of these controls, see Online Help for the `run_fault_sim` command.

2. Select required 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:

```
TEST-T> run_fault_sim -sequential
```

The following example shows a typical transcript of a fault simulation run is shown in Example 1.

```
TEST-T> 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](#) .

The following command generates a summary of fault simulation.

```
TEST-T> 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 `write_faults` command writes (saves) a fault list.

```
TEST-T> 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 and 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. The following sections describe the various methods for combining test patterns:

- [Creating Independent Functional and ATPG Patterns](#)
- [Creating ATPG Patterns After Functional Patterns](#)
- [Creating Functional Patterns After ATPG Patterns](#)
- [Using TestMAX ATPG with Z01X](#)

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 TestMAX ATPG.

To determine the test coverage overlap, you must perform a detailed comparison of the fault lists from both methods. You should expect overlap. In fact, you might prefer redundancy.

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.

The following steps show a typical flow:

1. Use TestMAX 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 TestMAX ATPG to create ATPG patterns for the fault list you created in step 3.

Example 1 shows a command file that implements this flow.

Example 1 Creating ATPG Patterns After Functional Patterns

```
#
# --- ATPG follows Fault Grade flow
#
read_netlist spec_design.v -del      # read netlist
read_netlist spec_lib.v             # read library modules
run_build_model                     # form in-memory design image
add_clocks 0 CLK                    # define clock
add_clocks 1 RESETB                 # define async reset
run_drc                             # DRC without a procedure file
set_patterns -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_faults 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 spec_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_faults 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.

The following steps show a typical flow:

1. Use TestMAX 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*.
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 2 shows a command file sequence that illustrates this flow.

Example 2 Creating Functional Patterns After ATPG Patterns

```
#  
# --- ATPG before Fault Grade  
#  
read_netlist spec_design.v -del      # read netlist  
read_netlist spec_lib.v             # read library modules  
run_build_model                    # form in-memory design image  
add_clocks 0 CLK                   # define clock  
add_clocks 1 RESETB                # define async reset  
add_pi_constraints 1 TEST           # define constraints  
run_drc spec_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_faults pass1.flt -all -uncollapsed -rep # save cumulative  
fault list  
#  
# --- switch to Fault Grade mode  
#  
drc -force                          # clocks will still be defined
```

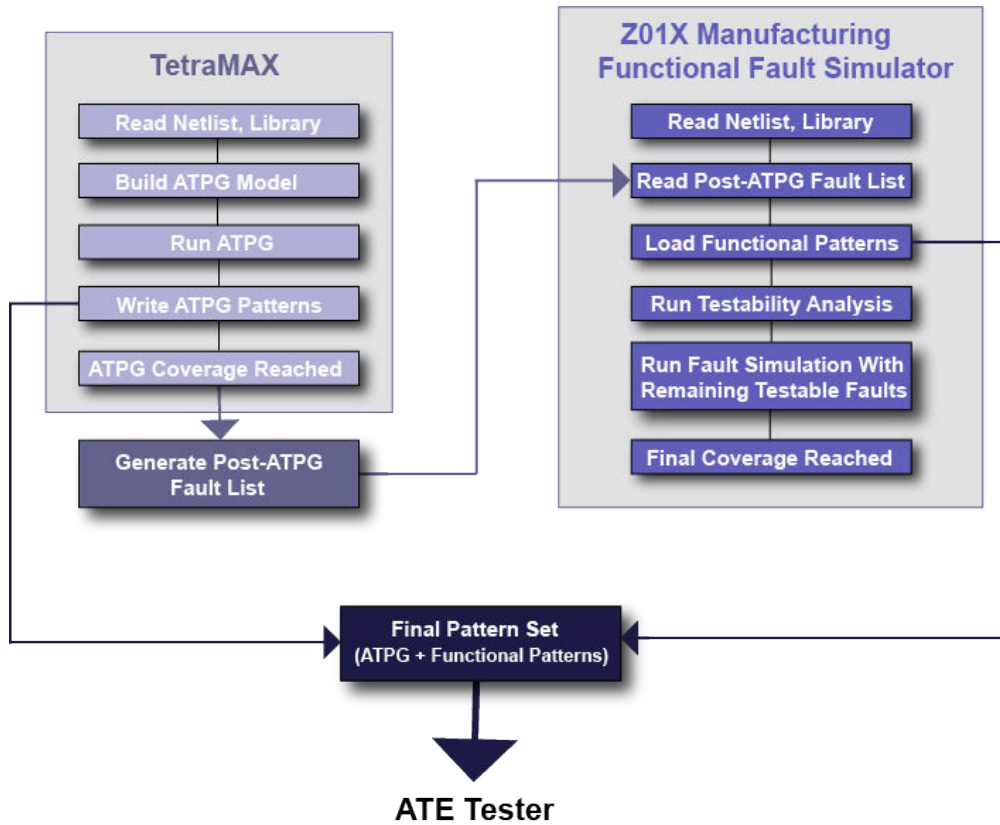
```
remove_pi_constraints -all      # don't constrain when using ext patterns
set_drc -nofile
run_drc                        # switch to test mode
set_patterns -external b010.vin # read in external patterns
set_simulation -measure pat     # set up for fault sim
run_simulation -sequential     # perform 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_faults pass2.flt -all -uncollapsed -replace # save fault list
```

Using TestMAX ATPG with Z01X

You can improve ATPG coverage by using TestMAX ATPG and the Z01XTM functional simulator to create a combined set of ATPG and functional patterns. Z01X accepts most fault types, including stuck-at, transition, IDDQ, and bridging faults.

To use the general TestMAX ATPG-Z01X flow, you create a post-ATPG fault list in TestMAX ATPG and import it into Z01X. Based on the fault list, Z01X creates a set of functional patterns that are combined with the ATPG patterns created from TestMAX ATPG and used by the tester.

Figure 99 General TestMAX ATPG-Z01X Fault Simulation Flow



To create a list of stuck-at faults for Z01X, use the `write_faults` command, as shown in the following example:

```
TEST-T> write_faults ud.au_tcl-test.flt -class {UD AU ND} \
-replace
```

Transition Fault Flow

The process for using transition faults in Z01X involves some additional steps that specify clock domain information:

1. Use TestMAX ATPG to generate one fault list per clock domain.

You can use the following command to create a report containing clock domain information:

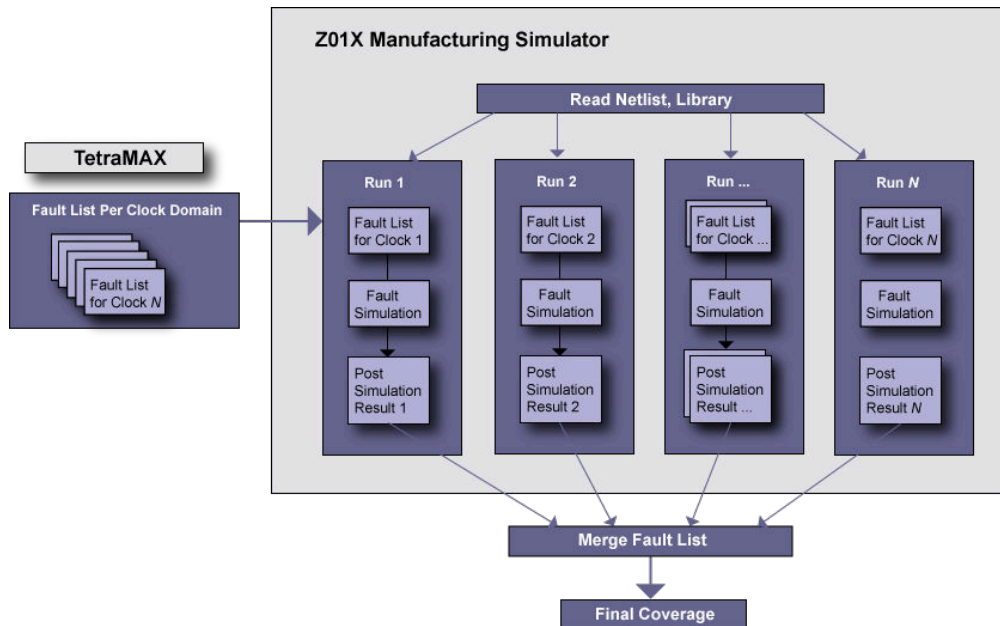
```
report_faults -all -per_clock_domain > flt_pr_ck_dmn.txt
```

2. Define the clock frequency for each clock domain when the individual fault lists are imported by Z01X.
3. Run a fault simulation for each fault list.

4. Merge each run.

For cross-clock domain faults, make sure to include the faults in both domains, and fault-simulate each clock domain starting with the slowest clock and progressing to the fastest clock.

Figure 100 Transition Fault Flow in Z01X



For complete details on using Z01X, see the *Z01X Simulator Manufacturing Assurance User Guide*.

Running Multicore Simulation

Multicore simulation is a methodology that enables you to improve simulation runtime by launching multiple slaves to parallelize fault and logic simulation to work on a single host. You can specify the number of processes to launch based on the number of CPUs and available memory on the machine.

Multicore simulation provides similar runtime reductions and works the same way as the multicore ATPG architecture described in Running Multicore ATPG.

The following topics describe how to use multicore simulation and analyze its performance:

- [Invoking Multicore Simulation](#)
- [Interrupt Handling](#)

- [Processes Summary Report](#)
- [Resimulating ATPG Patterns](#)
- [Limitations](#)

Invoking Multicore Simulation

Multicore simulation is activated by the following `set_simulation` command:

```
set_simulation -num_processes <
number
| max>
```

The `number` specification refers to the number of slave processes used during simulation. If `max` is specified, then TestMAX ATPG computes the maximum number of processes available in the host, based on number of CPUs. If TestMAX ATPG detects that the host has only one CPU, then single-process simulation is performed instead of multicore simulation with only one slave. Note that you should not specify more processes than the number of CPUs available on the host. You should also consider whether there are other CPU-intensive processes running simultaneously on the host when running the number of processes. If too many processes are specified, performance will degrade and might be worse than single-process simulation. On some platforms, TestMAX ATPG cannot compute the number of CPUs available and will issue an error if `max` is specified.

Interrupt Handling

To interrupt the multicore simulation process, use Control-c ; in the same manner as a single process. If a slave ends or is killed, the master and remaining slaves will continue to run.

If the master ends or is killed, the slaves will also halt. In this case, there are no ongoing zombie processes, dangling files, or memory leakage.

Processes Summary Report

Memory consumption needs to be measured to tune the global data structure to improve the scalability of multicore architecture. Legacy memory reports are not sufficient because they do not deal with issues related to copy-on-modification. ; To facilitate collecting performance data, a summary report of multicore simulation is printed automatically at the end of the simulation process when the `-level expert` option is used with the `set_messages` command. The summary report appears as shown in Example 1.

Example 1 Example Processes Summary Report

Processes Summary Report

```

-----
Process      Patterns      Time (s)          Memory (MB)
-----
ID  pid  Internal    CPU  Elapsed  Shared  Private  Total  Pattern
-----
0   7611   1231      0.53  35.00   67.78   30.54   98.32   5.27
1   7612    626     35.68  35.00   64.87   22.31   87.18   0.00
2   7613    605     35.50  35.00   64.71   22.47   87.18   0.00
Total      1231     71.71  35.00   67.78   75.32  143.10   5.27
-----

```

The report in Example 1 contains one row for each process. The first process with an ID of 0 ; is the master process. The child processes have IDs of 1, 2, 3, and so forth. The last row is the sum for each measurement across all processes.

The `pid` ; column lists the process IDs. The `Patterns` ; are the total number of patterns stored by the master or the number of patterns generated by the slave in this particular simulation session. The columns listed under `Time (s)` ; include CPU time and wall time.

The `Memory` ; measurements are obtained by parsing the system-generated file `/proc/pid/smaps`. The file contains memory mapping information created by the OS while the process still exists. The `/proc/pid/` directory cannot be found after the process terminates. The tool parses this file at the proper time to gather memory information for the reporting at the end of parallel simulation.

The `Memory` ; measurement includes `Shared`, ; `Private`, ; `Total`, ; and `Pattern`. ; The `Shared` ; column refers to all processes that share the same copy of the memory. The `Private` ; column refers to the process stores local changes in the memory. The `Total` ; column is the sum of `Shared` ; and `Private`. ; The `Pattern` ; column refers to memories allocated for storing patterns. The total memory consumption of the entire system is the `Total` ; item in the row `Total`, ; which is the sum of total shared memory (maximum of shared memories for each process) and the total private memory (sum of all private memory for all processes). Although the memory for patterns is listed separately, it is part of the master private memory.

Due to a lack of OS support, the Memory section of the summary report is only available on Linux and AMD64 platforms. No other platform gives shared or private memory information in a copy-on-write context. On other formats, the memory reports all 0 ;s for items other than the pattern memory.

Note: The report in Example 1 is printed only when the `set_messages` command is set to `-expert`. Otherwise, a default summary report, similar to the following example, is printed out:

```
End parallel ATPG: Elapsed time=35.00 sec, Memory=143.10MB.
```

```
Processes Summary Report
```

Resimulating ATPG Patterns

You can resimulate ATPG patterns to mask out the observe values for any mismatched patterns verified with `run_simulation` command. This feature is enabled when both multicore ATPG and ATPG pattern re-simulation are enabled, as shown in the following example:

```
set_atpg -resim_atpg fault_sim  
set_atpg -num_processes 2  
run_atpg -auto
```

The command output is similar to single-process ATPG pattern simulation with mismatch masking messages. The process summary report is automatically printed out at the end of ATPG, logic simulation, and fault simulation; this report is similar to the process summary report for the corresponding standalone commands.

Limitations

There are several `run_fault_sim` and `run_simulation` command options that are not supported by multicore simulation.

The unsupported `run_fault_sim` options are as follows:

- `-detected_pattern_storage` — This option stores the first detection pattern for each fault. In multicore fault simulation, the patterns are not simulated in the order of the pattern number occurrence.
- `-distributed` — This option is used to launch distributed fault simulation only. It cannot be used in conjunction with multicore fault simulation.
- `-nodrop_faults`

The unsupported `run_simulation` options are as follows:

- `-sequential`
- `-sequential_update`
- `-update`

See Also

- [Running Multicore ATPG](#)

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 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 recalculated, or the patterns can be simulated.

The following sections describe per-cycle pattern masking:

- [Flow Options](#)
- [Masks File](#)
- [Running the Flow](#)
- [Limitations](#)

Flow Options

There are two flows available for running per-cycle pattern masking: the tester flow and the simulation flow.

The following steps are for the tester flow:

1. The original patterns are written out from TestMAX ATPG.
2. A few mismatches occur on the tester.
3. The patterns and mismatches are read into TestMAX ATPG.
4. Mismatches are masked in the pattern.
5. Masked patterns are optionally fault simulated again.

6. Masked patterns are written out from TestMAX ATPG.
7. All patterns pass on the tester.

The following steps are for the simulation flow:

1. The original patterns are written out from TestMAX ATPG.
2. Mismatches occur during fault simulation.
3. The patterns and mismatches are read into TestMAX ATPG.
4. Mismatches are masked in the pattern.
5. Masked patterns are optionally fault simulated again.
6. Masked patterns are written out from TestMAX ATPG.
7. All patterns pass during simulation.

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. The pattern-based format with chain name from parallel STILDPV simulation is also supported. See “Providing Tester Failure Log Files” 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.

You can also create the masks file by collecting mismatches that occur during simulation, in serial or parallel mode, of STIL patterns. See [Predefined Verilog Options](#) in the *Test Pattern Validation User Guide* for information on the `+tmax_diag` option that controls this process.

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. It is possible to update binary, WGL or serial-STIL patterns with failures from the parallel simulation of STIL patterns; and then, to write the parallel STIL masked patterns for simulation.

To read the patterns in the external buffer and read in the masks file, use the following `set_patterns` command:

```
set_patterns -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_patterns -external pat.stil -resolve_differences 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 shown in the following example:

```
TEST-T> set_patterns -external pat.stil.gz -resolve_differences mask.txt
TEST-T> write_patterns 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_differences` and then `write_patterns`. In this case, the difference with previous method is that the `run_atpg -resolve_differences` 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 are useless and thus removing them will reduce 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_simulation` command after `run_atpg -resolve`. If new failures occur, you must mask the patterns another time using `set_patterns -resolve_differences`.

An alternate flow is shown in the following example:

```
TEST-T> set_patterns -external pat.stil.gz -resolv_differences mask.txt
TEST-T> add_faults -all
```

Chapter 15: Fault Simulation

Per-Cycle Pattern Masking

```
TEST-T> run_atpg -resolve_differences
TEST-T> run_simulation
TEST-T> write_patterns pat.masked.stil.gz -format STIL -compress gzip \
-external
```

You can also use this feature when the patterns are split after ATPG. In this case, make sure you generate the failures used for masking by using the log of the cycle-based failures from the tester. Also, the cycle count must be reset from the execution of a particular split pattern set to the next split pattern set. The flow is shown in following example:

```
set_patterns -external <pattern_filename_0_to_mask> -resolve
<failures_reset_0>
write_patterns ...
set_patterns -delete
set_patterns -external <pattern_filename_1_to_mask> -resolve
<failures_reset_1>
write_patterns ...
set_patterns -delete
etc. ...
```

You should specify the `set_diagnosis -cycle_offset` command when using a cycle-based failures log file for masking and an offset is applied to the cycle.

For multiple pattern sets, you need to use `-split` option, as shown in the following example:

```
TEST-T> set_patterns -external pat1.stil.gz -resolv_differences mask1.txt
-split
TEST-T> set_patterns -external pat2.stil.gz -resolv_differences mask2.txt
-split
TEST-T> set_patterns -external pat3.stil.gz -resolv_differences mask3.txt
-split
TEST-T> add_faults -all
TEST-T> run_atpg -resolve_differences
TEST-T> run_simulation
TEST-T> write_patterns pat.masked.stil.gz -format STIL -compress gzip \
-external
```

Limitations

The following limitations apply to this flow:

- It is not possible to write masked full-sequential patterns in parallel format.
- The binary pattern file cannot be masked with a cycle-based format masks file.
- For patterns with multiple load-unloads with measures on the scanout in each unload, only the failures for the first unload can be masked.

16

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, and so forth. In the scan-ATPG environment, scan chain load and unload are controlled through an ATE clock. However, internal clock signals that reach state elements during capture are PLL-related.

The following sections describe on-chip clocking support:

- [OCC Background](#)
- [OCC Definitions, Supported Flows, Supported Patterns](#)
- [OCC Limitations](#)
- [TestMAX DFT to TestMAX ATPG Flow](#)
- [OCC Support in TestMAX ATPG](#)
- [OCC-Specific DRC Rules](#)

OCC 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 is 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 might 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. TestMAX DFT and TestMAX 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 TestMAX ATPG for at-speed test generation.
- OCC circuitry can operate asynchronously to shift and other clocks from the tester.

OCC Definitions, Supported Flows, Supported Patterns

Note the following definitions as they apply to OCC:

- *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 might 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.

OCC is supported in the following flows:

- TestMAX DFT-to-TestMAX ATPG flow (for details, see Chapter 7, “Using On-Chip Clocking,” in the *TestMAX DFT User Guide Vol. 1: Scan*)
- Non-TestMAX DFT to TestMAX ATPG Flows:
 - Basic Scan with On-Chip Clocking
 - Adaptive Scan with On-Chip Clocking

Note the following pattern support available in OCC:

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

OCC Limitations

Note the following limitations for OCC support:

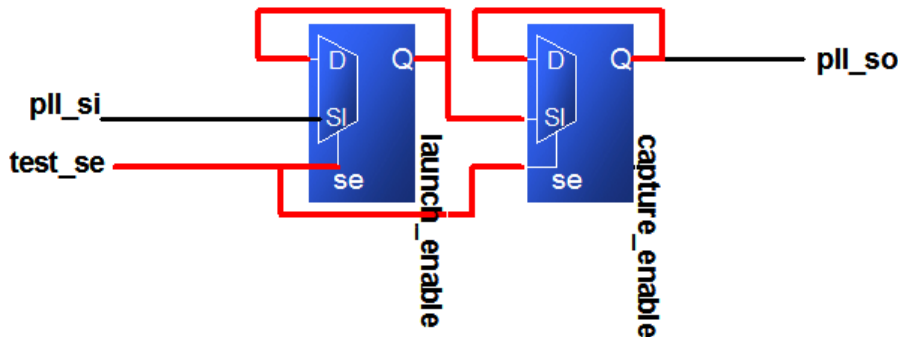
- You must use generic capture procedures for internal/external clocking. For more information, see [Creating Generic Capture Procedures](#).
- You cannot use the OCC from TestMAX DFT with the `set_delay -launch_cycle last_shift` command. However, you can use it with the `set_delay -launch_cycle extra_shift` command if it is used in combination with pipelined scan enable. In this case, the `scan_en` pin must be connected to the non-pipelined scan enable input.
- Multi-cycle paths can only be tested when they are defined in a `MultiCyclePath` block for synchronized multi frequency clocking. You must also specify the `set_drc -multiframe_paths` command.
- The clock frequency of the PLL generating internal clocks cannot change dynamically — must be constant (that is, programmable bits must be nonscan 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.
- 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 one time 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 TestMAX ATPG 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 TestMAX ATPG database, the timing that is written out is nominal and might not match the design's actual clock timing.

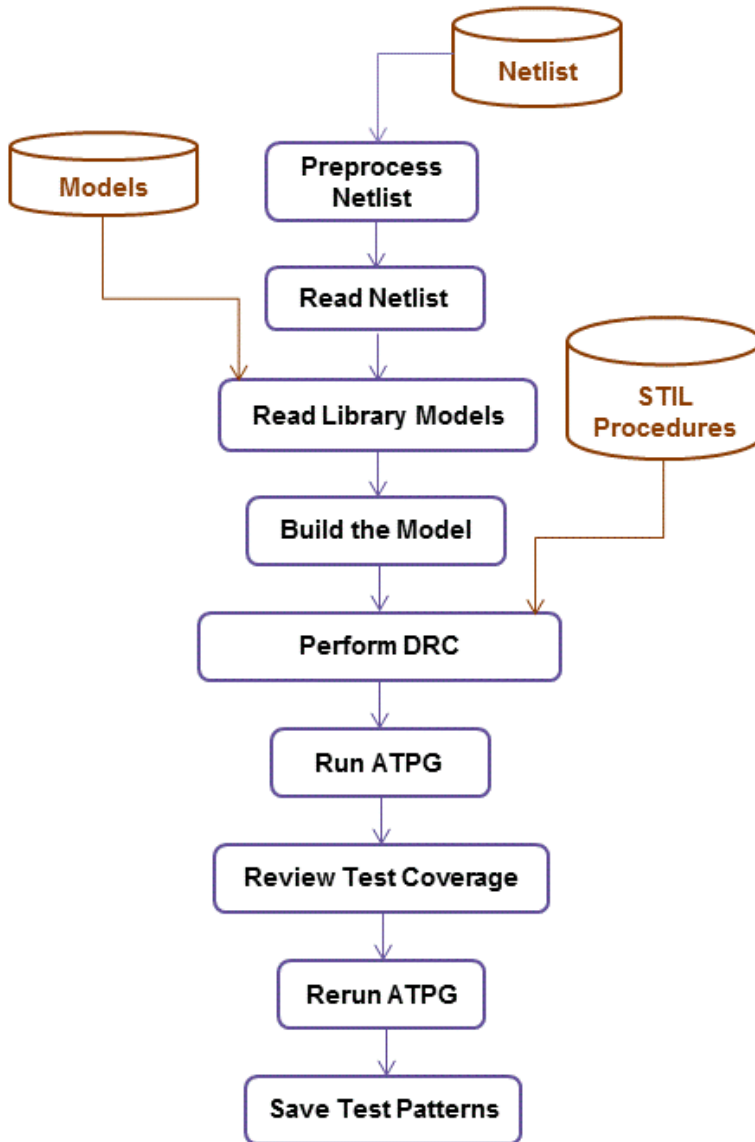
TestMAX DFT to TestMAX ATPG Flow

This flow automatically writes out the STIL procedure file for TestMAX ATPG and the Verilog netlist.

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

The following figure illustrates the basic TestMAX DFT to TestMAX ATPG design flow.

Figure 101 TestMAX DFT to TestMAX ATPG Flow



The basic TestMAX DFT-to-TestMAX ATPG design flow consists of the following steps:

1. Edit your netlist to meet the requirements of TestMAX ATPG (see [Netlist Requirements](#)).
2. Read the netlist (see [Reading in the Netlist](#)).
3. Read the library models (see [Reading Library Modules](#))

4. Build the ATPG design model (see [Building the ATPG Model](#))
5. Read in the STIL test protocol file, automatically generated by TestMAX DFT (see [Selecting the Pattern Source](#)).
6. Perform DRC and make any necessary corrections (see [Performing Test Design Rule Checking](#)).

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
```

7. Prepare the design for ATPG by setting up the fault list, and setting the ATPG options (see [Preparing for ATPG](#)).

Depending on the ratio between the `_default_WFT_` and the OCC clocks, the `set_atpg -min_ateclock_cycles` command might be needed.

The capture sequence for OCC clocks uses the `multiclock_capture` procedure (if generic capture procedures are used). There are as many of these as the number of launch and capture clocks required. The Synopsys OCC controller requires an ATE clock falling edge to occur after the scan enable has become inactive to start its count, then emits its first clock to correspond with the sixth following clock coming from the PLL. If the scan enable becomes active again before all of the pulses required from the OCC controller are emitted, then the capture pulses are truncated and the patterns will fail simulation.

When the ratio of the slowest PLL clock period to the ATE clock period is not high enough to ensure that all OCC clock pulses are emitted, the `set_atpg -min_ateclock_cycles` command should be used to add to the number of ATE clock cycles.

8. Run ATPG (see [Running ATPG](#)).
9. Review the test coverage and rerun ATPG if necessary (see [Reviewing Test Coverage](#)).
10. Save the test patterns and fault list (see [Writing ATPG Patterns](#)).

You should not use any OCC IP that is not created by DFT Compiler with TestMAX ATPG. If you have this type of IP, you should refer to the “User-Defined Instantiated Clock Controller and Chain Insertion Flow” section in the *TestMAX DFT User Guide*.

OCC Support in TestMAX ATPG

OCC support in TestMAX ATPG 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.

The following sections describe OCC support in TestMAX ATPG:

- [Design Set Up](#)
- [OCC Scan ATPG Flow](#)
- [Waveform and Capture Cycle Example](#)
- [Using Synchronized Multi Frequency Internal Clocks](#)
- [Using Internal Clocking Procedures](#)

Design Set Up

When a design contains both internal clocks (commonly driven by PLL sources), and external (primary input) clocks, the TestMAX ATPG 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 might not eliminate all ambiguity, especially when both clock sources are presented to the same internal element.

TestMAX ATPG 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
```

If you are using synchronous multi frequency internal clocks, you should not use this example. Instead, TestMAX ATPG offers a specific flow for designs that use synchronous multi frequency internal clocks. For details on this process, see [Using Synchronized Multi Frequency Internal Clocks](#). However, if your design contains asynchronous internal clocks, then you should use the example cited above.

Black boxes are often the sources of the PLL clocks. When PLL clocks are driven by logic, DRC might fail because of how these clocks are simulated. Simulation events are driven

on the defined PLL clock source, and these events are used to trace the OCC controller functionality. Other simulation events that propagate through the logic confuse this DRC analysis. To prevent this problem, you should replace the instances driving the PLL clocks with TIEX primitives:

```
set_build -instance_modify {pll1864/U93 TIEX}  
set_build -instance_modify {pll1923/U45 TIEX}
```

OCC Scan ATPG Flow

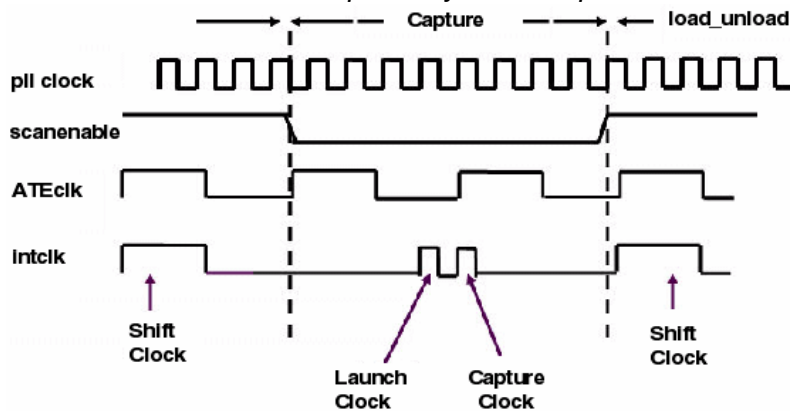
The OCC Scan ATPG flow consists of the following steps:

1. Read the design files (see [Reading the Library Modules](#)).
2. Build the design (see [Building the ATPG Model](#)).
3. Run DRC with the TestMAX ATPG STIL procedure file created by TestMAX DFT after scan insertion in presence of PLL circuitry (see [Performing Design Rule Checking](#)).
4. Run ATPG (see [Running ATPG](#)).

Waveform and Capture Cycle Example

The following figure shows an example of the relationship between various clocks when the design contains an OCC controller.

Figure 102 Waveform and Capture Cycle Example



The `refclk` must pulse in every vector. This figure also contains information about `pllclk`, `ateclk`, and `intclk`.

Using Synchronized Multi Frequency Internal Clocks

By default, internal clocks derived from an OCC Controller are considered by TestMAX ATPG to be asynchronous to each other. However, you can specify the timing relationships of internal clocks, thus improving the test quality. This section describes the process for implementing synchronized internal clocks at one or multiple frequencies in an OCC Controller.

It is important to note that this capability requires the PLL clocks to be synchronized in the design and requires the OCC Controllers to actually synchronize their output pulses. TestMAX ATPG uses the information provided to it and does not do any checking to ensure that this reflects the actual circuit design.

The following sections describe how to specify synchronized multi frequency internal clocks:

- [Enabling Internal Clock Synchronization](#)
- [Clock Chain Reordering](#)
- [Clock Chain Resequencing](#)
- [Finding Clock Chain Bit Requirements](#)
- [Reporting Clocks](#)
- [Reporting Patterns](#)

Enabling Internal Clock Synchronization

To enable internal clock synchronization, specify the `ClockTiming` block in the STIL Procedure File . There are several command switches, described later in this section, that can be used when this feature is enabled.

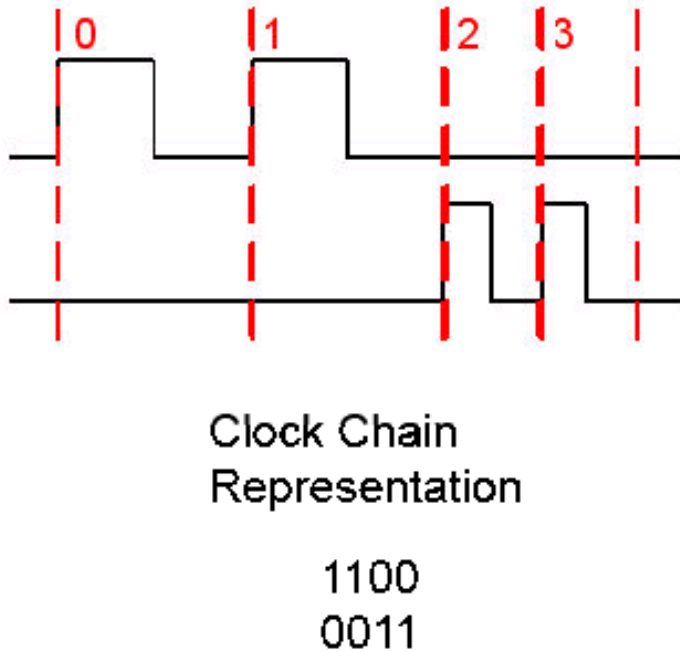
The `ClockTiming` block is placed in the top level of the `ClockStructures` block, which already describes other aspects of the internal clocks.

For details on how to enable internal clock synchronization in the STL procedure file, see the [Specifying Synchronized Multi Frequency Internal Clocks for an OCC Controller](#) section.

Clock Chain Reordering

The clock chain has one register bit per clock cycle. The value loaded into this register controls whether the OCC controller allows a clock pulse from the PLL to propagate during its cycle. ATPG calculates the pattern by ordering the clock pulses, and this initial order must be re-sequenced to reflect period and latency differences between the clocks.

Figure 103 Clock Chains Before Reordering
ATPG Frames



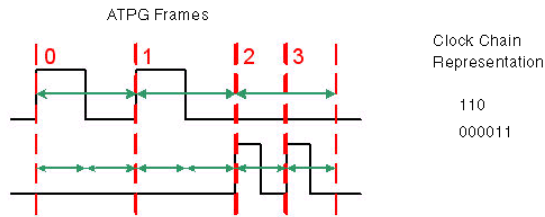
In the preceding figure, note that the order of the clock chain bits is the same as defined in the `Cycle` statements of the `PLLStructures` block, with ATPG frame 0 representing Cycle 0, and so forth.

Clock Chain Resequencing

By default, clock chain resequencing is done to convert the ATPG frame sequence to an equivalent duration in terms of clock periods. Since different clocks might have different periods, this might result in very different sequence lengths to cover the same capture time duration.

Latency is ignored when all clocks pulsed in a capture sequence are defined in the same `PLLStructures` block, or when the latency period (that is the latency number times the minimum clock period within the `PLLStructures` block) is the same even though the clocks are in different `PLLStructures` blocks. In this case, resequencing is based on period times and whether `MultiCyclePath` blocks are defined.

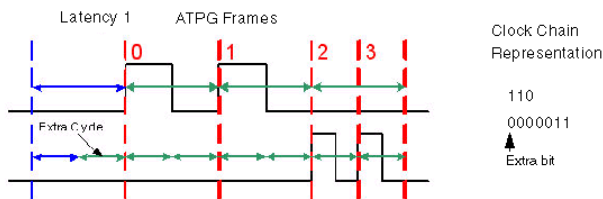
Figure 104 Clock Chain Resequencing with the Same Latency



Note that in the preceding figure, twice as many bits are needed to represent the 2X clock. For this reason, clock chains are allowed to have different lengths when a ClockTiming block is used.

Latency must be considered when a capture sequence contains clock pulses of clocks having different latency periods. In this case, extra padding cycles are added for the clock with the shorter latency period so that the clock periods coincide at the first ATPG frame.

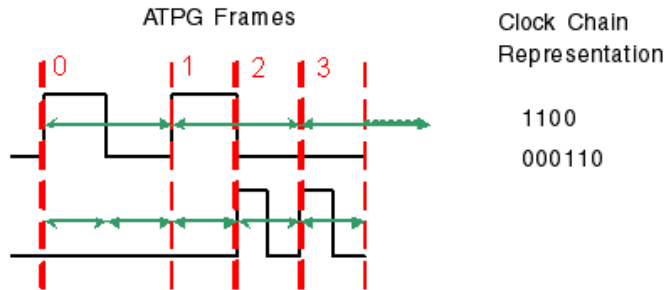
Figure 105 Clock Chain Resequencing with Different Latencies



Note that in the preceding figure, two clocks from different PLLStructures blocks with the same latency number have different latency periods because of their different frequencies. This requires an extra padding bit to be added to its clock chain.

When clock overlapping is enabled, either by the MultiCyclePath statement in the STL procedure file or by the set_drc -fast_multifrequency_capture on command, clock chain re-sequencing is required to get the final result. For example, in the Latency 0 case, see the following figure.

Figure 106 Clock Chain Resequencing When Clock Overlapping is Enabled



Finding Clock Chain Bit Requirements

The required clock chain lengths can be calculated and combined with the number of internal clock pulses that are used, based on the `set_atpg -capture_cycles` specification.

You can also determine the clock chain bit requirements using the following command:

```
set_messages -level expert
```

After pattern generation, before the summary is printed, the following message will appear:

```
Warning: 238 clock pulses rejected. Clock 895 has a 6 bit clock chain,
but needs 13 bits. (M720)
```

The clock number refers to the clock source, whose instance name can be found using the `report_primitives` command. The clock chain length reported is the maximum needed. If a M720 message is not printed, then the clock chains meet or exceed the required length.

Reporting Clocks

To report the structure of the synchronized clock groups as they are used by ATPG, use the command `report_clocks intclocks`. If any synchronization groups are active, two extra columns are printed with the headings `sync` and `period`. The example STL procedure file shown in “ClockTiming Block Example” uses `ClockTiming CTiming_2`, and looks like the following:

```
int_clock_inst_name gate_id off source sync period cycle conditions
-----
TOTO/U2 0 1468=1 (0,4) 895 0 20 1 1
... one line for each extra pulse condition ...
TOTO/U5 0 1487=1 (0,4) 825 0 19 1 2
... one line for each extra pulse condition ...
```

```
TOTO/U8          755          0          18          1          4
      0          1506=1 (0,4)
... one line for each extra pulse condition ...
```

The `sync` heading indicates the synchronization group number. The number is arbitrary, but all internal clocks that are synchronized to each other are in the same synchronization group.

The `period` heading indicates the period of the clock in units of the fastest clock in the same synchronization group. They are normalized to 1 since the actual period is not used by ATPG, only the relationships between the different periods. Note that each synchronization group will have a clock with a period of one. This does not mean that their periods are the same, since the different groups are asynchronous to each other.

To get clock pulse overlapping information, use the `report_clocks -capture_matrix` command. The output from this command takes one of two forms. The default form is as follows:

```
report_clocks -capture_matrix
Warning: Requested report contained no entries. (M13)
```

This means that overlapping is not allowed between any clock pairs. This would be expected in the example STL procedure file (see “[ClockTiming Block Example](#)”) if `set_drc -internal_clock_timing CTiming_2` was used because of the lack of `Waveform` and `MultiCyclePath` statements. The non-default form is as follows:

```
report_clocks -capture_matrix
id#  clock_gate  period  0  1
---  -
0    895         10.0  10.0  10.0
1    825         30.0  10.0  30.0
```

This means that clock pulse overlapping is allowed. All numbers in the matrix are the time between the launch and capture pulses when this pair of clocks is used. In this example, captures between any pairs of clocks can be made at the minimum of the two clocks’ periods, or in other words, at single-cycle timing.

The timing of the periods and edges of the internal clocks is reported by using the command `report_clocks -intclocks -verbose`. For example:

```
report_clocks -intclocks -verbose
#int_clk_inst_nm  gt_id  off  source  sync  period  LE  TE  lat  cycle
conditions
#-----
#pll_control_M1/U2 6698  0    138      1      1      1      0  10
5  0    13337=1 (0,4)
#
#                                     1      13336=1 (0,5)
#                                     13      13324=1 (0,17)
```

```

#pll_control_M2/U2 7347 0 139 1 1 0 10
5 0 13362=1 (0,4)
#
# 1 13361=1 (0,5)
#
# 13 13349=1 (0,17)
#pll_control_M3/U2 8567 0 187 1 2 5 25
5 0 13387=1 (0,4)
#
# 1 13386=1 (0,5)
#
# 13 13374=1 (0,17)

```

Note that the leading/trailing edge information comes from the STL procedure file. Here is the block that produced the example report:

```

SynchronizedClocks M_clocks {
  Clock ICLK1 {Location "pll_controller_M1/U2/Y"; Period '20ns';
Waveform '0ns' '10ns';
}
  Clock ICLK2 {Location "pll_controller_M2/U2/Y"; Period '20ns';
Waveform '0ns' '10ns';
}
  Clock ICLK3 {Location "pll_controller_M3/U2/Y"; Period '40ns';
Waveform '5ns' '25ns';
}
}

```

Reporting Patterns

The `report_patterns` command is useful for finding out the intention of ATPG, but the report can be too verbose when only the clocking information is required. To get a report that is tightly focused on the clocking, use the command `report_patterns -clocking`. For example:

```

TEST-T> report_patterns 7 -clocking
Clocking only:
Pattern 7 (fast_sequential-parallel_clocking)
Cycle-based clocking sequence:
0: TOTO/U2/Z:0100000000
1: TOTO/U5/Z:1-0-0-0-0-
Clock Instruction Registers:
0: 0010000000
1: 1000000000
# PLL internal clock pulse: capture_cycle=0, node=TOTO/U5 (191)
# PLL internal clock pulse: capture_cycle=1, node=TOTO/U2 (242)

```

The cycle-based clocking sequence field is the test in terms of ATPG frames and the Clock Instruction Registers field is the clock chain contents after re-sequencing. A dash is inserted to indicate that the clock operation is determined by a previous value

and its period has not finished yet. It allows columns representing the same time to line up even though they refer to clocks of different periods.

Using Internal Clocking Procedures

Internal clocking procedures enable you to specify which combinations of internal clock pulses you want to use and how to generate them.

The following sections describe how to use internal clocking procedures in TestMAX ATPG:

- [Enabling Internal Clocking Procedures](#)
- [Performing DRC with Internal Clocking Procedures](#)
- [Reporting Clocks](#)
- [Performing ATPG with Internal Clocking Procedures](#)
- [Grouping Patterns By ClockingProcedure Blocks](#)
- [Writing Patterns Grouped by Clocking Procedure](#)
- [Reporting Patterns](#)
- [Limitations](#)

Enabling Internal Clocking Procedures

To enable internal clocking procedures, you can use either the `ClockTiming` block or the `ClockConstraints` block within the top level of the `ClockStructures` block.

The `ClockTiming` block is used for synchronized multi frequency clocks. In many cases, you can use either the `ClockTiming` block or `ClockConstraints` block to describe synchronized OCC controllers. However, you should first consider using the `ClockTiming` block because it provides greater freedom to ATPG and results in fewer patterns for the same coverage. You should use the `ClockConstraints` block when the synchronized OCC controllers are limited to providing a small fixed set of clock waveforms.

Note that you cannot combine the `ClockConstraints` and `ClockTiming` blocks.

For complete details on enabling internal clocking procedures in the STL procedure file, see the [Specifying Internal Clocking Procedures](#) section.

Performing DRC with Internal Clocking Procedures

The presence of the `ClockConstraints` block in the STL procedure file disables some of the on-chip clocking checks normally performed during DRC. In particular, no checking is done to ensure that the specified `InstructionRegister` values cause the required clock pulses to be generated. In this case, the intention is to support clock controllers whose

behavior cannot be understood through zero-delay gate-level simulation. Clock effects from the defined clock pin name are simulated to ensure that capture behavior is valid. Clock-grouping checks are not performed.

You can define more than one named `ClockConstraints` block, but you can use only one for any single DRC or ATPG run. You must select the required `ClockConstraints` block using the `set_drc -clock_constraints` command, as shown in the following example:

```
set_drc -clock_constraints constraints1
```

If you do not specify the `set_drc -clock_constraints` command, none of the `ClockConstraints` blocks is used.

Timing information is not provided, which means clocks are assumed to be in the order specified. All clocks that pulse in the same frame are assumed to pulse simultaneously without disturbing each other. The trailing edges of all clock pulses in the first frame are assumed to occur before the leading edges of the clocks in the second frame. If these assumptions are violated in the actual design, timing exceptions must be used to prevent simulation mismatches.

You can use the `set_drc -num_pll_cycles` command to specify the sequential depth of the constraints. Procedures with a small number of frames are padded with clock-off values. Procedures with a large number of frames are degenerated if all of the extra frames are at clock-off values; otherwise, they are unusable. This enables the definition of multiple constraints of different depth in a single `Constraints` block while ensuring that only the procedures of the appropriate depth are used. The `set_drc -num_pll_cycles` and `set_atpg -capture` commands must match, but they can differ from the `PLLCycles` declaration in the `ClockStructures` block. The commands specify the sequential depth to be used in this particular run, while the `PLLCycles` declaration indicates the maximum sequential depth supported by the clock controller.

Reporting Clocks

You can use the `-constraints` option of the `report_clocks` command to report information on clocking procedures as they are used by ATPG. To report details for a given procedure, use the `report_clocks -constraints -procedure name` command. To report more detail for all procedures, use the `report_clocks -constraints -all` command.

For example,

```
TEST> report_clocks -constraints -all
-----
Clock Constraints constraints1:
  Maximum sequential depth: 2
  Defined Clocking Procedures: 3
  Usable Clocking Procedures: 3
  PLL clocks off Procedure: ClockOff
```

```
U0to1:
  CLKIR=10010
  dutm/ctrl1/U17/Z=P0
  dutm/ctrl2/U19/Z=0P
-----
U1to0:
  CLKIR=01010
  dutm/ctrl1/U17/Z=0P
  dutm/ctrl2/U19/Z=P0
-----
ClockOff:
  CLKIR=00000
  dutm/ctrl1/U17/Z=00
  dutm/ctrl2/U19/Z=00
-----
```

When procedures with different frame counts are reported, the shorter procedures are shown with zeros padded to the left so that all procedures are reported with the same depth. This does not mean that the procedures should be written this way. ATPG is more efficient when all procedures are written with as few frames as possible.

Performing ATPG with Internal Clocking Procedures

The internal clocking procedures feature fully supports two-clock optimized ATPG, basic scan ATPG, and fast-sequential ATPG. Full-sequential ATPG is not supported and no patterns are generated when internal clocking procedures are defined.

When two-clock optimized ATPG is used, all usable clocking procedures must have two frames for each clock. When basic scan ATPG is used, all usable clocking procedures must have one frame for each clock.

As a result of using internal clocking procedures, ATPG can use only a subset of the available clock pulse sequences. The sequences cannot be used to force ATPG to generate a pattern that it could not otherwise generate.

When a procedure has multiple clocks and multiple frames, ATPG can only capture transition or fault effects using clocks that pulse in the last frame. Clocks whose last pulse is in a preceding frame can only be used to launch transitions or set up conditioning to detect faults captured by other clocks. Make sure you provide other procedures where these clocks pulse in the last frame; otherwise, fault coverage is reduced.

Grouping Patterns By ClockingProcedure Blocks

In some situations, you might want to group patterns into sets, each of which uses only one of the defined `ClockingProcedure` blocks. To group patterns, specify the following command before the `run_atpg` command:

```
set_atpg -group_clk_constraints { first_pass middle_pass final_pass }
```

The three arguments are specified in terms of percentages of the fault list. These numbers are specified just one time, but they are applied for each individual clocking procedure. ATPG categorizes each fault by the clocking procedures that can test it; it considers only the appropriate subset as it generates tests for each clocking procedure.

The `first_pass` specification is the percentage of the fault list that is targeted in the first pass through each clocking procedure. The first pass results in long blocks of patterns with just one clocking procedure.

The `middle_pass` specification is the percentage of the fault list that is targeted by subsequent passes through each clocking procedure. These passes are repeated until the `final_pass` number is reached. The middle passes result in shorter blocks of patterns with just one clocking procedure.

The `final_pass` specification is the percentage of the fault list targeted by the final pass in which any clocking procedure is used. In this pass, there is no guarantee that any two consecutive patterns share the same clocking procedure.

Forcing a Single Group Per Clocking Procedure

The following example forces a single group for each clocking procedure with no exceptions:

```
set_atpg -group_clk_constraints { 100 0 0 }
```

The drawback of this particular specification is that ATPG efficiency, both in runtime and in fault detections per pattern, decreases significantly after most of the fault list has been targeted. All faults that are detectable by the first clocking procedure must be targeted before moving on to the next clocking procedure, which results in a larger pattern count than if other arguments are chosen.

Enabling ATPG to Achieve Better Efficiency

You can define a set of numbers that allow ATPG to achieve better efficiency and results in a lower overall pattern count, as shown in the following example:

```
set_atpg -group_clk_constraints { 85 5 2 }
```

This command creates a set of pattern groups by clocking procedure:

```
ClockingProcedure_1 (0-85%)  
ClockingProcedure_2 (0-85%)  
...  
ClockingProcedure_N (0-85%)  
ClockingProcedure_1 (85-90%)  
ClockingProcedure_2 (85-90%)  
...  
ClockingProcedure_N (85-90%)  
ClockingProcedure_1 (90-95%)  
ClockingProcedure_2 (90-95%)  
...  
...
```



```
ClockingProcedure_N (90-95%)  
ClockingProcedure_1 (95-98%)  
ClockingProcedure_2 (95-98%)  
...  
ClockingProcedure_N (95-98%)  
Mixed ClockingProcedure's (98-100%)
```

The drawback to this approach is that the grouping is less strict.

Writing Patterns Grouped by Clocking Procedure

By default, the `write_patterns` command saves all patterns into a single pattern file. You can use the `write_patterns -occ_load_split` command to split patterns into a separate file for each clocking procedure. This command is compatible with all pattern formats.

When patterns are grouped using the command `set_atpg -group_clk_constraints { 100 0 0 }`, only one pattern file is saved for each clocking procedure. If clocking procedures are grouped less strictly, or are not grouped at all, more pattern files are saved. A new pattern file is saved each time the clocking procedure changes from one pattern to the next, which can result in a large number of pattern files. Because of this, you should use the `write_patterns -occ_load_split` command only in combination with the `set_atpg -group_clk_constraints` command.

Reporting Patterns

You can use the `report_patterns -clocking` command to find out which clocking procedure is used in each capture cycle. For example,

```
TEST> report_patterns 7 -clocking  
Clocking only:  
Pattern 7 (fast_sequential)  
Clocking Procedures: U0to1  
// PLL internal clock pulse: capture_cycle=0, node=dutm/ctrl1/U17 (64)  
// PLL internal clock pulse: capture_cycle=1, node=dutm/ctrl2/U19 (94)
```

To get a summary of the number of clocking procedures of each type that was used in the pattern set, specify the `report_patterns -clk_summary` command:

```
TEST> report_patterns -all -clk_summary  
Pattern Clocking Constraints Summary Report  
-----  
#Used Clocking Procedures  
#U0to1 6  
#U1to0 5  
-----
```

Limitations

The following limitations apply when using internal clocking procedures in TestMAX ATPG:

- TestMAX ATPG DRC does not perform checking to ensure that the specified `InstructionRegister` values cause the generation of the required clock pulses.
- TestMAX ATPG DRC does not perform clock-grouping checks, and accepts all clock pulses specified in the same frame as simultaneous pulses without disturbing each other.
- TestMAX ATPG assumes that the trailing edges of all clock pulses in one frame occur before the leading edges of the clocks in the next frame. It is not possible to specify overlapping clock pulses.
- Full-sequential ATPG is not supported because it can generate bad patterns.
- When a procedure has multiple clocks and multiple frames, TestMAX ATPG can only capture transition or fault effects using clocks that pulse in the last frame. Clocks with a last pulse in a preceding frame can only be used to launch transitions or set up conditioning to detect faults captured by other clocks.
- Only single-load patterns are supported. You do not need to explicitly disable the generation of multi load patterns because TestMAX ATPG will not attempt to generate them.
- The grouping performed by the `-group_clk_constraints` option of the `set_atpg` command does not apply to fast sequential patterns. It applies to two-clock-optimized transition delay patterns and basic scan patterns for stuck-at.

See Also

- [Specifying Internal Clocking Procedures](#)

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
- C29 - Undefined PLL source for internal clock
- C30 - Scan PLL conditioning affected by nonscancells
- C31 - Scan PLL conditioning not stable during capture
- 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 scancell
- C40 - Internal clock is restricted

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 TestMAX ATPG 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 TestMAX DFT, or you can create one manually as described in [STIL Procedure Files](#).

17

TestMAX Diagnosis

The presence of defects in silicon has a direct impact on yield ramp during the manufacturing process. When a device fails testing, TestMAX Diagnosis can quickly isolate the cause and location of the failure, and simplify the analysis process.

TestMAX Diagnosis determines the root cause of failures observed during the testing of a chip. The data obtained from diagnostics identifies cells and scan chains with defects, and any logic with defects.

Diagnosis is applied to a variety of scenarios, including isolating ATPG pattern simulation failures, debugging first silicon, product and yield ramp, volume diagnostics, physical failure analysis, and field returns.

The following sections describe the process for applying TestMAX Diagnosis to manufacturing test failures:

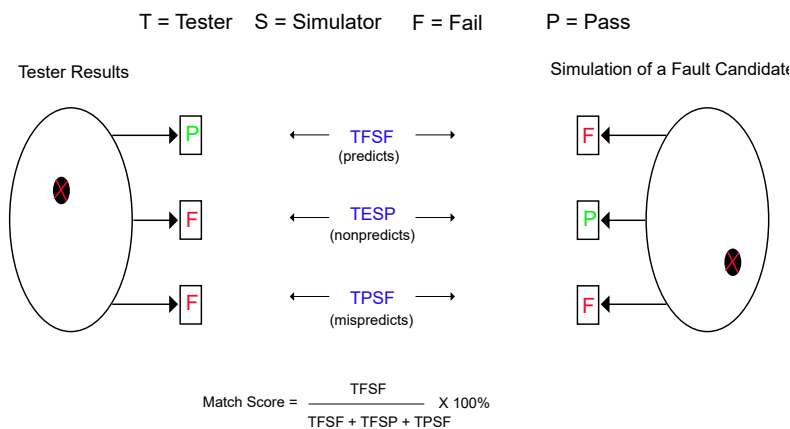
- [Understanding Diagnosis](#)
- [Running Diagnosis](#)
- [Running the TestMAX Diagnosis Flow](#)
- [Writing and Reading Binary Image Files](#)
- [Reading Pattern Files](#)
- [Failure Data Files](#)
- [Class-Based Diagnosis Reporting](#)
- [Fault-Based Diagnosis Reporting](#)
- [Using a Dictionary for Diagnosis](#)
- [Failure Mapping Report for DFTMAX Patterns](#)
- [Composite Fault Model Data Report](#)
- [Parallel Diagnosis](#)

Understanding Diagnosis

Scan diagnostics assume that there are many ATPG patterns with ATE failures. Since most faults produce a unique test response signature, diagnostics find the fault which most closely matches the defect signature from the ATE. This analysis results in the *match score*.

A match score reflects how well a reported fault matches the failure log file from the tester. Each candidate is assigned a score between 0% and 100%. A higher match score indicates higher confidence that the fault candidate location and behavior is the same as the defect.

The following figure shows how a match score is computed by mapping a simulated pattern to the tester failure log.



The following figure shows how a ma

Diagnosis Reporting

Diagnosis report the defect location, behavior, and match score for a given fault candidate.

The defect location depends on the provided input:

- When only logical information is provided, instance names, pin names, and connected nets are reported.
- When additional physical information is provided, the instance and pin location (X/Y), metal layers, and bridge and net polygons are reported.

Defect behavior types include stuck, transition, bridge, open, and cell-aware faults.

A match score reflects how well a reported fault matches the failure log file from the tester. Three components are computes for the failure of each fault candidate.

Defects are simulated, compared with tester results, and assigned a score based on how well they match. The best candidates include the match scores. Each candidate includes the scan cell and the connecting pins.

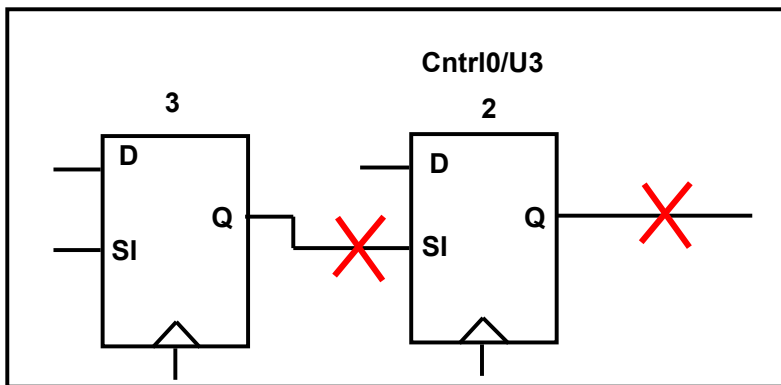
The following example for chain diagnostics reports a fast-to-rise defect type with a 100% match score:

```
sim/fail.log scan chain diagnosis results: #failing_patterns=5
-----
defect type=fast-to-rise
match=100% chain=chn0 position=2 master=cntrl0/U3 (17)
-----
```

Note in the example:

- The chain name is `chn0`
- The position (2) indicates the scan cell location in a given chain (0 is the cell closest to the scanout pin)
- The master indicates the scan cell instance name (`cntrl0/U3`)

The following schematic shows the defect.



Running Diagnosis

To start the diagnostics process, you can use either the Run Diagnosis dialog box in the TestMAX ATPG GUI or the specify the `run_diagnosis` command at the command line.

Using the Run Diagnosis Dialog Box

To start the diagnosis using the Run Diagnosis dialog box, perform the following steps:

1. Click the Diagnosis button in the command toolbar at the top of the TestMAX ATPG main window. The Run Diagnosis dialog box appears.

2. Fill in the dialog box.

For descriptions of these controls, see TestMAX ATPG Help for the `run_diagnosis` (and `set_diagnosis`) command(s).

3. Click OK.

Using the `run_diagnosis` Command

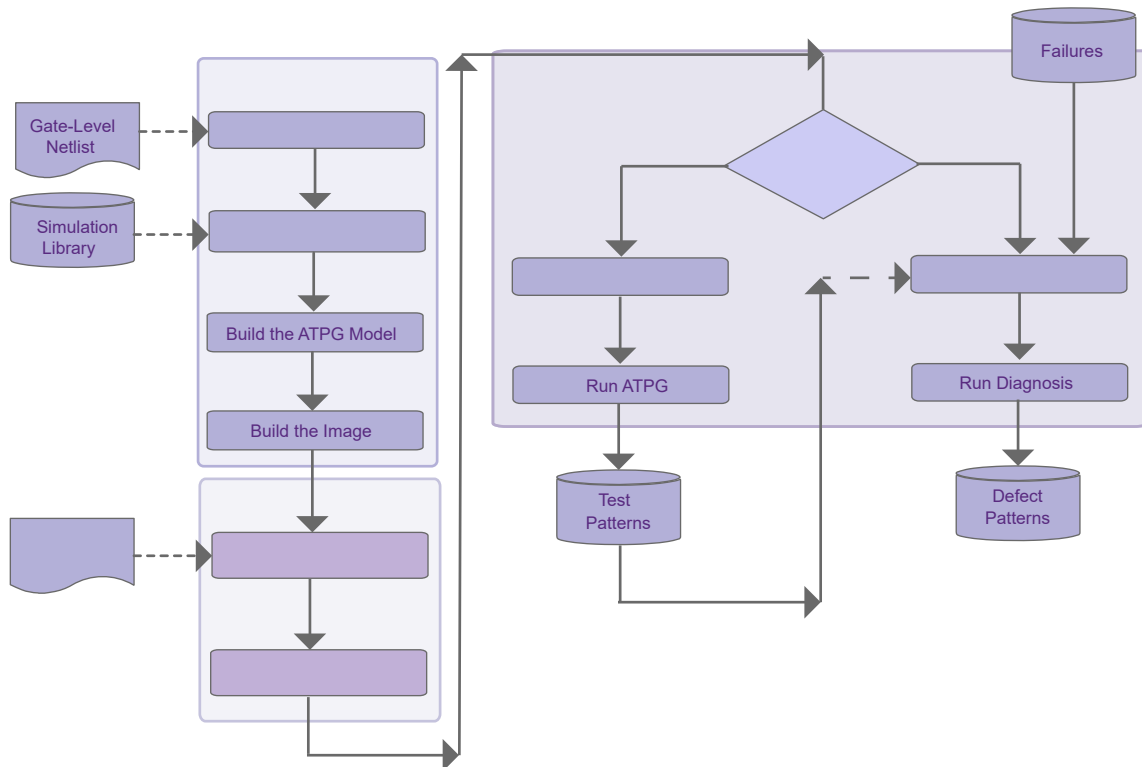
You can start the diagnosis at the command line using the `run_diagnosis` command, as shown in the following example:

```
TEST-T> run_diagnosis chipA_failure.dat -display
```

Running the TestMAX Diagnosis Flow

TestMAX Diagnosis determines the root cause of failures observed during the testing of a chip. The data obtained from diagnostics identifies cells and scan chains with defects, and any logic with defects.

Diagnosis is applied to a variety of scenarios, including isolating ATPG pattern simulation failures, debugging first silicon, product and yield ramp, volume diagnostics, physical failure analysis, and field returns.



To run TestMAX Diagnosis

1. Establish the original TestMAX ATPG environment used for creating the patterns, including reading the design netlist, reading the library model, running DRC, and running ATPG. For details on preparing for and running ATPG, see the [ATPG Design Flow](#) section.
2. Write the image to a binary file using the `write_image` command, as shown in the following example:

```
write_image image.gz -compress gzip
```

This is an optional step. You can run diagnosis using the original netlist, library model, and rerun DRC. However, a binary image file contains all this information in single file and eliminates the need to rerun the ATPG process before each diagnosis run. For more information on binary image files, see the [Writing and Reading Binary Image Files](#) section.

3. Connect to the PHDS database (optional)

For more information on creating and validating a PHDS database, see the [Creating and Validating a PHDS Database](#) section.

4. 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. For more information on using pattern files, see [Reading Pattern Files](#).
5. Obtain the failure data file produced from the ATE. For more information, see [Failure Data Files](#).
6. Use the `set_diagnosis` command to specify parameters for the diagnostics run. For example, you might want to [perform scan diagnostics](#) or [perform parallel diagnostics](#).
7. Read the failure data file and start diagnostics using the Run Diagnosis dialog box in the TestMAX GUI or specify the `run_diagnosis` command at the command line.
8. Analyze the results in the diagnostics summary report.

TestMAX ATPG determines the cause of the failing patterns and generates the diagnostics report. By default, TestMAX Diagnosis searches for defects in functional logic. If pattern 0 is the chain test pattern and it fails, then chain diagnostics is performed. For more information, see either the [Class-Based Diagnosis Reporting](#) section or the [Fault-Based Diagnosis Reporting](#) section.

Note the following:

- The `run_diagnosis` command uses the TestMAX ATPG threaded simulator for good machine simulation only. All other diagnostics operations use a single process.
- TestMAX ATPG simulation is not used for good machine simulation of chain defects. If a failure log includes failing chain test patterns, TestMAX ATPG simulation is not used.
- You should avoid using patterns generated from TestMAX ATPG with single-process diagnostics. Otherwise, M266 warning messages, indicating that failures were ignored due to X measures, might be printed and diagnostics might be less accurate.
- DFTMAX compression is supported for both logic and scan chain diagnosis. Serialized DFTMAX compression is also supported. Each individual failure is mapped to a scan cell. To produce a detailed failure mapping report, use the `-mapping_report` option of the `set_diagnosis` command. Always review the mapping report for accuracy.
- If a large set of failures cannot be mapped using DFTMAX patterns, you can create patterns that bypass the output compressor. For more information on this process, see the [Translating DFTMAX Patterns Into Normal Scan Patterns](#) section.
- By default, a defect is not linked to the type of fault tested. This means, for example, that any failures collected while running transition patterns could include stuck-at faults. However, you can use the `-delay_type` option of the `set_diagnosis` command to cause the diagnostics report to include delay defects.

Script Example

```
# Read the image used to generate
# the pattern for diagnostics:
read_image Results/design1_img.gz
# Use the class-based candidate
# organization in the diagnosis report.
set_diagnosis -organization class
set_diagnosis -fault_type all
# Multithread settings
set_atpg -num_threads 8
set_simulation -num_threads 8
# Set the pattern file, using the
# binary pattern set if available, but
# only to diagnose pattern-based failure
# data. STIL (or WGL) patterns are
# required to diagnose cycle-based failure
# data, and may be used to diagnose
# pattern-based failure data.
set_patterns -external patterns.bin
# Perform diagnosis on the failure file
# and generate data for Yield Explorer.
# Multiple runs using different failure files
# with the same pattern may be
# run in sequence:
run_diagnosis datalogs/fail1.log
write_ydf results1/design1_diagnosis.ydf -replace
run_diagnosis datalogs/fail2.log
write_ydf results2}/design1_diagnosis.ydf -append
run_diagnosis datalogs/fail3.log
write_ydf results3/design1_diagnosis.ydf -append
```

Writing and Reading Binary Image Files

A binary image offers many advantages when running diagnosis. It simplifies file management because you use only a single file that encapsulates the design netlist, library, and SPF. It is also faster to load, and can be password-protected. You can also garble the instance names in an image file, if necessary.

Prior to creating an image file, you need to read the design netlist and library model, run DRC, and run ATPG (see the [ATPG Design Flow](#) section for details). You then use the

`write_image` command to write a binary image that contains the netlist, library, SPF, and DRC data. During the diagnosis process, you can use the `read_image` command to read the image file as many times as necessary without rerunning the ATPG flow.

The following example shows how to write and read a binary image file for diagnosis:

```
// First time through (ATPG flow)
BUILD> read_netlist top.v
```

```
BUILD> read_netlist spec_lib.v -library
BUILD> run_build_model spec_chip
DRC> run_drc spec_chip.spf
TEST> write_image spec_image.gz -compress gzip

// For subsequent runs during diagnosis
BUILD> read_image spec_image.gz
```

For more information on writing and reading secure binary images, see [Binary Image Files](#).

Reading Pattern Files

The TestMAX Diagnosis process requires either a single pattern file corresponding to the patterns that were run on the tester when the device failed or an entire set of split patterns files, including the associated failure files.

The binary pattern format is optimal for diagnosis. STIL or WGL patterns also work, however if you use these patterns, the fast-sequential patterns might be interpreted as a full-sequential patterns, and errors are reported.

The following sections describe how to read and use pattern files for diagnostics:

- [Reading Patterns](#)
- [Reading Multiple Pattern Files](#)
- [Translating DFTMAX Compressed Patterns Into Normal Scan Patterns](#)

See Also

- [Writing ATPG Patterns](#)

Reading Patterns

TestMAX Diagnosis accepts either basic-scan or fast-sequential ATPG patterns. When reading patterns into TestMAX ATPG, use binary formats whenever possible.

You can perform a sanity check to verify that the simulation passes with the patterns you read in by running the `run_simulation` command before performing diagnostics.

Use the `set_patterns` command to read a set of patterns, as shown in the following example:

```
set_patterns -external patterns.bin
```

For details on reading patterns, see "[Selecting the Pattern Source](#)."

See Also

- [Using Split Datalogs to Perform Parallel Diagnosis for Split Patterns](#)

Reading Multiple Pattern Files

Some designs require the ATE to run multiple TestMAX ATPG pattern files. 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 TestMAX ATPG pattern file.

TestMAX Diagnosis can read multiple pattern files and multiple failure data files so you can get a single result from a single diagnosis run. This is supported for DFTMAX compression.

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 patterns in a particular pattern file, the corresponding failure log file might not exist. The correspondence between pattern files and failure log files is specified by a required directive in the failure log file, as explained in the [Failure Data Files](#) section. An error is generated otherwise.

To use multiple patterns files, specify the following `set_patterns` command:

```
set_patterns -external file -split_patterns
```

When split pattern files are read, you can specify multiple failure log files using the `run_diagnosis` command.

By default, TestMAX Diagnosis considers that the cycle count recorded in the failures file in cycle-based format is reset to 1 (or the recorded pattern count is reset to 0 for the pattern-based format) from the execution of one pattern set to the next set. You can use the `.first_pattern` directive to change this behavior if the failures are in the pattern-based format.

See Also

- [Using Split Datalogs to Perform Parallel Diagnosis for Split Patterns](#)

Translating DFTMAX Compressed Patterns Into Normal Scan Patterns

If your design uses DFTMAX compression, you can perform diagnostics 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 scan mode. The translated patterns detect the same defects, but diagnostic resolution is higher because the compressor no longer affects the unloaded values.

The translation process involves writing out a special netlist-independent version of the pattern in binary format along with the DFTMAX 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 TestMAX ATPG after a design is put into the reconfigured scan mode by reading the scan mode STL procedure file. When the patterns are read back, an internal simulation is performed to compute the expected values to complete the translation process. The internal patterns can then be written out for the tester to use in scan mode for diagnostics.

Example Flow

To translate DFTMAX compressed patterns to normal scan patterns:

1. Read the design with DFTMAX in compressor mode and write out the netlist-independent pattern format.

```
run_build_model ...  
  
# read STL procedure file 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_patterns -netlist_independent  
write_patterns compressed_pat.net_ind.bin
```

2. Read the design with DFTMAX compression in scan mode. Translate the patterns into scan mode.

```
run_build_model ...  
  
# read STL procedure file for normal scan mode  
run_drc scan.spf  
  
# read netlist independent patterns  
set_patterns -external 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
```

Translation Limitations

The following limitations apply when translating DFTMAX compression patterns to normal scan mode patterns:

- Translation is one-way. You cannot translate scan patterns to compression mode.
- Only basic-scan and fast-sequential patterns are supported .
- Configuration differences between compressor mode and scan mode might result in slightly different coverage numbers.

For more information, see the *DFTMAX User Guide*.

Failure Data Files

A failure data (or log) file is an ASCII text file that provides the failure information from a device necessary to perform diagnostics. This file captures the test results of a failing device, including failing patterns and failing outputs and scan cells. Most ATE vendors automatically generate failure data files in a format recognizable by TestMAX ATPG.

When testing a chip, if the value measured by the ATE is different than the expected value indicated in the patterns file, a failure is recorded in the failure data file. Each recorded failure includes the vector number, the output port where the mismatch occurs, the cycle number within the mismatched vector, and optional expected data.

TestMAX ATPG supports either a pattern-based or cycle-based failure data file.

The following sections describe failure data files:

- [Pattern-Based Failure Data File](#)
- [Cycle-Based Failure Data File](#)
- [Failure Data File Extensions](#)

- [Adding Header Information to a Failure Data File](#)
- [Failure Data File Limitations](#)

Pattern-Based Failure Data File

Each line in a failure data file describes a pattern in which the output values detected by the test equipment did not match the expected values. An example is as follows:

```
// Pattern Output Cell
50      vout    55
50      abus    57
58      vout    57
82      xstrb
82      vout    57
82      vout     5
83      abus    90
```

The format of each line is as follows:

```
pattern_num output_port [cell_position] [expected_data]
```

Where:

pattern_num

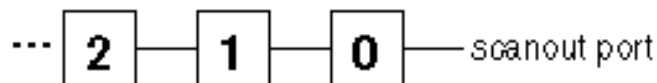
The TestMAX ATPG pattern number in which the failure occurred, starting with 0 for the first pattern.

output_port

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

The cell position must be provided if the failure occurred during a scan shift cycle. The position is the number of tester shift cycles that have occurred since the start of the scanout process. From this value, TestMAX ATPG 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`

This is an optional parameter that describes the expected value of the measured failure. The expected value is either 0 or 1. By default, if this parameter is present, it is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

To specify the expected data on a primary output during a capture cycle, which is also used as a scan chain output, you must use the `exp= syntax` to avoid any ambiguity with this being a cell position. 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 data file that begins with two slash characters is considered a comment line.

The following example shows another tester data file. In this example, five failing patterns are reported: pattern numbers 3, 4, 10, 11, and 12.

```
// 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
```

The `-failure_memory_limit` option of the `set_diagnosis` command 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 also enables TestMAX ATPG to automatically truncate the patterns considered during diagnosis.

Pattern-Based Failure Data File for DFTMAX Serialized Adaptive Scan

The format for DFTMAX serialized adaptive scan technology is similar to the regular format except that it includes additional piece of information that identifies the bit of a serialized bitstream containing the failure (*bit_position*). The pattern-based format of the failure data file is as follows:

```
pat_num output_ports cell_pos bit_pos [expected_data]
```

Where:

pat_num

The TestMAX ATPG pattern number on which the failure occurred. The first pattern is 0.

output_ports

The name of the output port on which the failure was detected.

cell_pos

This is the position of the scan chain cell that captured the data that was in error. The cell position of the scan chain cell closest to the output port is 0, the next one in is 1, and so on.

bit_pos

For each scan chain shift cycles, the serializer is capturing the parallel output of the output compressor. Then, this information is serialized and shift out on the scanout pin. The *bit_position* is the bit of the serialized bitstream where there is a failure. The first *bit_position* is 0 and it corresponds to serializer bit close to the scan out.

expected_data

This optional parameter is a 0 or 1 depending on the expected value specified by the pattern. By default, this parameter is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

Cycle-Based Failure Data File

Most ATE vendors generate failure data directly in the pattern-based failure data file format. However, some testers do not support this format. For the unsupported testers, you can use a cycle-based (or vector-based) failure data file format (TestMAX ATPG

patterns contain multiple cycles or vectors). Cycle-based failure log files in TestMAX ATPG format are easier to generate than pattern-based failure log files.

TestMAX ATPG supports both basic-scan and fast-sequential patterns in STIL or WGL format. The binary pattern format and the WGL flat format are not supported for cycle-based failure log file diagnostics.

If the external pattern buffer contains an unsupported pattern format, TestMAX ATPG displays an error message when you execute the `run_diagnosis` command with a cycle-based failure log file.

Cycles (V statements in STIL format or vector statements in WGL format) are counted when you read patterns using the `set_patterns external` command. This count identifies the vectors at pattern boundaries, and the time 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 following `set_diagnosis` options are associated with the cycle-based failure log file:

- `-cycle_offset integer` — You can use this option to adjust the cycle count when the cycle numbering does not start at 1.
- `-show cycles` — This option causes the translated pattern-based failure log file to be reported by the `run_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 set of commands show an example flow:

```
run_drc ...
set_patterns -external pat.stil
set_diagnosis -cycle_offset 1
run_diagnosis fail.log
```

Cycle-Based Failure Data File Format

A failure data file contains only failed cycles. The format of each line is as follows:

```
C output_name cycle [expected_value]
```

Where:

C

The first character on the line, indicating that the line specifies tester cycles and not TestMAX ATPG pattern numbers. It helps you identify the type of failure log file (pattern- or cycle-based).

`output_name`

A string that can be a PO or a scan chain name (no output compressor).

`cycle`

An integer that identifies the cycle in the external pattern set that failed on the tester. If the first cycle is numbered 0, use the `set_diagnosis -cycle_offset 1` command to make an adjustment. TestMAX ATPG expects failures only in cycles in which measurements occur (for example, during shift or capture cycles). Invalid failure cycles can provide inaccurate diagnostics results.

`expected_value`

This optional parameter is a 0 or 1 depending on the expected value specified by the pattern. By default, this parameter is checked against the expected data recorded in the patterns. If this step succeeds, it is a good indicator that the files used for diagnostics and on the tester are consistent. To change the default, use the `-nocheck_expected_data` option of the `set_diagnosis` command.

TestMAX ATPG ignores all other characters in the line, and treats them as comments.

Failure Data File Extensions

The failure data file can contain the directives to specify settings specific to that failure log file. These directives must be at the top of the failure data file before any failure data. The directives cannot be abbreviated. Any other line in the failure log file is interpreted as failure data.

`.pattern_file_name string`

This directive is required when you are using the split patterns feature. It specifies the name of the corresponding pattern file to associate the failure log files to the pattern file. If there are no failures in the patterns corresponding to a pattern file, this directive is used to make correspondences between pattern and failure log files. This name is assumed to be just the file name, without the directory hierarchy.

If the failure log file does not contain this directive, or if the name does not match, diagnosis is aborted.

`.attr_file_name string`

This directive can be used to set user-defined attributes for a particular failure log file; you can then access the specified *string* value using the Tcl API. For example, the attribute could describe the ATE clock frequency or the pattern type used for testing the chip. You could then retrieve the *string* using the attribute `<attr_file_name>` returned by the `get_diag_files` Tcl API command. If the name of the directive does not match, the diagnosis process is aborted.

`.cycle_offset <d | continue>`

This directive adjusts the cycle count when the cycle numbering does not start at 1 for cycle-based failure log files. The `d` parameter is an integer in tester cycles. The purpose of `continue` is for ease of use. The string argument `continue` indicates that the cycle count is not reset from the previous pattern set. The default is to reset the cycle count to 1. This directive only applies to split pattern diagnosis.

When used, this directive overrides the `-cycle_offset` option of the `set_diagnosis` command for this pair of pattern/failure log files. Normally, the cycle count is reset to 1 for every pattern set.

`.split_pattern_offset d`

Specifies the number of offset cycles for each failure log file. This directive is used for diagnosing cycle-based failures in split patterns. For more information on using the `.split_pattern_offset` directive, see the "Split Pattern Diagnosis for Cycle-Based Failures" topic in TestMAX ATPG Online Help.

`.truncate d`

All patterns numbered greater than `d` in this failure log file are ignored. The pattern numbers begin at 0 for each failure log file. The argument `d` specifies the last pattern for which complete failures were captured. It should not exceed the number of patterns in the corresponding pattern file.

When used, this directive overrides the `-truncate` option of the `run_diagnosis` command.

`.incomplete_failures`

Ignores patterns in the range beginning with the last failing pattern recorded in this failure log file, to the last pattern in the corresponding pattern file, unless there is only one failing pattern in this file.

When used, this directive overrides the `-incomplete_failures` option of the `set_diagnosis` command.

`.failure_memory_limit d`

Ignores patterns in the range beginning with the last failing pattern recorded in this failure log file, to the last pattern in the corresponding pattern file, if the number of failures in this file is at least `d`. The argument `d` is a decimal number that specifies the number of failures that the tester can capture.

When used, this directive overrides the `-failure_memory_limit` option of the `set_diagnosis` command.

Adding Header Information to a Failure Data File

You can insert a header section into a failure data file to include additional data from the ATE, such as key-value pairs information, the device name, job name, or truncation status. This information is passed to the output of the `write_ydf` command during physical diagnostics (for more information on physical diagnostics, see [Using Physical Data for Diagnosis](#)).

Not all information in the header section is passed to the Yield Explorer Data Format (YDF) file used for physical diagnostics. Only data described in a configuration file, called the *header schema file*, is retrieved when running diagnostics.

The following sections describe how to insert a header section into a failure data file:

- [Creating a Header Section](#)
- [Creating a Header Schema File](#)
- [Examples](#)

Creating a Header Section

You can place the header section in a failure log file either immediately before or after the `.pattern_file_name` directive. All key-value pairs in the header section are associated with the corresponding pattern file specified by this directive.

To start the header section, specify the `.header` keyword. To finish the header section, use the `.end_header` keyword. Each line in the header section is a key-value pair. A key is a single word separated by tab or space, and the value can be one or more words excluding the special symbols tab, “#” and “\”.

The following example shows a typical header section:

```
.pattern_file_name patterns.bin
.header
DEVICE TOPDUT1
LOT K382
WAFER 03
DIEX 112
DIEY 124
```

```
VDD_CORE 1.32
VDD_PAD 3.3
TEMP 0300
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
```

Note the following:

- After the header information is read by the `write_ydf` command, it is included in the DFTCandidates table in the YDF file. Each keyword constitutes a column with entries specified as strings.
- Only one header section is used for a set of failure log files associated with a single `run_diagnosis` command. When split patterns are used, the header section is defined only one time.
- The header section can be included in any failure log file.
- If duplicate value names are included in the header, the last defined value is used.
- If a custom field in the header matches a standard DFTCandidates Table field, the custom field is ignored. For example:

```
TEST-T> set_ydf schema.txt -schema
----- YDF
Schema Set Summary
-----
LOT used as a standard column in DFTCandidate segment ...
Skipping ... WAFER used as a standard column in DFTCandidate
segment ... Skipping ... DIEX used as a standard column in
DftCandidate segment ... Skipping ... DIEY used as a standard
column in DftCandidate segment ... Skipping ... YDF Schema has
been set for 6 keywords. CPU_time: 0.00 sec Memory Usage: 0MB
-----
```

- You can use the `set_diagnosis -show key_value_pairs` command to print the values from the header section to the diagnostics report.

Creating a Header Schema File

The header schema file defines the custom columns that are included in the YDF file. The schema file specifies the keywords and their respective string argument field sizes. If

the size is not specified for a keyword, a default string size of 256 is used. The following example is a typical schema file:

```
DEVICE 128
LOT 256
WAFER 128
DIEX 64
DIEY 64
... ..
... ..
... ..
```

After you create the header schema file, you define it using the `-schema` option of the `set_ydf` command, as shown in the following example:

```
set_ydf header_schema_file -schema
```

You need to set the header schema file only once. All successive diagnostics results appended to the same YDF file adhere to the original specified header schema file. When performing an append operation on an existing YDF file, TestMAX ATPG retrieves the header schema from the YDF file and fills in the appropriate values for the keywords from the failure log file.

However, if you update the diagnosis results for a YDF file to a new file, you must define a new header schema file using the `set_ydf` command. If a new schema is not specified, TestMAX ATPG uses the header schema file specified earlier in the session. If a schema file is not specified, TestMAX ATPG does not write the custom columns.

The script in [Example C: Flow for Handling Custom Columns in the YDF File](#) implements the custom columns in the YDF file during diagnosis.

If a value name is duplicated in the schema file, an error is issued when the `write_ydf` command is executed.

Examples

The examples in this section include the following cases:

- [Example A: Header Schema File for Split Pattern Set With Two Pattern Files](#)
- [Example B: Header Schema File for Split Pattern Set With Three Pattern Files](#)
- [Example C: Flow for Handling Custom Columns in the YDF File](#)

Example A: Header Schema File for Split Pattern Set With Two Pattern Files

In this example, the 15 key-value pairs are defined in the header section and passed to the `write_ydf` command using one list of 15 string pairs.

```
.header
DEVICE 1604
LOT PL924
```

Chapter 17: TestMAX Diagnosis Failure Data Files

```

WAFER 03
DIEX 122
DIEY 122
VDD_CORE 1.32
VDD_PAD 3.3
V2_PAD N/A
TEMP 0300
JOB_NAM 1604_SW
JOB_REV 02
FLOOR_ID AF6E
FLOW_ID EWS1
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
.pattern_file_name pattern_file1.bin
6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
.pattern_file_name pattern_file2.bin
7977 PAD_34 1
7981 PAD_34 1
7985 PAD_34 1
7989 PAD_34 1

```

Example B: Header Schema File for Split Pattern Set With Three Pattern Files

In this example, TestMAX ATPG associates the header with all pattern files: `pattern_file1.bin`, `pattern_file2.bin`, and `pattern_file3.bin`. The header section associated with the second file includes eight key values.

```

.pattern_file_name pattern_file1.bin
6977 PAD_34 1
6981 PAD_34 1
6985 PAD_34 1
6989 PAD_34 1
...
.pattern_file_name pattern_file2.bin
.header
DEVICE 1604
LOT PL924
WAFER 03
DIEX 122
DIEY 122
VDD_CORE 1.32
START_T Nov 25 2015 18:40:10
TRUNCATE Y
.end_header
7977 PAD_34 1
7981 PAD_34 1
7985 PAD_34 1
7989 PAD_34 1

```



```
....  
.pattern_file_name pattern_file3.bin  
9977 PAD_34 1  
9981 PAD_34 1  
9985 PAD_34 1  
9989 PAD_34 1  
.....
```

Example C: Flow for Handling Custom Columns in the YDF File

```
TEST-T> set_messages -log example.log -replace  
TEST-T> set_command noabort  
TEST-T> read_image mydesign.phy.img.gz  
TEST-T> set_physical_db -hostname host01 -port_number 9998  
TEST-T> set_physical_db -top_design top_specdevice  
TEST-T> set_physical_db -device [list "specDevice" "1"]  
TEST-T> match_names -verify all  
TEST-T> run_drc mydesign_scan.spf  
TEST-T> set_patterns -external mydesign_pat.bin  
TEST-T> run_diagnosis sample1.ff  
TEST-T> set_ydf schemal.sch -schema  
TEST-T> set_diagnosis -show key_value_pairs  
  
// A new YDF file is created with the results of last  
// diagnostics run (note the -replace option).  
  
TEST-T> write_ydf mydesign-diag1.ydf -replace \  
-device TESTDEVICE -version 1 \  
-candidates -cell -instance_cell \  
-cell_instance_pin_net -net_path \  
-net_contact_position -net_layer  
TEST-T> run_diagnosis sample2.ff  
  
// The same YDF file is updated with the results of last  
// diagnostics run (note the -append option). The same header schema file  
// is used.  
  
TEST-T> write_ydf mydesign-diag1.ydf -append -candidates \  
-cell -instance_cell -cell_instance_pin_net \  
-net_path -net_contact_position -net_layer  
TEST-T> run_diagnosis sample3.ff  
TEST-T> set_ydf schema2.sch -schema  
  
// A new YDF file is created with the results of last  
// diagnostics run (note the -replace option). A new header schema file  
// is used.  
  
TEST-T> write_ydf mydesign-diag2.ydf -replace \  
-device TESTDEVICE -version 2 \  
-candidates -cell -instance_cell \  
-cell_instance_pin_net -net_path \  
-net_contact_position -net_layer  
TEST-T> run_diagnosis sample4.ff
```

```
// The same YDF file is updated with the results of last
// diagnostics run (note the -append option). The same header schema file
// is used.

TEST-T> write_ydf mydesign-diag1.ydf -append -candidates \
        -cell -instance_cell -cell_instance_pin_net \
        -net_path -net_contact_position -net_layer
TEST-T> exit
```

Failure Data File Limitations

The following limitations are associated with failure data files:

- Only STIL and WGL patterns with cycle-based failure data files are supported. Binary patterns are also supported with pattern-based failure files.
- The `-truncate` option of the `run_diagnosis` command is not supported with cycle-based diagnosis.
- WGL flat patterns are not supported for diagnostics.

Class-Based Diagnosis Reporting

The class-based diagnostics report includes cell, net, subnet, and bridgeable area physical data, and other information required for physical failure analysis (PFA). You can enable this report using the following command:

```
set_diagnosis -organization class
```

The following sections describe how to configure, create, and read a class-based report:

- [Filtering Candidates](#)
- [Filtering Bridge Candidates](#)
- [Resetting User-Specified Filters](#)
- [Reporting Detailed Candidate Information](#)
- [Example Flow](#)
- [Understanding the Class-Based Diagnosis Report](#)
- [Class-Based Cell-Aware Diagnosis](#)

Filtering Candidates

You can use the `-filter_candidates` option of the `set_diagnosis` command to apply several different types of filters when generating the class-based report.

If you use the `score_distance` parameter of the `-filter_candidates` option, you can exclude all candidates that are further than the specified percentage distance from the maximum candidate percentage match score:

```
set_diagnosis -organization class -filter_candidates \  
  {score_distance 10}
```

The default for the `score_distance` parameter is 20 percent.

You can also use the `min_score` parameter to filter candidates with a match score less than a specified minimum value (the default is 50). In this case, at least one candidate is reported even it is below the minimum value. You can also use the `max_number` parameter to limit the number of candidates reported for each defect. Since candidates are reported in order of decreasing scores, this effectively specifies to report the top *N* candidates.

In the following example, all candidates with a match score less than 10 and a score further than 20 are removed from the diagnostics report:

```
set_diagnosis -organization class -filter_candidates \  
  {min_score 75 max_number 20}
```

The number of filtered candidates are reported by an M804 message:

```
Warning: Filtered 2 candidates with match score outside the  
  score_distance of 20. (M804)
```

Filtering Bridge Candidates

An ambiguous bridge is any bridge candidate in which every explained pattern has the same value on the aggressor node. These bridge candidates are not distinguishable from a stuck candidate on the victim node. A same-cell bridge is any bridge candidate between two nodes that are connected to the same library cell. Such bridge candidates are not distinguishable from bridges inside the cell.

You can use the `-filter_bridges` option of the `set_diagnosis` command to remove bridge candidates exercised by either an ambiguous bridge or any bridge candidates between two nets connected to the same cell.

To remove bridge candidates with the same value on the aggressor node and that behave the same as a stuck candidate, use the `ambiguous` parameter:

```
set_diagnosis -filter_bridges {ambiguous on}
```

To remove bridge candidates between two nets connected to the same cell, use the

same_cell parameter:

```
set_diagnosis -filter_bridges {same_cell on}
```

Resetting User-Specified Filters

To reset any filters you specified using the `-filter_candidates` or `-filter_bridges` options, specify the `-reset_filters` option of the `set_diagnosis` command:

```
set_diagnosis -reset_filters
```

Note that all class-based filters must be specified before the `run_diagnosis` command to affect a candidate list reported by the `report_defects` or `write_ydf` command.

Reporting Detailed Candidate Information

You can obtain more detailed information about net connectivity and physical locations by specifying the `set_diagnosis -verbose` command, or report the same data separately using the `-candidates` option of the `report_nets` command or the `report_physical` command. For example, the `-candidates` option of the `report_nets` command returns the following net connectivity information:

```
TEST-T> report_nets -candidates all
Candidate 1:
  Net connections:
  -----
  I_RISC_CORE/I_ALU/n57 (695)
  O      I_RISC_CORE/I_ALU/U108/Z
  I      I_RISC_CORE/I_ALU/U90/A2
  I      I_RISC_CORE/I_ALU/U82/A2
  -----
  Net connections:
  -----
  I_RISC_CORE/I_ALU/n111 (889)
  O      I_RISC_CORE/I_ALU/U149/ZN
  I      I_RISC_CORE/I_ALU/U147/IO
  -----
```

The `-candidates` option of the `report_physical` command reports the following physical data:

```
TEST-T> report_physical -candidates all
Candidate 1:
  Physical details:
  -----
  ~ METAL3 (619530 622190) (623420 622400) TB
  -----
```

To view candidates for all defects or a particular defect after specifying the `run_diagnosis` command, use the `-defect` option of the `report_defects` command:

```
TEST-T> report_defects -defect all
Defect 1:
  #failing_patterns=5
  #observe_points=1
  #simulated_failures=5
  #candidates=1
  bbox=(619530 622190) (623420 622400), area=816900
Candidate 1:
  class=Bridge
  net1_id=695, net2_id=889
  driver1=I_RISC_CORE/I_ALU/U108, pin=Z
  driver2=I_RISC_CORE/I_ALU/U149, pin=ZN
  layers=METAL3
  bbox=(619530 622190) (623420 622400), area=816900
  behavior=bAND, match_score=100.00% (TFSF=5/TFSP=0/TPSF=0)
```

You can generate a fault list for high-resolution ATPG using the `-faults` option of the `report_defects` command:

```
TEST-T> report_defects -faults
ba0  NC  I_RISC_CORE/I_ALU/U108/Z  I_RISC_CORE/I_ALU/U149/ZN
ba0  NC  I_RISC_CORE/I_ALU/U149/ZN  I_RISC_CORE/I_ALU/U108/Z
```

Example Flow

A typical class-based diagnostics reporting flow is as follows:

```
read_image CHIP.img

set_patterns -external pat.bin

set_physical_db -database ./PHDS
set_physical_db -port_number 9998

open_physical_db

set_physical_db -hostname localhost
set_physical_db -device {"TOP" "1"}

set_diagnosis -organization class
set_diagnosis -filter_candidates {min_score 75}

run_diagnosis die123.tmx

report_physical -candidates all
set_ydf -version 1.2

write_ydf die123.ydf -replace
```

```
close_physical_db
```

Understanding the Class-Based Diagnosis Report

The class-based diagnostics report includes cell, net, subnet, and bridgeable area physical data, and information required for physical failure analysis (PFA). This data includes the defect area and the overall bounding box coordinates.

The header section of the report contains the number of tester failures, tester patterns, patterns simulated by diagnosis, and identified defects:

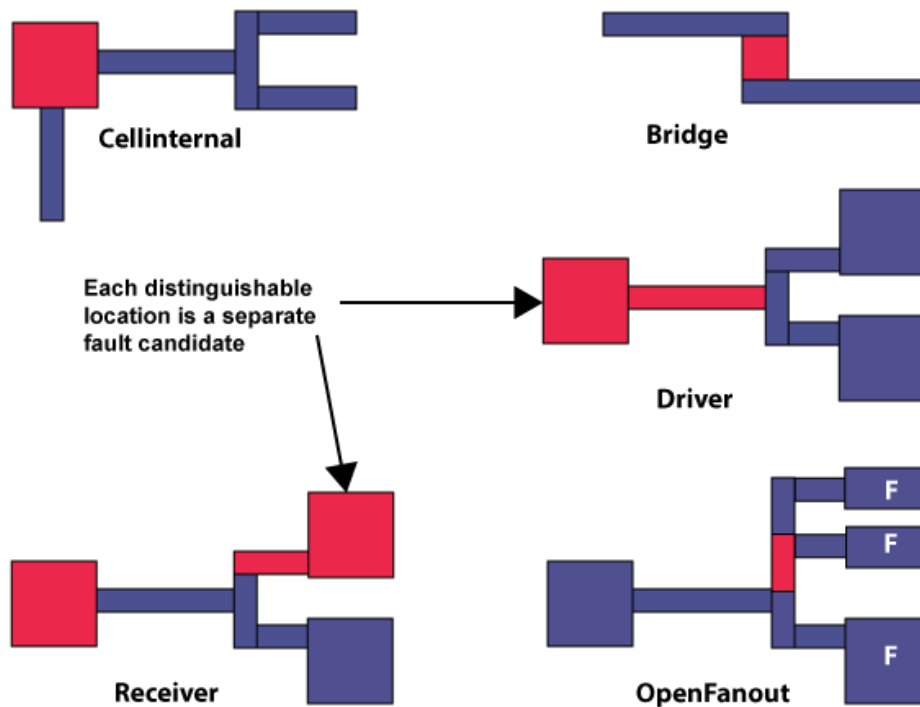
```
-----  
Diagnosis Candidate Report  
-----  
#tester_failures=256 (#ignored=41/#used=215)  
#patterns=51, #simulated_patterns=17 (#fail=7/#pass=10)  
#defects=1
```

Each defect is reported, and includes the number of failing patterns, observe points, simulated failures, and list of one or more candidates:

```
Defect 1:  
  #failing_patterns=7  
  #observe_points=65  
  #simulated_failures=70 (#unique=68/#potential=2)  
  #candidates=2
```

Each defect candidate is assigned a predefined class which defines the logical location and physical details of the candidate (shown in red in Figure 1). Classes include Cellinternal, Bridge, Receiver, Driver, PinPath, and OpenFanout. These classifications match Yield Explorer classifications.

Figure 107 Candidates are Assigned Predefined Classes, such as Cellinternal or Bridge



A candidate includes a specific set of physical details. For example, the description of a Receiver class candidate consists of a cell instance and the net or subnet connected to an input pin of that cell. A Bridge class candidate includes the bridgeable area between two metal shapes on the same layer.

```

Candidate 1:
class=Receiver
instance=c/iproc/U14417, pin=E, module=A010LL
net_id=27094, fanout_id=2
behavior=sa0, match_score=91.43% (TFSF=64/TFSP=6/TPSF=0)
Candidate 2:
class=Bridge
net1_id=12443, net2_id=27094
driver1=c/iproc/iarc_portxdati_regx12x, pin=QN
driver2=c/iproc/U6085, pin=Z
behavior=bAND, match_score=100.00% (TFSF=70/TFSP=0/TPSF=0)
    
```

Note the following:

- A single candidate can have multiple fault associations, and each fault has a particular type of behavior. For example, a Bridge class candidate may share both bAND and bDOM behavior between the same net pairs. In this case, each behavior is scored separately.
- Candidates with different behaviors can be reported in the same defect — as long as they are associated with a common set of tester failures and they are in separate logic cones. This means candidates with different behaviors are scored based upon the same set of tester failures.
- Nets are identified by the primitive ID of the driving gate for candidate classes with a net, subnet, or net pair. For OpenFanout class candidates, the subnet ID is always the physical subnet ID, even if the `set_diagnosis -show physical_subnet_id` command is specified. For Bridge class candidates, the instance and pin name for both net drivers are reported.
- If physical data is available from the PHDS database, a physical summary is also included with each candidate. This summary includes the layers associated with the candidate and the candidate bounding box and area:

```
layers=METAL2 bbox=(619320 659710) (619530 660730), area=214200
```

- A candidate is assigned to the PinPath class if it does not contain physical net data. Driver candidates with physical net data are assigned a subnet ID of 0, and Receiver candidates with physical net data on a net with only one receiver pin are assigned a subnet ID of 1.

Class-Based Cell-Aware Diagnosis

The class-based diagnostics report supports cell-aware diagnostics. Cell-aware behaviors can be included in Driver, Receiver, CellInternal, and CellAware class candidates.

When cell-aware diagnostics is enabled by the `set_diagnosis -fault_type all` command, several behaviors can be reported differently than the fault-based report: ca0, ca1, ca01, car, caf, carf. These six behaviors correspond to the sa0, sa1, sa01, str, stf, strf faults, except that their match score is calculated assuming that the defect can be inside the cell rather than on the pin.

If cell test models have been read using the `read_cell_model` command, CTM behaviors will also be scored during diagnosis and any matching candidates will be reported as a CellAware class candidate with the CTM ID and any equivalent CTM IDs.

Fault-Based Diagnosis Reporting

In the fault-based diagnostics report, a collapsed fault is a unique candidate. A fault and all its equivalent faults are considered a single candidate and separate faults are created for different behaviors.

The fault-based report is created by default when you specify the `run_diagnosis` command. If you previously ran class-based reporting (described in the [Class-Based Diagnosis Reporting](#) section), you can revert back to fault-based reporting using the following command:

```
set_diagnosis -organization fault
```

The following example shows a typical fault-based diagnosis report produced by the `run_diagnosis` command.

```
TEST-T> run_diagnosis /project/mars/lander/chipA_failure.dat \  
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>  
  
sa1 DS de_d/data3_reg_0_/Q (S003)  
  
sa1 -- 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>
```

```
sa1  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)
sa1  --  de_encrypt/C264/U26/I2  (L434ND)
-----
match=50.00%, #explained patterns: <failing=1, passing=37>
sa1  DS  de_encrypt/C264/U28/I1  (L434ND)
-----
```

This example 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 percent match score and has four faults-equivalents. The second fault location of the second defect has a 50 percent match score.

The fields in this report are described as follows:

#failing_patterns

Identifies the total number of failing patterns in the failure file. A pattern is assumed to include both a measure of all POs and an unload of the scan chain.

#failures

Located in the main header, this field identifies the number of failures in the failure log file. In each defect's header, it shows the number of failures the candidates in that defect caused.

#defects

Indicates the number of different defects that appear to be causing the failures.

#faults

Indicates the number of collapsed faults. In the main header, it indicates the total number of faults. In each defect's header, it shows the number of faults in that defect group.

Simulated : #failing_pat=, #passing_pat= #failures

Displays the number of failing and passing patterns that were simulated, and the number of failures in the simulation.

Fault candidates for defect : <> fault model

The header for each defect displays the fault model used for that defect group. Then, there is the list all the fault candidates for a given defect. The fault list is given in the following format:

- First column: fault type. It could be `sa0` for stuck-at-0 or `sa1` for stuck-at-1.
- Second column: detection technique. It could be: "DS" (detected by simulation). This is the representative fault. "- -". This is an equivalent fault.
- Third column: fault location (pin pathname)
- Fourth column: module name of the defective cell.

match=%

Indicates the match score of the set of fault candidates based on how well they match the defective device response on the tester.

#explained pattern: <failing: , passing:>

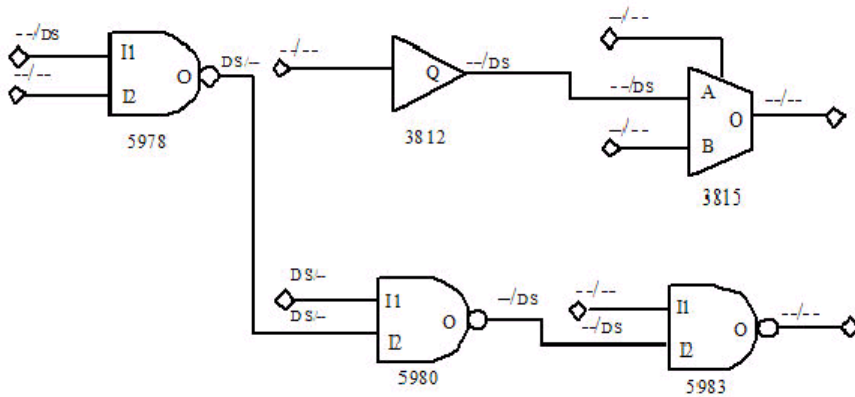
Indicates the number of failing and passing patterns that are explained with the fault candidate.

If logic diagnostics fails to find any candidates, the report appears as shown in the following example:

```
#failing_patterns=7, #defects=0, #unexplained_fails=7, CPU=19.56
-----
Unexplained pattern list:
3 6 8 12 13 25 67
-----
No candidate because all failing patterns are unexplained.
```

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 of the TestMAX ATPG GUI, you can display the instances and fault locations graphically, as shown in the following figure. 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 1 Diagnosis Data Displayed Graphically



You can identify each defect location 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. TestMAX ATPG shows all potential failure sites that would cause the same failure data patterns.

In this example, the diagnosis by TestMAX ATPG finds two independent areas of failure in the design. The graphical schematic viewer (GSV) display shows the two corresponding independent groups of logic. According to the diagnosis, the faulty circuit location for each failure is displayed along the path.

Verbose Format

When the `-verbose` option is used for either the `set_diagnosis` or `run_diagnosis` commands, additional information is added to the diagnostics report. An example is as follows:

```
#failing_pat=15, #failures=17, #defects=2, #faults=8, CPU_time=0.01
Simulated : #failing_pat=15, #passing_pat=36, #failures=17
-----
Fault candidates for defect 1: stuck fault model, #faults=1,
#failing_pat=14, #passing_pat=37, #failures=14
Observable points:
  782
-----
Explained pattern list:
  2  16  20  21  23  25  34  37  38  39  40  45  47  49
-----
```

```

match=100.00%, (TFSF=14/TFSP=0/TPSF=0), #perfect/partial match:
<failing=14/14, passing=37>
sa0 DS mic0/pc0/add_247/U41/Z (ND2I)
sa0 -- mic0/pc0/add_247/U40/B (ENI)
  
```

Note the following definitions:

- **Observable points** — The list of gate IDs in which the failures generated by all fault candidates from this defect group occurred.
- **Explained pattern list** — The list of the all patterns which could be explained by all fault candidates from this defect group. Some fault candidates in the same defect could explain some patterns and other candidate other patterns. But the list is the union of all explained patterns.
- **TFSF=N1/TFSP=N2/TPSF=N3** — These are the match score components. See `run_diagnosis -rank_fault` for more details.
- **#perfect/partial match: <failing=N1/N2, passing=N3>** — This data indicates the number of failing patterns that are perfectly or partially explained by the fault candidate. A perfect match means that the failures observed on the tester are perfectly matching (without any other failure either in simulation or on the tester). It also indicates the number of passing patterns that are explained with the fault candidate.

Physical Diagnosis Format

```

TEST-T> run_diagnosis fail_56.log
Setting top-level physical design name to 'RISC_CHIP'
Check expected data completed: 241 out of 241 failures were checked
Diagnosis summary for failure file fail_56.log
#failing_pat=30, #failures=241, #defects=1, #faults=1, CPU_time=1.08
Simulated : #failing_pat=30, #passing_pat=96, #failures=194
-----
Defect 1: stuck-at fault model, #faults=1, #failing_pat=30,
#passing_pat=96, #failures=241
Observable points:
1693 1687 1696 1699 1672 1530 1529
-----
Explained pattern list:
40 41 47 48 50 51 54 55 58 59 67 69 70 71 72 73 77 78 79 80 85 88
92 93 94 95 97 98 99 101
-----
match=100.00%, #explained patterns: <failing=30, passing=96>
sa01 DS I_RISC_CORE/I_ALU/U14/ZN (inv0d1)
Pin_data: X=747110 Y=652790, Layer: METAL (38)
Cell_boundary: L=746115 R=747345 B=650175 T=653865
Subnet_id=4
-----
  
```

```
Total Wall Time = 22.97 sec      PHDS Query Time = 22.13 sec  
PHDS queries: subnets (added/total)=10/40 bridges (added/total)=28/58
```

The physical diagnosis report includes the physical location of the failing pin and cell.

Where:

Pin_data:

- **X** is the horizontal coordinate of one of the vertices of the pin associated with the location of the fault candidate.
- **Y** is the vertical coordinate of one of the vertices of the pin associated with the location of the fault candidate.
- **Layer** is the physical layer where the pin object is defined.

Cell_boundary:

- **L** is the horizontal coordinate of the leftmost boundary of the cell identified with the fault candidate.
- **R** is the horizontal coordinate of the rightmost boundary of the cell identified with the fault candidate.
- **B** is the vertical coordinate of the bottommost boundary of the cell identified with the fault candidate.
- **T** is the vertical coordinate of the topmost boundary of the cell identified with the fault candidate.

Also note that the performance data in the physical diagnosis report is slightly different than the standard diagnosis report:

- The **CPU_time** is strictly the time that TestMAX ATPG is active. This time does not include the PHDS query time. In the previous example, the **CPU_time** is approximately 1 second, even though the diagnosis run took almost 23 seconds.
- The PHDS diagnosis report includes the wall time and the time it takes to query the PHDS database during diagnosis.
- The second line in the report displays the number of extracted subnets and the number of bridge pairs queried in the PHDS database compared to the total number of subnets and bridging pairs identified during the previous diagnosis run. This data helps you identify any matching issues between the logical and physical names during diagnosis.

- The subnets and bridges extracted during a diagnosis run are displayed as “added”. The “total” includes subnets and bridges extracted in previous diagnosis runs. For example, if the next diagnosis run extracts 10 subnets and 22 bridges, the second line appears as follows:

```
PHDS queries: subnets(added/total)=10/50 bridges(added/total)=22/80
```

Scan Chain Diagnosis Format

```
fail.log scan chain diagnosis results: #failing_patterns=79
```

```
-----  
  
defect type=stuck-at-1  
  
match=100% (TFSF=500/TFSP=0/TPSF=0) chain=c0 position=178  
master=CORE/c_rg0 (46)  
  
match=100% (TFSF=500/TFSP=0/TPSF=0) chain=c0 position=179  
master=CORE/c_rg2 (57)  
  
match= 98% (TFSF=500/TFSP=10/TPSF=0) chain=c0 position=180  
master=CORE/c_rg6 (54)  
  
CPU=0.26 #sim_patterns=57 #sim_cells=64  
  
-----
```

#failing_patterns

Indicates the total number of failing patterns in the failure file.

defect type

The predicted type of defect. It could be stuck-at, slow-to-rise, slow-to-fall, fast-to-rise, or fast-to-fall. The polarity of the reported defect affects the scan output of the top candidate for each scan chain. It is not necessarily the same for all scan cells because an inverter might be present in the scan path. The exact polarity can be retrieved using the Tcl API.

match

A percentage score that measures how well failures seen on the tester match a simulated chain defect at that location. The components of the match score (TFSF, TFSP, TPSF) calculation are displayed in the verbose report.

chain

Indicates the chain name where the defect is diagnosed.

position

Indicates the position in the chain where the defect is diagnosed.

master

Indicates the scan cell instance name of the diagnosed defect.

Next, follow these performance indicators:

CPU

Indicates the CPU time of the chain diagnosis run.

#sim_patterns

Indicates the number of patterns used during the chain diagnosis simulation.

#sim_cells

Indicates the number of cells used during the chain diagnosis simulation.

The chain diagnostics can also appear as follows:

```
./failures/fail8g16.log scan chain diagnosis results: #failing_patterns=1
-----
Warning: Insufficient data to locate stuck-at-0 fault in chain 48.
-----
```

This example reports that the number of failures contained in the failures log file is sufficient to determine the behavior and the chain name of the fault candidate. But it fails to accurately locate the failing scan cells. More failures are needed.

The report can also appear as follows:

```
./failures/fX7.log scan chain diagnosis results: #failing_patterns=1
-----
Scan chain diagnosis failed to identify any fault candidate.
-----
```

This example indicates that the scan chain diagnostics failed to find a fault candidate that match the failures seen on the tester.

```
/i_p0/E (mx2a3)
# subnet_id=2
```

Using a Dictionary for Diagnosis

TestMAX ATPG diagnostics normally use a limited set of patterns for fault candidate ranking. By default, the first 96 passing patterns are used. A diagnostics dictionary improves diagnostics resolution by targeting and storing an additional set of [N detect patterns](#) for each pin and polarity. You can control the contents of the dictionary by specifying the maximum number of patterns to use per pin.

The following models and patterns are supported:

- Stuck-at and transition fault models
- Basic scan, two-clock, and fast-sequential patterns

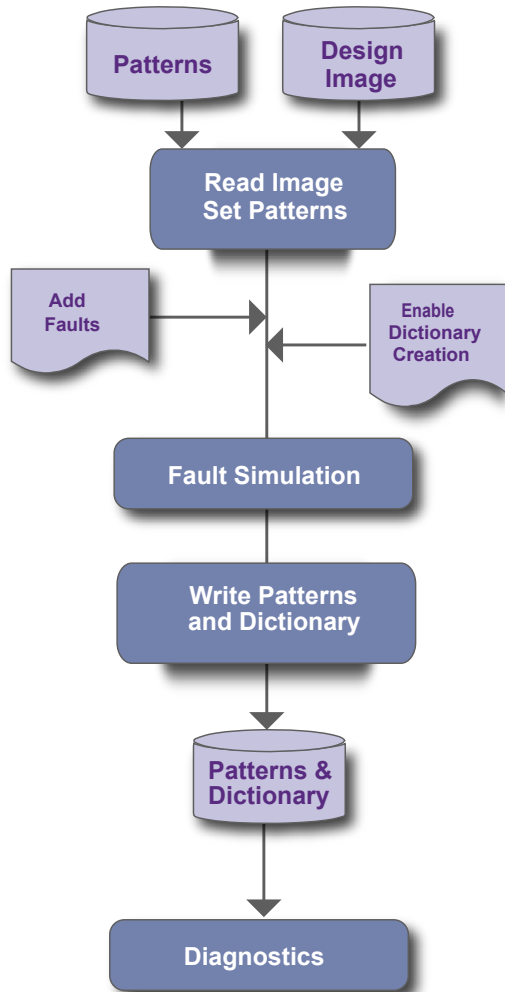
A diagnostics dictionary is written to a binary file, including the patterns, and is extracted with minimum overhead during the `run_fault_sim` command process.

The following topics describe how to use a dictionary for diagnostics:

- [Example Flow](#)
- [Diagnosis Dictionary Commands](#)
- [Limitations](#)

The following figure shows a typical flow for generating a diagnostics dictionary. The output is used for diagnostics.

Figure 108 Diagnosis Dictionary Creation Flow



Example Flow

The following example script generates a diagnostics dictionary and uses it in the diagnostics flow:

Diagnosis Dictionary creation script:

```
read_image ./design.img
set_patterns -external ./pat.bin
add_faults -all
set_faults -store_dictionary 10
run_fault_sim
write_patterns ./pat+dic.bin -external -replace
```

Diagnosis Dictionary diagnosis script:

```
read_image ./design.img
set_patterns -external ./pat+dic.bin
# Diagnosis will use the Diagnosis Dictionary by default
run_diagnosis ./failure.fail
# Runs diagnosis without using the Diagnosis Dictionary
set_diagnosis -nouse_dictionary
run_diagnosis ./failure.fail
```

Diagnosis Dictionary Commands

The following commands create, delete, enable, or disable a diagnostics dictionary:

- The `set_faults -store_dictionary` command enables you to create a diagnostics dictionary during the fault simulation process. For example, the following command creates a diagnostics dictionary with a maximum of 10 failing patterns per pin:

```
set_faults -store_dictionary 10
```

- The `set_faults -delete_dictionary` command deletes either internal or external pattern diagnostics dictionaries, or both. The following example deletes both internal and external pattern diagnostics dictionaries:

```
set_faults -delete_dictionary both
```

To delete only an internal diagnostics dictionary, use the `internal` parameter instead of `both`. Or to delete only an external diagnostics dictionary, use the `external` parameter.

- The `set_patterns -keep_dictionary` command retains the current diagnostics dictionary when reading in an external pattern set, as shown in the following example:

```
set_patterns -external pat_and_dic.bin -keep_dictionary
```

- Use the `-use_dictionary` or `-nouse_dictionary` options of the `set_diagnosis` command to enable or disable the use of a dictionary during diagnostics. The `-use_dictionary` option is the default. The following example disables the use of a dictionary:

```
set_diagnosis -nouse_dictionary
```

Limitations

The following limitations applies to creating or using a diagnostics dictionary:

- You cannot write output with split patterns
- When using TestMAX ATPG to generate a dictionary, you must specify the `-ndetects` option of the `run_fault_sim` command. For example:

```
set_simulation -num_threads 8 set_faults -store_dictionary 8
run_fault_sim -ndetects 8
```

Failure Mapping Report for DFTMAX Patterns

When you run diagnostics for DFTMAX patterns, the report includes a section in its header that displays the mapping of the failures. This report is an intermediate report and is not intended to be a complete report. It includes only those cases that have a unique choice during the first phase of DFTMAX failure mapping procedure.

To print a complete mapping report, you have to use the

`run_diagnosis -only_report_failures` command or the `set_diagnosis -mapping_report` command. The format of the later report is explained in "Understanding the Failure Mapping Report." The format of `-only_report_failures` is documented in the description of the `run_diagnosis` command. You should use this report because it is short and easy to understand.

The following example shows how this additional section appears in the diagnosis report:

```
-----
pattern  chain                pos#      output pin_names
-----  -
9        1                      4        test_so1
test_so2 test_so3
14       1                      3        test_so1
test_so2 test_so3
15       1                      4        test_so1 test_so2
test_so3
48       1                      4        test_so1
test_so2 test_so3
52       1                      4        test_so1
test_so2 test_so3
-----
Failure mapping completed: #failing_pats=5, #skipped_pats=0,
#masked_cycles=0, CPU_time=0.00
-----
```

This section includes a header which describes the column printed after it. The failures mapping results are printed next, followed by a failure mapping one line summary.

The columns are described as follows:

- `pattern` -- the failing pattern number
- `chain` -- the failing scan chain
- `pos#` -- the failing scan shift present in the failures log file
- `output pin_names` -- the names of the output pins where the failures are observed

The default number of failures reported are 10. You can change this by running the command `set_diagnosis -max_report_failures <N>`.

The failure mapping one-line summary shows the number of failing patterns that were processed (`#failing_pats`). When failures for a particular cycle could not be mapped back, the `#masked_cycles` is incremented. In the previous example, no cycle has been masked. When too many cycles per pattern are masked, the entire pattern is skipped. This is indicated by the `#skipped_pats`. In the previous example, no pattern has been skipped by the mapping process. When `#masked_cycles` and `#skipped_pats` are equal to 0, this is an indicator that the mapping step of the diagnosis is doing a good job.

When tail pipeline registers are present, the failures log file indicates the scan cell index which is failing + `N`, where `N` is the depth of the tail pipeline register. Then, the column `pos#` in the previous report is not the failing scan cell index. To precisely determine the failing scan cell index, execute the command `run_diagnosis <failure_log> -only_report_failures`.

Composite Fault Model Data Report

Composite faults are based on TestMAX ATPG component faults. They are only used in a diagnosis report to better describe the observed behavior of a defect on the tester. If this defect behavior is observed, the composite fault model behaviors can be reported by default. For the ranking flow, TestMAX ATPG diagnostics ranks only component fault types, unless the `set_diagnosis -composite` command is specified. The composite fault types are as follows:

- `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 this fault type 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. Ranking scores are based on the fault simulation at the fault site for failing and passing patterns. This might result in optimistic ranking scores since this model always matches the tester for passing patterns. The scores are optimistic only when the aggressor is unknown. Only the victim node is reported.

The following example report show how the diagnosis report appears with composite fault model data:

```
#failing_pat=6, #failures=20, #defects=4, #faults=5, CPU_time=0.44
Simulated : #failing_pat=6, #passing_pat=75 #failures=23
-----
Fault candidates for defect 1: stuck fault model, #faults=1, #failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=72>
sa01 DS ENC/I_RC/U1569/B (and2c3)
-----

Fault candidates for defect 2: transition fault model, #faults=1,
#failures=1
-----
match=100.00%, #explained patterns: <failing=1, passing=73>
str DS ENC/I_RC/U1685/Y (inv1a3)
stf -- ENC/I_RC/U1685/A (inv1a3)
-----

Fault candidates for defect 3: stuck fault model, #faults=2, #failures=2
-----
match=100.00%, #explained patterns: <failing=2, passing=72>
sa0 DS ENC/I_RC/U1697/Y (inv1a3)
sa1 -- ENC/I_RC/U1697/A (inv1a3)
sa0 DS ENC/I_RC/U2074/B (ao4f3)
-----

Fault candidates for defect 4: bridging fault model, #faults=1,
#failures=1
-----
```

```
match=33.33%, #explained patterns: <failing=1, passing=71>  
bAND DS ENC/I_RC/U1685/Y ENC/I_RC/U1697/Y (inv1a3)
```

Fault types:

sa01:

Fault location behaves like a sa0 on some patterns, and a sa1 on others. This could be a coupled open or a bridge type defect.

strf:

Fault location can cause a delay on both rising and falling transitions (slow-to-rise-fall).

bAND or bOR:

The defect location behaves as a wired AND or wired OR type bridge. Examples are provided below:

```
-----  
match=100.00%, (TF SF=15/TF SP=0/TP SF=0), #perfect/partial match:  
<failing=15/15, passing=31>  
bAND DS mic0/pc0/add_247/U14/Z (ENI) mic0/alu0/U89/Z (ND2I)  
-----  
match=100.00%, (TF SF=24/TF SP=0/TP SF=0), #perfect/partial match:  
<failing=24/24, passing=51>  
bOR DS mic0/pc0/add_238/U76/Z (ENI) mic0/alu0/U12/Z (ND2I)
```

Note that both nodes of the bridge are reported. The library cells are also provided in parenthesis.

bDOM:

The defect location behaves as the victim node of a dominant bridge.

By default, only the victim node is reported. An example is as follows:

```
-----  
match=100.00%, (TF SF=24/TF SP=0/TP SF=0), #perfect/partial match:  
<failing=24/24, passing=56>  
bDOM DS mic0/pc0/add_233/U39/Z (ND2I)  
bDOM -- mic0/pc0/add_233/U26/A (AN2I)
```

However, if the likely bridging pairs or ranking flow is used, then both nodes can be included in the report. An example is provided below:

```
-----  
match=100.00%, (TF SF=75/TF SP=0/TP SF=0), #perfect/partial match:  
<failing=75/75, passing=92>  
bDOM DS mic0/pc0/add_233/U39/Z (ND2I) mic0/alu0/U16/ZN  
(INV2I)
```

Note that the library cells are provided in parenthesis.

The following example shows how stuck-open faults in the diagnosis subnets report:

```
#-----  
# Defect 1: stuck fault model, #faults=1, #failing_pat=6,  
#passing_pat=14,#failures=35  
  
#-----  
# match=100.00%, #explained patterns: <failing=6, passing=14>  
  
# sa01 DS top/i_p0/E (mx2a3)  
  
# subnet_id=2
```

Parallel Diagnosis

You can diagnose multiple failure logs in parallel in a single TestMAX ATPG session with a single `run_diagnosis` command. This approach, called *parallel diagnosis*, improves volume diagnostics throughput and is much more memory efficient than invoking multiple TestMAX ATPG sessions. It is especially useful when processing a large number of failure files.

The following sections describe how to run parallel diagnosis:

- [Specifying Parallel Diagnosis](#)
- [Converting Serial Scripts to Parallel Scripts](#)
- [Using Split Datalogs to Perform Parallel Diagnosis for Split Patterns](#)
- [Diagnosis Log Files](#)
- [Parallel Diagnosis Limitations](#)

See Also

- [Running Multicore ATPG](#)

Specifying Parallel Diagnosis

To specify parallel diagnostics, use the `-num_processes` option of the `set_diagnosis` command. This option sets the number of cores to use during parallel diagnostics. You can specify the number of processes to launch based on the number of CPUs and the available memory on the multicore machine.

The following example configures parallel diagnostics to use four cores:

```
set_diagnosis -num_processes 4
```


You can also define a post processing procedure to run concurrently with a parallel diagnostics run, as shown in the following example:

```
proc pp {} {
    write_ydf -append my_ydf.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
set_diagnosis -post_procedure pp
```

In the previous example, a procedure called `pp` specifies the `write_ydf` command and several Tcl API commands. This procedure is executed using the `-post_procedure` option of the `set_diagnosis` command.

Note that when you use multiple slave cores, each process writes the same report file.

To disable a post processing procedure, specify the following command:

```
set_diagnosis -post_procedure none
```

When parallel diagnostics is enabled, you can specify a list of data logs in the `run_diagnosis` command, as shown in the following example:

```
set_diagnosis -num_processes 4
run_diagnosis [list {datalogs/ff_[1-9].log} \
    {datalogs/ff_[1-9][0-9].log} \
    {datalogs/ff_100.log}]
```

Note that wildcards are also accepted when specifying data logs, as shown in the following example:

```
run_diagnosis datalog/ff_*.log
```

Converting Serial Scripts to Parallel Scripts

You can convert an existing serial mode script to a parallel mode script, and then run parallel diagnostics for any number of cores.

The following example is a script snippet used for volume diagnosis in serial mode:

```
for {set i 1} {$i <= 100} {incr i} {
    set fail_log datalog/ff_{$i}.log
    run_diagnosis $fail_log
    write_ydf -append my_ydf_file.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath]\

```

```
    $datalog >> candidates.list  
  }  
}
```

The following example shows how the script snippet in the previous example appears as a parallel script that uses four cores:

```
set_diagnosis -num_processes 4  
proc pp {} {  
  write_ydf -append my_ydf_file.ydf  
  set_datalog [get_attribute [index_collection \  
    [get_diag_files -all] 0] name]  
  foreach_in_collection cand [get_candidates -all] {  
    echo [get_attribute $cand pinpath] \  
      $datalog >> candidates.list  
  }  
}  
set_diagnosis -post_procedure pp  
run_diagnosis [list {datalogs/ff_[1-9].log} \  
  {datalogs/ff_[1-9][0-9].log} \  
  {datalogs/ff_100.log}]
```

Using Split Datalogs to Perform Parallel Diagnosis for Split Patterns

You can use split datalogs to perform parallel diagnostics for split patterns.

The following example shows a serial mode script snippet used for diagnostics with split datalogs:

```
set_patterns -external p1.bin -split  
set_patterns -external p2.bin -split  
set_patterns -external p3.bin -split  
for {set i 1} {$i <= 100} {incr i} {  
  run_diagnosis datalogs/ff_${i}.p1.log \  
    -file "datalogs/ff_${i}.p2.log datalogs/ff_${i}.p3.log"  
  write_ydf -append my_ydf.ydf  
  set_datalog [get_attribute [index_collection \  
    [get_diag_files -all] 0] name]  
  foreach_in_collection cand [get_candidates -all] {  
    echo [get_attribute $cand pinpath] \  
      $datalog >> candidates.list  
  }  
}
```

The following example shows how the script snippet in the previous example appears as a parallel mode script that uses four cores:

```
set_patterns -external p1.bin -split  
set_patterns -external p2.bin -split  
set_patterns -external p3.bin -split
```

```

set_diagnosis -num_processes 4
proc pp {} {
    write_ydf -append my_ydf.ydf
    set datalog [get_attribute [index_collection \
        [get_diag_files -all] 0] name]
    foreach_in_collection cand [get_candidates -all] {
        echo [get_attribute $cand pinpath] \
            $datalog >> candidates.list
    }
}
set_diagnosis -post_procedure pp
run_diagnosis datalogs/ff_*.p1.log \
    -file "datalogs/ff_*.p2.log datalogs/ff_*.p3.log"

```

See Also

- [Reading Multiple Pattern Files](#)

Diagnosis Log Files

When you run parallel diagnosis, the diagnosis log is stored in multiple files; one file is created for each core. The name of the diagnosis log file is based on the name of the tool log file specified by the `set_messages` command and is appended with the core ID.

The following example specifies a log file called `diag.log`:

```
set_messages -log diag.log -replace -level expert
```

When multiple cores are used for parallel diagnostics, a diagnosis log file is created for each core, as shown in the following example:

```
diag.log.1
diag.log.2
diag.log.3
diag.log.4
```

Each datalog file is processed for a different slave core, as specified in the tool log file:

```

run_diagnosis datalogs/ff_*.log
Perform diagnosis with 100 failure files.
Starting parallel processing with 4 processes.
-----
Failure file                >>                output log
-----
datalogs/ff_100.log         diag.log.1
datalogs/ff_101.log         diag.log.2
datalogs/ff_102.log         diag.log.3
datalogs/ff_103.log         diag.log.4
datalogs/ff_104.log         diag.log.4
datalogs/ff_105.log         diag.log.2
datalogs/ff_106.log         diag.log.3

```

```

datalogs/ff_107.log          diag.log.1
...
-----
End parallel diagnosis: Elapsed time=14.33 sec, Memory=596.61MB.
Processes Summary Report
-----

```

In the previous example, the total memory consumed by parallel diagnostics is 596.61MB, and the total elapsed runtime is 14.33 seconds.

A slave core diagnosis log file is similar to a single core diagnosis log file. The following example is an excerpt from the `diag.log1` file:

```

=====
Performing diagnosis with failure file datalog/ff_100.log
Diagnosis will use 2 chain test patterns.
Check expected data completed: 196463 out of 196463 failures were checked
Failures for COMPRESSOR patterns
-----
pattern  chain    pos#  output pin_names
-----
0        41         0     OUT_0  OUT_1  OUT_5
0        41         3     OUT_0  OUT_1  OUT_5
0        41         4     OUT_0  OUT_1  OUT_5
0        41         7     OUT_0  OUT_1  OUT_5
0        41         8     OUT_0  OUT_1  OUT_5
0        41        11     OUT_0  OUT_1  OUT_5
0        41        12     OUT_0  OUT_1  OUT_5
0        41        15     OUT_0  OUT_1  OUT_5
0        41        16     OUT_0  OUT_1  OUT_5
0        41        19     OUT_0  OUT_1  OUT_5
-----
datalogs/ff_100.log scan chain diagnosis results: #failing_patterns=400
-----
defect type=stuck-at-1
match=100.00% chain=41 position=214
  master=CORE_U1/vys2/U_L0/U_FONTL/FF_pp1_reg_1_ (FSDX_1)
CPU_time=1.25 #sim_patterns=10 #sim_failures=5001
-----
YDF Candidates Schema with 0 entries retrieved. -----

Following physical data tables generated for all elements:
- YDF
CPU_time: 0.00 sec
Memory Usage: 0MB
-----
Performing diagnosis with failure file datalog/ff_107.log
Diagnosis will use 2 chain test patterns.
Check expected data completed: 157974 out of 157974 failures were checked
Failures for COMPRESSOR patterns
...

```

You can specify the `-level expert` option of the `set_messages` command to produce a parallel processing summary report for each core after the `run_diagnosis` command process is completed:

Process		Patterns	Time (s)		Memory (MB)			
ID	pid	External	CPU	Elapsed	Shared	Private	Total	Pattern
0	3449	401	0.09	14.33	296.45	0.14	296.59	1.81
1	3461	0	14.20	14.32	251.91	75.18	327.10	0.00
2	3462	0	8.71	8.80	251.93	70.98	322.91	0.00
3	3463	0	10.66	10.77	251.94	72.02	323.9	0.00
4	3464	0	10.81	10.98	251.96	81.84	333.80	0.00
Total		401	44.47	14.33	296.45	00.17	596.61	1.81

In the previous example, the total memory usage for parallel diagnostics is less than 600MB. However, running multiple TestMAX ATPG sessions for the same diagnostics session requires almost 1.2GB.

Parallel Diagnosis Limitations

The `run_diagnosis` command issues a warning message if the number of specified failure data files is less than the number of enabled processes.

Note the following restrictions and limitations:

- If you specify more cores than the number of datalogs to be analyzed, the enhanced performance provided by parallel diagnostics is compromised because parallelization is applied to each datalog.
- For small designs you might not see a significant performance improvement, especially if diagnosis for a single datalog takes only a few seconds.

18

Using Physical Data for Diagnosis

Physical diagnostics provides significantly higher defect isolation accuracy and precision than standard scan diagnostics. When used with the Synopsys Yield Explorer tool, physical diagnostics improves the effectiveness of volume diagnostics.

To perform physical diagnostics, TestMAX ATPG requires physical data stored in a physical diagnostics (PHDS) database. TestMAX ATPG then dynamically extracts likely bridging pairs, subnet information, and layout data to identify diagnostics fault candidates.

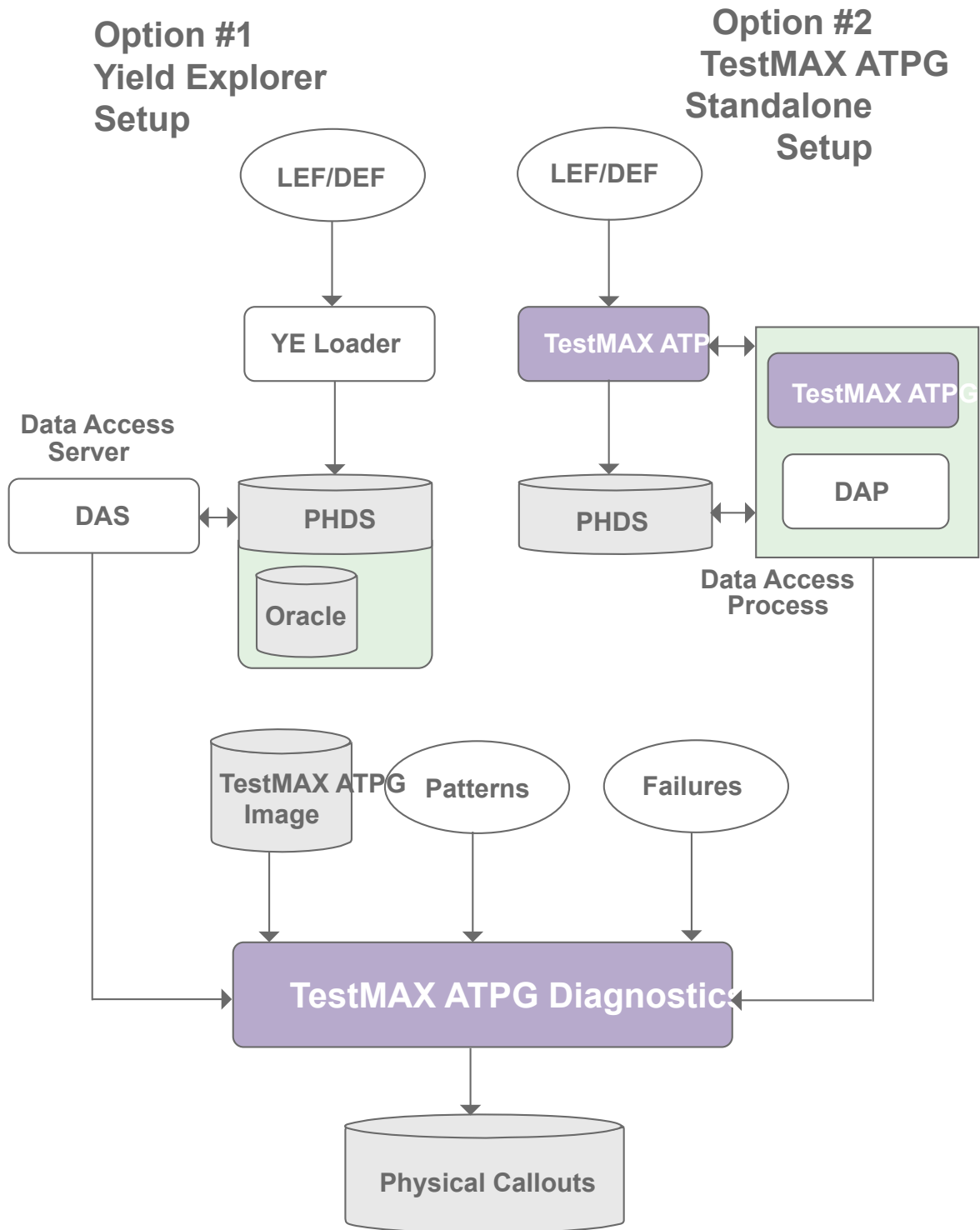
The following sections describe how to prepare and use physical data to perform diagnostics:

- [Physical Diagnosis Flow Overview](#)
- [Creating and Validating a PHDS Database](#)
- [Reading a PHDS Database into TestMAX ATPG](#)
- [Name Matching Using a PHDS Database](#)
- [Setting Up and Running Physical Diagnosis](#)
- [Static Subnet Extraction Using a PHDS Database](#)
- [Reporting Physical Subnet ID Data](#)
- [Writing Physical Data for Yield Explorer](#)

Physical Diagnosis Flow Overview

You can use Yield Explorer or TestMAX ATPG to create the PHDS database used for physical diagnostics. After loading the PHDS database into the data access process (DAP) server or the data access server (DAS), you can use TestMAX ATPG to extract the physical information and perform diagnostics.

Figure 109 Flows for Using the PHDS Database for Physical Diagnosis



See Also

- [Reading the PHDS Database](#)

Creating and Validating a PHDS Database

To create and validate a PHDS database, you need access to all the LEF/DEF files relevant to diagnostics within your design. It is possible to extract a physical database from an incomplete LEF/DEF set, although you can only perform logical diagnostics for the blocks associated with any missing files. This is expected behavior for memories and hardened IP.

A technology file (commonly referred to as a "techlef") is required for physical diagnostics. This file contains a description of the physical properties of the design, such as the metal layers and the routing grid.

The information in the technology LEF file is sometimes included directly in a set of LEF files. If you include the string "tech" (case insensitive) in the name of the file (for example, "technology.lef" or "any.tech.lef.gz"), the file is automatically recognized by TestMAX ATPG as a technology file. You can also use the `set_physical_db -technology_lef_file` command to specify the technology file name.

To create and validate a PHDS database:

1. Specify the locations of the source LEF and DEF directories.

```
set_physical_db -lef_directory ./lef -def_directory ./def
```

2. Specify the name of the top-level DEF file.

```
set_physical_db -top_def_file top_design.def
```

You can use any name for the `top_design.def` file.

Specify the location of the output PHDS directory.

```
set_physical_db -database ./phds
```

3. Specify the name of the design associated with the LEF/DEF database you are translating.

```
set_physical_db -device [list DES 4]
```

You can specify any name regardless of the actual design name. If you use the `-device` option, you can specify the device version (in the example, the device version is 4).

4. Create and validate the PHDS database.


```
write_physical_db -replace -verbose
```

The `write_physical_db` command creates and validates the specified PHDS database (the `phds` directory is used in the example). You must use the `-replace` option to overwrite a previously loaded device with same name and version.

After creating a PHDS database, a confirmation message appears, as shown in the following example:

```
Writing Physical Database...
LEF input directory : ./lef
DEF input directory : ./def
Top DEF file name : top_design.def
PHDS output directory: ./phds
Device name : DES
Device version : 4
Running PHDS validation...
PHDS validation completed successfully.
-----
-----
Validation Summary Report
-----
-----
Warning: Rule Y18 (DEF Without Corresponding LEF) was violated 2 times.
There were 2 violations that occurred during Validation process.
Running PHDS creation...
PHDS creation completed successfully.
Total_time = 36.76
```

You can specify the `-no_validation` option of the `set_physical_db` command to create a PHDS database in a separate run without validating it, as shown in the following example:

```
set_physical_db -lef_dir ./LEF -def_dir ./DEF
set_physical_db -top_def_file RISC_CHIP.def
set_physical_db -database PHDS_C
set_physical_db -device {"RISC" "1"}
set_physical_db -novalidation
write_physical_db
```

You can also use the `-nocreate_phds` option of the `set_physical_db` command to only validate the PHDS database in a separate run:

```
set_physical_db -lef_dir ./LEF -def_dir ./DEF
set_physical_db -top_def_file RISC_CHIP.def
set_physical_db -database PHDS_C
set_physical_db -device {"RISC" "1"}
set_physical_db -nocreate_phds
write_physical_db
```

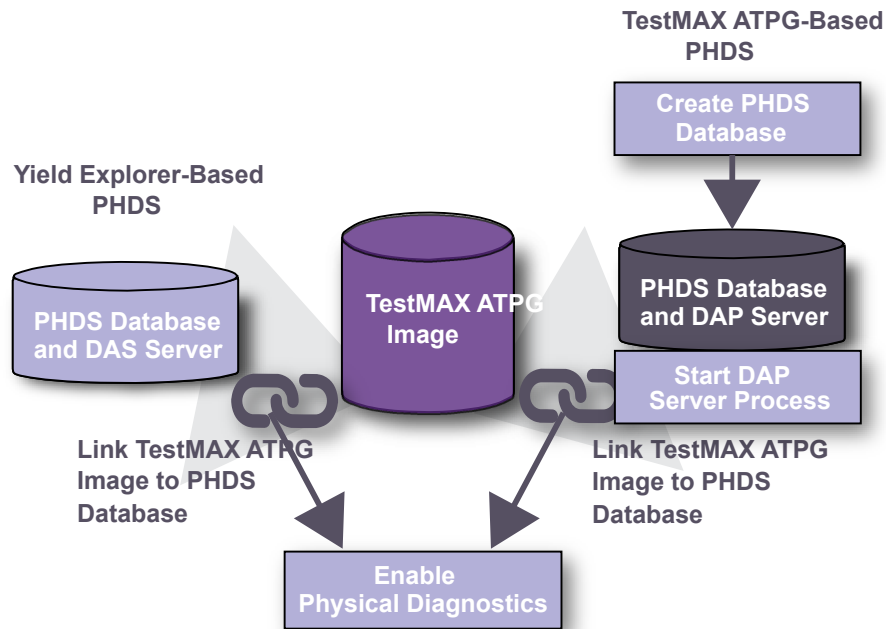
See Also

- [Physical Diagnosis Flow Overview](#)
- [Reading a PHDS Database](#)
- [Setting Up and Running Physical Diagnosis](#)

Reading a PHDS Database into TestMAX ATPG

This section describes how to set up and read a PHDS database for use in TestMAX ATPG. The initial setup steps for the flow are different depending on whether you use Yield Explorer or TestMAX ATPG to create a PHDS database. The following figure shows how to read a PHDS database.

Figure 110 Reading a PHDS Database for Physical Diagnosis



The following sections describe how to use TestMAX ATPG to read a PHDS database for physical diagnosis:

- [Starting and Stopping the DAP Server Process](#)
- [Setting Up a Connection to the PHDS Database](#)

See Also

- [Using TestMAX ATPG to Create a PHDS Database](#)
- [Setting Up and Running Physical Diagnosis](#)

Starting and Stopping the DAP Server Process

The DAP (Data Access Process) server process is used to access a PHDS database created by TestMAX ATPG. You must start this process before you can access and query a PHDS database created by TestMAX ATPG.

To start the DAP server process:

1. Identify the location of the PHDS database.

```
set_physical_db -database ./phds
```

2. Specify any available port on the host in which TestMAX ATPG is currently running.

```
set_physical_db -port_number 9990
```

The default, 9998, is used if this command is not specified.

3. Start the data access process.

```
open_physical_db
```

When the DAP server process starts, the following message prints:

```
Starting Data Access Process...
Hostname           : ighost101
Port Number        : 9990
Physical Database Directory: ./phds
Successfully started Data Access Process.
```

If the process is already running, you will see the following message:

```
Starting Data Access Process...
Hostname           : ighost101
Port Number        : 9990
Physical Database Directory: ./phds
Data Access Process is already running.
```

To stop the DAP server process, specify the `close_physical_db` command:

```
BUILD-T> close_physical_db
Stopping Data Access Process...
Hostname           : ighost101
Port Number        : 9990
All kernel objects removed. Exiting the process...
Successfully stopped Data Access Process.
```

The PHDS database created from TestMAX ATPG uses the DAPListener process. To keep the server process alive, make sure you exit the current TestMAX ATPG session. The DAPListener process halts if TestMAX ATPG runs in the background. You can perform this operation from any TestMAX ATPG mode (BUILD, DRC or TEST).

Setting Up a Connection to the PHDS Database

Before performing physical diagnostics, you must establish a connection between the TestMAX ATPG logical image and the PHDS database. To create this link:

1. Make sure the appropriate design image is loaded in DRC or TEST mode in your current TestMAX ATPG session.

```
TEST-T> read_image i044_image.dat
```

2. Use the `set_physical_db` command to identify the hostname and port number of the PHDS server containing the PHDS database.

```
TEST-T> set_physical_db -hostname ighost101 -port_number 9990
```

```
Setting host name ('ighost101') for physical connection. Setting  
port number ('9990') for physical connection. Connecting to physical  
database. Successfully connected to physical database. Available  
Devices: ----- DES 1 DES 2 DES 3 DES 4 TST 1
```

Note the following:

- If the connection is successful, a list of available devices is printed, as shown in the example.
 - You should always specify the port number when connecting to an existing DAP. The default is not used in this case.
 - If you are diagnosing different designs at the same time, make sure you assign a different port number for each design image.
3. If you are using an Oracle-based PHDS database created by Yield Explorer, you must include a user name and password to establish a connection.

```
TEST-T> set_physical_db -hostname ighost101 \ -port_number 9990 -user  
tester -password safel234 Setting user name ('tester') for physical  
connection. Setting password ('safel234') for physical connection.  
Setting host name ('ighost101') for physical connection. Setting  
port number ('9990') for physical connection. Connecting to physical  
database. Successfully connected to physical database. Available  
Devices: ----- DES 1 DES 2 DES 3 DES 4 TST 1
```

4. Use the `-device` option of the `set_physical_db` command to specify the current device and version.

```
TEST-T> set_physical_db -device "DES 4" Connecting to physical
database. Successfully connected to physical database. Setting device
name ('DES') and device version ('4') for physical connection.
```

5. Use the `-top_design` option of the `set_physical_db` command to specify the top-level DEF design name.

```
TEST-T> set_physical_db -top_design top_def_design_name
```

Name Matching Using a PHDS Database

TestMAX ATPG diagnostics uses physical information mapped to logical instances to improve the accuracy and precision of diagnosis callouts. After diagnosis, TestMAX ATPG writes the physical information for the diagnosis candidates for use in physical failure analysis. This process can be compromised due to logical pin names mismatching with the corresponding physical names in the LEF/DEF database.

To resolve instance name conflicts, matching should be performed on the logical names from the Verilog netlist to the physical names in the LEF/DEF database before running diagnosis. If the logical names and physical names match, name matching rules will be created for later use in diagnosis. The following sections describe how to perform name matching for all instance pins using a PHDS database:

- [Name Matching Overview](#)
- [Understanding the Name Matching Coverage Report](#)
- [Reporting the Name Matching Coverage](#)
- [Using Name Matching Results for Diagnosis](#)

Name Matching Overview

For standard physical diagnosis, TestMAX ATPG diagnostics dynamically searches a PHDS database and matches logical candidate instance names with the corresponding physical names using existing name matching rules. The name matching feature performs this name matching process before running diagnosis. You can then create a set of rules to resolve name mismatches in subsequent diagnosis runs.

To perform name matching, you use both the `set_match_names` and `match_names` commands. The `set_match_names` command specifies the name matching rules you want to apply, if any, and the `match_names` command prints a report of the name matching coverage.

For example, you can use the `match_names` command to create an initial name matching report for a subset of instances. Next, you can use the `set_match_names` command to

specify the replacement of a specific instance prefix with another instance prefix from the flattened logical instance names. You can then perform name matching again to generate a final report that uses the preferred instance prefix.

The name matching report includes pin-level analysis data, hierarchical mismatch behavior, and a name match summary.

Understanding the Name Matching Coverage Report

The `match_names` command creates a name matching report that you can use to analyze the matching of logical candidate instance names with the corresponding physical names.

The following example shows sample output for the `match_names` command:

```
TEST-T> match_names
Setting top level physical design name to 'RISC_CHIP'
Performing Pin Level Analysis
    Matched 564 of 884 instance pins
Checking for logical wrapper
Checking for physical wrapper
Checking for differences in the lowest hierarchy levels
Performing Hierarchy Level Analysis
    Module                               Inst Count Matched Unmatched
Unmatched Names
-----
    STACK_TOP                             1
    0                                     1
-----
Name Match Summary
-----
Number of instance names matched: 564
Number of mismatches found: 320
Percent Correct = 63.80%
CPU_time: 0.02 sec
Query_time: 2.61 sec
Total_time: 2.62 sec
Memory usage summary: 0MB
-----
Closing connection to physical database.
```

Note the following sections of the sample report:

- *Performing Pin Level Analysis* — TestMAX ATPG diagnostics attempts to map every pin in the design to its logical equivalent, and displays the total results.
- *Checking for logical and physical wrappers* — For the logical and physical wrappers, TestMAX ATPG diagnostics attempts to find the top-level hierarchies to explain mismatch behavior.

- *Checking for differences in the lowest hierarchy levels* — TestMAX ATPG diagnostics attempts to explain mismatches at the lowest level hierarchies. The module level results are displayed in descending order relative to the highest number of unmatched names found. After reviewing this report, you should specify a series of `set_match_names` commands to find the correct match for each name.
- *Name Match Summary* — This section summarizes the name matching results. It contains the following fields:
 - *Number of instance names matched* — indicates the number of logical names for which an instance can be found.
 - *Number of mismatches found* — indicates the number of logical names for which no match was found.
 - *Percent Correct* — indicates the final coverage of the name matching process

You can also use the `-auto` option of the `set_match_names` command to perform automatic name matching to resolve hierarchy conflicts.

Reporting the Name Matching Coverage

TestMAX ATPG diagnostics creates a coverage report that displays the success of the static name matching process between the logical and physical names. The flow for this process is as follows:

1. Start TestMAX ATPG.

For details, see [Invoking TestMAX ATPG](#).

2. Read the design image.

```
read_image design.img.gz
```

3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998 set_physical_db  
-top_design top_design_name set_physical_db -device [list  
"Device_name" "1"]
```

4. Use the `match_names` command to perform name matching for a subset of the instances.

```
match_names -sample 1
```

This command reports name matching for 1% of the logical names.

5. Use the `set_match_names` command to specify the name matching rules, if needed.

```
set_match_names -sub_prefix [list "dut/" ""]
```

Note that the physical names include an extra level of hierarchy (`dut/`). The

`set_match_names` command removes this string from the physical names and finds a match with the logical names.

6. Perform name matching again to get the final report.

```
match_names -sample 1
```

Using Name Matching Results for Diagnosis

You can use the name matching flow to identify logical to physical naming conflicts and create appropriate name matching rules that you can use later in diagnostics. The flow consists of the following steps:

1. Start TestMAX ATPG.

For details, see [Invoking TestMAX ATPG](#).

2. Read the design image.

```
read_image design.img.gz
```

3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998 set_physical_db  
-top_design top_design_name set_physical_db -device [list  
"Device_name" "1"]
```

4. Use the following command to automatically create the name matching rules (optional).

```
set_match_names -auto
```

5. Use the `match_names` command to perform name matching for all instances:

```
match_names
```

6. Use the following command to view the automatically created match name rules (optional):

```
report_settings match_names
```

7. Based on the remaining mismatches, use the `set_match_names` command to specify the name matching rules, then rerun the `match_names` command, as shown in Step 5.

```
set_match_names -sub_str [list "dut_0/" "DUT0/"]
```

8. Restart TestMAX ATPG.

9. Read the design image.


```
read_image design.mapped.img.gz
```

10. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998 set_physical_db  
-top_design top_design_name set_physical_db -device [list  
"Device_name" "1"]
```

11. Read the patterns into an external buffer.

```
set_patterns -external design.pat.bin.gz
```

12. Define the name matching rules. For example:

```
set_match_names -sub_prefix {top_i i_core}
```

13. Run the diagnosis

```
run_diagnosis design.datalogs
```

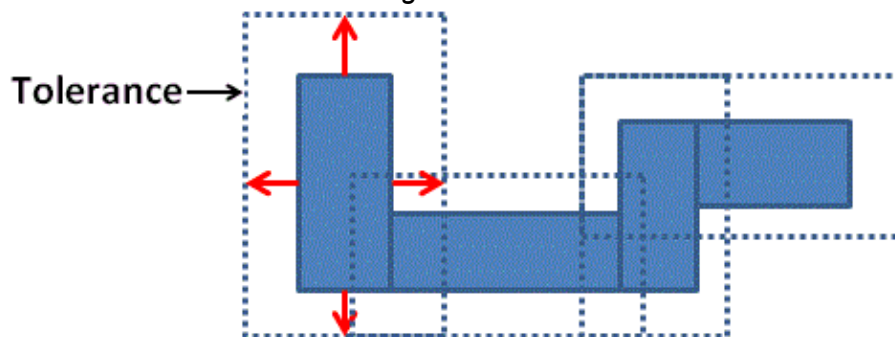
Setting Up and Running Physical Diagnosis

To perform physical diagnostics, you first need to extract the physical data structures from the PHDS database. You can then perform diagnostics using the `run_diagnosis` command.

You can use the `set_physical -tolerance` command and the `set_physical_db -device` command to specify a series of parameters for extracting specific types of data from the PHDS database.

When extracting bridges, TestMAX ATPG searches and extracts neighbor nets based on a default distance per layer tolerance. This tolerance is measured from the boundary of the net, as shown in the following figure.

Figure 111 Net Tolerance for Bridge Extraction



The tolerance distance is equal to the pitch if this data exists in the technology information. If not, the default is 1000nm. You can determine the appropriate tolerance by analyzing technology data, such as pitch distance. To set a tolerance level, use the

`-tolerance` option of the `set_physical` command.

Running Physical Diagnosis

The following steps describe how to set the extraction parameters from the PHDS database, extract physical data, run physical diagnostics, and write the physical data for Yield Explorer:

1. Use the `set_physical_db -device` command to query the PHDS database for technology information, including routing layers and tolerances for each layer.

```
set_physical_db -device [list "RISC" "1"] Connecting to physical
database. Successfully connected to physical database. Setting device
name ('RISC') and device version ('1') for physical connection.
Retrieving layers and tolerance values for device ('RISC') and device
version ('1') Layer Tolerance ----- METAL 410 METAL2 410
METAL3 410 METAL4 515 METAL5 810 METAL6 970
```

2. If required, use the `set_physical -tolerance` command to specify a tolerance for extracting neighbor nets for specific layers. Use the Tcl list syntax to specify each layer and its tolerance setting, as shown in following example:

```
set_physical -tolerance [list METAL 50 METAL2 100 METAL3 200 \ METAL4
300 METAL5 400]
```

3. Perform physical diagnosis on the PHDS database using the `run_diagnosis` command. For example:

```
run_diagnosis /project/mars/lander/chipA_failure.dat
```

4. Use the `write_ydf` command to write the physical data, as shown in the following example:

```
write_ydf chipA.ydf -candidate -append
```

When running physical diagnostics, TestMAX ATPG dynamically retrieves the physical data based on the instance names. If a match exists, TestMAX ATPG accesses the physical data. You can also perform match naming using the physical IDs created before running diagnostics. For more information on this process, see [Static Subnet Extraction Using a PHDS Database](#).

See Also

- [Using TestMAX ATPG to Create a PHDS Database](#)
- [Reading a PHDS Database](#)

Static Subnet Extraction Using a PHDS Database

You can improve the runtime for physical diagnostics by statically extracting subnet information from a PHDS database before running diagnostics. The default flow, dynamic subnet extraction, is performed during diagnostics. Static subnet extraction is only recommended when you run volume diagnostics on a large number of failing parts. Otherwise, the additional runtime required for static extraction is greater than the total reduction in diagnosis runtimes.

The static subnet extraction flow consists of the following steps:

1. Start TestMAX ATPG.

For details, see [Starting TestMAX ATPG](#).

2. Read the logical image.

```
read_image original.img.gz
```

3. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998 set_physical_db  
-top_design top_design_name set_physical_db -device [list  
"Device_name" "1"]
```

4. Perform extraction of all subnet information. At the end of the extraction process, all subnet data is saved in the TestMAX ATPG database.

```
extract_nets -all
```

Only driver pins with more than two fanouts are extracted since they are the only pins with subnets. Subnets from driver pins are extracted in groups of 500 to minimize server overloading.

5. Report statistics, as needed, for the extracted subnets (optional).

The following example reports statistics for a design in which 286 nets have a subnet:

```
TEST-T> report_layout -summary Subnets : #nets=286, #subnets=1191,  
max_subnets=51, memory=0MB Subnets_distribution: <10 (88.46%)  
<20 (98.25%) <30 (98.95%) <50 (99.65%) <60 (100.00%) Receivers_per_net:  
<10 (84.62%) <20 (97.90%) <30 (98.95%) <50 (99.65%) <60 (100.00%)
```

6. Write the physical image containing the subnet information.

```
write_image new.img.gz -compress gzip -replace
```

7. Exit TestMAX ATPG.

```
exit
```

8. Start TestMAX ATPG.

For details, see "[Starting TestMAX ATPG.](#)"

9. Read the physical image.

```
read_image new.img.gz
```

10. Connect to an existing PHDS database.

```
set_physical_db -hostname host01 -port_number 9998 set_physical_db  
-top_design top_design_name set_physical_db -device [list  
"Device_name" "1"]
```

11. Set diagnostics to query only candidate physical information and bridging information.

```
set_diagnostics -use_phds [list candidates bridges]
```

12. Perform Diagnosis.

```
run_diagnostics fail.log
```

13. Exit TestMAX ATPG.

```
exit
```

Reporting Physical Subnet ID Data

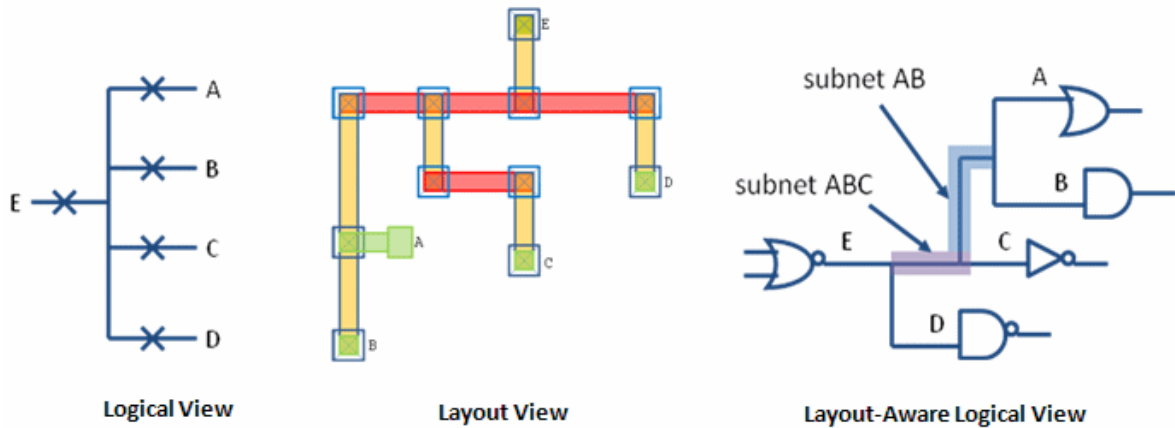
You can use TestMAX ATPG diagnostics with a PHDS database to extract the physical structure of nets in a design. TestMAX ATPG diagnostics uses this subnet data for any net that has more than two fanouts. When diagnostics is performed using the subnet data, the open fault candidates localized on a subnet are reported with the physical ID of the subnet when using the class-based candidate organization.

TestMAX ATPG diagnostics normally uses the PHDS database of the design to extract the subnet data. TestMAX ATPG reads in the extracted subnet data and maps the physical subnet to the logical image. If the subnet is an intermediate branch of the net, it is assigned a logical subnet ID. All branches of the net, including stem branches connected to the driver and receivers, are assigned a physical subnet ID.

The ability to display this type of information improves the precision of the diagnostics results because the physical failure analysis (PFA) area encompasses a much smaller portion of the net.

Understanding Physical Subnet ID Data

The following figure shows the logical, layout, and layout-aware logical views of the same net.



If a fault is localized in subnet ABC, it can be described by the actual subnet definition, and the subnet ID is reported in the diagnostics report. However, if a fault is localized in the metal segments of fanout C, the actual subnet definition is not sufficient. However, the physical subnet ID can address this situation.

The physical subnet ID begins to count from 0, starting with the driver. This means that all metal segments that belong to the driver side of the net are assigned the physical subnet ID of 0. The first fanout metal segments (in this case A), are assigned a physical subnet ID of 1. The physical subnet ID assignment process continues successively until all portions of the net, including the subnets, have received an ID.

The subnet definition of the net displayed in Figure 1 is described in the following table. The left column contains the regular subnet definition. The middle column contains the logical subnet IDs. The right column displays the physical subnet IDs.

Subnet Definition	Logical Subnet Ids	Physical Subnet Ids
.net		
dut/impl/hier3/U12/E	-----	0
dut/impl/hier3/U27/A	-----	1
dut/impl/hier3/U21/B	-----	2
dut/impl/hier3/U72/C	-----	3
dut/impl/hier3/U9/D	-----	4
.subnets		
1 2 3	1	5
1 2	2	6

Note the following:

- The physical subnet IDs are the same IDs reported in the Yield Explorer data file (YDF). If a net does not have a subnet definition, the physical subnet ID cannot be displayed.
- TestMAX ATPG reports the physical subnet ID to match the contents of the YDF. When using the fault-based candidate organization, this can be enabled with the `set_diagnosis -show physical_subnet_id` command. This option is not necessary with the class-based organization.
- Physical subnet ID data is reported only for nets that have been extracted successfully and are in the subnet file. This means that nets with only one or two fanouts are not included. Physical subnet IDs for nets with three or more fanouts that cannot be extracted are also not reported.
- When the fault candidates are cell inputs, and a physical subnet ID must be reported, TestMAX ATPG changes the instance name of the cell input to the instance of the cell which is driving the cell input.

Writing Physical Data for Yield Explorer

After completing the physical diagnostics process, you can write the diagnostics candidates, and all related physical information, to a Yield Explorer Data Format (YDF) file. Yield Explorer uses this file for volume diagnostics analysis.

To write physical data for Yield Explorer, specify the `write_ydf` command after each

`run_diagnosis` command. The `write_ydf` command must include the name of the YDF file. You can specify this command using the `-candidates` option or without any options. For example:

```
run_diagnosis /project/mars/lander/chipA_failure.dat  
write_ydf top_chip.ydf -candidates
```

The `write_ydf` command prints the physical data for the diagnostics candidates in all tables in the output YDF file. This data enables Yield Explorer to perform volume diagnostics analysis. You should use the `-replace` option if you want to create a new file to store each diagnostics candidate. You can use the `-append` option if you want to store all candidates in a single file. The following example reports the physical data elements for the diagnostics candidates to a single file:

```
write_ydf top_chip.ydf -candidates -append
```

By default, the `write_ydf` command does not write the chain definition table. To write this table, which displays chain definition data for the entire database, you must specify the

`-chain_def` option. You only need to write this table one time, as shown in the following example:

```
write_ydf top_chip.chain_def.ydf -chain_def
```

You might prefer to write individual tables to a specific file. If you specify one or more physical data options, TestMAX ATPG reports only the physical data tables related to the specified options.

If you specify the following example, TestMAX ATPG reports only the physical data for the vias and the LEF macro cells in their respective tables:

```
write_ydf top_chip.ydf -replace -via -cell
```

See Also

- [Using TestMAX ATPG to Create a PHDS Database](#)
- [Reading a PHDS Database](#)

19

Power Aware ATPG

A typical ATPG run targets as many faults as possible within a particular pattern. However, this approach can cause unintended ATE failures for designs containing a large number of flip-flops that toggle at any given time.

The TestMAX ATPG power aware ATPG feature calculates the fanout of clock-gating structures and other clock sources during DRC. This approach enables you to specify capture and shift power budgets for generating power aware ATPG vectors. You can specify a budget as a percentage of scannable flip-flops and thereby limit the number of flip-flops that can toggle.

To ensure optimal accuracy, the `report_power` command uses the TestMAX ATPG threaded simulator to resimulate previously generated patterns. Consequently, the `set_atpg -calculate_power` command is ignored by TestMAX ATPG.

TestMAX ATPG also supports hardware-assisted shift power reduction. This methodology decreases average shift power and pattern count compared to an ATPG-only approach by using independently controlled scan chain groups implemented in DFTMAX or DFTMAX Ultra.

TestMAX ATPG lowers the overall peak and average flip-flop switching by selectively turning on and off the respective clock-gating cells which control the flip-flops. This selective switching affects capture for stuck-at testing and launch and capture for transition fault testing.

Power aware ATPG is not intended to be used for power analysis. TestMAX ATPG efficiently estimates the relative power of test patterns, which generally correlates well with actual power consumption. However, this approach is not a precise calculation of the actual power metrics. Performing a full power analysis during ATPG causes an unacceptable increase in runtime and is therefore not used for power aware ATPG.

The following sections describe how to prepare for and use power aware ATPG:

- [Input Data Requirements](#)
- [Setting a Power Budget](#)
- [Preparing Your Design](#)
- [Running Power Aware ATPG](#)

- [Applying Quiet Chain Test Patterns](#)
- [Testing with Asynchronous Primary Inputs](#)
- [Power Reporting By Clock Domain](#)
- [Setting a Capture Budget for Individual Clocks](#)
- [Testing for Partitions](#)
- [Retention Cell Testing](#)
- [Power Aware ATPG Limitations](#)

Input Data Requirements

The following input data is required to use the power aware ATPG feature within TestMAX ATPG:

- Netlists
- Library
- STIL procedure file
- Tcl command script containing the `build`, `run_drc`, `run_atpg` and other commands.

Setting a Power Budget

To run power aware ATPG, you need to set a power switching budget using the

`-power_budget` option of the `set_atpg` command. You can specify the power switching budget using either of the following methods:

- Specify the maximum percentage of scannable flip-flops that are budgeted to change during capture. For example:

```
set_atpg -power_budget 48
```

- Specify the `min` keyword to use the minimum recommended switching budget based upon the clock-gating analysis. For example:

```
set_atpg -power_budget min
```

You can set the power switching budget any time before running the `run_atpg` command.

For complete information on how to determine the power switching budget, see the following section, [Preparing Your Design](#).

Preparing Your Design

Power aware ATPG is intended for designs that contain clock-gating cells in the context of ATPG. You will need to perform an initial analysis of your design to identify all clock-gating cells and calculate the recommended setting for the `set_atpg -power_budget` command.

The power analysis performed by TestMAX ATPG uses information from the STL procedure file and data specified by the `add_pi_constraints` command. If your design has a constraint in which the clock-gating cells are always transparent, this power analysis will not show these clock-gating cells and they are not usable within the context of power aware ATPG. This means you need to constrain scan-enable ports to their respective off-state for basic-scan, two-clock, and fast-sequential modes for test pattern generation and gated-clock (latch) identification.

Also note the following:

- All global signals capable of enabling a large proportion of the clock gating cells must be disabled.
- All synchronous set and reset signals described as clocks with TestMAX ATPG must be inactive or constrained to their respective off-state.

Reporting Clock-Gating Cells

After your design successfully passes the DRC process, use the

`report_clocks -gating -verbose` command to report the clock-gating cells and calculate the recommended low-power ATPG budget percentage, as shown in the following example:

```
report_clocks -gating -verbose
Clock name: ife_clockdiv2_afe_wrap (0)
Number of cells directly controlled by the clock: 12077 (22.33%)

Number of cells controlled by clock through
a clock gating latch16605 (30.70%)

Number of cells directly controlled by clock + largest
clock gating domain: 12097 (22.36%)

Clock Gating Latch DecoderFrontEnd1/fedcod/dcod_yc/ \
clk_gate_ramAddr_regx0x/U1 (693893)
drives 20 (0.04%) scan cells
...
...
Minimum Recommended Low-Power ATPG Budget: 22.36% (12097)
```

You should round-up the recommended low-power ATPG budget percentage to the next integer value. In the previous example, 22.36 should be rounded up to 23. This value is specified by the `-power_budget` option of the `set_atpg` command using either of the two methods:

- You can manually specify the budget as shown in the following example:

```
set_atpg -power_budget 23
```

- You can specify TestMAX ATPG to automatically use the minimum recommended low-power ATPG budget, as shown in the following example:

```
set_atpg -power_budget min
```

Constraining Clock-Gating Cells for Power Aware ATPG

After the DRC process completes its analysis, a list of clock-gating cells are reported. You need to constrain these cells to their opposite value so that capture power yields optimal power budget results.

It is also recommended to do this when using SPC (Shift Power Controller) with compression if a pattern inflation is seen after enabling SPC.

```
Begin clock-gating analysis...
7578 ATPG controllable clock-gating cells were found
Setting top_inst/core_1/icg_te_data_reg(id=2620544) to 1 allows 73.75% of
all scan cells to toggle.
Clock-gating analysis completed, CPU time=0.00 sec.
-----
-----
Begin clock rules checking...
Clocks successfully passed off-state check of scan cells.
```

In this case you should add a cell constraint of 0 on this cell `top_inst/core_1/icg_te_data_reg` since this cell drives over 70 percent of TE pins of the `clock_gating` cells and leaving it a 1 would lead to all of these cells having a potential active clock. Adding a cell constraint to hold this cell inactive will then help ATPG achieve the described power budget sooner.

The suggestion is to rerun the DRC step after adding

```
add_cell_constraints 0 top_inst/core_1/icg_te_data_reg
```

and then compare pattern count and coverage for when power-aware ATPG is being used with and without this constraint.

In the following example, DRC reports a set of clock-gating cells:

```
Scan chain operation checking completed, CPU time=51.51 sec.
-----
```

```
Begin clock-gating analysis...
106438 ATPG controllable clock-gating cells were found
Setting top/test/clk_always_0_clock_gate_reg35(id=2599) to 1 allows
 15.54% of all scan cells to toggle.
Setting top/test/clk_always_0_clock_gate_reg28(id=2506) to 1 allows
 19.50% of all scan cells to toggle.
Setting top/test/clk_always_0_clock_gate_reg22(id=2512) to 1 allows
 12.62% of all scan cells to toggle.
Setting top/test/clk_always_0_clock_gate_reg13(id=2521) to 1 allows
 12.61% of all scan cells to toggle.
Clock-gating analysis completed, CPU time=105.17 sec.
-----
Begin clock rules checking...
```

For capture power to work properly, you need to set the signals in the previous example to 0, as shown in the following example:

```
add_cell_constraints 0 top/test/clk_always_0_clock_gate_reg35
add_cell_constraints 0 top/test/clk_always_0_clock_gate_reg28
add_cell_constraints 0 top/test/clk_always_0_clock_gate_reg22
add_cell_constraints 0 top/test/clk_always_0_clock_gate_reg13
```

Setting a Strict Power Budget

Use the `-power_effort` option of the `set_atpg` command to generate patterns that do not exceed the power budget specified for capture. The syntax for this option is as follows:

```
set_atpg -power_effort <high | low>
```

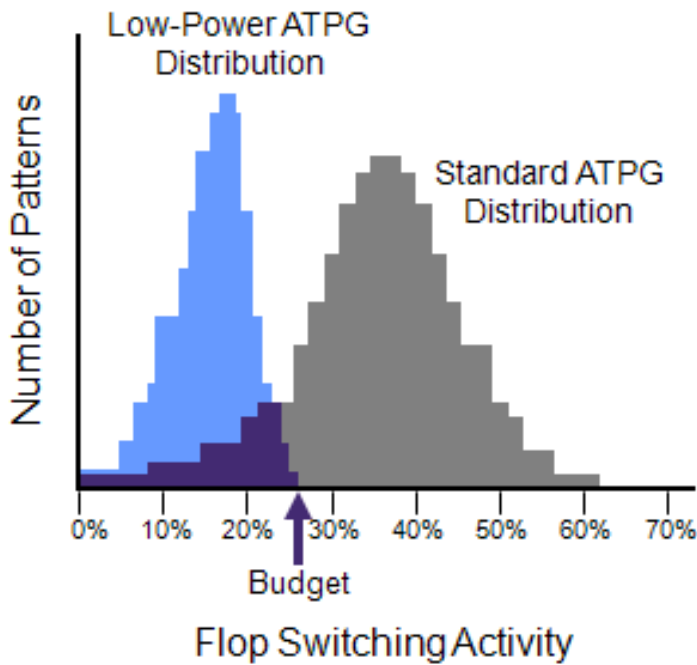
The default is `low`. If you set this option to `high`, TestMAX ATPG generates patterns that do not exceed the budget specified by the `set_atpg -power_budget` command. Note that over-constraining the power budget might cause longer runtimes and generate fewer patterns when the `-power_effort` option is set to `high`. Because of this, you should not set power budgets below that recommended by the

```
report_clocks -gating command.
```

Running Power Aware ATPG

After preparing your design, as described in the [Preparing Your Design](#) section, you are ready to perform a complete power aware ATPG run and use the `report_power` command to report the power data.

The following example script shows the use of the `set_atpg` and `report_power` commands in a typical power aware ATPG flow.



```
read_netlist -lib $my_lib.v
read_netlist $my_design.v
run_build $my_design
run_drc $my_drc_file.spf
report_clock -gating
set_atpg -fill adjacent
set_atpg -power_budget 25
add_faults -all
run_atpg -auto
report_power -per_pattern -percentage
```

The `report_power` command produces the report shown in the following example:

```
-----
                        Power Analysis Summary
-----
Number of Scan Cells           75053
Number of Patterns            0-2680
Average Shift Changes:        2400.38 3.20%
Average Capture Changes:      9058.04 12.07%
Maximum Shift Cell Changes:    37510 49.97% (pattern: 0 cycle: 3411)
Maximum Capture Cell Changes: 30742 40.96% (pattern: 1)
```

Applying Quiet Chain Test Patterns

Regular scan chain test patterns apply the 0011 sequence to all scan inputs, which can lead to power issues and unintended ATE failures. However, a quiet chain test pattern minimizes switching activity by loading a single scan input with specified pattern data and loads all other chains with a constant value.

The `-quiet_chain_test` option of the `set_atpg` command enables the automatic generation of quiet chain test patterns when the `run_atpg` command is executed.

The `set_atpg -load_mode` command enables the generation of the quiet chain test for a particular load mode or all load modes. The `set_atpg -load_value` command configures the constant value loaded into the quiet chains.

In legacy scan mode, the 0011 sequence, or any other specified pattern data, is independently applied to each scan chain, while all other scan chains are set to 0. This means that one pattern loads the 0011 sequence in a single scan chain at a time. To load N scan chains, where N is the total number of scan chains, TestMAX ATPG generates N quiet chain test patterns.

In scan compression mode, the 0011 sequence or any other specified pattern data is independently applied to each scan channel, while all other scan channels are set to 0. The compressor load mode is maintained to a constant value, which is 0. One scan channel fans to multiple chains due to the input load compressor. Thus, to load the P scan channels, where P is the total number of load compressor scan inputs, TestMAX ATPG generates P quiet chain test patterns.

Testing with Asynchronous Primary Inputs

Use the `-power_aware_asyncs` option of the `set_atpg` command to test asynchronous sets and resets from primary inputs on legacy scan designs. Note that this feature is not implemented for DFTMAX designs.

To use this option, your design must be able to propagate the asynchronous signals to allow sufficient time for the signals to fit within the given ATE vector.

The following example shows the `-power_aware_asyncs` option of the power aware ATPG flow:

```
...
run_drc
...
set_atpg -power_aware_asyncs
run_atpg -auto
report_power -per_pattern -percentage
```

Power Reporting By Clock Domain

You can set the `-per_clock_domain` option of the `report_power` command to create individual capture power reports for each clock. By default, the `report_power` command creates a consolidated report for all clock domains.

The following example shows the type of report created using the `-per_clock_domain` option.

```
report_power -percentage -per_pattern -per_clock_domain
```

```
-----
Power Analysis: Per Pattern
-----
```

Shift Results:

Peak

pattern	load cycle	shift cycle	switching	percentage
0	0	87	344	48.93%
1	0	35	361	51.35%
2	0	5	341	48.51%
3	0	80	351	49.93%
4	0	11	337	47.94%
5	0	23	343	48.79%
6	0	64	340	48.36%
7	0	75	361	51.35%
8	0	6	365	51.92%
9	0	48	348	49.50%

Average

pattern	average switching	percentage
0	171.51	24.40%
1	348.91	49.63%

2	326.01	46.37%
3	338.85	48.20%
4	327.02	46.52%
5	328.31	46.70%
6	328.89	46.78%
7	343.67	48.89%
8	349.93	49.78%
9	339.20	48.25%

Capture Results:

Peak

pattern	capture cycle	switching	percentage
0	0	0	0.00%
1	2	68	9.67%
2	1	52	7.40%
3	1	54	7.68%
4	1	25	3.56%
5	1	16	2.28%
6	1	30	4.27%
7	1	26	3.70%
8	1	47	6.69%
9	1	60	8.53%

Average

pattern	average switching	percentage
0	0.00	0.00%
1	41.67	5.93%
2	27.33	3.89%

3	30.33	4.31%
4	13.33	1.90%
5	7.00	1.00%
6	13.00	1.85%
7	14.00	1.99%
8	26.00	3.70%
9	32.67	4.65%

Capture Results For Clock CLK1:

Peak

pattern	capture cycle	switching	percentage
0	0	0	0.00%
1	0	0	0.00%
2	0	0	0.00%
3	0	0	0.00%
4	0	0	0.00%
5	0	0	0.00%
6	0	0	0.00%
7	0	0	0.00%
8	1	19	2.70%
9	0	0	0.00%

Average

pattern	average switching	percentage
0	0.00	0.00%
1	0.00	0.00%
2	0.00	0.00%
3	0.00	0.00%

4	0.00	0.00%
5	0.00	0.00%
6	0.00	0.00%
7	0.00	0.00%
8	12.33	1.75%
9	0.00	0.00%

Capture Results For Clock CLK2:

Peak

pattern	capture cycle	switching	percentage
0	0	0	0.00%
1	0	0	0.00%
2	1	32	4.55%
3	0	0	0.00%
4	0	0	0.00%
5	0	0	0.00%
6	0	0	0.00%
7	1	26	3.70%
8	0	0	0.00%
9	0	0	0.00%

Average

pattern	average switching	percentage
0	0.00	0.00%
1	0.00	0.00%
2	14.00	1.99%
3	0.00	0.00%
4	0.00	0.00%

5	0.00	0.00%
6	0.00	0.00%
7	14.00	1.99%
8	0.00	0.00%
9	0.00	0.00%

Capture Results For Clock CLK3:

Peak

pattern	capture cycle	switching	percentage
0	0	0	0.00%
1	1	36	5.12%
2	0	0	0.00%
3	1	23	3.27%
4	1	25	3.56%
5	1	16	2.28%
6	1	30	4.27%
7	0	0	0.00%
8	0	0	0.00%
9	1	23	3.27%

Average

pattern	average switching	percentage
0	0.00	0.00%
1	22.67	3.22%
2	0.00	0.00%
3	13.00	1.85%
4	13.33	1.90%
5	7.00	1.00%

6	13.00	1.85%
7	0.00	0.00%
8	0.00	0.00%
9	11.33	1.61%

...
...
...

Power Analysis Summary

Number of Scan Cells	703
Number of Patterns	0-9
Cycles Per Load	88
Average Shift Switching	320.23 45.55%
Average Capture Switching	22.00 3.13%
Peak Shift Switching cycle: 6)	365 51.92% (pattern: 8
Peak Capture Switching	68 9.67% (pattern: 1)
Peak Capture Switching (CLK1)	19 2.70% (pattern: 8)
Peak Capture Switching (CLK2)	32 4.55% (pattern: 2)
Peak Capture Switching (CLK3)	36 5.12% (pattern: 1)
Peak Capture Switching (CLK4)	0 0.00% (pattern: 0)
Peak Capture Switching (CLK5)	0 0.00% (pattern: 0)
Peak Capture Switching (CLK6)	31 4.41% (pattern: 3)
Peak Capture Switching (CLK7)	37 5.26% (pattern: 9)
Peak Capture Switching (CLK8)	0 0.00% (pattern: 0)

Peak Capture Switching (CLK9)	36 5.12% (pattern: 1)
Peak Capture Switching (CLK10)	0 0.00% (pattern: 0)
Peak Capture Switching (CLK11)	28 3.98% (pattern: 8)
Peak Capture Switching (SETN)	0 0.00% (pattern: 0)
Peak Capture Switching (RSTN)	0 0.00% (pattern: 0)

Setting a Capture Budget for Individual Clocks

You can use the `-power_budget` and `-domain` options of the `set_atpg` command to set a capture budget for individual clocks. You must specify the `-power_budget` option before the `-domain` option.

The following example sets a capture budget of 55 for clock1:

```
set_atpg -power_budget 55 -domain clock1
```

The next example sets a capture budget of 15 to the clock2 and clock3 clock domains:

```
set_atpg -power_budget 15 -domain {clock2 clock3}
```

The next example assigns the minimum recommended capture budget for clock4:

```
set_atpg -power_budget min -domain clock4
```

The next example assigns the minimum recommended capture budget for internal clock domains. These would be the clock names listed by `report_clocks -intclocks`

As they are set, the tool echos the minimum percentage of capture budget per internal clock domain:

```
set_atpg -power_budget min -domain codec/occl1/U2/U3  
Setting power budget for clock codec/occl1/U2/U3 to 23%  
set_atpg -power_budget min -domain codec/occl2/U2/U3  
Setting power budget for clock codec/occl2/U2/U3 to 91%  
set_atpg -power_budget min -domain codec/occl3/U2/U3  
Setting power budget for clock codec/occl3/U2/U3 to 89%  
set_atpg -power_budget min -domain codec/occl4/U2/U3  
Setting power budget for clock codec/occl4/U2/U3 to 87%  
set_atpg -power_budget min -domain codec/occl5/U2/U3  
Setting power budget for clock codec/occl5/U2/U3 to 80%
```

Note:

It's recommended to use `set_atpg -power_budget min -domain name` to get the minimum setting per clock domain as `report_clock -gating` does not list

minimum recommended settings. You can then set the power budget to any number larger than the reported minimum setting.

After setting a capture budget for the individual clock domains, you can produce a power report using the `-per_clock_domain` option of the `report_power` command, as shown in the following example:

```
report_power -per_clock_domain

-----
Power Analysis Summary
-----
Number of Scan Cells          54093
Number of Patterns           0-64
Cycles Per Load              52
Average Shift Switching      12939.91 23.92%
Average Capture Switching    7867.95 14.55%
Peak Shift Switching         18373 33.97% (pattern: 5
  cycle: 12)
Peak Capture Switching       10880 20.11% (pattern: 11)
Peak Capture Switching (clk1) 9860 18.23% (pattern: 16)
Peak Capture Switching (clk2)  154 0.28% (pattern: 62)
Peak Capture Switching (clk3) 6160 11.39% (pattern: 26)
Peak Capture Switching (clk4) 1389 2.57% (pattern: 23)

report_power -per_clock_domain -percentage -capture
Starting threaded simulation with 8 threads. (M733)
-----
-----
Power Analysis Summary
-----
-----
Number of Scan Cells 261050
Number of Patterns 0-104
Cycles Per Load 301
Average Capture Switching 17961.72 6.88%
Peak Capture Switching 28317 10.85% (pattern: 38)
Peak Capture Switching (codec/occ11/U2/U3) 3323 14.92% (pattern: 14)
Peak Capture Switching (codec/occ12/U2/U3) 2068 46.37% (pattern: 32)
Peak Capture Switching (codec/occ13/U2/U3) 9920 44.63% (pattern: 76)
Peak Capture Switching (codec/occ14/U2/U3) 11192 44.40% (pattern: 9)
Peak Capture Switching (codec/occ15/U2/U3) 2395 56.13% (pattern: 6)
Peak Capture Switching (codec/occ16/U2/U3) 1617 17.24% (pattern: 42)
```

You can use the following command to set all clocks to use the specified capture budget:

```
set_atpg -power_budget {min} -domain [get_attribute \
[get_clocks -all] clock_name]
```

Testing for Partitions

When generating power aware ATPG patterns, you can set the target test coverage, capture power, and shift power on a per partition basis and get reports on individual partition contributions to power.

Specifying a Test Coverage Target for Partitions

To specify the test coverage target for specific partitions, use the `-coverage` option of the `set_atpg` command with the `-partition` option. For example:

```
set_atpg -coverage 90 -partition {p1 p2}
```

Specifying Capture Power for Partitions

You can use the `-power_budget` and `-partition` options of the `set_atpg` command to set capture budgets for different partitions. You must specify the `-power_budget` option before the `-partition` option. Partitions can be created using the `add_partition` command and should be defined before using the `-partition` option with `set_atpg`. The `-partition` option allows you to specify a list of one or more partition names.

The following example sets a capture budget of 20 for partitions p1 and p2:

```
set_atpg -power_budget 20 -partition {p1 p2}
```

Specifying Shift Power for Partitions

To set the shift power for specific partitions, use the `-shift_controller_peak` option of the `set_atpg` command with the `-partition` option. For example:

```
set_atpg -shift_controller_peak 30 -partition {p1 p2 p3}
```

Note: If you do not use the `-partition` option, the specified shift power controller (SPC) peak is applied to the entire design (global constraint).

Reporting Power Per Partition

After setting capture budgets for the partitions, you can report the shift and capture power per partition by using the `-per_partition` option of the `report_power` command, as shown in the following example:

```
report_power -shift -per_partition -percent  
report_power -capture -per_partition -percent  
report_power -per_partition -percent
```

Example

The following example shows how to set the test coverage, shift power, and capture power for partitions as well as report the power for individual partitions.

```
read_netlist My_design.v
run_build_model My_Unit

#add partitions
add_partition p1 {A B C}
add_partition p2 {P Q R}
add_partition p3 {X Y Z}

#SPC related constraints and chain name
set_drc -spc_chain SPC
add_pi_constraint 0 SPC_DISABLE
run_drc My_spf.spf

#Set coverage per partition
set_atpg -coverage 95
set_atpg -coverage 95 -partition {p1 p2}
set_atpg -coverage 90 -partition p3

#Set capture power per partition
set_atpg -power_budget 20 -partition p1
set_atpg -power_budget 30 -partition p2
set_atpg -power_budget 25 -partition p3

#set shift power per partition
set_atpg -shift_controller_peak 30
set_atpg -shift_controller_peak 30 -partition {p1 p2 p3}

#Atpg settings
set_faults -model stuck
add_faults -all
run_atpg -auto

#report power per partition
report_power -shift -per_partition -percent
report_power -capture -per_partition -percent
report_power -per_partition -percent

#Report coverage per partition
report_faults -summary
```

Limitations

Specifying power budgets for partitions has the following limitations:

- Chain test patterns are not supported with SPC and partitions. Chain test patterns might exceed the per partition SPC budget.
- Chain diagnosis ATPG patterns are not supported with SPC and partitions.
- Setting the shift power to a very small value can lead to coverage drop and/or pattern inflation.
- Any instance for which a partition is not defined will belong to the default partition. The power budget for the default partition is 100%.
- The test coverage targeted for partitions must be the same and close to the coverage targeted for the top level of the design.

Retention Cell Testing

TestMAX ATPG uses the `chain_capture` procedure to generate tests specifically for retention cells within scan chains. These chain tests apply only two patterns to retention cells.

A regular chain test and a retention cell test have different goals. A regular chain test checks that scan chains shift reliably and that a capture procedure does not corrupt any values when a capture clock does not pulse. A retention cell chain test purposely corrupts data and ensures it is restored to its original state.

TestMAX ATPG supports three flows to generate tests for retention cells:

- *Pattern generation*

This flow uses the `chain_capture` procedure and full-sequential ATPG to generate patterns that are fault-simulated by TestMAX ATPG. In most cases, you should use this flow unless you are limited by machine memory, the `chain_capture` procedure size, or your design size.

- *Pattern formatting*

This flow uses the `chain_capture` procedure, but it does not generate patterns. Instead, it formats patterns based on user-provided specifications. Use this flow when memory capacity limitations prevent you from using a large `chain_capture` procedure or a large design for pattern generation. The patterns formatted in this flow cannot be correctly simulated by TestMAX ATPG because they are not annotated with simulation information.

- *Pattern formatting by masking non-retention cells*

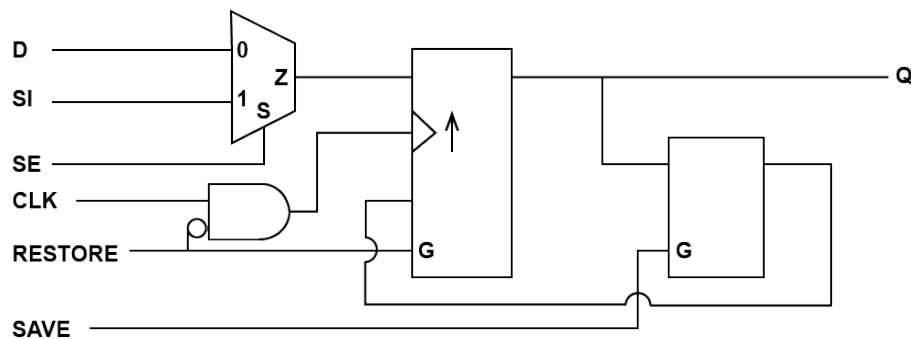
This flow is similar to the pattern formatting flow except you do not explicitly identify retention cells. Instead, you identify all cells that are *not* retention cells by masking their scan-out values. This masking is done for every instance.

The following sections describe how to use TestMAX ATPG to test the unique properties of retention cells:

- [Typical Retention Cell Used for Testing by TestMAX ATPG](#)
- [Creating the chain_capture Procedure](#)
- [Identifying Retention Cells for Testing](#)
- [Pattern Generation for Retention Cells](#)
- [Pattern Formatting for Retention Cells](#)
- [Pattern Formatting by Masking Non-Retention Cells](#)
- [Retention Cell Testing Limitations](#)

Typical Retention Cell Used for Testing by TestMAX ATPG

Scan cells that use SAVE and RESTORE functions are directly addressed by the TestMAX ATPG retention cell testing flows, as shown in the following example.



The behavior of this scan cell is as follows:

- When the SAVE and RESTORE functions are deasserted, the cell behaves as a normal scan flip-flop.
- When the retention function is required, the SAVE sequence stores the flip-flop value in the retention latch.
- After power is restored to the flip-flop portion of the cell, the RESTORE function disables the flip-flop clock and any asynchronous controls. It then loads the value from the retention latch into the flip-flop.

Creating the chain_capture Procedure

TestMAX ATPG uses a special retention cell chain test to handle retention cell testing. This chain test works with the chain_capture procedure in the SPF.

A separate SPF is required for a retention cell test and can include only the load_unload and chain_capture procedures. Do not include any other procedures in this SPF.

TestMAX ATPG initially runs a retention cell chain test using the data specified in the chain_capture procedure. It then reapplies the test with the same data in reverted order. For example, if you specify a repeating 0101 sequence for the chain test, TestMAX ATPG first applies the chain test with a repeating 0101 sequence and reapplies it with a repeating 1010 sequence.

The chain_capture procedure performs the following tasks:

- Saves the scanned-in values by executing the SAVE sequence on the retention cells.
- Corrupts the scanned-in values by clocking in the opposite value from the scan chain. A repeating 0101 value specified for the chain test requires one pulse of the shift clock.
- Restores the original scanned-in values by executing the RESTORE sequence on the retention cells.

The chain_capture procedure is more complex than typical capture procedures, as shown in the following example:

```
"chain_capture" {
  F { "test_mode"=1; _po=\r 78 X; }
  V { "scan_enable"=0; "ret_clk"=0; "reset"=0; "clk"=0; }

  // Execute the SAVE sequence.
  V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
  V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
  V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
  V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }

  // Pulse the capture clock after the data is saved.
```

```
V { "ret_clk"=0; "clk"=P; }

// Execute the RESTORE sequence.
V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; "clk"=0; }
V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
V { "ret_in"=0; "scan_enable"=1; "ret_clk"=P; }
V { "ret_in"=1; "scan_enable"=1; "ret_clk"=P; }
}
```

Identifying Retention Cells for Testing

To identify retention cells for testing, you must include two special notations in the cell library file: ``define retention` and ``undef retention`. The ``define retention` notation must be placed before the cell definition. This enables TestMAX ATPG to identify cells that will retain value after the retention cell test.

The ``undef retention` notation is placed after the retention cell definition and prevents identifying the next cell in the cell library as a retention cell.

The following example shows a snippet of a cell model that uses the ``define retention` and ``undef retention` notations:

```
`define retention
`celldefine
module DFFS_RETENTION (D, CLK, SAVE, RESTORE, SI, SE);
.... <contents of cell model here> ....
endmodule
`endcelldefine
`undef retention
```

If you cannot identify retention cells using the `'define retention` and `'undef retention` statements, see [Pattern Formatting by Masking Non-Retention Cells](#).

Pattern Generation for Retention Cells

The following steps show how to use the pattern generation flow.

1. Create the SPF and include the `chain_capture` procedure (see [Creating the chain_capture Procedure](#)).
2. Identify the retention cells in the cell library file (see [Identifying Retention Cells for Testing](#)).
3. Use the following command sequence to set up and run test DRC:

```
set_drc -clock -chain_capture run_drc SPF_filename
```

Note the following:

- The `-clock` option of the `set_drc` command specifies restrictions on clock usage for pattern generation; the `-chain_capture` option sets the DRC process to use the retention cell chain test.
 - The `run_drc SPF_filename` command runs DRC using the specified SPF.
4. Use the `-chain_test` option of the `set_atpg` command to specify the chain test pattern (the 0101R sequence is recommended), as shown in the following example:

```
set_atpg -chain_test 0101R
```

This command helps full-sequential ATPG efficiently generate a test.

5. Specify the `run_atpg full_sequential_only` command to run ATPG in full-sequential mode.

```
run_atpg full_sequential_only
```

To create retention patterns, TestMAX ATPG performs the following steps:

- Loads the scan chains with the SLEEP signal turned off.
- Runs the `chain_capture` procedure.
- Unloads the scan chains.

These steps are performed twice. The first run includes the data specified for the normal scan chain test. The second run includes the same data inverted.

6. Set up and run the fault simulation using the `set_simulation` and `run_fault_sim` commands.

TestMAX ATPG fault-simulates the generated patterns and the patterns are retained only if they detect faults. All detected faults are categorized as Detected by Simulation (DS) faults. If the first pattern detects faults but the second pattern does not, only the first pattern is retained.

Pattern Formatting for Retention Cells

You should use the pattern formatting for retention cells flow only if tool capacity issues prevent you from using the pattern generation flow (see [Pattern Generation for Retention Cells](#)). Because TestMAX ATPG does not simulate the `chain_capture` procedure in this flow, you should validate the patterns using a Verilog simulator.

The following steps shows how to use the pattern formatting flow.

1. Create the SPF and include the `chain_capture` procedure (see [Creating the chain_capture Procedure](#)).
2. Identify the retention cells in the cell library file (see [Identifying Retention Cells for Testing](#)).
3. Run test DRC using the following command sequence:

```
set_drc -clock -retention_test run_drc SPF_filename
```

The `set_drc -clock -retention_test` command sets the DRC process to use the retention cell chain test, and the `run_drc SPF_filename` command runs DRC using the specified SPF.

4. Format the patterns for retention testing.
 - a. Use the `-chain_test` option of the `set_atpg` command to specify the chain test pattern (the 0101R sequence is recommended), as shown in the following example:

```
set_atpg -chain_test 0101R
```

- b. Specify the `run_atpg -only_chain_test` command.

```
run_atpg -only_chain_test
```

This command writes the chain test into the pattern buffer and calls the `chain_capture` procedure. The chain test is performed twice: First with the data specified for the normal scan chain test and a second time with the same data inverted.

5. Validate the patterns using a Verilog simulator. This process is required because the TestMAX ATPG simulators cannot correctly simulate patterns that are not annotated with simulation information. In this case, the `run_simulation` and `run_fault_sim` commands always report mismatches.

Pattern Formatting by Masking Non-Retention Cells

You might need to use the pattern formatting flow for retention cells, but your cell library does not identify the retention cells (using the ``define retention` and `'undef retention` statements). In this case, you can explicitly identify all cells that are not retention cells by masking their scan-out values. It is crucial to validate these patterns using a Verilog simulator because TestMAX ATPG does not simulate the `chain_capture` procedure and cannot verify if non-retention cells are correctly identified.

The following steps show how to perform pattern formatting by masking non-retention cells.

1. Create the SPF and include the `chain_capture` procedure (see [Creating the chain_capture Procedure](#)).
2. Use the `add_cell_constraints` command to identify all cells in the cell library that are *not* retention cells by masking their scan-out values. This masking is done for every instance and prevents the patterns from expecting the non-retention cells to restore their value.

```
add_cell_constraints ox { cpu0/ifetch0/u04 cpu0/mmu/u43 cpu0/luu/u63  
cpu0/cdu/u167 }
```

Note that the `ox` constraint indicates that the observed value is always masked or considered to be X.

3. Run test DRC using the following command sequence:

```
set_drc -clock -chain_capture run_drc SPF_filename
```

The `set_drc -clock -chain_capture` command sets the DRC process to use the retention cell chain test. In this case, the `-chain_capture` option is the same as used for the pattern generation flow. But it executes differently than the pattern formatting flow when retention cells are identified in the library. The `run_drc SPF_filename` command runs DRC using the specified SPF.

4. Format the patterns for retention testing.

Use the `-chain_test` option of the `set_atpg` command to specify the chain test pattern (the 0101R sequence is recommended), as shown in the following example:

```
set_atpg -chain_test 0101R
```

5. Specify the `run_atpg -only_chain_test` command.

```
run_atpg -only_chain_test
```

This command writes the chain test into the pattern buffer and calls the `chain_capture` procedures. These steps are performed twice: The first run includes the data specified for the normal scan chain test. The second run includes the same data inverted.

6. Validate the patterns using a Verilog simulator. This process is required because the TestMAX ATPG simulators cannot correctly simulate patterns that are not annotated with simulation information. You could run the `run_simulation` and `run_fault_sim` commands, but they would always report mismatches in this case.

Retention Cell Testing Limitations

The following limitations apply to retention cell testing:

- You cannot run retention cell testing using either of the following commands:
 - `run_atpg basic_scan_only`
 - `run_atpg fast_sequential_only`
- The Procedure section of the SPF used for retention cell testing must contain only the `chain_capture` procedure and the `load_unload` procedure. Incorrect results are possible if any other capture procedures are included in the Procedure section when the `chain_capture` procedure is present.
- You cannot use the `run_atpg -auto` command if full-sequential test generation is turned off.
- Retention cell testing is only supported in uncompressed scan mode. If you attempt to use a compression mode, the `run_atpg` command will stop and an M870 error is issued.

Power Aware ATPG Limitations

Note the following limitations related to power aware ATPG:

- Test-mode based clock gating is not supported. Only scan-enable based clock gating is supported.
- Latch-free clock gating is not supported. Only latch-based clock gating is supported, which includes cascaded latch-based clock gating structures.
- Only simple clock-gating latches are supported. Combinations of the output of two or more latches when logically combined with the clock are not supported.
- Full-sequential ATPG is not supported for either the `report_power` command or Power Aware ATPG.
- Scan-enable signals must be constrained to the off-state for basic-scan, two-clock, and fast-sequential for test pattern generation and gated-clock (latch) identification. In addition, all global signals that are capable of enabling a large proportion of the clock-gating cells must be disabled.
- Maximum switching overshoots might occur if ATPG requires more flip-flops to change in excess of the power budget to detect a fault.
- Memories are not supported.

- Asynchronous set and reset signals must be inactive (in their off state).
- The `-domain` option of the `set_atpg` command does not work when specified with the `-calculate_power` option of the `set_atpg` command.

20

Bridging Fault ATPG

A bridging defect, also known as a short, is a common defect in semiconductor devices. This defect causes two normally unconnected signal nets in a device to become electrically connected due to extra material or incorrect etching.

Bridging 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.

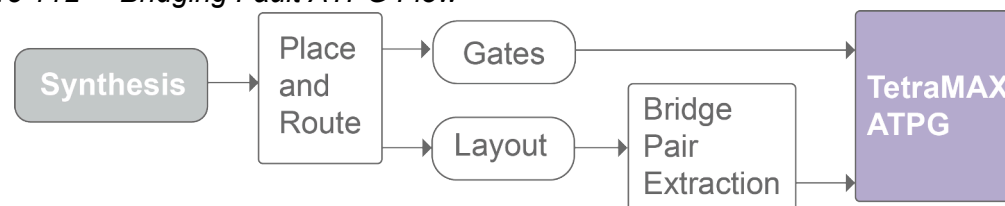
The following sections describes the bridging fault model, fault simulation, and dynamic bridging fault ATPG flows:

- [Bridging Fault ATPG Flow Overview](#)
- [Running the Bridging Fault ATPG Flow](#)
- [Detecting Bridging Faults](#)
- [Bridging Fault Model Limitations](#)
- [Running the Dynamic Bridging Fault ATPG Flow](#)

Bridging Fault ATPG Flow Overview

Bridging fault ATPG and fault simulation is usually run following the completion of place and route on full-chip designs.

Figure 112 Bridging Fault ATPG Flow



Two possible ways you can generate bridge pairs are:

- Extract bridging pairs from the layout using an IFA-based scheme
- Use an extracted coupling capacitance report

Third-party capacitance extraction tools can be used to generate coupling capacitance reports if the node list is in the TestMAX ATPG format.

It is not possible to accurately model fault effects for bridges that involve clock/set/reset lines and bridges that produce combinational loops. Therefore, you should filter out these types of bridges.

You can also run the TestMAX ATPG dynamic bridging fault model, which combines two fault models:

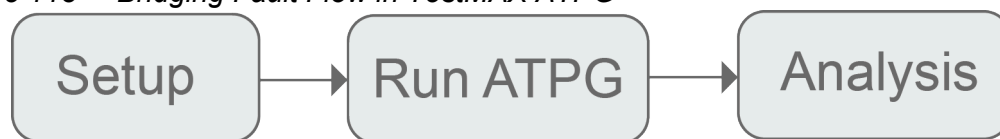
- The *static bridging fault model*, which observes whether the value on the aggressor node will override the value on the victim
- The *transition fault model*, which observes whether the transition at the fault site is too slow for the rated clock speed

For details on dynamic bridging, see [Running the Dynamic Bridging Fault ATPG Flow](#).

Running the Bridging Fault ATPG Flow

The following figure shows the bridging fault flow in TestMAX ATPG. The typical commands used in this flow are identified in the subsections that follow.

Figure 113 Bridging Fault Flow in TestMAX ATPG



Setup

The following commands are typically used at the beginning of a command file since they have an effect on subsequent commands:

- `set_faults -model bridging` – mandatory command for bridging faults.
- `set_atpg -optimize_bridge_strengths` – determines if TestMAX ATPG optimizes drive strength on the driving gates of the victim and aggressor nodes.
- `set_faults -bridge_input` – specifies TestMAX ATPG 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.

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

The following commands and options are useful for manipulating the fault list:

- `add_nofaults` – 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.
- `remove_faults <[-bridge_location <bridge_location1>
<bridge_location2>] | -all | -retain_sample <d> | -class
<fault_class>> [-bridge <0|1|01>] [-clocks] [-agressor_node <first |
second | both>] [-non_strength_sensitive]`

This command can be used to remove bridging faults.

Examining the Fault List

The following 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>] [-agressor_node <first | second | both>] [-bridge_feedback] [-bridge_strong]`

The format of the report is in four columns:

- 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 are 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 TestMAX ATPG 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

The following example shows a script for bridging fault support. This script generates tests for bridging faults followed by stuck-at faults. You might want to experiment with the reverse order as well to see which method produces better results.

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_input
# to optimize strengths during atpg
set_atpg -optimize_bridge_strengths
# read in fault list
add_faults -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
```

Detecting Bridging Faults

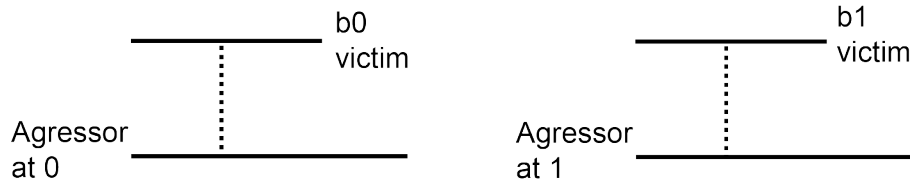
The following sections describe how TestMAX ATPG detects bridging faults:

- [Defining Bridging Faults](#)
- [Bridge Locations](#)
- [Strength-Based Patterns](#)

Defining Bridging Faults

TestMAX ATPG defines a bridging fault by type and a set of two nodes that can be instance pins or net names. The type is either bridging fault at 0 (ba0) or bridging fault at 1 (ba1), as shown in the following figure). The first node is the victim node and the second node is the aggressor node.

Figure 114 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, which can be any of the following:

- Cell instance input pin
- Cell instance output pin
- Net name

Faults on bidirectional pins are ignored. Input pins can be used only if the `set_faults -bridge_input` command is specified.

Although a net can have many names as it traverses the hierarchy of a design, TestMAX ATPG does not store them all. If you specify a net name as a bridge location that TestMAX ATPG recognizes (those accepted by the `report_primitives` command), it is 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 when a bridge is 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 process 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 the following detection types:

- Minimal detection
 - The minimum condition for the detection of ba0 & ba1 faults
- Fully optimized detection.
 - A detection in which the conditions specified in Table 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.

Figure 115 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 Model Limitations

Using the bridging fault model 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 is 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 TestMAX ATPG.
- 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.

Running the Dynamic Bridging Fault ATPG Flow

The following sections describe how to run the dynamic bridging fault ATPG flow:

- [Dynamic Bridging Fault Model Introduction](#)
- [Preparing to Run Dynamic Bridging Fault ATPG](#)
- [Fault Simulation](#)
- [Running ATPG](#)
- [Analyzing Fault Detection](#)
- [Example Script](#)
- [Limitations](#)

Dynamic Bridging Fault Model Introduction

The TestMAX ATPG dynamic bridging fault model combines two fault models:

- The *static bridging fault model*, which observes whether the value on the aggressor node will override the value on the victim
- The *transition fault model*, which observes whether the transition at the fault site is too slow for the rated clock speed

Based on the combined usage of these two fault models, the dynamic bridging fault model can be used to analyze transition effects in the presence of a specified value on a bridge-aggressor node.

TestMAX ATPG defines two types of dynamic bridging faults:

- *Bridge slow-to-rise (bsr)* — a slow-to-rise fault exists on the victim node while the aggressor node is at 0.
- *Bridge slow-to-fall (bsf)* — a slow-to-fall fault exists on the victim node while the aggressor node is at 1.

The fault location is the same as that used for (static) bridging faults, except that the cell instance input pin cannot be faulted. See [Bridge Locations](#) for more information.

Note that since a list of dynamic bridging nodes is required to run ATPG, the dynamic bridging fault model and fault simulation process is usually run after completing place and route on full-chip designs. Also note that you cannot add all faults using the dynamic bridging fault model. To add all faults, you will need to explicitly create a fault list before running ATPG using the `-auto` option.

Preparing to Run Dynamic Bridging Fault ATPG

To enable dynamic bridging fault ATPG, specify the following command:

```
set_faults -model dynamic_bridging
```

The following tasks are required to set up TestMAX ATPG to run dynamic bridging fault ATPG:

- [Specifying a List of Input Faults](#)
- [Manipulating the Fault List](#)
- [Examining the Fault List](#)

Specifying a List of Input Faults

You can use any combination of the following three files to supply a list of dynamic bridging pairs (referred to as a fault list):

- **Command file** — This file, which can be sourced from a script, contains a set of `add_faults` commands that specify dynamic bridging pairs. The syntax for using the `add_faults` command for this purpose is as follows:

```
add_faults [-bridge_location bridge_location1 bridge_location2]  
[-dynamic_bridge <r|f|rf>] [-dominant_node <first | second | both>]
```

- **Fault list file** — This file is generated by either the `report_faults` command or the `write_faults` command and is read by the `read_faults` command. Note that only bsr and bsf fault types are valid in this list.
- **Node file** — This file contains a list of dynamic bridging node pairs, specified in terms of nodes, using the following syntax for the `add_faults` command:

```
add_faults <-node_file name>[-dynamic_bridge <r|f|rf>]  
[-dominant_node <first | second | both>]
```

The format for a node file is to specify a pair of bridge locations on each line, separated by a space. Additional details are also covered in the “Node File Format for Bridging Pairs” topic in TestMAX ATPG Online Help.

The command file, fault list file, and node file should not include clocks and asynchronous set or reset signals, because proper detection status cannot be guaranteed for these faults.

Manipulating the Fault List

You can use the following commands and options to manipulate the fault list:

- `add_nofaults` — This command can be used to set a “no fault” status on victim nodes to prevent the associated dynamic bridging fault from being added to the fault list. The syntax for this command is as follows:

```
add_nofaults < instance_name | pin_pathname | -Module name >
```

- `remove_faults` — This command can be used to remove dynamic bridging faults. The syntax for this purpose is as follows:

```
remove_faults < [-bridge_location bridge_location1  
bridge_location2]
```

```
| -all | -retain_sample <d> > [-dynamic_bridge <r|f|rf>]  
[-clocks][-dominant_node <first | second | both>]
```

Examining the Fault List

The following options from the `report_faults` command can be used to examine a fault list, both before and after fault ATPG and simulation:

```
[bridge_location1 bridge_location2]  
[-dynamic_bridge <r|f|rf>]  
[-dominant_node <first | second | both>]
```

The format of the generated report contains the following four columns:

- Column 1: Fault type (bsr or bsf)
- Column 2: Fault detection status code
- Column 3: Dynamic bridge location of the victim node
- Column 4: Dynamic bridge location of the aggressor node

Note the following example report:

```
bsr NC nodeA nodeB  
bsf NC nodeA nodeB  
bsr NC nodeB nodeA  
bsf NC nodeB nodeA
```

Fault Simulation

You will need to perform fault simulation on the dynamic bridging faults to determine which dynamic bridging faults are detected by other existing patterns (such as those generated for stuck-at faults or transition faults). Typically, a large number of dynamic bridges are detected by patterns that target other fault models.

To run a fault simulation on the existing patterns, specify the `run_fault_sim` command. (You can see how to run this command in the [Example Script](#) section.)

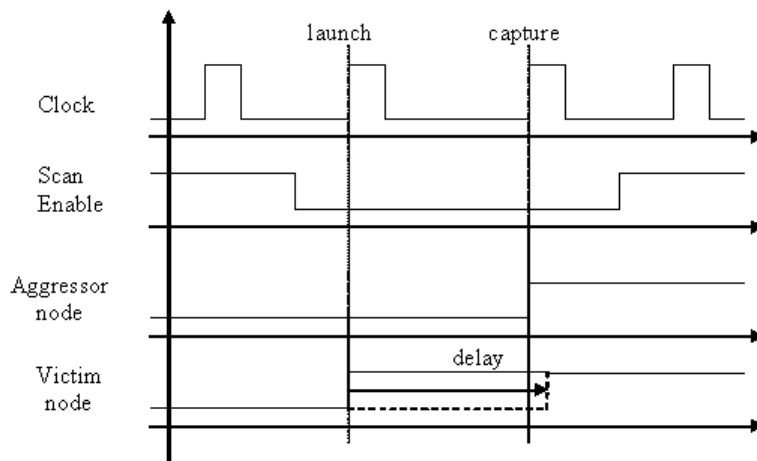
Note that fault simulation for dynamic bridging fault does not support Full-Sequential mode. An error is issued if you attempt to use this mode.

Running ATPG

The dynamic bridging fault ATPG process attempts to launch a transition along the victim while holding the aggressor at a static value. If you plan on issuing a `run_atpg -auto_compression` command, you will first need to create an explicit fault list by specifying either an `add_faults` or `read_faults` command.

Dynamic bridging fault ATPG can be run using the following ATPG modes: Basic-scan (launch on last shift), Two Clocks, and Fast-Sequential. An example of waveforms that are typically applied in the case of Fast-Sequential launch on system clock is shown in the following figure. In the presence of a bsr fault, the transition initiated because of the launch cycle at the victim node is delayed (dashed line) and the capture cycle detects the fault.

Figure 1 Dynamic Bridge Fault Detection Waveforms for Launch on System Clock



Note that dynamic bridging fault ATPG does not support Full-Sequential mode. An error is issued if this is attempted. Also, strength-based pattern generation similar to what exists for the TestMAX ATPG bridging fault model is not supported.

Analyzing Fault Detection

After running ATPG or fault simulation, you can use the `report_faults` and `write_faults` commands to analyze the fault detection status. You can invoke automated analysis and schematic display by using the following `analyze_faults` command options:

```
analyze_faults <bridge_location1 bridge_location2 -dynamic_bridge <r|f>>
```

Example Script

The following example shows a script for dynamic bridging fault support. This script generates tests for dynamic bridging faults, followed by stuck-at faults. You might want to experiment by reversing the order to see which method produces better results.

```
# read netlist and libraries, build, run_drc
read_netlist design.v -delete
run_build_model design
run_drc design.spf

# set fault model to dynamic bridging
set_faults -model dynamic_bridging

# read in fault list
add_fault -node_file nodes.txt

# run_atpg
run_atpg -auto_compression

# write out the bridging patterns
write_patterns dyn_bridge_pat.bin -format binary -replace

# fault simulate dynamic 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 dynamic bridging patterns
remove_faults -all
set_faults -model stuck
add_faults -all

# read in dynamic bridging pattern
set_patterns -external dyn_bridge_pat.bin
```

```
# fault simulate  
  
run_fault_sim  
  
# generate additional stuck-at patterns  
set_patterns -internal  
run_atpg -auto_compression
```

Limitations

The dynamic bridging fault ATPG feature currently has the following limitations:

- Full-Sequential ATPG and Full-Sequential fault simulation are not supported.
- The dominant node effect is based on its fault-free value. There is no ability to consider feedback effects that result from a dynamic bridge.
- There is no fault collapsing for dynamic bridging faults.
- Strength-based pattern generation is not supported.
- Input and bidirectional pins cannot be faulted.
- Proper detection status cannot be guaranteed for dynamic bridging pairs, including clocks and asynchronous sets/resets.
- TestMAX ATPG does not provide detection by implication (DI) credit.
- TestMAX ATPG does not provide a method for internally generating dynamic bridging node pairs.
- Only net names given by the `report_primitives` command are supported.

21

Cell-Aware Test

Cell-aware test is a methodology for increasing defect coverage, lowering defective parts per million (DPPM) and improving diagnostics accuracy for emerging process nodes, [FinFETs](#) (multi-gate field-effect transistors), and automotive standards. Cell-Aware is not supported with the persistent fault model flow.

The following topics explain how to use cell-aware test:

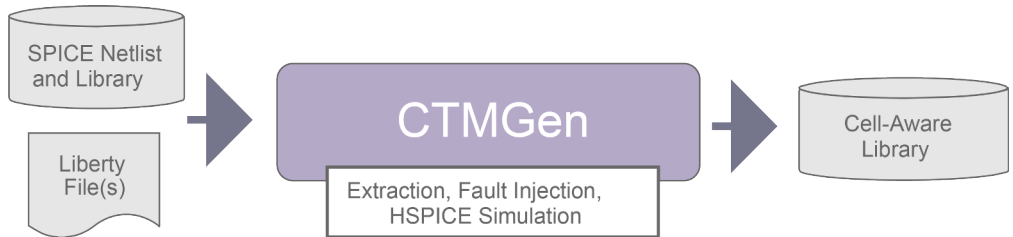
- [Cell-Aware Test Flow](#)
- [Targeting Internal Cell Defects](#)
- [Cell Test Models](#)
- [Generating Cell Test Models](#)
- [Running Cell-Aware ATPG](#)
- [Running Cell-Aware Simulation](#)
- [Cell-Aware Diagnosis](#)

Cell-Aware Test Flow

Cell-aware test is comprised of four primary phases:

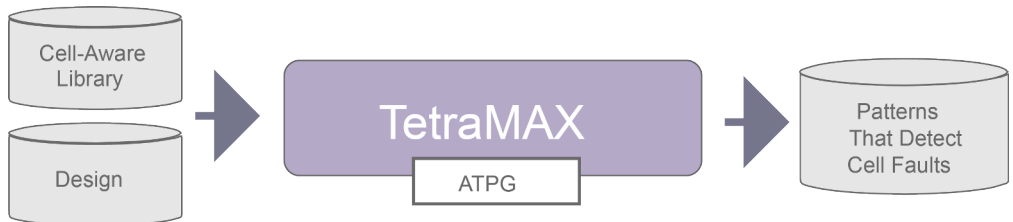
- *Cell Model Generation*

CMGen, Synopsys' cell test model (CM) generation utility, creates a CTM that lists all the detectable defects for each cell. The CTM guides the TestMAX ATPG and diagnostics processes on how to target and isolate these defects. This process is described in more detail in [Generating Cell Test Models](#) and the "Running CTMGen" document.



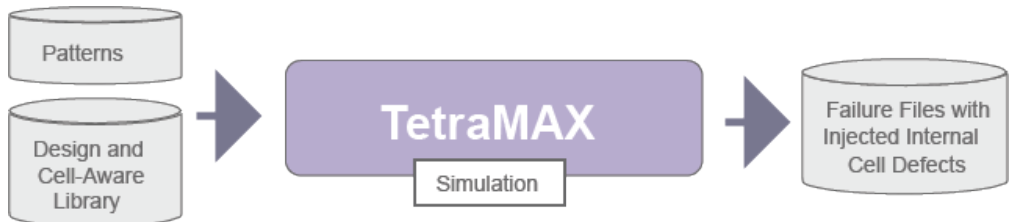
- **Pattern Generation**

Cell-aware ATPG uses the existing TestMAX ATPG paradigm. All defects become faults in the fault lists, and both static and dynamic defects are supported. The ATPG process merges and simulates both primary and secondary faults. This process is described in [Running Cell-Aware ATPG](#).



- **Pattern Simulation**

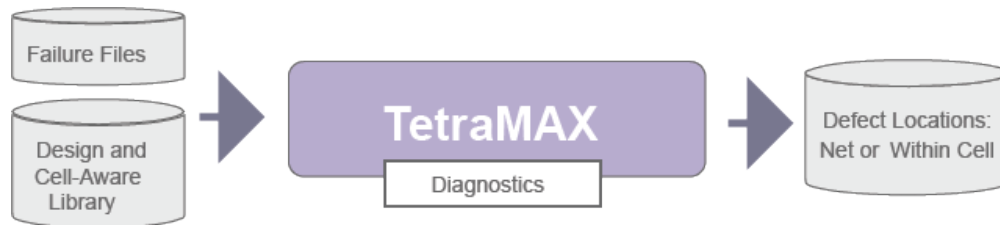
This process enables you to produce a failure file and inject internal cell defects for cell-aware diagnostics. For details, see [Running Cell-Aware Simulation](#).



- **Diagnosis**

Cell-aware diagnostics fits into the existing TestMAX ATPG diagnostics process. Cell-aware faults are mapped to the cell excitation conditions. Detailed structural

annotations allow for increased diagnostic resolution. This process is described in detail in [Cell-Aware Diagnosis](#).



The inputs to the characterization process include the following:

- SPICE netlist – The netlist for each cell typically includes extracted parasitics (not a strict requirement to generate a CTM).
- SPICE library – The library contains all the information needed to run HSPICE simulations on a cell.
- Liberty file – The .lib file contains timing information about the cell used to accurately model defect behavior.

Targeting Internal Cell Defects

Traditional ATPG and TestMAX ATPG cell-aware ATPG are significantly different in their approaches for targeting physical cell defects.

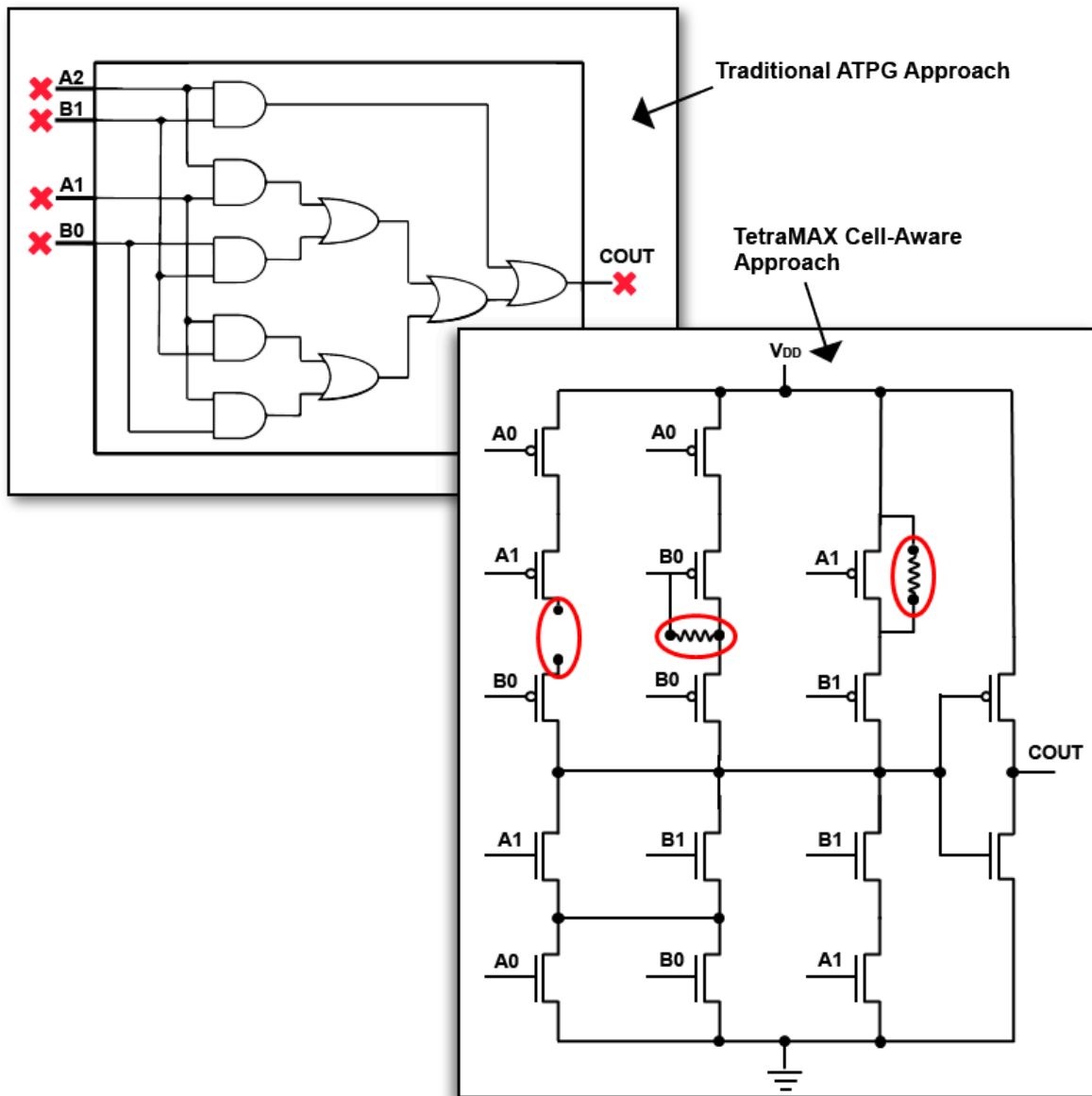
- Traditional ATPG targets faults between cells that are assigned to the input and output pins of cell instances. This approach can be effective in most situations, even though some of these faults originate as physical defects inside cells — referred to as “internal cell defects.”

However, traditional ATPG does not explicitly target internal cell defects. This is because as the complexity of a cell increases, the probability also increases that ATPG will not produce the input combinations required to cover all the defects likely to occur inside the cell. This observation particularly applies to high fan-in cells and cells that implement complex Boolean functions.

- TestMAX ATPG cell-aware ATPG explicitly targets internal cell defects during ATPG and diagnosis to increase defect coverage and improve diagnostics accuracy. It is particularly effective at testing faults in complex cells.

The example on the left in the following figure is a complex cell using the traditional ATPG approach, which targets faults between cells that are assigned to the input and output pins of cell instances. The example on the right is a transistor-level view of same cell and shows examples of the defects targeted by TestMAX ATPG cell-aware ATPG. In this case, the defects are targeted inside the cell.

Figure 116 Traditional ATPG Versus TestMAX ATPG Cell-Aware ATPG



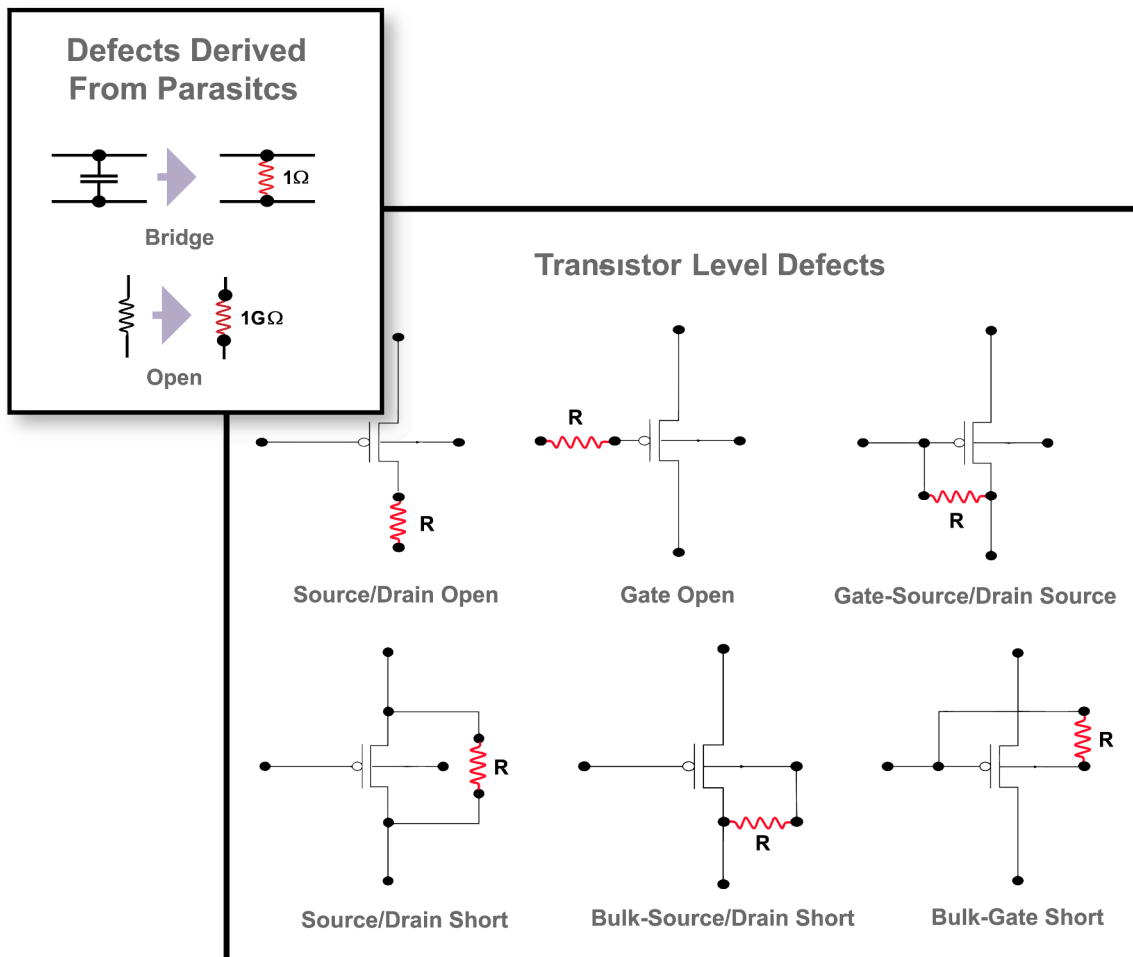
In TestMAX ATPG cell-aware ATPG, internal cell defects are characterized by simulating the HSPICE model of the cell under various short and open conditions. This characterization creates a single file, called a cell test model (CTM), that lists all the detectable defects for each cell. The CTM guides ATPG and diagnostics on how to target and isolate these defects.

Cell Test Models

Cell test models (CTMs) are the basis for performing cell-aware ATPG. A CTM is a text file that uses industry-standard YAML format and contains header information, such as the process corners derived from an HSPICE run.

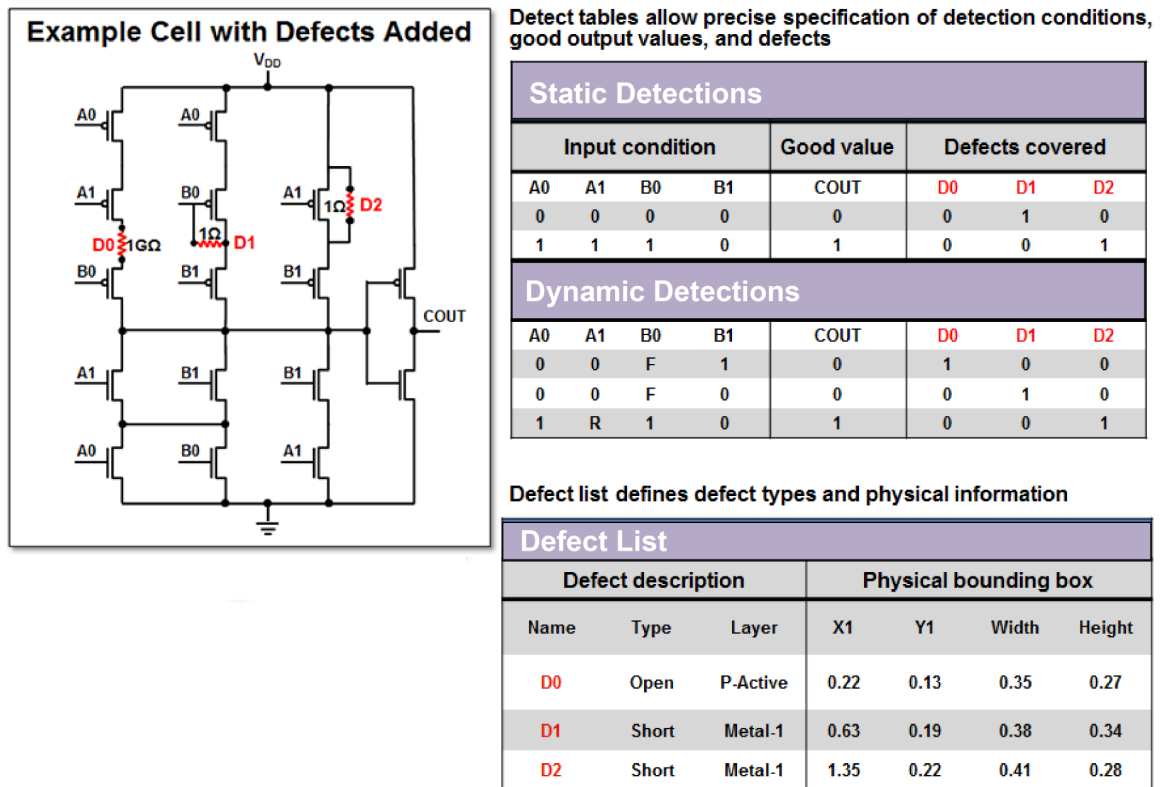
As shown in the following figure, resistors can be connected to transistors and assigned values that approximate the behavior of various physical defect types, such as opens on drains, source-drain shorts, and so forth. If you insert a parameterized resistor into a circuit netlist and perform a transient analysis, you can compare good behavior versus faulty behavior at the outputs. The simulations need to be accurate enough to predict faulty behavior observable as stuck-at-1/0.

Figure 117 Defect Injections



A CTM for a cell in a library includes a list of possible defects that could occur in a cell and the binary logic levels on the inputs and outputs. TestMAX ATPG references to target and detect each defect. A CTM can also include additional physical information about each defect, such as the mask layer and cell coordinates, that can be used for TestMAX ATPG diagnostics. The following figure shows a conceptual representation of a CTM of a cell with three defects: D0, D1, and D2.

Figure 118 Conceptual Representation of a CTM

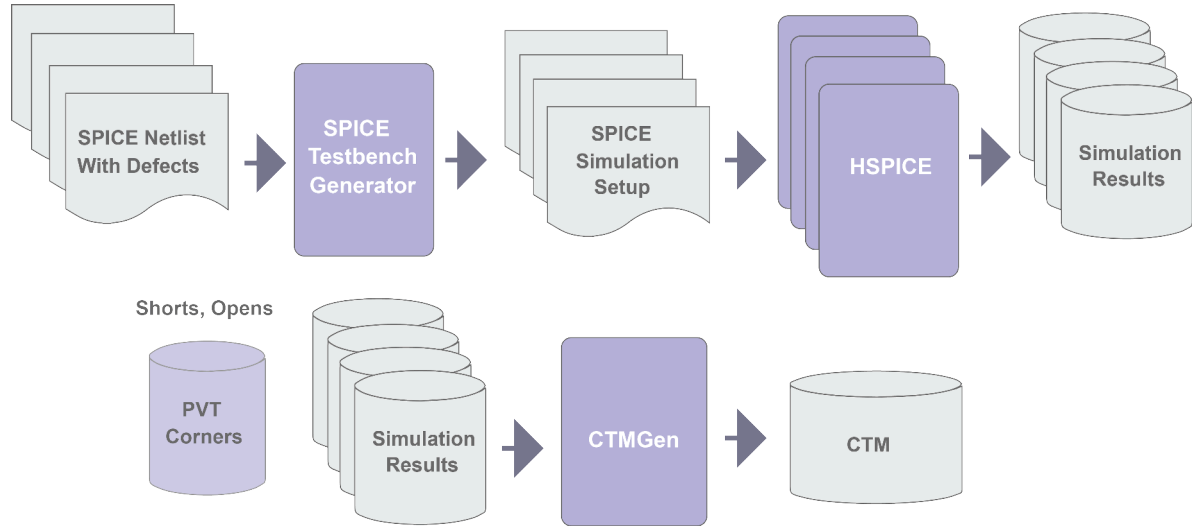


Generating Cell Test Models

CTMs are generated using a utility called CTMGen, which is provided by TestMAX ATPG. See the "Running CTMGen" document for details on using this utility.

To prepare for CTM generation, a SPICE testbench is generated and conditions are set up to inject defects to create faulty netlists. One testbench is used for each defect in the cell. HSPICE simulations are run on the fault-free and fault-injected netlists. The CTMGen utility processes the simulation results and generates the CTM. A library compiler compiles the source model to binary form.

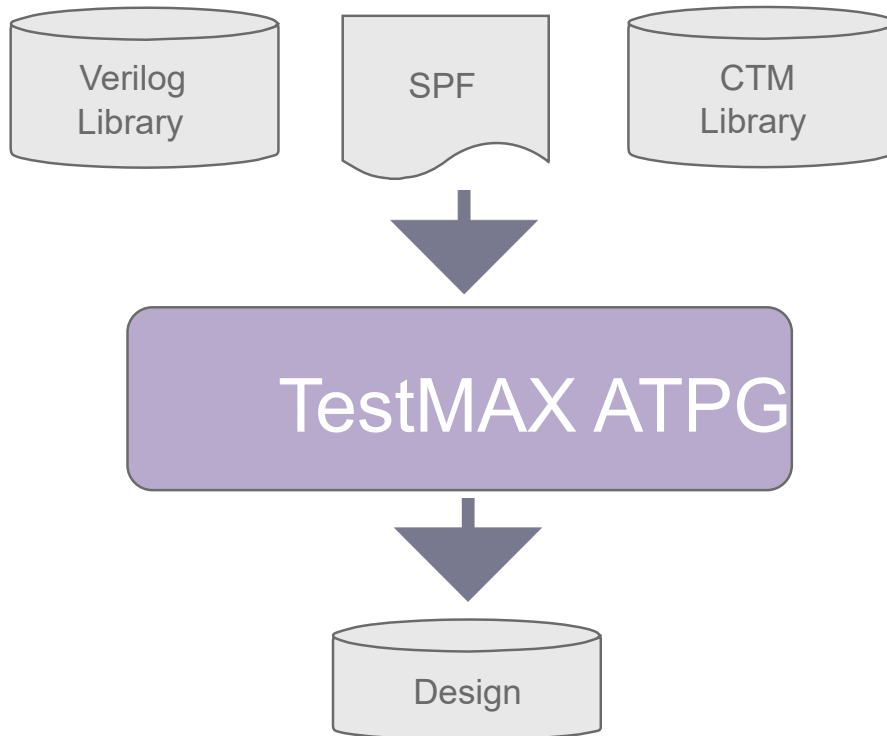
Figure 119 CTM Generation Process



Running Cell-Aware ATPG

Cell-aware ATPG uses the existing TestMAX ATPG paradigm. All defects become faults in the fault lists, and both static and dynamic defects are supported. The ATPG process uses both primary and secondary faults, and merges and fault simulates the faults.

Figure 120 Inputs and Outputs for Running Cell-Aware ATPG



The following steps show a typical cell-aware ATPG flow:

1. Do one of the following:

- If you have an existing design image, load it using the `read_image` command:

```
read_image ./design/leon3mp.img
```
- If you haven't read in the netlists, compiled the library, and run DRC, specify the following commands:
 - `read_netlist`
 - `run_build_model`
 - `run_drc`

2. Read in the cell test models.

```
read_cell_model /path/to/*.CTM
```

3. Add all faults to the fault list, including the cell-aware faults.

```
add_faults -all -cell_aware
```

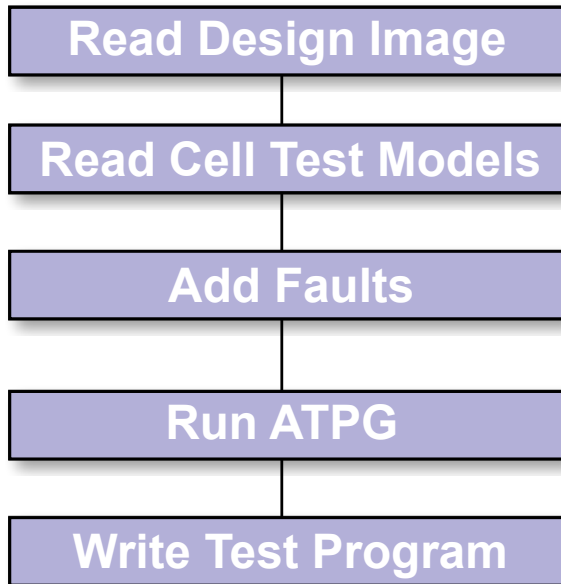
4. Run ATPG.

```
run_atpg -optimize
```

5. Write the patterns.

```
write_patterns patterns/ca_patterns.bin
```

Example Script



```
# Read the design image
read_image ./design/leon3mp.img
# Read CTMs
read_cell_model ./CTM/*.CTM
# Add Faults
add_faults -all -cell_aware
# Run ATPG
run_atpg -optimize
# Write the patterns
write_patterns \
./patterns/cell_aware_patterns.bin -replace
```

See Also

- [Cell-Aware Diagnosis](#)

Running Cell-Aware Simulation

The simulation phase of cell-aware test enables you to write a failure file and inject an internal cell defect for simulation.

To run a cell-aware simulation:

1. Do one of the following:

- If you have an existing design image, load it using the `read_image` command:

```
read_image ./design/SOPHIA3mp.img
```

- If you haven't read in the netlists, compiled the library, and run DRC, specify the following commands:

- `read_netlist`
- `run_build_model`
- `run_drc`

2. Read in the cell test models.

```
read_cell_model /path/to/*.CTM
```

3. Read in the patterns you created during ATPG (see [Running Cell-Aware ATPG](#)).

```
set_patterns -external ./patterns/cell_aware_patterns.bin
```

4. Run a simulation, and include the path and name for the failure file. Optionally, you can inject an internal cell defect, as shown in the following example:

```
run_simulation -failure_file [list outfile1 outfile2] \  
-cell_aware_fault instance_name1 D1 -replace
```

The following script shows the commands used to run a cell-aware simulation.

```
read_image ./design/SOPHIA3mp.img  
set_patterns -external ./patterns/cell_aware_patterns.bin  
read_cell_model ./CTM/*.CTM  
run_simulation -failure_file ./cell_aware.fail -cell_aware_fault  
"u0_3/p0_c0mmu_dcache0/U2708/Q D1" -replace
```

See Also

- [Running Cell-Aware ATPG](#)

Cell-Aware Diagnosis

Cell-aware diagnostics involves the following steps:

1. Identify the defective cells

Identify any defective cells and their observed behavior. First, use physical information to rule out defects on nets (such as bridges or open faults). Next, use the values observed on the cell inputs in the failing and passing patterns to characterize the defective behavior. This process is briefly described in [Generating Cell Test Models](#) and in more detail in the "Running CTMGen" document.

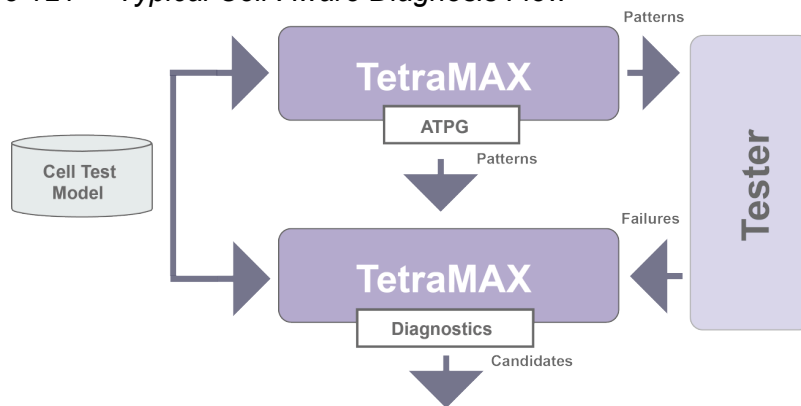
2. Identify defects within a cell and the cell input values

Use the cell test model (CTM) to map defective behavior to defects modeled within the simulation, then examine the values applied to the cell inputs and consider the values in the response table. You can then create a model for the defective cell based on the observed behavior, and review the cell input values. A description of a cell test model is described in [Cell Test Models](#). Also see [Identifying a Defect Within a Cell](#) for additional details.

3. Perform Physical Diagnosis

This process is described in [Running Cell-Aware Physical Diagnosis](#).

Figure 121 Typical Cell-Aware Diagnosis Flow



See Also

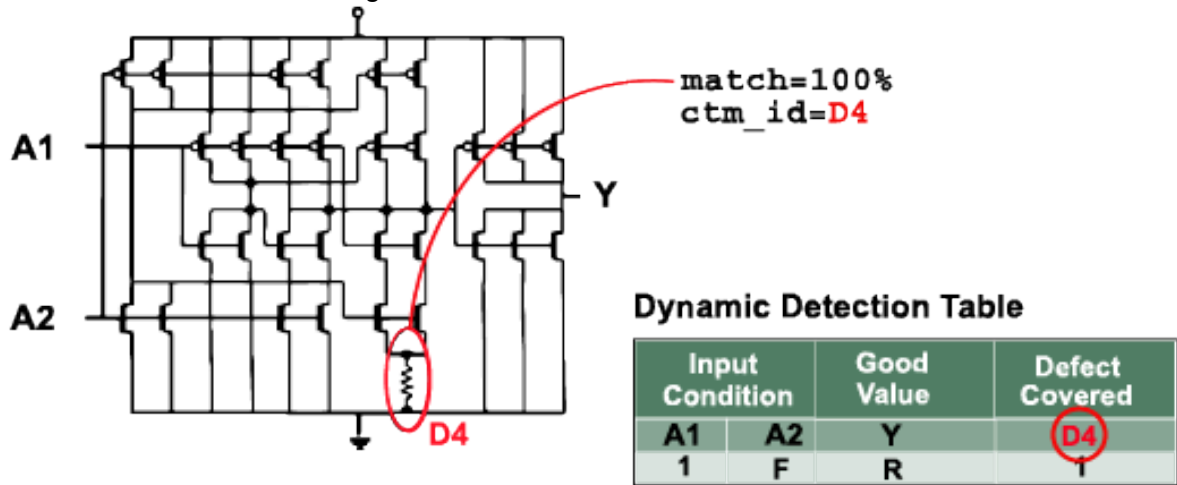
- [Using Physical Data for Diagnosis](#)

Identifying a Defect Within a Cell

Cell-aware faults are mapped to the cell excitation conditions in diagnostics. The CTM contains information that guides diagnostics, including detailed structural and behavioral annotations that enable increased diagnostic resolution.

A CTM can include both a static-detect (single-cycle) table and a dynamic-detect (two-cycle) table used for stuck-at and transition delay testing. Each table contains the input conditions required to target all the specified defects.

Figure 122 How Cell-Aware Diagnosis Identifies an Actual Defect



Cell Test Model

```

CellTestModel:
- Cell: OR2
InSignals: [A1, A2]
OutSignals: [X]
Defects:
- Id: D1
Type: short
- Id: D2
Type: short
- Id: D3
Type: short
- Id: D4
Type: short
- Id: D5
Type: short
- Id: D6
Type: short
- Id: D7
Type: short
- Id: D8
Type: open
Detections:
- [Table, Static]
- [A1,A2, X, D1,D3, D4, D5,D7]
- [0, 0, 0, 1, 1, 1, 0, 1]
- [0, 1, 1, 0, 0, 1, 1, 1]
- [1, 0, 1, 1, 0, 1, 1, 0]
- [1, 1, 1, 1, 0, 1, 1, 1]
- [Table, Dynamic]
- [A1,A2, X, D2,D6,D8]
- [0, R, R, 0, 1, 0]
- [R, 0, R, 1, 0, 0]
    
```

Running Cell-Aware Physical Diagnosis

To run cell-aware diagnostics:

1. Do one of the following:

- If you have an existing design image, load it using the `read_image` command:

```
read_image ./design/leon3mp.img
```

- If you haven't read in the netlists, compiled the library, and run DRC, specify the following commands:

- `read_netlist`
- `run_build_model`
- `run_drc`

2. Read in the patterns you created during cell-aware ATPG using the `set_patterns` command (see [Running Cell-Aware ATPG](#)).

```
set_patterns -external ./patterns/cell_aware_patterns.bin
```

3. Identify the host name and port number of the PHDS server containing the PHDS database using the `set_physical_db` command

```
set_physical_db -port_number 3967
```

4. Specify the output directory for the PHDS database using the `set_physical_db` command.

```
set_physical_db -database ./PHDS
```

5. Start the server process that queries the PHDS database using the `open_physical_db` command.

```
open_physical_db
```

6. Specify the name of the machine on which to connect to the PHDS database and the port number.

```
set_physical_db -hostname localhost -port_number 3967
```

7. Specify the name and version of your design.

```
set_physical_db -device [list SOPHIA3MP 0]
```

8. Read in the cell test models using the `read_cell_model` command.

```
read_cell_model /path/to/*.CTM
```

9. Use the `set_diagnosis` command to define the diagnostics report type and the fault types.

```
set_diagnosis -organization class -fault_type all
```

10. Run diagnostics using the failure file created when you ran cell-aware simulation (see [Running Cell-Aware Simulation](#)).

```
run_diagnosis cell_aware.fail
```

The following example script runs a cell-aware simulation:

```
read_image ./design/leon3mp.img
set_patterns -external ./patterns/cell_aware_patterns.bin
set_physical_db -port_number 4057
set_physical_db -database ./PHDS
open_physical_db
set_physical_db -hostname localhost -port_number 4057
set_physical_db -device [list LEON3MP 0]
read_cell_model ./CTM/*.CTM
set_diagnosis -organization class
set_diagnosis -fault_type all
run_diag cell_aware.fail
```

The following example shows typical output after running cell-aware diagnostics:

```
Diagnosis summary for failure
  file ./output/failure_logs/failure_log-1.diag
#failing_pat=6, #failures=24, #defects=1, #faults=4, CPU_time=53.05,
Memory=216M
Simulated : #failing_pat=6, #passing_pat=96, #failures=24
-----
Defect 1: stuck fault model, #faults=4, #failing_pat=6, #passing_pat=96,
#failures=6
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa1 DS frex1sdc/fr/FRCTL/p0010A1126972/Z (ddd222xssluhd)
Internal_cell_type (cell_aware) cell_name=ddd222xssluhd defect_id=D68
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frex1sdc/fr/FRCTL/p0010A1126972/Z (hdoi222xssluhd)
Internal_cell_type (cell_aware) cell_name=ddd222xssluhd defect_id=D60
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frex1sdc/fr/FRCTL/p0001A1126970/Z (hdao221xsslur)
Internal_cell_type (cell_aware) cell_name=ddd221xsslur defect_id=D503
-----
match=100.00%, #explained patterns: <failing=6, passing=96>
sa0 DS frex1sdc/fr/FRCTL/p0001A1126970/Z (ddd221xsslur)
Internal_cell_type (cell_aware) cell_name=ddd221xsslur defect_id=D504
-----
```

See Also

- [Using Physical Data for Diagnosis](#)

22

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, TestMAX ATPG 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.

The following topics describe how to use the transition delay fault model:

- [Using the Transition Delay Fault Model](#)
- [Specifying Transition Delay Faults](#)
- [Pattern Generation for Transition Delay Faults](#)
- [Pattern Formatting for Transition-Delay Faults](#)
- [Specifying Timing Exceptions From an SDC File](#)
- [Slack-Based Transition Fault Testing](#)

Using the 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 a slow-to-rise or slow-to-fall fault, the ATPG process 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.

For details on transition delay fault models, see [Transition Delay Fault Models](#) and [Detecting Transition Delay Fault Models](#).

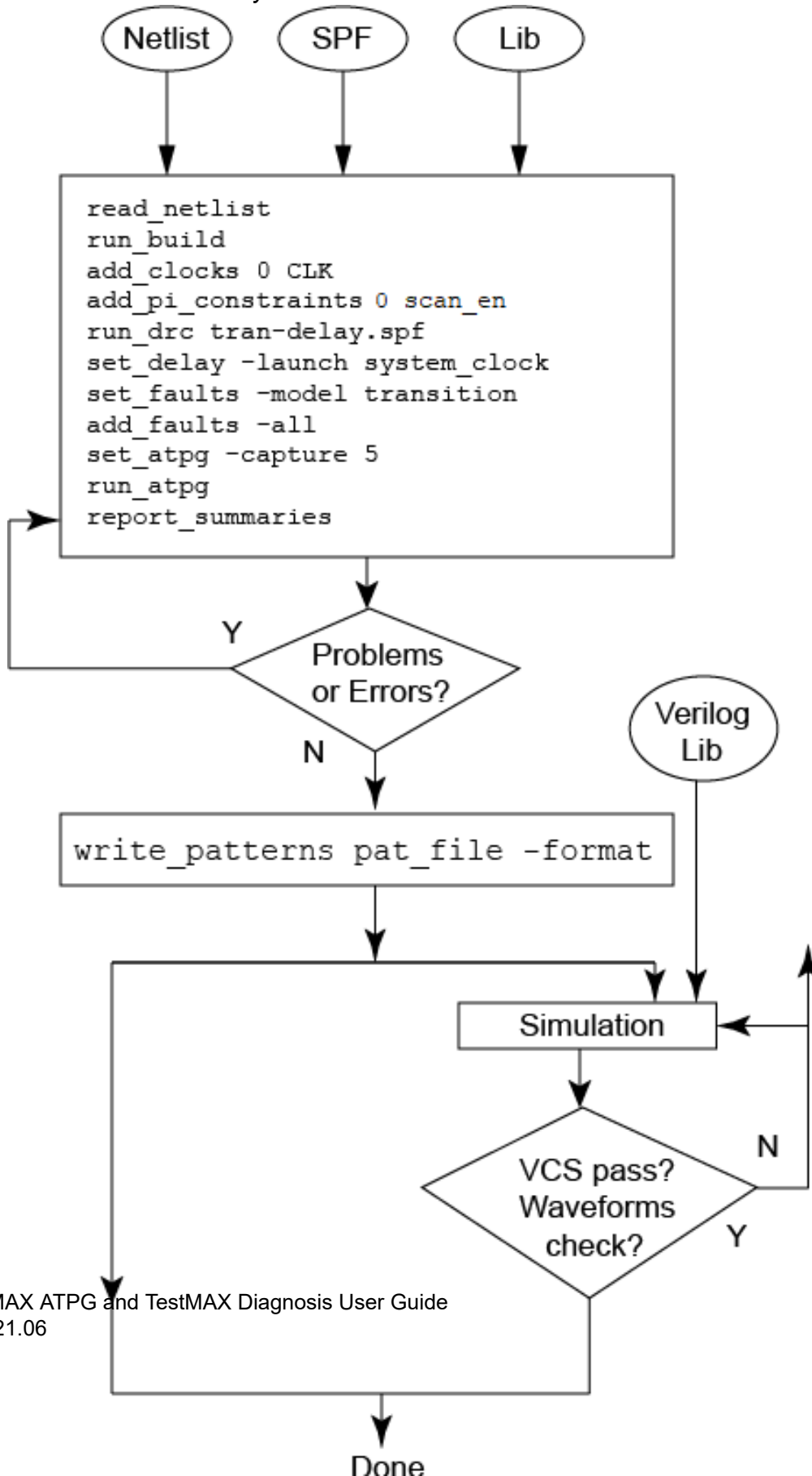
The following sections describe how to use the transition delay fault model:

- [Transition Delay Fault ATPG Flow](#)
- [Transition Delay Fault ATPG Timing Modes](#)
- [STIL Protocol for Transition Faults](#)
- [Creating Transition Fault Waveform Tables](#)
- [DRC for Transition Faults](#)
- [Limitations of Transition Delay Fault ATPG](#)

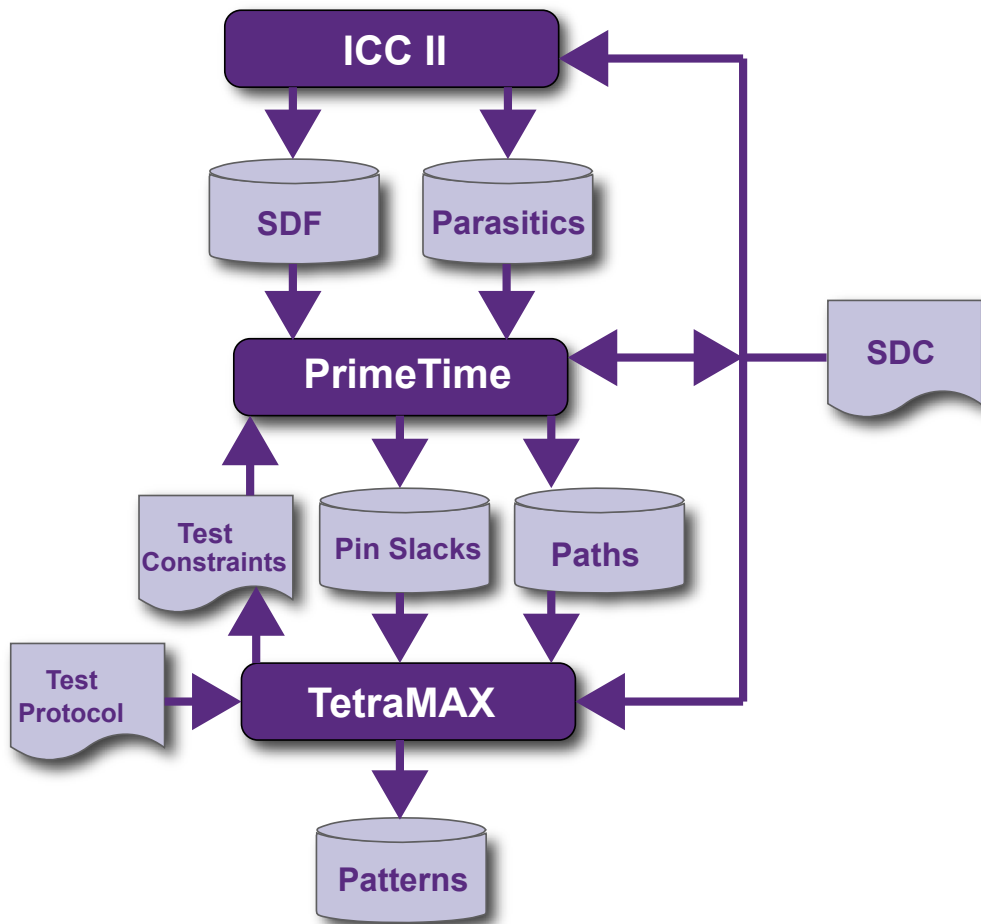
Transition Delay Fault ATPG Flow

The ATPG process for transition delay faults is similar to the process for stuck-at faults. Figure 1 shows the typical steps for performing transition-delay fault ATPG.

Figure 123 Transition Delay Fault Test Flow



Typical Transition Delay Fault ATPG Run



```
// last_shift launch:
read_netlist lib.v -lib
read_netlist test.v
run_build top
set_delay -launch_cycle \
last_shift
set_fault -model transition
set_drc test.spf
read_sdc <FILE_NAME>
set_delay -nopi_changes
set_delay -nopo_measures
add_po_mask -all
set_delay -common_launch_capture_clock
set_delay -allow_multiple_common_clocks
#optional# set_delay -slow_equivalence
#optional# set_delay -nodisturb
add_pi_constraints 0 scan_en
```

```

run_drc
run_atpg -auto
write_pattern patp.stil -format stil -parallel
write_pattern pats.stil -format stil -serial
exit
// system_clock launch:
read_netlist lib.v -lib
read_netlist test.v
run_build top
set_delay -launch_cycle system_clock
set_drc test.spf
set_fault -model transition
read_sdc <FILE_NAME>
set_delay -nopi_changes
set_delay -nopo_measures
add_po_mask -all
set_delay -common_launch_capture_clock
set_delay -allow_multiple_common_clocks
#optional# set_delay -slow_equivalence
#optional# set_delay -nodisturb
add_pi_constraints 0 scan_en
run_drc
run_atpg -auto
write_pattern patp.stil -format stil -parallel
write_pattern pats.stil -format stil -serial
exit

```

Transition Delay Fault ATPG Timing Modes

TestMAX ATPG transition delay fault ATPG supports several ATPG modes for applying transition-delay tests. You select the required mode with the `set_delay -launch_cycle` command. The following modes are supported:

- **Launch-On Shift (LOS)** — Specified by the `last_shift` option, TestMAX ATPG launches a logic value in the last scan load cycle when the scan enable is active, that is, 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, that is, in capture mode. Figure 2 shows the clock and scan enable timing for this mode.
- **System Clock** — Specified by the `system_clock` option (the default ATPG mode for transition-delay faults), TestMAX ATPG launches 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 3 shows the clock and scan enable timing for this mode.
- **Launch-On Extra Shift (LOES)** — Specified by the `extra_shift` option, TestMAX ATPG launches a logic value based on one more shift than launch on shift mode. This ensures that all clock domains receive their last scan shift before the internally-

controlled capture clock pulse. Unlike launch-on shift mode, launch-on extra shift mode does not place additional timing requirements on an on-chip clocking controller.

- *Any* — Specified by the `any` option, TestMAX ATPG attempts launch-on shift mode first, and then goes to launch-on capture or launch-on extra shift, depending on the pipelined SE constraint, or goes to both modes if it's unconstrained. .

The following sections explain some of the key characteristics of the timing modes:

- [Launch-On Shift Mode Versus System Clock Launch Mode](#)
- [Launch-On Extra Shift Timing](#)

Launch-On Shift Mode Versus System Clock Launch Mode

One of the major differences between launch-on shift mode and system clock mode is that for the launch-on shift mode, the scan enable signal must switch between a launch and capture cycle, which might not be possible depending on the design and cycle time. For details, see “DRC for Transition Faults”.

Figure 2 and Figure 3 show the clock waveform pertaining to launch on shift mode and system clock mode 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.

Figure 124 Last Shift Launch Timing

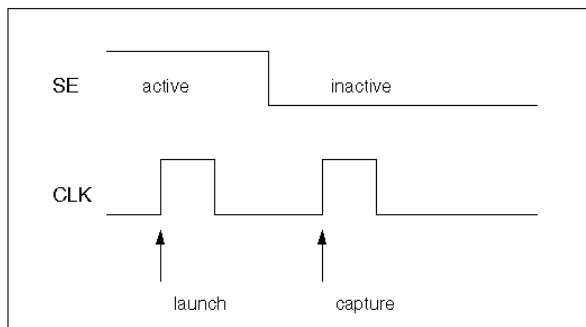
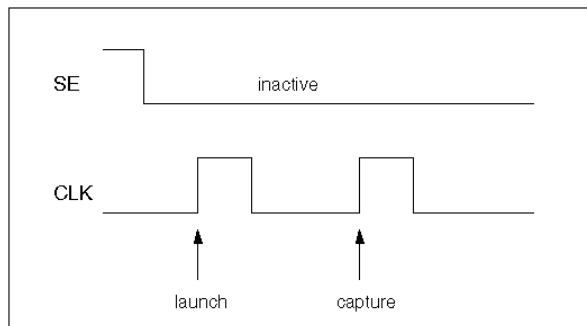


Figure 125 System Clock Launch Timing



Launch-on shift mode generates only basic-scan patterns, using a single capture procedure between scan load and scan unload.

By default, when using the `run_atpg -auto_compress` command for the system clock mode, TestMAX ATPG uses a highly optimized two-clock ATPG process that has some features of both the basic-scan and fast-sequential engines. The patterns generated by this process are only two clock cycles long and listed as Fast Sequential patterns in the TestMAX ATPG pattern summary.

To use system clock mode, fast-sequential ATPG must be enabled before 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 “Using the `set_atpg` Command”. The two-clock process, used with the `run_atpg -auto_compress` command by default, will automatically set the `-capture_cycles` option to 2.

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 2 before issuing the `run_atpg -auto_compress` command. In this case, TestMAX ATPG will first run the optimized two-clock process 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.

Launch-On Extra Shift Timing

Launch-on extra shift mode provides many advantages of launch-on shift mode without requiring an exotic on-chip clocking controller. However, a small but significant percentage of transition faults (which represent valid functional paths) cannot be detected by launching on either the last shift or an extra shift. When using LOES, you also need to perform a top-off ATPG run using launch-on capture (LOC).

Because the LOES test application waveforms are similar to those used for LOC, you can create an SPF for LOES based on the SPF used for LOC. The constraint on the `LOSPipelineEnable` signal is the only parameter you need to change. If the pipeline scan

enable logic was created by TestMAX DFT, the `LOSPipelineEnable` signal type appears in the `dc_shell` script, or it defaults to a port named `global_pipe_se`. This parameter should be a PI constraint and a scan enable for LOC and LOES; each mode is set to a different value. To specify the `LOSPipelineEnable` signal in the SPF, set `LOSPipelineEnable` in an F (for Fixed) block in each capture procedure, then add the setting to the first V statement in the `load_unload` procedure, and set it to the same value at the end of the `test_setup` macro. The `LOSPipelineEnable` port should be set to its active state (1 by default) for LOES and to its inactive state for LOC.

The `LOSPipelineEnable` signal is unconstrained in the SPF written from TestMAX DFT. Instead of editing the SPF, use the `add_pi_constraints` command to constrain it to 1 for LOES or to 0 for LOC.

When a LOS protocol is modified to make a protocol for LOES, you also must change the `load_unload` procedure. The LOS `load_unload` procedure has an extra V statement, with `"_pi" = #;` and `"_po" = #;` values, following the Shift loop; this V statement must be removed for LOES. You should also carefully check the waveform table usage and launch procedures since they can be very different for LOS compared to any other application.

The only additional command option used for LOES is the `-launch_cycle extra_shift` option of the `set_delay` command. This option should be set before the `run_drc` command. It directs the DRC process to perform different clock matrix checking for the launch cycle, since no disturbance originates from the functional data inputs of the scan registers. Instead, the disturbance comes only from the scan inputs. The `-launch_cycle extra_shift` option might result in better coverage.

LOES can be run with the `set_delay -launch_cycle system_clock` command. It is possible to generate patterns without constraining the `LOSPipelineEnable` signal so that the LOES and LOC patterns are intermingled. Both of these should be avoided because LOES patterns generated in this way uses pessimistic clock disturbances, which results in more patterns.

Transition delay fault ATPG should be run first with LOES, followed by a second run on the undetected faults with LOC. In comparative testing, this configuration resulted in higher coverage with fewer patterns than running ATPG with only LOC. An example of this flow is as follows:

Typical Flow for Using Launch-On Extra Shift Mode

```
set_delay -launch_cycle extra_shift
# Other delay settings are the same for LOES and LOC
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loes.spf -patternexec Internal_scan
set_faults -model transition
run_atpg -auto
write_patterns design_with_loes.stil -format stil
write_faults design_with_loes.faults -all
```

```
drc -force
# Prepare to change the LOSPipelineEnable constraint value
remove_pi_constraints -all
set_delay -launch_cycle system_clock
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loc.spf -patternexec Internal_scan
set_faults -model transition
#Use -retain_code so the redundant faults do not need to be identified
  again
read_faults design_with_loes.faults -retain_code
# Many faults that are AU for LOES can be detected by LOC
update_faults -reset_au
run_atpg -auto
write_patterns design_with_loc.stil -format stil
```

Note the following:

- ATPG in LOES mode uses the two-clock optimized ATPG engine. Fast-sequential patterns, and full-sequential patterns (if enabled), may also be generated. This is the same as with LOC, but different from LOS, which uses the basic-scan ATPG engine.
- The fault coverage achieved by LOES ATPG changes with scan chain reordering, even when run in uncompressed scan mode.
- For stuck-at testing, the LOSPipelineEnable port should be constrained to its inactive state (0 by default) for more efficient pattern generation.

See Also

- [Using load_unload for Last Shift-Launch Transition](#)
- [Understanding Fault Models](#)

STIL Protocol for Transition Faults

By default, TestMAX ATPG generates a capture procedure consisting of the following events:

- Force PI
- Measure PO
- Pulse Clock

You can insert a force PI or measure PO event between a launch and capture cycle. However, this has a negative impact on the 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 (vectors that follow the scan shift in the load_unload procedure) in the STL procedure file, 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` negatively impacts the overall quality of the test, but will not affect 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 STL procedure file. This can cause faults between primary inputs and registers to be ATPG untestable in the `last_shift` mode.

For more information, see [Creating Generic Capture Procedures](#).

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;}}  
  
}
```


Chapter 22: Transition Delay Fault ATPG Using the Transition Delay Fault Model

```
"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; } }
}
}
}
```

The following example shows a load_unload procedure defined in the STIL procedure 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 example shows a three-event capture procedure (the default) followed by its recommended single-event capture procedure for the “_fast_WFT_” waveform table in the previous example:

```
"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 TestMAX ATPG. TestMAX ATPG 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 STIL procedure file, they are first created and then used.

The difference for transition faults is that these special waveform tables must be present in the STIL procedure file to be used; TestMAX ATPG 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](#). To get more details about specialized timing support for both transition and path delay environments, see the following sections:

See Also

- [Generating Generic Capture Procedures](#)
- [Pattern Formatting for Transition-Delay Faults](#)

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 is switching between launch and capture. This transition fault ATPG requirement does not normally apply to stuck-at ATPG.

For 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 TestMAX ATPG, an output strobe occurs before 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 affect 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 TestMAX ATPG 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.

The following sections show you how to specify transition-delay faults:

- [Selecting the Fault Model](#)
- [Adding Faults to the Fault List](#)
- [Reading a Fault List File](#)

Selecting the Fault Model

You select the transition fault model using the `set_faults -model transition` command. You can change the fault model during a TestMAX ATPG 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 before 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 `system_clock` mode. This option selection is valid only if the transition model is selected with the `-model transition` option. In the `any` mode, TestMAX ATPG will do the following:

- Attempt to detect all faults using `last_shift` mode
- Apply `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.

The following steps show you how 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 required percentage of transition faults (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 TestMAX ATPG and should have the format shown in the following example. 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 TestMAX ATPG 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.

The following sections describe the various TestMAX ATPG commands used for transition-delay fault ATPG:

- [Using the set_atpg Command](#)
- [Using the set_delay Command](#)
- [Using the run_atpg Command](#)
- [Pattern Compression for Transition Faults](#)
- [Using the report_faults Command](#)
- [Using the write_faults Command](#)

Using the 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, TestMAX ATPG 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, TestMAX ATPG 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.

Using the `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 unlocked tester cycle to be added to each generated transition fault or path delay pattern. The use of a `set_drc -clock -one_hot` command might interfere with the addition of this unlocked 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.

Using the `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 cases:

- 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 want 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 might 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.

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.

Using the `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) before 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.

Using the `write_faults` Command

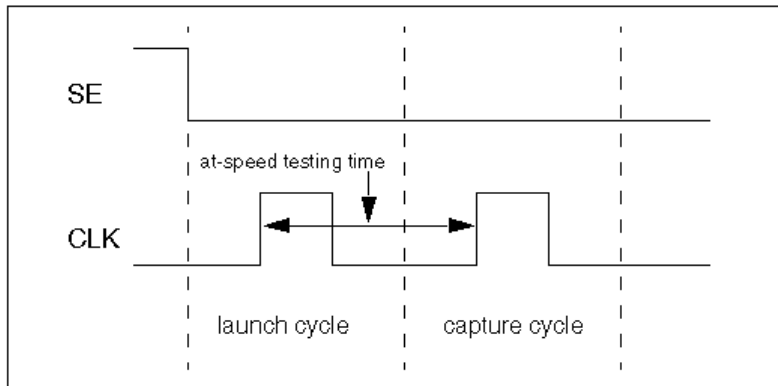
The `write_faults` command writes fault data to an external file. The file can be read back in later to specify a future fault list.

You can use the `-slow` option to write out a specific transition fault to a file.

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 the following figure.

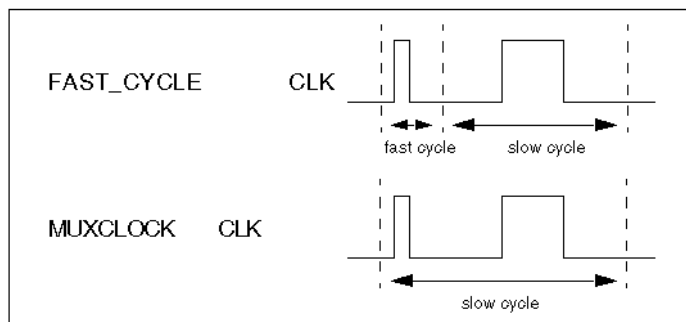
Figure 126 At-Speed Transition Test Timing



A fast tester might be able to generate two clock pulses (cycles) at the required at-speed value without dynamic cycle-time switching or other special timing formatting. For these testers, TestMAX ATPG 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 required 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. The following figure illustrates these techniques.

Figure 127 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 STL procedure 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. For more information, see [Creating At-Speed WaveformTables](#). 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 TestMAX ATPG 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_cycle system_clock`
- `set_delay -nopi_changes`
- `add_po_masks -all`
- `set_atpg -capture_cycles > 1`

These options will support the creation of patterns that might 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 [Generating Path Delay Tests](#).

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 TestMAX ATPG.
- 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
```

- MUXClock is not supported with scan compression designs

Specifying Timing Exceptions From an SDC File

TestMAX ATPG can read timing exceptions directly from a Synopsys Design Constraints (SDC) 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.

The following sections describe how to specify timing exceptions from an SDC file:

- [Reading an SDC File](#)
- [Interpreting an SDC File](#)
- [How TestMAX ATPG Interprets SDC File Commands](#)
- [Controlling Clock Timing, ATPG, and Timing Exceptions for SDC](#)
- [Reporting SDC Results](#)

The following limitation applies to SDC support in TestMAX ATPG:

- 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. TestMAX ATPG does not support this usage.

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_model` command, but before running the `run_drc` command) to use the `read_sdc` command.

Note the following:

- The 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.
- 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 unreadable. If the SDC file can be read into PrimeTime using its `read_sdc` command, then it can be read into TestMAX ATPG. If it must be read into PrimeTime using the source command, then it cannot be read into TestMAX ATPG. PrimeTime can write SDC, and this output is valid as SDC input for TestMAX ATPG.

Interpreting an SDC File

To control how TestMAX ATPG 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 to use the `set_sdc` command.

Note that the `set_sdc` command settings are cumulative; this command might be run multiple times to prepare for a `read_sdc` command. If multiple `read_sdc` commands are required, you can also specify the `set_sdc` command before each `read_sdc` command to specify its verbosity and instance.

How TestMAX ATPG Interprets SDC File Commands

TestMAX ATPG creates timing exceptions for transition delay testing based on a set of SDC (Synopsys Design Constraints) file commands. Note that not all SDC commands are used for this purpose, however they all must be specified in an SDC file.

The following list describes the set of SDC commands that are used by TestMAX ATPG, and how they are interpreted:

`set_false_path` -- This command creates a timing exception for a false path according to the specified from, to, or through points. TestMAX ATPG does not distinguish between edges; this means, for example, that `-rise_from` is interpreted the same as `-from`.

`set_multicycle_path` -- This command creates a timing exception for a multicycle path according to the specified from, to, or through points. TestMAX ATPG does not distinguish between edges. This means, for example, that `-rise_from` is interpreted the same as `-from`. 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 TestMAX ATPG `add_clocks` command or in the STIL protocol are used for pattern generation.

`set_disable_timing` -- This command creates a timing exception by disabling timing arcs between the specified points. TestMAX ATPG does not support library cells in the `object_list`.

`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_drc -sdc_environment` command (see the “Controlling Clock Tracing” section 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, ATPG, and Timing Exceptions for SDC

You can control the following processes related to reading timing exceptions for a Synopsys design constraints (SDC) file:

- *Controlling Clock Timing*

You can control clock tracing using the `set_sdc -environment` command.

- *Controlling ATPG Interpretation*

In some cases, you might want to treat multicycle paths below a certain number as if they are single-cycle paths. To do this, use the `set_delay -multicycle_length <N>` command.

Based on this option, all `set_multicycle_path` exceptions with numbers of *N* or less are 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 `set_simulation[-timing_exceptions_for_stuck_at | -notiming_exceptions_for_stuck_at]` command to control timing exceptions simulation for stuck-at faults. 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).

In addition, you can use the `report_settingssdc` command to report the current settings specified by the `set_sdc` command.

A pindata type related to SDC is available. You can control the display of this pindata type by running the `set_pindata -sdc_case_analysis` command:

The format of the data is N/M (N is the case analysis setting from the SDC, and M is the TestMAX ATPG constraint value). Unconstrained values are printed as X. You can also specify the display of this pindata type directly from the GSV Setup menu.

Slack-Based Transition Fault Testing

As geometries shrink, it is increasingly important to identify small delay defects. Standard transition fault testing is insufficient for detecting small delay defects because it focuses only on finding the simplest and shortest paths.

TestMAX ATPG uses a special slack-based mode of transition fault testing to identify small delay defects. When this mode is activated, TestMAX ATPG generates a specific set of transition fault tests that systematically identify the longest paths.

Using this testing methodology, you can extract slack data from PrimeTime, read this data into TestMAX ATPG, and use various TestMAX ATPG commands, command options, and flows related to testing small delay defects.

TestMAX ATPG uses a single ATPG run to generate tests for both slack-based transition faults and regular transition faults.

The following sections describe the slack-based transition fault testing process:

- [Basic Usage Flow](#)
- [Special Elements of Slack-Based Transition Fault Testing](#)
- [Limitations](#)

Basic Usage Flow

The basic flow for slack-based transition fault testing includes the following steps:

- [Extracting Slack Data from PrimeTime](#)
- [Utilizing Slack Data in the TestMAX ATPG Flow](#)
- [Command Support](#)

Extracting Slack Data from PrimeTime

TestMAX ATPG uses a specific set of timing data extracted from PrimeTime. To obtain this information, you use the `report_global_slack` PrimeTime command to extract slack data for all pins from PrimeTime.

The sequence of commands is shown in the following example:

```
pt_shell> set timing_save_pin_arrival_and_slack TRUE
pt_shell> update_timing
pt_shell> report_global_slack -max -nosplit > <global_slack_file>
```

The `-max` option is used with the `report_global_slack` command because PrimeTime considers setup margins to be "max" and hold margins to be "min". In this case, setup margins are required, so use the `-max` option to extract the minimum setup slacks.

The output format shown in the following example:

```
Max_Rise Max_Fall Point
-----
4.65 4.40 SE_FMUL10/OP0_L0_reg_23_/Q
* * SE_FMUL10/OP0_L0_reg_23_/SE
-0.82 -0.80 SE_PE0/U16112/A1
```

A * character is used instead of INFINITY.

Utilizing Slack Data in the TestMAX ATPG Flow

After producing a slack data file, use the `read_timing` command to read this data into TestMAX ATPG. Make sure you specify this command after entering DRC or TEST mode (after a successfully running the `run_build_model` or `run_drc` commands).

When TestMAX ATPG reads a slack data file, it uses a set of slack-based transition fault testing processes to construct a pattern for the target fault. If TestMAX ATPG does not read the slack file, regular transition-delay ATPG is performed.

How TestMAX ATPG Integrates Slack Data

During ATPG, TestMAX ATPG selects the first available fault from the list of target faults. It then uses the available slack data for the selected fault, and attempts to construct a delay test that makes use of the longest available sensitizable path. Secondary target faults for that same pattern might not have their longest testable path sensitized because some values were already set in the test for the primary target fault. Faults that are detected only by fault simulation without being targeted by test generation are 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.

For maximum efficiency, ATPG typically targets the easiest solution. This means that transition faults are more likely detected along the shorter paths or paths with larger slack. Fault simulation and ATPG increase the efficiency by accounting for transition faults that are randomly detected by the tests generated for the targeted fault. Those transition faults detected only by fault simulation represent a large fraction of the detected faults and 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 1 lists the key commands available to help validate the flow and pattern content.

Table 5 Key TestMAX ATPG Commands for Slack-Based Transition Fault Testing

Command	Description
<code>read_timing file_name [-delete]</code>	Reads in minimum slack data in the defined format and optionally deletes previous data
<code>report_timing instance_name -all -max_gates number</code>	Reports pin slack data accepted by TestMAX ATPG
<code>set_pindata slack</code>	Sets the displayed pindata type to show slack data
<code>set_delay [-noslackdata _for_atpg -slackdata_fo r_atpg</code>	Turns on and off the slack-based transition fault testing function during ATPG. If slack data exists, the default is the <code>-slackdata_for_atpg</code> option.

Table 5 Key TestMAX ATPG Commands for Slack-Based Transition Fault Testing
(Continued)

Command	Description
<pre>set_delay [-noslackdata _for_faultsim -slackdata_for_faultsim]</pre>	Turns on and off the slack-based transition fault testing function during fault simulation. If slack data exists, the default is the <code>-slackdata_for_faultsim</code> option.
<pre>set_delay -max_tmgn <float defect%></pre>	Defines the cutoff for faults of interest for slack-based transition fault testing generation. Faults with minimum slacks larger than the <code>-max_tmgn</code> parameter are targeted by the normal transition fault ATPG algorithm rather than by the slack-based algorithm.
<pre>set_delay -max_delta_per_fault float</pre>	Sets a level between the longest path and the path on which the fault is detected. Full detection is still credited, and the fault is dropped from further consideration. The default is zero (full credit is given only when detection is on the minimum slack path).
<pre>report_faults [-slack tmgn [integer float]]</pre>	Reports a histogram of faults based on the minimum slack numbers read in by the <code>read timing</code> command. This histogram is either fixed in the 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.
<pre>report_faults -slack tdet [integer float]]</pre>	Reports a histogram of faults based on the slack numbers for the detection path for each fault (detection slacks). This histogram is 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.
<pre>report_faults -slack delta [integer float]]</pre>	Reports a histogram of faults based on the difference between detection slacks and minimum slacks. The reported histogram is 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.
<pre>report_faults -slack effectiveness</pre>	Reports a measure of the effectiveness of the slack-based transition fault set. The measure varies from 0 percent (no faults of interest with detection slacks smaller than the <code>-max_tmgn</code> parameter) to 100 percent (all faults of interest detected on the minimum-slack path).

Special Elements of Slack-Based Transition Fault Testing

This section describes some of the unique characteristics related to slack-based transition fault tests. These special elements are described in the following topics:

- [Allowing Variation From the Minimum-Slack Path](#)
- [Defining Faults of Interest](#)
- [Reporting Faults](#)

Allowing Variation From the Minimum-Slack Path

When creating slack-based transition fault tests, the transition fault test generator targets the path with the minimum slack for the primary target fault. As with regular transition fault ATPG, there might be secondary target faults that are targeted following the successful generation of a test for the primary target fault.

Many faults detected during fault simulation are likely not be targeted faults for the test generator. You need to decide if you are willing to accept a test that detects a transition fault, or a test that detects the fault along a path with small slack. To specify the type of test you are willing to accept, 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, you can use a setting of 0 for the `-max_delta_per_fault` parameter (the default setting). If you want to accept any test that comes within 0.5 time units of the minimum slack for the fault, set `-max_delta_per_fault` to 0.5. This allows you to control when faults can be dropped from simulation in a slack-based transition fault 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 might continue to be simulated in hopes of getting a better test for the fault.

A fault detected with a slack equal to or smaller than the `-max_delta_per_fault` parameter is placed in the DS category normally used for detected transition faults. A DS category fault is always dropped from further simulation.

Specifying the `-max_delta_per_fault 0` option likely produces the highest quality test set. However, this specification also likely produces the longest runtimes and the largest test sets. The `-max_delta_per_fault` setting allows you to choose an acceptable trade-off point for test set quality versus runtime and test set size.

Defining Faults of Interest

You can specify how small the slack needs to be for TestMAX ATPG to target the fault with the slack-based transition fault test generation algorithm. If you specify the `set_delay -max_tmgn` command, the test generator uses the slack-based algorithm to target only those faults with a slack smaller than the number specified by the `-max_tmgn` parameter. All faults not designated as “faults of interest” are targeted by the normal transition fault test generation algorithm. All faults are fault-simulated even if they are not designated as “faults of interest” targeted for test generation.

You can use the `report_faults -slack tmgn` command to examine the distribution of slacks to determine a reasonable value of the `-max_tmgn` option. This command prints a histogram of the slack values that are read in by the `read_timing` command. You can also use the `report_faults -slack tmgn` command to specify how many categories are included in this report.

Reporting Faults

Slack-based transition fault tests are applied to paths with a smaller slack than those typically activated in regular transition fault test generation. The `report_faults` command includes several options that facilitate the examination of this data:

- The `-slack tdet` option prints a histogram that shows the slack of the detection paths. You can compare this data directly against the output of the `-slack tmgn` option to see how close TestMAX ATPG got to the minimum slack paths.
- The `-slack delta` option clearly shows the slack of the detection paths. The reporting histogram associated with this option is based on the difference between the slacks for the detection paths and the minimum slack read from the slack data file. A distribution heavily skewed toward the zero end of the continuum indicates a highly successful slack-based transition fault test generation.
- The `-slack effectiveness` option reports a measure of delay effectiveness 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 percent. If no 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 is 0 percent.

The output of the `report_faults -all` command also includes additional fields related to the slack-based transition fault testing. For more information, see the “Slack-Based Transition Fault Format” section of the “Understanding the `report_faults` Output” topic in TestMAX ATPG Help.

Limitations

The following limitations currently apply to slack-based transition fault testing:

- [Engine and Flow Limitations](#)
- [ATPG Limitations](#)
- [Limitations in Support for Bus Drivers](#)

Engine and Flow Limitations

Last-shift launch, two-clock, and fast-sequential transition fault testing are supported. There is currently no support for Full-Sequential mode.

ATPG Limitations

There are two limitations for slack-based transition ATPG:

- *Second Smallest Slack*

If the path with the smallest slack for a given fault is untestable, TestMAX ATPG begins normal back-tracking in an attempt to find a test along some other path. There is no guarantee that the second path TestMAX ATPG will try is 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 Might 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, TestMAX ATPG 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 required.

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 Bus Drivers

Full slack-based transition fault testing support is not available for BUS drivers. TestMAX ATPG does not choose the minimum slack path when back-tracing through a bus driver if that path goes through the enable input. TestMAX ATPG always chooses the path through the data input to the driver. This limitation applies to test generation only. The detection slack and the slack delta are accurately reported in all cases.

23

Path Delay Fault and Hold Time Testing

The TestMAX ATPG DSMTest option enables you to 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. TestMAX ATPG also includes features to read, manage, and analyze paths from static timing analysis tools such as PrimeTime.

Most of the fault models supported by TestMAX ATPG are intended to test maximum delays (or setup times), whether they are delay-based fault models (transition and dynamic bridging) or path-based fault models (path delay). Even the static fault models (stuck-at and bridging) are simulated so that the fault effect appears as a setup violation. The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow

The following sections describe path delay fault and hold time testing:

- [Path Delay Fault Theory](#)
- [Path Delay Testing Flow](#)
- [Obtaining Delay Paths](#)
- [Hold Time ATPG Test Flow](#)
- [Generating Path Delay Tests](#)
- [Handling Untested Paths](#)

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.

The following sections describe the path delay fault testing theory:

- [Path Delay Fault Term Definitions](#)
- [Models for Manufacturing Tests](#)
- [Models for Characterization Tests](#)
- [Testing I/O Paths](#)
- [Path Delay Test Patterns](#)

Path Delay Fault Term Definitions

The following table lists the definitions for key terms used in path delay fault testing.

Table 6 *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 6 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 (combinationally 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.
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, TestMAX ATPG 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 example:

```
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

TestMAX ATPG 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 example:

```
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

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)	00	11	00	11

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)	00	11

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, 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 TestMAX ATPG 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,

- 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 might 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 might 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.

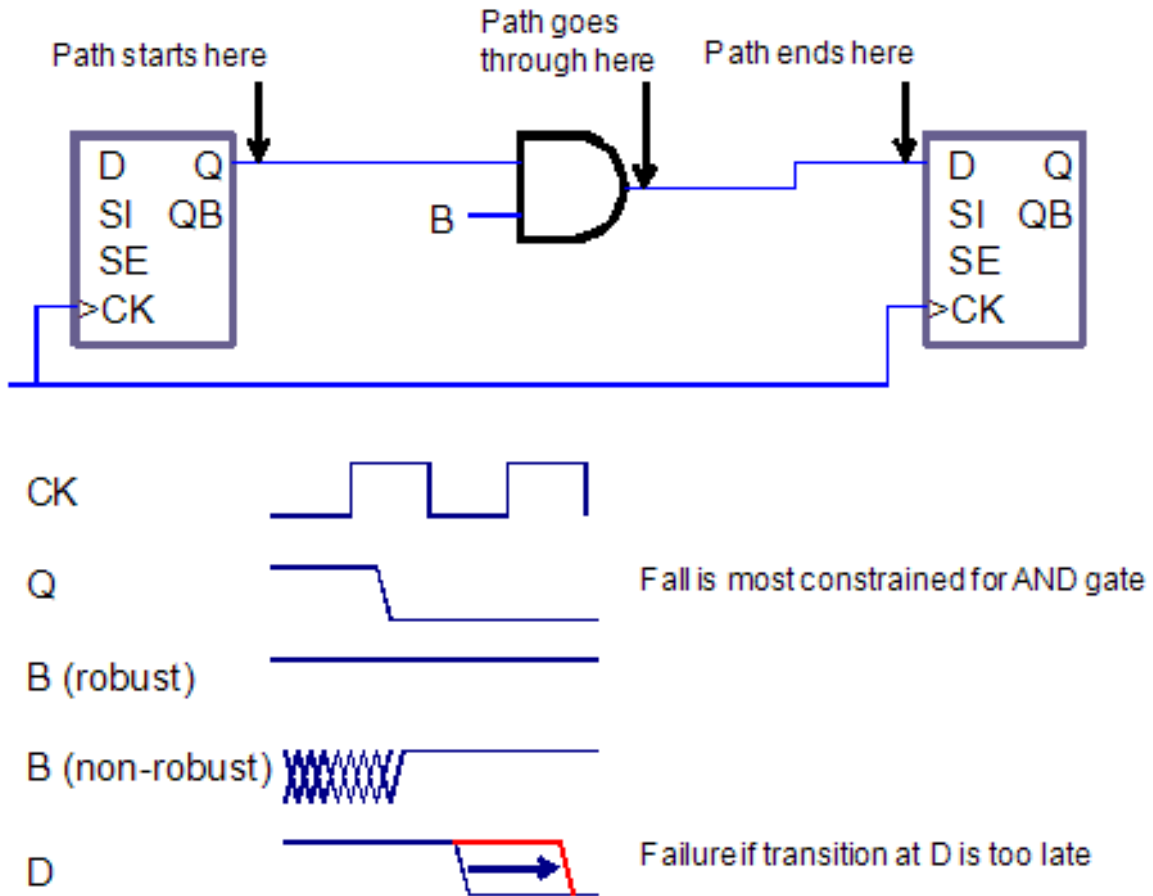
Path Delay Test Patterns

All delay paths must start with a state element (DFF, DLAT, or memory) and must end with an edge-triggered state element (DFF or edge-triggered memory); only combinational gates can be situated between the starting and ending elements. The source and destination points must capture on the same edge of the same clock. If the source and destination points are clocked by different clocks, the clocks must be either synchronized internal clocks (see [Specifying Synchronized Multi Frequency Internal Clocks](#)) or equivalent external clocks (see the description of the `add_pi_equivalences` command in TestMAX ATPG Online Help). If these conditions are not satisfied, the path is declared ATPG Untestable (fault status AN).

The edge information provided in the path file is only used for the source point of the path. If the path goes through XOR gates or multiple paths, then the polarity at the destination point and the path actually taken by the transition might differ from what was specified.

In the fault modeled by the TestMAX ATPG fault simulator, the launching node makes its transition too early. The captured node is assumed to be on time, and all off-path inputs are also assumed to be on time. If these assumptions result in a 0/1 difference in the output, then the fault is detected. See the representations of a path delay test pattern in the following figure.

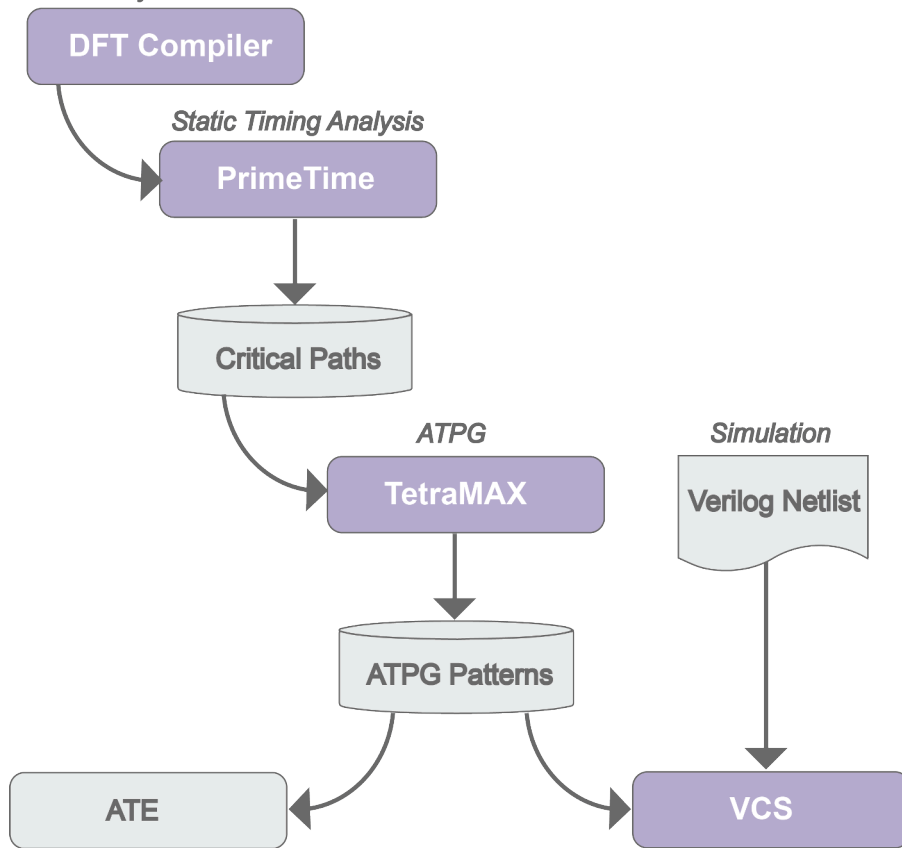
Figure 128 Path Delay Test Pattern



Path Delay Testing Flow

PrimeTime generates the critical path information you need to input for a path delay ATPG test run as shown in the following figure.

Figure 129 Path Delay Test Flow
Test Synthesis

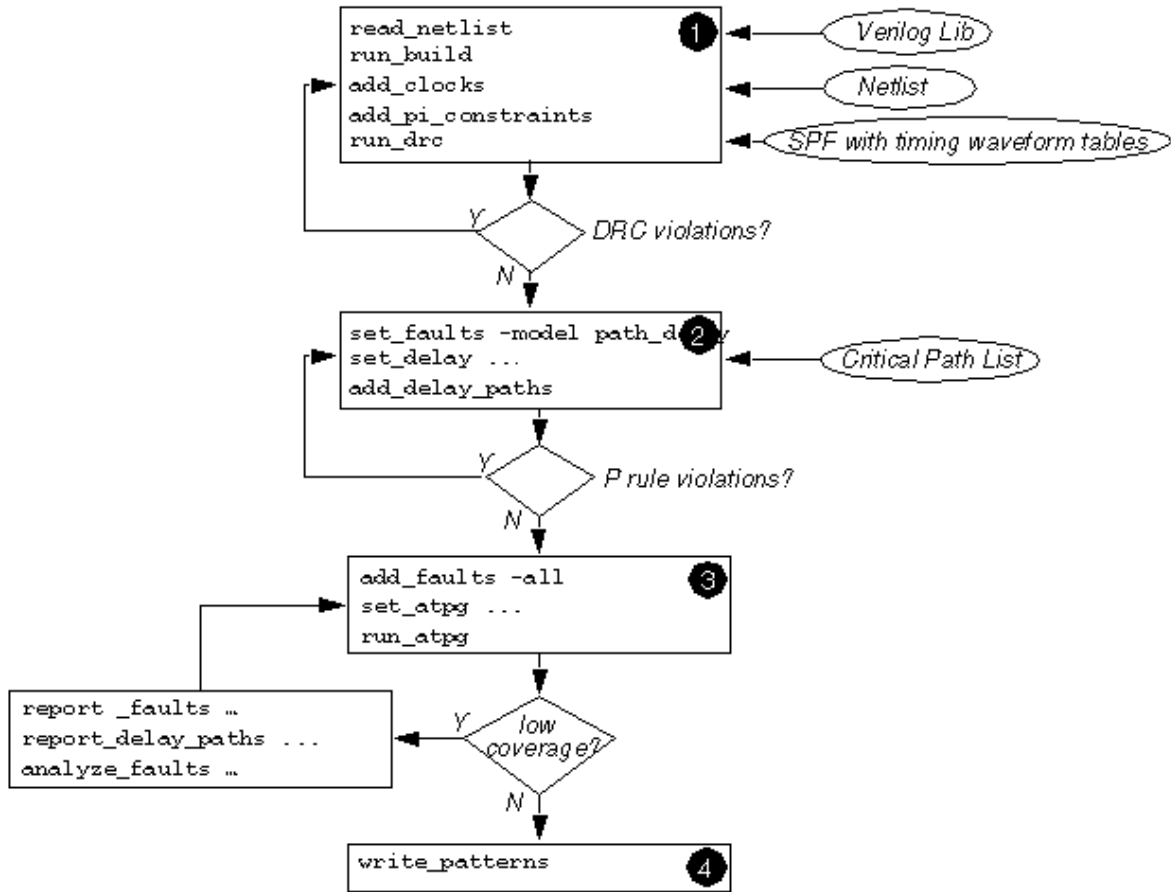


TestMAX ATPG supports ATPG and fault simulation for scan-based path delay fault testing with the following features:

- 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 STL procedure 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

The following figure shows the basic TestMAX ATPG steps and checkpoints to generate an effective set of path delay tests.

Figure 130 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. TestMAX ATPG 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, regarding transition patterns under a stuck model), care must be exercised if this information needs to be maintained. Before the `write_patterns` 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

TestMAX ATPG requires an input list of critical paths to target for path delay fault generation. TestMAX ATPG 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.

To obtain a list of critical delay paths, use the `write_delay_paths` Tcl procedure, which is part of the `pt2tmax.tcl` file. This process is described in detail in [Importing PrimeTime Path Lists](#).

For details on translating timing exceptions, see [Specifying Timing Exceptions From an SDC File](#).

Hold Time ATPG Test Flow

The TestMAX ATPG `DSMTest` option enables you to use hold time testing to perform test generation to detect critical path minimum delays. This option generates the most effective tests possible while providing the highest coverage of critical paths. TestMAX ATPG also includes features to read, manage, and analyze paths from static timing analysis tools such as PrimeTime.

Most of the fault models supported by TestMAX ATPG are intended to test maximum delays (or setup times), whether they are delay-based fault models (transition and dynamic bridging) or path-based fault models (path delay). Even the static fault models (stuck-at and bridging) are simulated so that the fault effect appears as a setup violation. The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow

The hold time fault model is different in that it tests minimum delays. In other respects, the hold time flow is very similar to the path delay ATPG flow .

The hold time ATPG test flow is the same as the path delay ATPG flow, except that instead of running the `set_faults -model path_delay` command, you need to specify the `set_faults -model hold_time` command. The `hold_time` argument specifies the ATPG and fault simulation commands to use the hold time fault model and must be specified before you add faults.

The standard hold time ATPG flow includes the following commands:

- `run_drc`
- `set_faults -model hold_time`
- `add_delay_paths hold_path_file`

- `add_faults -all`
- `run_atpg`

You can use normal reporting commands such as `report_summaries faults` and `report_delay_paths`. The fault types are reported as FTF (fast to fall) and FTR (fast to rise).

In the hold time ATPG test flow, all `set_delay` commands are ignored because the hold time path transition is launched and captured in a single clock cycle. Hold time faults are usually detected by the basic scan pattern type, although fast-sequential ATPG is also supported. Multiple-clock patterns are generated when the hold time path must be set up or when its effects are propagated through nonscan elements such as memories.

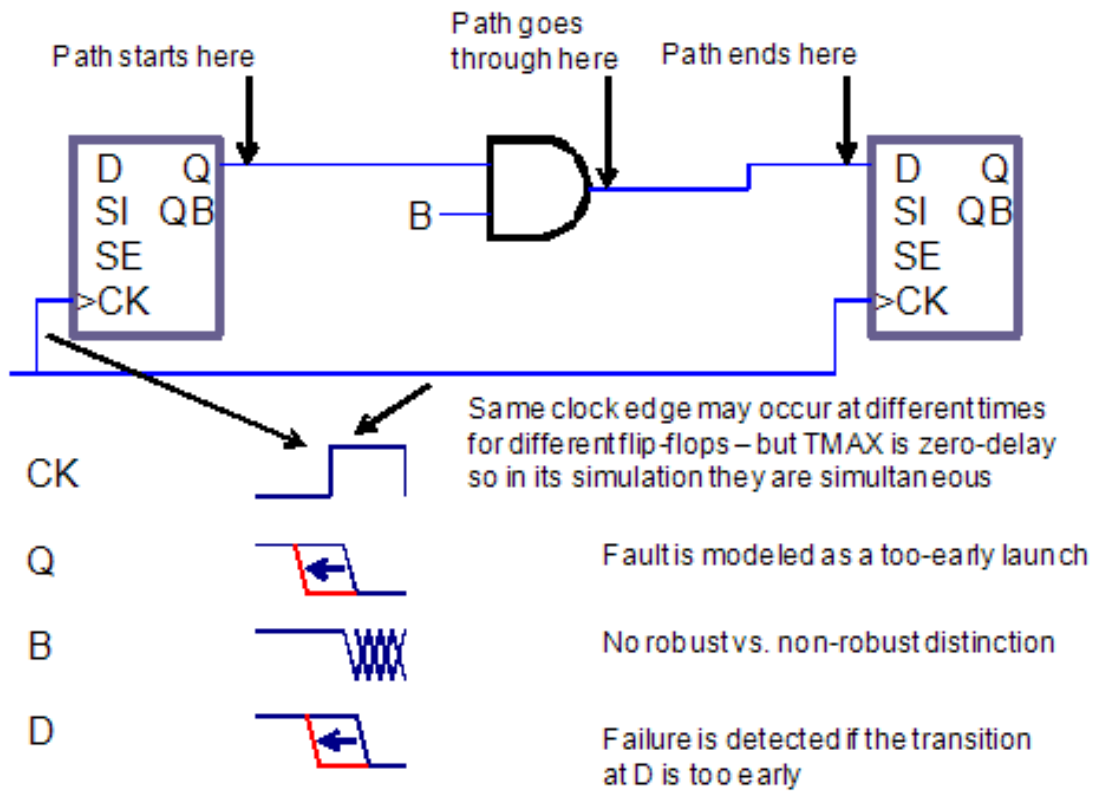
You can usually reduce the pattern count by first fault-simulating the stuck-at patterns with the hold time fault model, and then using ATPG to create new patterns to detect the undetected paths.

All paths must start with a state element (DFF, DLAT, or memory) and must end with an edge-triggered state element (DFF or edge-triggered memory); only combinational gates can be situated between the starting and ending elements. The source and destination points must capture on the same edge of the same clock. If the source and destination points are clocked by different clocks, the clocks must be either synchronized internal clocks (see [Specifying Synchronized Multi Frequency Internal Clocks](#)) or equivalent external clocks (see the description of the `add_pi_equivalences` command in TestMAX ATPG Online Help). If these conditions are not satisfied, the path is declared ATPG Untestable (fault status AN).

The edge information provided in the path file is only used for the source point of the path. If the path goes through XOR gates or multiple paths, then the polarity at the destination point and the path actually taken by the transition might differ from what was specified.

In the fault modeled by the TestMAX ATPG fault simulator, the launching node makes its transition too early. The captured node is assumed to be on time, and all off-path inputs are also assumed to be on time. If these assumptions result in a 0/1 difference in the output, then the fault is detected. See the representations of a path delay test pattern in [Figure 128](#) and a hold time test pattern in [Figure 131](#).

Figure 131 Hold Time Test Pattern



Generating Path Delay Tests

The following sections describe how to generate path delay tests:

- [Flow for Generating Path Delay Tests](#)
- [Using set_delay Options](#)
- [Reading and Reporting Path Lists](#)
- [Analyzing Path Rule Violations](#)
- [Viewing Delay Paths](#)
- [Path Delay ATPG Options](#)
- [Internal Loopback and False/Multicycle Paths](#)
- [Creating At-Speed Waveform Tables](#)

- [Maintaining At-Speed Waveform Table Information](#)
- [MUXClock Support for Path Delay Patterns](#)

Flow for Generating Path Delay Tests

The following steps show you how to generate a set of path delay tests:

1. Start TestMAX ATPG.
2. Read in the libraries and netlists.
3. Build a circuit model.
4. Run DRC (you can use delay waveform tables in the STL procedure file):

```
run_drc filename.spf
```

5. Depending on the ATE functionality, set the delay testing options:

```
set_delay -nopi_changes  
set_delay -nopo_measures
```

6. Read in the delay paths:

```
add_delay_paths filename
```

7. Analyze any P rule errors or warnings.

Use the following command to remove any paths that were read:

```
remove_delay_paths pathname
```

8. Display a delay path (optional):

```
report_delay_paths path_name -display -pindata
```

9. Add path delay faults:

```
set_faults -model path_delay
```

10. Run ATPG:

```
run_atpg -auto
```

11. Analyze low path delay coverage (optional):

```
report_faults -class AU  
  
analyze_faults path_name -slow rise -display -verbose  
-fault_simulation
```


12. Write the path delay test patterns:

```
write_patterns patname.stil -format stil  
write_patterns patname.wgl -format wgl
```

Many of the commands described in this flow have other options that you can use to adjust TestMAX ATPG to your unique requirements.

Using set_delay Options

After passing DRC, and before reading in a list of critical paths, you can use the `set_delay` command to specify any options related to path delay testing.

Note that the launch cycle setting has no effect on full-sequential ATPG. TestMAX ATPG uses either 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_constraints` command.

Reading and Reporting Path Lists

After setting the delay options, you can read delay faults into TestMAX ATPG using the `add_delay_paths` command. This command reads in a path delay definition file. 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

You can analyze the P rule violations using the GSV. For example, to view additional information on P20 violations, enter the following commands:

```
report_violations P20  
analyze_violations P20-3
```

Viewing Delay Paths

You can use the `report_delay_paths path_name -display -pindata` command to report delay paths and view them in the GSV. The displayed data includes any path requirements (transitions and conditions) annotated to the wires of the design or primitive elements in the path.

Path Delay ATPG Options

Fast-sequential ATPG is the default for path delay tests and usually provides adequate coverage of most testable paths. In some cases, full-sequential ATPG can achieve slightly higher coverage. The recommended flow is to first generate path delay patterns with fast-sequential ATPG, then top off with full-sequential patterns, if they provide improvement. You can enable full-sequential ATPG using the `-full_seq_atpg` option of the `set_atpg` command. The following options can improve vector generation and pattern compression with full-sequential ATPG:

```
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 can increase the time limit beyond 10 seconds (the default), and rerun ATPG on the remaining faults. Raising the merge effort allows TestMAX ATPG to generate fewer vectors for the same fault coverage. 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 before 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 before 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, is observed by an unload operation. Full-sequential ATPG will continue to apply the “robust fill” algorithm before random fill. This decreases the probability that the launch clock creates a transition from scan cells feeding off-path inputs, including any with this attribute.

Creating At-Speed WaveformTables

Path delay tests are generated during both the fast-sequential and full-sequential test modes. 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 STIL procedure file.

If there are no additional waveform tables in the STL procedure file, then the default timing (`_default_WFT_`) is 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 as follows:

- `_launch_WFT_`
- `_capture_WFT_`
- `_launch_capture_WFT_`

When using generic capture procedures, the `allclock_launch`, `allclock_capture`, and `allclock_launch_capture` procedures are used. Each procedure calls a WFT specifically associated with it.

Each table can use different timing definitions 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 example shows a `_capture_WFT_timing` WaveformTable in the context of a STL procedure 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 full-sequential delay fault requiring a launch on the rising (leading) edge of a clock and a capture on the falling (trailing) edge of the same clock generates 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 the following figure).

```
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; } }
  }
}

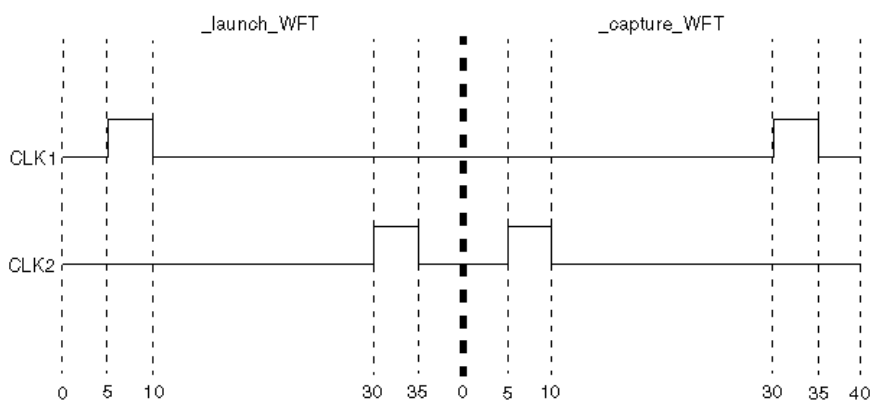
WaveformTable "_capture_WFT_" {
  Period '40ns';
  Waveforms {
    "CLK1" { 01Z { '0ns' D/U/Z; } }
```

```

"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 132 Two Different At-Speed Times



Maintaining At-Speed 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 TestMAX ATPG (see [Selecting the Pattern Source](#)), specify the appropriate fault model for these patterns. See “Specifying Transition-Delay Faults” for transition patterns for the appropriate `set_faults -model` command.

MUXClock Support for Path Delay Patterns

Testing of internal paths in DSMTest 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. MUXClock is supported only for full-sequential ATPG, so you must use the `set_atpg -full_seq_atpg -nofast_path_delay` command.

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 STIL procedure 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
- MUXClock is not supported with scan compression designs

Enabling MUXClock Functionality

The waveform table sections in the STL procedure file 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

The following example shows a 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; } }
}
```

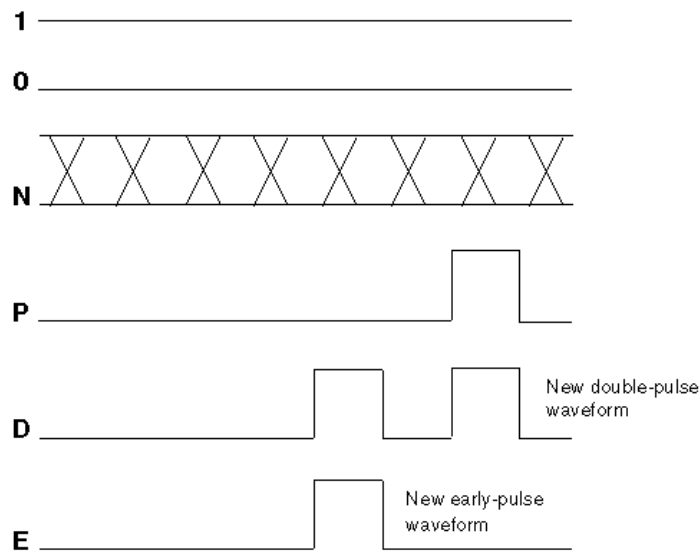
```
"CK" { E { '0ns' D; '45ns' U; '55ns' D; } }
}
}
}
```

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 2

Figure 133 MUXClock: Active-High Clock Waveforms



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 for the WGL signals and timeplate sections is as follows.

```
"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 TestMAX ATPG.
- 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.

Handling Untested Paths

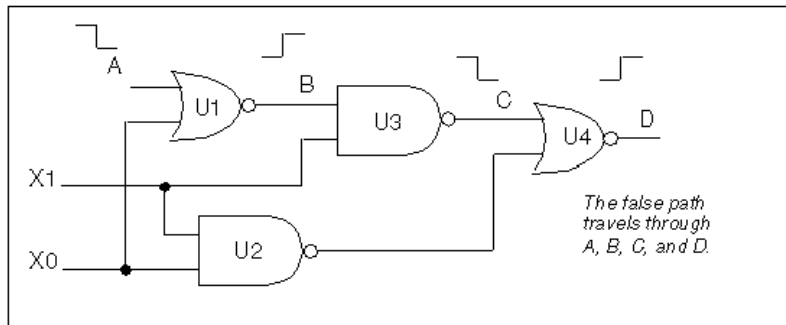
This following sections explain false and untestable paths, and describes how you can handle them:

- [Understanding False Paths](#)
- [Understanding Untestable Paths](#)
- [Reporting Untestable Paths](#)
- [Analyzing Untestable Faults](#)

Understanding False Paths

A false path might 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. The following figure illustrates a false path, ABCD.

Figure 134 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. TestMAX ATPG identifies combinational 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).

Understanding Untestable Paths

TestMAX ATPG DSMTest might 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 might 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.

If there are reconverging paths, you might 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 TestMAX ATPG and are declared untestable.

Multicycle paths can often be tested if the appropriate clock timing is applied

- Other TestMAX ATPG restrictions might 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 TestMAX ATPG and are declared untestable.

Reporting Untestable Paths

A specific path delay fault might 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.

The following example shows how to review untestable paths after ATPG:

```
TEST-T> report_faults -class AU
```

```
str AN path8
```

```
str AN path9
```

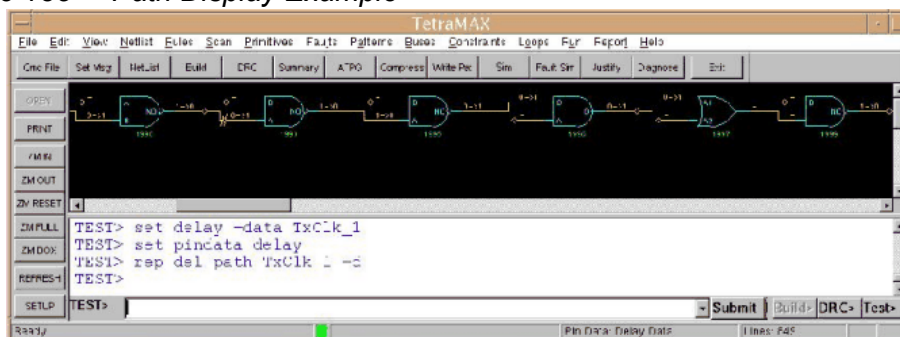
To display the delay for a particular path use the following command:

```
TEST-T> report_delay_paths 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.

The following figure shows a path being displayed in the GSV.

Figure 135 Path Display Example

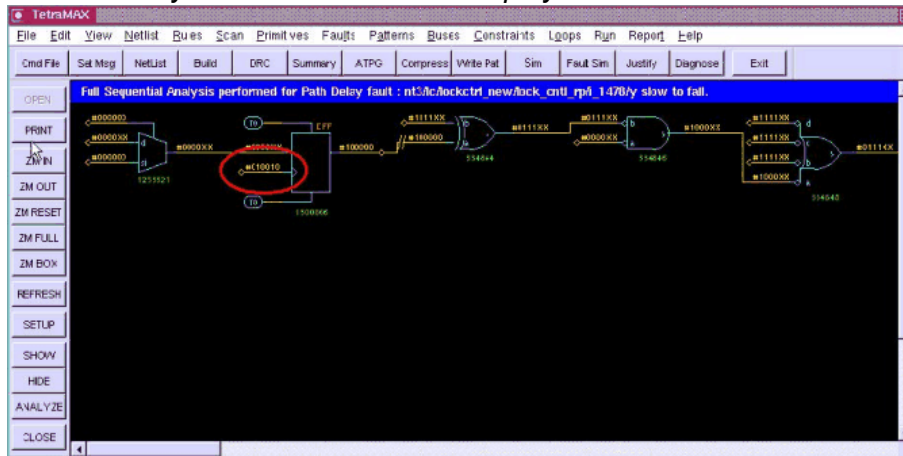


Analyzing Untestable Faults

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, the following figure shows the displayed result of entering

```
analyze_faults CLK_0 -slow f -display
```

Figure 136 Analyze Untestable Faults Display



TestMAX ATPG 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.

TestMAX ATPG 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 following:

- The `set_delay -mask_nontarget_paths` command, which ensures that TestMAX ATPG does not generate expected values on multi cycle or false paths
- The `set_delay -relative_edge` command, which causes TestMAX ATPG 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_model 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 required

add_capture_masks dff0 # if needed
add_slow_cells dff1 # if needed
add_slow_bidi -all

add_pi_constraints 0 scan_enable

run_drc ckt.spf

add_delay_paths ckt.paths

set_faults -model path
add_faults -all

run_atpg -auto

report_patterns -all -path_delay # if required
report_patterns -all -slack # if required

# You can optionally run the following command
analyze_faults path0 -slow r -verbose -display -fault_sim
```

24

Quiescence Test Pattern Generation

TestMAX ATPG 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.

The following topics describe the process for IDDQ test pattern generation:

- [Why Do IDDQ Testing?](#)
- [About IDDQ Pattern Generation](#)
- [Fault Models](#)
- [DRC Rule Violations](#)
- [Generating IDDQ Test Patterns](#)
- [Using IDDQ Commands](#)
- [IDDQ Bridging](#)
- [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.

The following sections describe IDDQ testing in detail:

- [CMOS Circuit Characteristics](#)
- [IDDQ Testing Methodology](#)

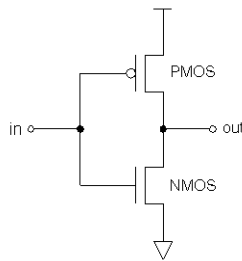
- [Types of Defects Detected](#)
- [Number of IDDQ Strokes](#)

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.

The following figure 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 137 CMOS Inverter Schematic Diagram



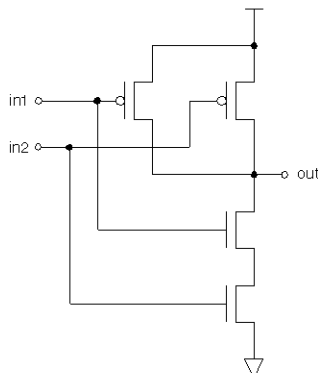
When the input is low, the upper transistor is on and the lower transistor is off, which pulls the output up to the supply voltage (VDD). When the input is high, the upper transistor is off and the lower transistor is on, which pulls the output to ground.

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 the following figure. 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 138 CMOS NAND Gate Schematic Diagram

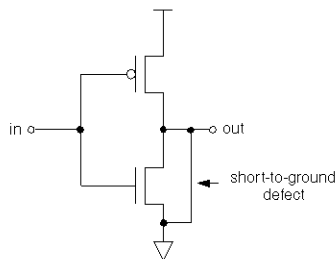


IDDQ Testing Methodology

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 the following figure. 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 139 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 the preceding figure, 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

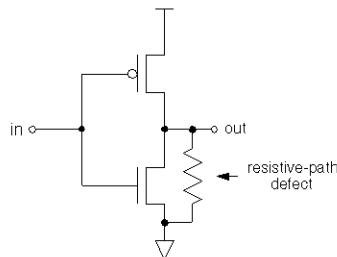
IDDQ testing can detect many kinds of circuit defects that are difficult or impossible to detect by functional or stuck-at testing, such as three-state enable nodes, redundant logic, high-resistance faults, scan chain control/data paths, undetectable faults, possibly detected faults, ATPG untestable faults, and bridging faults.

For example, consider the defect shown in the following figure, 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 140 Resistive Path to Ground



Number of IDDQ Strokes

IDDQ testing can provide very high fault coverage with just a few strokes. The first IDDQ strobe typically detects half of all short-to-ground and short-to-VDD faults. IDDQ test

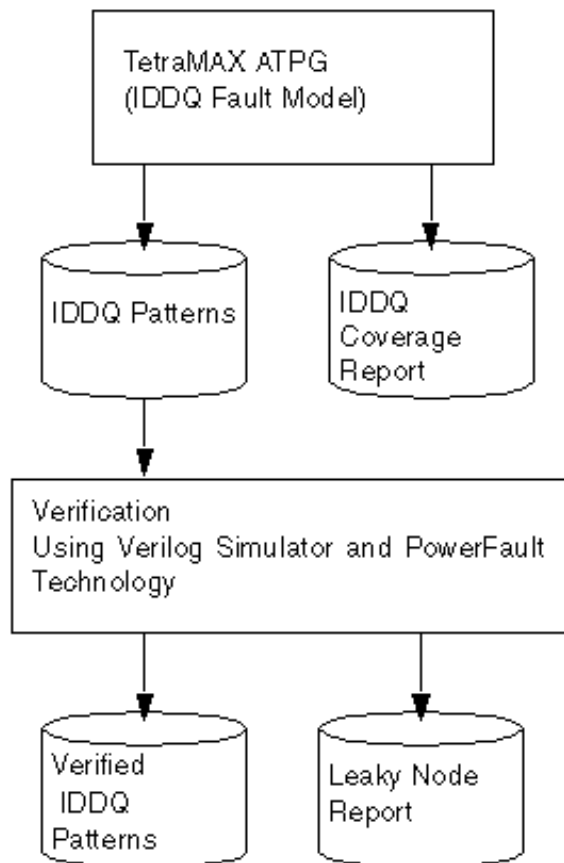
patterns attempt to change or toggle as many nodes as possible in subsequent patterns to quickly increase fault coverage.

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 required 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

The following figure shows the IDDQ testing flow using TestMAX ATPG test-pattern generation. The ATPG algorithm attempts to sensitize all IDDQ faults and apply IDDQ strobes to test all such faults. TestMAX ATPG compresses and merges the IDDQ test patterns, just like ordinary stuck-at patterns.

Figure 141 IDDQ Testing Flow



While generating IDDQ test patterns, by default TestMAX ATPG avoids any condition that could cause excessive current drain, such as strong or weak bus contention or floating buses.

TestMAX ATPG 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 TestMAX ATPG. Refer to the *Test Pattern Validation User Guide* for details about PowerFault.

TestMAX ATPG 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 TestMAX ATPG 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.

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 Strokes in TestMAX ATPG Stuck-At Patterns” in the *Test Pattern Validation User Guide*

Fault Models

TestMAX ATPG 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, TestMAX ATPG 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. In this case, TestMAX ATPG attempts to sensitize each node in the design, but does not try to propagate faults to the device outputs.

TestMAX ATPG supports two IDDQ fault models:

- *Pseudo-Stuck-At Fault Model (the default)*

This fault 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) 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*

This fault model is a simple, net-only model that does not consider gate functionality. Each fault site only needs to have its state controlled 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

TestMAX ATPG 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](#).

To view a list of rule violations after you perform design rule checking, use the `report_rules` command. The following example shows a typical DRC violation report.

DRC Violation Report

```
TEST-T> 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
```

The following table lists TestMAX ATPG design rule violations that warrant investigation if you plan to generate IDDQ test patterns.

Table 7 DRC Rule Violations and IDDQ Significance

Rul	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
B1 2	Undriven instance input pin, error	Likely to be a floating net
B1 8	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 TestMAX ATPG design rule checking, see [Performing Test Design Rule Checking](#).

Generating IDDQ Test Patterns

The following sections describe how to generate IDDQ test patterns:

- [IDDQ Test Pattern Generation Flow](#)
- [Using the iddq_capture Procedure](#)
- [Off-Chip IDDQ Monitor Support](#)

IDDQ Test Pattern Generation Flow

The following steps show you how 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 strobos 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-T> set_faults -model iddq
TEST-T> set_iddq -toggle # pseudo-stuck-at is the default
TEST-T> add_faults -all
TEST-T> set_atpg -patterns 20 # budget of 20 IDDQ strobos
TEST-T> 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 strobos from the pattern set, see “Selecting Strobos in TestMAX ATPG Stuck-At Patterns” in the *Test Pattern Validation User Guide*.

Using the `iddq_capture` Procedure

When you create IDDQ patterns, TestMAX ATPG defines a procedure, called `iddq_capture`, in the pattern output file. This procedure (shown in the following example) is used when an IDDQ measure is performed:

```
"iddq_capture" {
W "_default_WFT_";
F { "testmode"= 1; }
V { "_pi"=\r379 # ; "_po"=\j \r276 X ; }
```

```
IddqTestPoint;  
V { "_po"=\r276 # ; }  
}  
}
```

TestMAX ATPG generates the default `iddq_capture` procedure when IDDQ test patterns are written and the input STL procedure file does not define an `iddq_capture` procedure. If you not define the `iddq_capture` procedure in the input STL procedure file, make sure you specify the `write_drc_file` command after the `write_patterns` command so the `iddq_capture` procedure is preserved in the output STL procedure file.

You should use the new STL procedure file in subsequent runs to provide the same `iddq_capture` procedure. Since default flows change in various releases, it is important to preserve the default behavior for these patterns. When WGL IDDQ patterns are written, V4 errors will occur if these patterns are read in a context that does not define the `iddq_capture` procedure. You can eliminate these problems in subsequent flows by saving the new STL procedure file with the complete set of procedures.

You can also define customized `iddq_capture` procedures in the STIL procedure file and pass them into the flow.

Off-Chip IDDQ Monitor Support

You can 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 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.

The following sections describe how to include off-chip IDDQ monitor signals in your testing:

- [Specifying Additional Signals in the Netlist](#)
- [Defining the `iddq_capture` Procedure to Support Additional Signals](#)

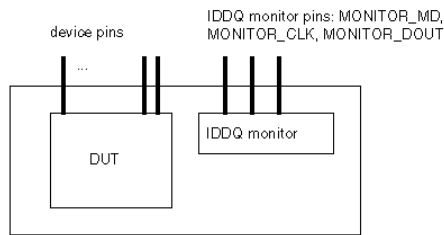
Specifying Additional Signals in the Netlist

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 the following figure. 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 142 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 the `iddq_capture` Procedure 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 TestMAX ATPG, 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; } }

```

Chapter 24: Quiescence Test Pattern Generation Generating IDDQ Test Patterns

```

    "_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;

```

Chapter 24: Quiescence Test Pattern Generation Generating IDDQ Test Patterns

```

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; } }

}

}

}

```

Chapter 24: Quiescence Test Pattern Generation Generating IDDQ Test Patterns

```

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)
}
}
```

Using IDDQ Commands

You can use the `set_faults` command to set the fault model. If you select the IDDQ fault model, you can use the `set_iddq` command to specify the quiescence constraints and toggle/no-toggle model type. The `add_atpg_constraints` command lets you set IDDQ-specific ATPG constraints on nodes in the design.

These commands are described in the following sections:

- [Using the set_faults Command](#)
- [Using the set_iddq Command](#)
- [Using the add_atpg_constraints Command](#)

Using the set_faults Command

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.

Using the set_iddq Command

The `float`, `strong`, `weak`, and `write` options of the `set_iddq` command allow you to specify the conditions required for quiescence. TestMAX ATPG 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 in the following manner:

- 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.

Using the `add_atpg_constraints` Command

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.

IDDQ Bridging

You can use the IDDQ bridging fault model to generate additional patterns and increase the IDDQ coverage. This fault model, which is specified using the `set_faults -model iddq_bridging` command, behaves differently than the regular IDDQ fault model. Regular IDDQ fault model has two versions of the same model (Toggle or Pseudo-Stuck-At). The fault model for IDDQ bridging is only of type Toggle. This means that the fault site at a gate input does not require propagation to an output of the same gate to be given credit for IDDQ bridging fault detection.

In regular scan mode, unload values are written in the patterns to facilitate load/unload overlapping by the tester. However, capture is not in effect when using the IDDQ bridge fault model, and the unload values are exactly the same as the load values.

When the IDDQ bridging fault model is specified, the `set_iddq -toggle` command is invalid because only one version of the model is available.

The IDDQ bridging fault model is similar to the bridging fault model except that bridging faults are directly observed by an IDDQ strobe rather than by propagating the fault effect to a scan cell. The primary purpose of performing IDDQ bridging fault ATPG is to detect faults by inserting correct values into the fault nodes. As a result, there are no observation requirements. For example, to detect the IDDQ bridging fault named `ba0 node1 node2`, where the aggressor node is `node1` and victim node is `node2`, ATPG sets the `node1` logic value to 0 and the `node2` logic value to 1.

The IDDQ bridging fault model uses the same fault codes as the bridging fault model. The existing `add_faults -node_file` and `-bridge_location` options are used to read net pairs and add the IDDQ bridging faults. The IDDQ bridging measurement criteria is adjusted by a separate `set_iddq` command, as is the case with the regular IDDQ fault model. During ATPG, the detection of one pair of bridges implies the detection of another pair. This behavior can be controlled using the `set_iddq -bridge_equivalence` command.

The flow to generate IDDQ bridging patterns is as follows:

```
set_fault -model iddq_bridging

# optional setting of measurement criteria

set_iddq nofloat

add_faults -node pair.txt

run_atpg -auto
```

```
write_patterns iddq_bridging.stil -format stil
```

Note the following limitations related to IDDQ bridging ATPG:

- The `analyze_faults` command is not supported when using the IDDQ bridging fault model.
- The strength-based optimizations used for the regular bridging fault model are not supported for the IDDQ bridging fault model.
- Full-sequential ATPG does not support the IDDQ bridging fault model.

Design Principles for IDDQ Testability

The following design principles apply to designing your circuits for IDDQ testability:

- [I/O Pads](#)
- [Buses](#)
- [RAMs and Analog Blocks](#)
- [Free-Running Oscillators](#)
- [Circuit Design](#)
- [Power and Ground](#)
- [Models With Switch/FET Primitives](#)
- [Connections](#)
- [IDDQ Design-for-Test Rule Summary](#)

IDDQ testing and PowerFault simulation is more efficient and reliable if you follow these requirements. 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 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. It must be possible to stop the clock. The core must have a bypass clock signal from the tester (a primary I/O).

25

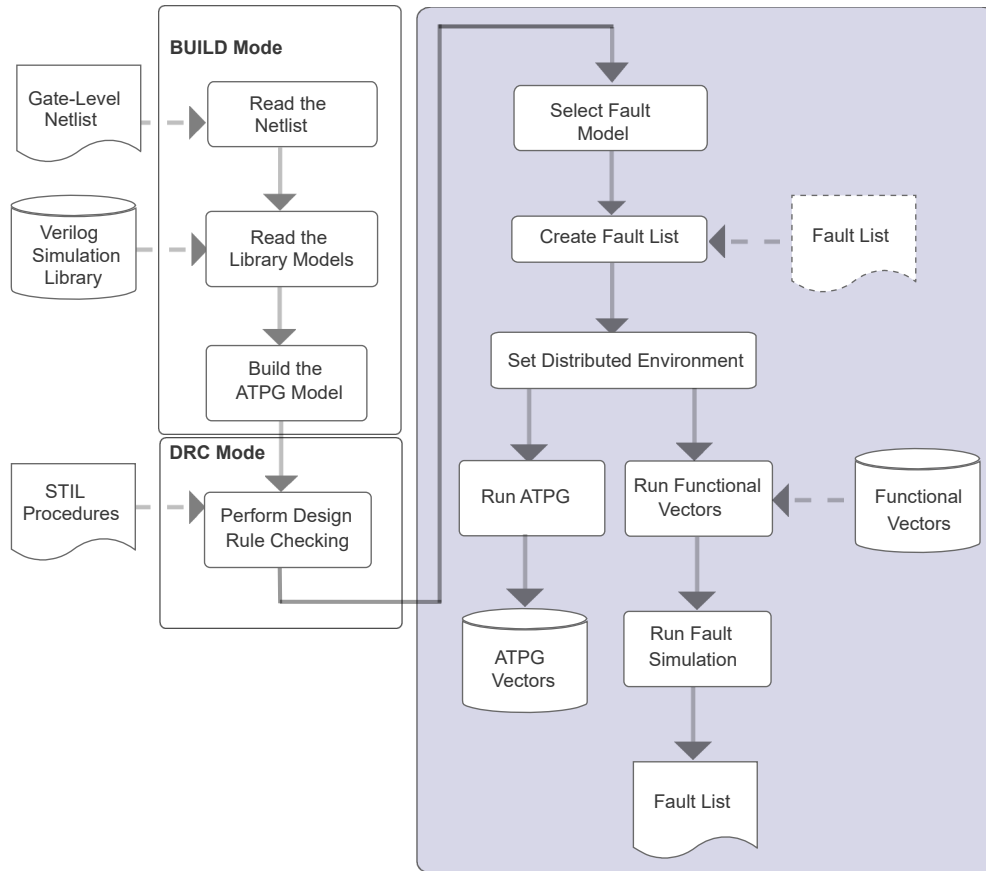
Running Distributed ATPG

TestMAX ATPG distributed ATPG launches multiple slave processes on a computer farm or on several standalone hosts. The slave processes read an image file and execute a fixed command script. ATPG distributed technology does not differentiate between multiple CPUs and separate workstations. Each slave requires as much memory as a single CPU TestMAX ATPG run. Distributed ATPG offers both scalability and runtime improvement.

The following topics describe how to set up and run distributed ATPG:

- [Debugging Name Matching Errors](#)
- [Checking Your Environment for Distributed Processing](#)
- [Machine Access and Setup for Distributed ATPG](#)
- [Preparing to Run Distributed Processing](#)
- [Setting Up the Distributed Environment](#)
- [Setting Up the Distributed Environment With Load Sharing](#)
- [Verifying Your Environment](#)
- [Starting Distributed ATPG](#)
- [Starting Distributed Fault Simulation](#)
- [Debugging Distributed ATPG Issues](#)
- [Distributed ATPG Limitations](#)

Figure 143 Distributed Processing Flow



Debugging Name Matching Errors

When you perform physical diagnostics, it is crucial that the instance names match between the logic netlist and the PHDS (physical design store) database. If mismatches exist, TestMAX ATPG diagnostics cannot accurately extract the physical data used for physical diagnostics.

You can use the `he match_names` command to create a report that identifies name mismatches between the logic and physical databases. The process for creating this name matching report is described in [Performing Name Matching](#).

Name matching errors typically occur when an instance is missing either in the netlist or in the PHDS database, or an instance name uses a hierarchy in the netlist different from the one in the PHDS database.

The following sections describe how to debug name matching errors:

- [Debugging Missing Instances](#)
- [Debugging Hierarchical Mismatches](#)

Debugging Missing Instances

A missing instance is the source of a mismatch when a memory module is included in the logic netlist but its definition is missing in the LEF/DEF database. To identify this problem, you need to examine one mismatch at a time and search specifically for memory name mismatches. This process is described in the following steps:

1. Start TestMAX ATPG, read the design image, and connect to an existing PHDS database, as described in steps 1 through 3 of [Performing Name Matching](#).
2. Specify the `match_names -sample -verbose` command, and redirect the output to a file, as shown in the following example:

```
match_names -verify sample -verbose > match_report.txt
```
3. Identify all the memory names in the `match_names` command output report.
4. Specify the `report_instances` command to determine if the identified instances are defined in TestMAX ATPG diagnostics. For example:

```
report_instances I_TOP/RECEIVER/PACKET_CTRL/I_MEM24x16
```
5. Search the LEF/DEF files for the instances identified in the output report. For example, you can apply the `grep` command to all the provided DEF files:

```
UNIX> grep I_TOP/RECEIVER/PACKET_CTRL/I_MEM24x16 ./DEF/TOP.def
```
6. If you discover that a memory module (or any other module) in the output file is not present in the LEF/DEF files, you can use the `-exclude` option of the `set_match_names` command to exclude the module from the matching process. For example:

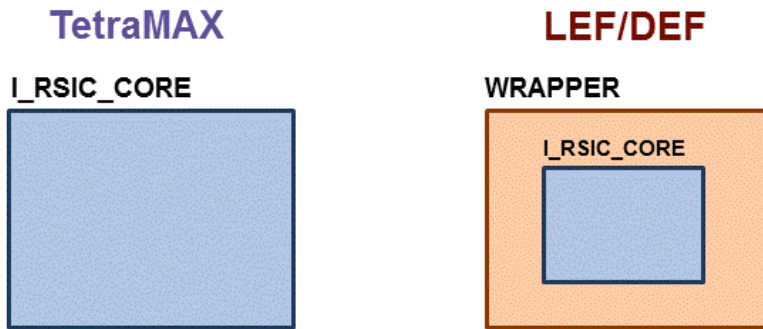
```
set_match_names -exclude DPMEM64x128
```

Alternatively, you can try to obtain the missing module from the source that originally provided the LEF/DEF files.

Debugging Hierarchical Mismatches

Hierarchical mismatches are usually caused when a wrapper is used in the logic or physical version of a design. For example, a physical design might contain an extra level of hierarchy compared to the logic design, as shown in the following figure.

Figure 1 Example of Hierarchy Difference When Using a Wrapper in a Physical Design

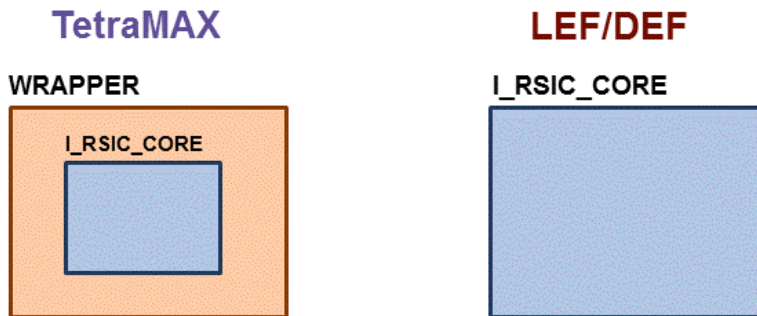


To fix a hierarchical mismatch issue similar to the example in aforementioned figure, use the `-sub_prefix` option of the `set_match_names` command, as shown in the following example:

```
set_match_names -sub_prefix {I_RISC_CORE WRAPPER/I_RISC_CORE}
```

It is also possible that the logic design might have an extra level of hierarchy compared to the physical design, as shown in the following figure.

Figure 2 Example of Hierarchy Difference When Using a Wrapper in a Logical Design



To fix a hierarchical mismatch similar to the example shown in the aforementioned figure, use the `-sub_prefix` option of the `set_match_names` command, as shown in the following example:

```
set_match_names -sub_prefix {"WRAPPER/" ""}
```

After identifying and fixing the name mismatches, you can rerun the `match_names` command, check for any additional mismatches, and debug them as necessary.

See Also

- [Understanding the Name Matching Coverage Report](#)
- [Physical Diagnosis Overview](#)

Checking Your Environment for Distributed Processing

Perform the following tasks to make sure you can run ATPG distributed processing:

1. Verify that you can use the UNIX `rsh` command to log in on each slave machine without having to supply a password, as shown in the following example:

```
rsh <machine_name>
```

See [Machine Access and Setup](#).

2. Verify that each slave machine has an identical search the path to TestMAX ATPG, as shown in the following example:

```
rsh <machine_name> which tmax
```

See [Machine Access and Setup](#).

3. If you are using LSF for launching slaves, find out from your system administrator what queues or special options you need to use. The machine that you run as master should be a valid LSF submit host. You can verify that it is an LSF host by trying a sample LSF job.

```
/path/to/bsub -q hw-vlsi tmax -shell -version
```

4. If you are using GRD for launching the slave, ask your system administrator if you need to use any special project name or queue. The machine that you run as master should be a valid GRID submit host. You can verify that it is a grid host by trying a sample grid job.

```
/path/to/qsub $SYNOPSIS/bin/tmax -shell -version
```

Machine Access and Setup for Distributed ATPG

Distributed ATPG makes use of the UNIX `rsh` command to login and launch distributed processing on slave machines. To setup your system:

1. Make sure that you have network access to each slave machine. You should successfully perform step 2 described previously, which verifies that you can log in on each slave machine without having to supply a password.

If you are prompted for a password, create a `.rhosts` file in your home directory.

Add to that file the names of the slave machines in a list format. This list categorizes the slave machines as "trusted hosts."

Another technique is to put a single "+" entry (without the quotation marks) in your `.rhosts` file.

If you are still not able to `rsh` to a machine after creating a `.rhosts` file, refer to the manpage for `rsh` or talk to your System Administrator or contact your IT group.

Inability to use `rsh` will result in a M316 error message when you attempt to run distributed ATPG.

2. Make sure TestMAX ATPG is set up identically on each slave machine. You should successfully do step 3 above, which verifies that each slave machine accesses TestMAX ATPG through the same network path. If this is not the case
3. There might be something in your `.cshrc` file (or equivalent) might be preventing the machine from locating TestMAX ATPG on the network. This could happen if you have `tty`, `stty`, or interactive statements in your `.cshrc` file. To debug this, add `echo` statements to your `.cshrc` file to determine where the search fails.
4. If you cannot fix your `.cshrc` file or you are not setting the path to TestMAX ATPG through your `.cshrc` (or equivalent file), you can use the `-script` option of the `set_distributed` command to pass a script for setting up TestMAX ATPG on the slave. Here's an example `csh/sh/tcsh` script:

```
#!/bin/csh -f setenv SYNOPSISYS /tools/tmax/V-2003.12 set path  
($SYNOPSISYS/bin $path) tmax $*
```

The inability to set the search path to TestMAX ATPG on the slave machines causes a M315 error message when you attempt to run distributed ATPG.

Preparing to Run Distributed Processing

Before running distributed processing, you need to build the design model and run DRC.

Example Script:

```
BUILD-T> read_netlist Libs/*.v -delete -library -noabort  
BUILD-T> run_build_model top_level  
DRC-T> set_drc top_level.spf  
DRC-T> run_drc
```

For details, see "Building the ATPG Model" and "Running DRC."

You also need to select the fault model and create the fault list. Note that N-detect ATPG and fault simulation are not supported for distributed ATPG.

The following is a summary of the support for distributed ATPG and the various fault models:

- The stuck-at fault model is supported by basic-scan, fast-sequential, and full-sequential ATPG
- The path delay fault model is supported by fast-sequential, and full-sequential ATPG
- The transition fault model is supported by basic-scan, fast-sequential, and full-sequential ATPG
- The IDDQ fault model is supported by basic-scan and fast-sequential ATPG
- The bridging fault model is supported by basic-scan and fast-sequential ATPG

The following example scripts are for selecting fault models for distributed processing:

Example 1:

```
TEST-T> set_faults -model stuck
TEST-T> add_nofaults top_level/module1/sub_mod
TEST-T> add_faults -all
```

Example 2:

```
TEST-T> set_faults -model transition
TEST-T> set_delay -nopi_change -nopo_measure
TEST-T> set_delay -launch last_shift
TEST-T> read_faults specFaultList.flt
```

Setting Up the Distributed Environment

Defining the working directory is the first step in setting up the environment for distributed processing. The working directory stores all the files required for exchanging data between the various machines, including the log files. 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, as shown in the following example:

```
TEST-T> 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 TestMAX ATPG 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 can populate the distributed processors list; for example:

```
TEST-T> 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
```

The following 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.

TestMAX ATPG supports heterogeneous machine architectures (sparcOS5, Linux, and HP-UX). For example,

```
TEST-T> 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-T> 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 might 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-T> remove_distributed_processors zelda

TEST-T> 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-T> report_settings distributed

distributed = shell_timeout=30, slave_timeout=100,
print_stats_timeout=30, verbose=-noverbose,
shell=rsh;
```

Setting Up the Distributed Environment With Load Sharing

TestMAX ATPG 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 is 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.

Chapter 25: Running Distributed ATPG Setting Up the Distributed Environment With Load Sharing

Note the following example:

```

BUILD-T> add_distributed_processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 4 \
-options "-q lb0202"
BUILD-T> 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. After you issue the `run_atpg -distributed` command (or the `run_fault_sim -distributed` command), four jobs are assigned to four distributed processors. However, this is 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-T> add_distributed_processors \
-lsf /u/tools/LSF/mnt/2.2-glibc2/bin/bsub -nslaves 3 \
-options "-q lb0202"
BUILD-T> report_distributed_processors
Working directory ==> "/remote/dtgnat/Distributed"
(32bit)
-----****-----

```

```
MACHINE **lsf** [ARCH: linux]
MACHINE **lsf** [ARCH: linux]
MACHINE **lsf** [ARCH: linux]
-----***-----
```

Verifying Your Environment

Each time TestMAX ATPG 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 TestMAX ATPG 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 SYNOPSIS /softwares/synopsis/2004.12
set path = ( $SYNOPSIS/bin $path )
```

For a discussion about the use of the `SYNOPSIS_TMAX` environment variable, see “Specifying the Location for TestMAX ATPG Installation”.

Remote Shell Considerations

If you are running distributed processing by directly running the host names, TestMAX ATPG 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 might 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 halt 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 wait for a process that has quit running.

If you have any trouble using the `add_distributed_processors` command, you might 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, make sure you have the following information available to you.

- Path to the load sharing application(`bsub` for LSF and `qsub` for GRID)
- Required options (like project name or queue name)

The TestMAX ATPG session for the master must be run on a machine that is a valid submit host capable of submitting jobs to load sharing software.

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-T> 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. Note that the reported faults will not add up: there is a difference between collapsed and non-collapsed faults. The master only sends active faults.

Each slave contains all the faults. In this case, the fault list is not split; only the ATPG process is split. The slave log files try to report uncollapsed faults, but since they only receive collapsed faults information, the number reported is actually collapsed faults.

The following example shows how the master collapse fault list correlates to the slaves uncollapsed fault list. In this case, 1715195 is the key number of faults that appear in both reports. Note that even though the slave file reports the faults as uncollapsed, the faults are actually the collapsed list from the master.

Comparing the Master Collapse Fault List to the Slaves Uncollapsed Fault List

```

From master log file:
report_faults -summary -collapse
Collapsed Stuck Fault Summary Report
-----
fault class code #faults
-----
Detected DT 802240
Possibly detected PT 0
Undetectable UD 34404
ATPG untestable AU 127879
Not detected ND 1715195
-----
total faults 2679718
test coverage 30.33%
-----

run_atpg -auto -dist
Master: Saving image of session for slaves ...
Master: Spawning the slaves ...
Master: Starting distributed process with 3 slaves ...
Slaves: About to get licenses ...
Slaves: About to restore master's session ...
Master: Removing temporary files ...
Master: Sending 1715195 faults to slaves ...
Master: End sending faults. Time = 14.00 sec.
From slave log file:
run_atpg -auto
*****
* NOTICE: The following DRC violations were previously *
* encountered. The presence of these violations is an *
* indicator that it is possible that the ATPG patterns *
* created during this process might fail in simulation. *
* *
* Rules: C8 *
*****
ATPG performed for stuck fault model using internal pattern source.
-----
#patterns #patterns #faults #ATPG faults test process
simulated eff/total detect/active red/au/abort coverage CPU time
-----
Begin deterministic ATPG: #uncollapsed_faults=1715195, abort_limit=10...

```

If you have some vectors in the external buffer before starting distributed ATPG, they are automatically transferred into the internal buffer. The new vectors created during distributed ATPG are 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_patterns -delete` command.

As the following transcript shows, TestMAX ATPG 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 TestMAX ATPG cannot proceed. Then, TestMAX ATPG starts generating the vectors.

```
run_atpg -auto -distributed
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 TestMAX ATPG.

`#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, TestMAX ATPG automatically prints a summary for the faults and the vectors.

When using the `set_atpg -patterns max_patterns` command, for some designs that run quickly, the master sends a signal to stop the slaves. However, because of a network delay for this signal, the pattern count is already met; therefore the pattern count might already have exceeded the limit.

Saving Results

The following sample script shows you how to save results:

```
TEST-T> set_faults -summary verbose
TEST-T> report_faults -summary
TEST-T> report pattern -summary
TEST-T> write_faults final.flt -all -collapsed -compress gzip -replace
TEST-T> write_patterns final_pat.bin.gz \
-format binary -compress gzip -replace
TEST-T> 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_messages log run.log -replace` command, a command that indicates you are running distributed ATPG with four slaves, the log files that are 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.

Starting Distributed Fault Simulation

After the functional vectors are read into the TestMAX ATPG external pattern buffer, you simply need to add the `-distributed` option to the `run_fault_sim` command to trigger the parallelization of the process; for example:

```
TEST-T> 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), TestMAX ATPG 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-T> 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 TestMAX ATPG has already processed

Debugging Distributed ATPG Issues

You might encounter the following issues when setting up to run distributed ATPG:

- If you specify the host directly, verify your environment based on information provided in "Checking your Environment" above. If you are using the `-script` option of the `set_distributed` command and your command file contains lines similar to this:

```
set_distributed -script /home/pjaini/datpg_setup
add_distributed_processors yosemite goldengate
```

Enter something similar to the following:

```
rsh yosemite /home/pjaini/datpg_setup -shell -version
```

This should return the proper version of TestMAX ATPG and you should not see any additional messages.

- For GRID, assume your command entries look like the following to launch the slave processors:

```
set distributed processor -script /user/larry/tmax_launch
```

```
add distributed processor -grd "/hw/tools/grid/qsub" -nslaves 5 -options "-P hw-rush"
```

Then, in an xterm window, enter:

```
% /hw/tools/grid/qsub -P hw-rush /user/larry/tmax_launch -version -shell
```

Check to make sure your GRID has been launched. You should see a message from GRID indicating your job has been submitted and the TestMAX ATPG version string is printed wherever your GRID job output would normally appear. The standard output from a GRID job could be sent to you via email or stored in some file in your home directory, depending on your GRID settings.

If the job does not launch, find the set of options (that is, project name, queue name, and so forth) required for launching a GRID job.

- For LSF, assume your command entry looks like the following to launch the slave processors:

```
add_distributed_processors -lsf "/path/to/bsub" -nslaves 5 -options "-q bnormal "
```

Then, in an xterm window, enter:

```
% /path/to/bsub -q normal tmax -version -shell
```

Check to make sure your LSF job has been launched. Your indication for this is an email notifying you that the job has ran and the TestMAX ATPG version string prints in the output. Note that some LSF setups might mandate the use of certain queues or project names.

- Debugging a "Master: Couldn't start the daemon ..." message. There are two conditions where this message could occur:

1. The job was never launched. To check that the job launched, enter the following when you see the "Master: Spawning the slaves.." message:

For GRID:

```
qstat -u <username>
```

For LSF:

```
bjobs -u <username>
```

If you do not get an indication that the job is running, check to make sure you can launch the jobs from xterm window as described above.

2. Jobs were launched, but are not scheduled to be executed yet. If you see that the jobs have been launched, try increasing the slave setup timeout with a `set_distributed -slave_setup` command entry.

A less likely possibility is an error with the read/write image process. Prior to running distributed ATPG, write out the image file (for example, `write_image save.img -viol`), and then read the image in a new TestMAX ATPG session (for example, `read_image save.img`). If you get an error message when you read the image in this new session, contact Synopsys Support with a testcase.

Received "Error: At least one slave died" message. This message indicates that one of slave processors terminated in the middle of pattern generation. Possible causes are:

- a slave machine was rebooted or restarted
- a slave process was explicitly killed
- a slave process ran out of memory

Distributed ATPG Limitations

The following limitations are associated with distributed ATPG:

- The `-analyze_untestable_faults` option of the `set_atpg` command is not supported.
- N-detect ATPG and fault simulation are not supported.

26

Persistent Fault Model Support

TestMAX ATPG supports a variety of fault models that abstractly represent real-world defects. Supported models include the stuck-at, transition, path delay, bridging, dynamic bridging, and IDDQ models. These models are implemented in a serial manner, which means that only one model is active at any time.

When you change fault models, TestMAX ATPG flushes the current fault list from memory, along with any internal patterns, and starts again from scratch.

The single-fault model approach is inefficient in terms of pattern count and runtime. A set of transition patterns, for example, will also detect a certain number of stuck-at faults; but transition ATPG does not recognize them. Before you run stuck-at ATPG, you can reduce the stuck-at pattern count by fault-grading the stuck-at fault list against the transition patterns to prune previously detected faults. The stuck-at pattern reduction can be quite significant. Conversely, when you generate transition and stuck-at patterns in isolation, you waste time and patterns generating tests for stuck-at faults that were already detected by the transition patterns.

Persistent fault model support helps you manage multiple fault model flows easily by providing an automated way for TestMAX ATPG to perform the following operations:

- [Persistent Fault Model Overview](#)
- [Persistent Fault Model Operations](#)
- [Direct Fault Crediting](#)
- [Example Commands Used in Persistent Fault Model Flow](#)

Persistent Fault Model Overview

The persistent fault model flow is enabled by the `set_faults -persistent_fault_models` command.

This flow enables the following behaviors:

- The fault list for the active fault model is saved in a cache. When you return to an inactive fault model, the saved faults are restored. When path delay faults are preserved, the delay paths are also preserved.
- The `report_faults -summary` command prints the total number of faults and the test coverage for each inactive fault model.
- All other fault-oriented commands continue only to affect the active fault model. This includes, but is not limited to, the following commands: `run_atpg`, `run_fault_sim`, `write_faults`, `read_faults`, `add_faults`, and `remove_faults`.
- The fault lists are preserved when you switch to DRC mode. You can't interact with the lists in DRC mode, but they will still be available when you return to TEST mode. ATPG untestable faults (AU) are automatically reset for any fault list that was in the cache during DRC mode.
- Faults detected in the transition fault model are credited as equivalent stuck-at detects without fault simulation. This is activated by the `update_faults -direct_credit` command.

See Also

- [IDDQ Testing](#)

Persistent Fault Model Operations

The following sections describe the primary processes associated with the persistent fault model flow:

- [Switching Fault Models](#)
- [Working With Internal Pattern Sets](#)
- [Manipulating Fault Lists](#)
- [Reporting Persistent Fault Models](#)

See Also

- [Working with Fault Lists](#)
- [What Are Fault Models?](#)

Switching Fault Models

You can set a different fault model in the persistent fault model flow, even if you have faults in the active fault model, as shown in the following example:

Example Commands Used for Switching Fault Models

```
set_faults -persistent_fault_models
set_faults -model transition
add_faults -all
run_atpg -auto
(Transition faults exist)
set_faults -model bridging
```

In this case, if you do not set the `-persistent_fault_models` option, TestMAX ATPG will issue an M106 error.

Working With Internal Pattern Sets

The internal pattern set, and generated patterns in general, are preserved even if the fault model is changed in the persistent fault model flow. This means you can run fault simulation with an alternate fault model, as shown in the following example.

Running Fault Simulation With an Alternative Fault Model

```
set_faults -persistent_fault_models
set_faults -model stuck
add_faults -all
set_faults -model transition
add_faults -all
run_atpg -auto
set_faults -model stuck
(Transition fault patterns are preserved as internal pattern set)
update_faults -direct_credit
run_fault_sim
```

If you need to change primary input (PI) constraints or the STL procedure file, you still need to return to DRC mode. After you are in DRC mode, the saved internal pattern set is deleted.

Manipulating Fault Lists

The following topics explain how to manipulate fault lists:

- [Automatically Saving Fault Lists](#)
- [Automatically Restoring Fault Lists](#)

- [Removing Fault Lists](#)
- [Adding Faults](#)

Automatically Saving Fault Lists

A fault list is automatically saved in the cache as an inactive fault model when a fault model is changed or when you go back to DRC mode. The following example shows how to save fault lists automatically.

Automatically Saving Fault Lists

```
set_faults -persistent_fault_models
add_pi_constraint 1 mem_bypass
run_drc compression.spf
set_faults -model stuck
add_faults -all
set_faults -model transition
(Stuck-at fault list is saved)
add_faults -all
run_atpg -auto
drc -f
(Transition fault list is saved)
remove_pi_constraints -all
add_pi_constraint 0 mem_bypass
run_drc compression.spf
```

Automatically Restoring Fault Lists

A fault list is automatically restored from the cache as an active fault model when a fault model is reactivated or before exiting DRC mode. See the following example.

Automatically Restoring Fault Lists

```
set_faults -persistent_fault_models
add_pi_const 1 mem_bypass
run_drc compression.spf
set_faults -model stuck
add_faults -all
set_faults -model transition
(Stuck-at fault list is saved)
add_faults -all
run_atpg -auto
drc -f
(Transition fault list is saved)
remove_pi_constraints -all
add_pi_const 0 mem_bypass
run_drc compression.spf
(Transition fault list is restored)
run_atpg -auto
set_faults -model stuck
(Transition fault list is saved)
```

```
(Stuck-at fault list is restored)
update_faults -direct_credit
run_fault_sim
```

The process of automatically restoring fault lists is equivalent to executing the command `read_faults fault.list -retain_code`. Therefore, when you return to DRC mode and change the STL procedure file, you might see some minor differences in the fault summaries obtained before DRC.

Removing Fault Lists

There are several different ways to remove fault lists:

- Use the command `remove_faults -all` to remove a fault list on an active fault model.
- Use the command `set_faults -nopersistent_fault_models` to delete all inactive fault lists from the cache.
- When you return to BUILD mode, all fault lists are automatically removed.

Adding Faults

There are some precautions you need to take when adding faults. Even when the command `set_faults -persistent_fault_models` is enabled, faults cannot be added when an internal pattern set is present. The following figure shows the actual situations that are considered when adding faults.

Figure 144 Process For Adding Faults

Patterns Generated By:		Followed By:	
Fault Model	Process	Fault Model	Process
Transition	ATPG	Stuck-at	Direct fault credit
Any	ATPG	Any other	Fault simulation
Any	ATPG	Any other	ATPG

Note that the processes described in the “Followed by” column always require a fault list. However, if an internal pattern is present in any process in the “Patterns Generated By” column, you cannot add faults.

If you try to add faults to a different model when an internal pattern set is present, TestMAX ATPG will issue an M104 error.

The following example shows an example flow for adding faults.

Typical Flow For Adding Faults

```
set_faults -persistent_fault_models
set_drc compression.spf
run_drc
set_faults -model stuck
add_faults -all
set_faults -model transition
add_faults -all
run_atpg -auto
set_faults -model stuck
(You can't add faults here because internal pattern set is present)
update_faults -direct_credit
run_fault_sim
```

Reporting Persistent Fault Models

When the persistent fault model flow is enabled and inactive fault models are present, TestMAX ATPG prints additional information when any of the following conditions exist:

- TestMAX ATPG exits the DRC process
- The ATPG process is completed
- The `report_summaries` command is executed

The following example shows an example of an Uncollapsed Stuck Fault Summary Report.

Typical Uncollapsed Stuck Fault Summary Report

```
Uncollapsed Stuck Fault Summary Report
-----
fault class code #faults
-----
Detected DT 2000576
detected_by_simulation DS (1610727)
detected_by_implication DI (389849)
Possibly detected PT 0
Undetectable UD 1331
undetectable-unused UU (504)
undetectable-tied UT (491)
undetectable-blocked UB (295)
undetectable-redundant UR (41)
ATPG untestable AU 18985
atpg_untestable-not_detected AN (18985)
Not detected ND 13052
not-controlled NC (565)
not-observed NO (12487)
-----
total faults 2033944
test coverage 98.42%
-----
Inactive Fault Summary Report
```

```
-----  
fault model total faults test coverage  
-----  
Transition 1841908 96.46%  
-----
```

This report shows inactive faults list information. These numbers can be changed using the `set_faults -report [-collapsed | -uncollapsed]` command.

When you execute direct fault crediting using the `update_faults -direct_credit` command, you will see shorter reports that show how many faults are credited to DS, DI and NP. These faults are also generated by the `set_faults -report [-collapsed | -uncollapsed]` command. The following example shows an example report using this command.

Report Created Using the `update_faults -direct_credit` Command

```
update_faults -direct_credit  
# 15597 stuck-at faults were changed to DS from the inactive transition  
  fault list.  
# 0 stuck-at faults were changed to DI from the inactive transition fault  
  list.  
# 0 stuck-at faults were changed to NP from the inactive transition f  
  ault list.
```

Direct Fault Crediting

The persistent fault model flow supports direct fault crediting. To understand how this works, consider an example slow-to-rise (STR) transition fault. A pattern that detects this fault on a particular node must control that node from a 0 to a 1 and observe the result in a specified amount of time. To detect a stuck-at-0 (SA0) on the same node, only the 1 needs to be observed, and the timing is irrelevant. Thus, any slow-to-rise detection can be detected as a stuck-at-0 detection without actually simulating the transition patterns.

Direct fault crediting is enabled by running the `update_faults -direct_credit` command.

This command automatically reads back the transition fault list if it is in the cache, and it credits the following fault models:

- Dynamic bridging (victim only) faults to transition delay faults
- Dynamic bridging faults to static bridging faults
- Dynamic bridging (victim only), static bridging (victim only), and transition delay faults to stuck-at faults

The following example shows a typical direct fault crediting flow.

Typical Direct Fault Crediting Flow

```
set_faults -persistent_fault_models

set_faults -model bridging
add_faults -node_file nodes.txt

run_atpg -auto

set_faults -model transition

# (transition fault patterns are preserved as internal pattern set)

add_faults -all

run_atpg -auto

set_faults -model bridging
# update_faults -direct_credit

# (Transition fault detections can't be credited to bridging faults, so
  fault simulation is necessary)
run_fault_sim
```

The following example shows an example log of applying direct credit with four fault models.

Applying Direct Credit to Stuck-at Faults

```
set_faults -model stuck

81568 faults moved to the inactive bridging fault list.

419252 stuck faults moved to the active fault list.

3126 stuck AU faults were reset.

update_faults -direct_credit

112790 stuck faults were changed to DS from the inactive
dynamic_bridging fault list.
0 stuck faults were changed to DI from the inactive dynamic_bridging
fault list.

0 stuck faults were changed to NP from the inactive dynamic_bridging
fault list.

10121 stuck faults were changed to DS from the inactive bridging fault
list.

0 stuck faults were changed to DI from the inactive bridging fault list.

0 stuck faults were changed to NP from the inactive bridging fault list.
```


203578 stuck faults were changed to DS from the inactive transition fault list.

0 stuck faults were changed to DI from the inactive transition fault list.

0 stuck faults were changed to NP from the inactive transition fault list.

The following table describes the direct fault crediting process.

Table 8 *Direct Fault Crediting Process*

Transition Fault Status	Existing Stuck-at Fault Status	Updated Stuck-at Fault Status
DS	Not DS	DS
DI	Not DS	DI
TP (small delay defect)	Not DS	DS
AP	Not DT or AP	NP
NP	Not DT or AP	NP

If the `-persistent_fault_models` option is not enabled, you can apply direct crediting to stuck-at faults by using `-external` option if you have transition fault list. The following is a script example that uses this method:

Script Example Using the `-external` Option

```
run_drc compression.spf

set_faults -model stuck

add_faults -all

update_faults -direct_credit -external transition.flt

run_atpg -auto
```

See Also

- [Fault Categories and Classes](#)

Example Commands Used in Persistent Fault Model Flow

The following example shows the commands used in a typical persistent fault model flow:

```
read_netlist des_unit
run_build_model des_unit
set_delay -launch system_clock
#
#Activate persistent fault model feature
#
set_faults -persistent_fault_models
#
# Model=Transition memory bypass=No OCC=Yes
#
add_pi constraints 0 memory_bypass
run_drc des_unit.spf -patternexec comp
set_faults -model stuck
add_faults -all
set_fault -model transition
add_faults -all
run_atpg -auto
write_patterns trans_bp0_occl.bin -format binary
set_faults -model stuck
#
# Credit transition detections to stuck-at faults.
#
update_faults -direct_credit
#
# Optional step to increase fault coverage
# run_fault_sim
#
drc -force
remove_pi_constraints -all
remove_clocks -all
#
# Model=Stuck-at memory bypass=Yes OCC=No
#
add_pi_constraints 1 memory_bypass
run_drc des_unit.spf -patternexec comp_occ_bypass
set_fault -model stuck
run_atpg -auto
write_patterns stuck_bp1_occ0.bin -format binary
drc -force
remove_pi_constraints -all
remove_clocks -all
#
# Model=Stuck-at memory bypass=No OCC=No
#
add_pi_constraints 0 memory_bypass_mode
run_drc des_unit.spf -patternexec comp_occ_bypass
set_faults -model stuck
```

Chapter 26: Persistent Fault Model Support

Example Commands Used in Persistent Fault Model Flow

```
run_atpg -auto  
write_patterns stuck_bp0_occ0.bin -format binary
```

27

Using TestMAX ATPG and DFTMAX Ultra Compression

DFTMAX Ultra compression is an advanced test compression technology that delivers the optimal quality of results as measured by test time, data volume, design area, congestion, and time to implementation.

TestMAX ATPG has built-in knowledge of DFTMAX Ultra compression and its pattern decompression and compression technology. Using a design netlist and a STIL procedure file, TestMAX ATPG generates a set of test patterns specifically intended for the DFTMAX Ultra test mode.

The following sections describe how to use TestMAX ATPG with DFTMAX Ultra compression:

- [Generating Patterns for DFTMAX Ultra Designs](#)
- [High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnosis](#)
- [Test Validation and VCS Simulation for DFTMAX Ultra Designs](#)
- [Limitations for Using DFTMAX Ultra](#)

Generating Patterns for DFTMAX Ultra Designs

To generate patterns for DFTMAX Ultra designs you must use either serial STIL or parallel STIL patterns generated by TestMAX ATPG. DFTMAX Ultra compression does not accept any other pattern format.

The following sections describe how to generate patterns specifically for a DFTMAX Ultra design:

- [Pattern Types Required by DFTMAX Ultra](#)
- [Script Example for Generating Patterns for DFTMAX Ultra](#)
- [Manipulating Patterns for DFTMAX Ultra](#)

Pattern Types Required by DFTMAX Ultra

TestMAX ATPG generates two types of STIL test patterns that can be used by DFTMAX Ultra compression:

- *Serial STIL*– These patterns are used for both scan testing and simulation of full scan testing. The test patterns are applied to the device for testing on the ATE and are used for simulating the entire test procedure, including serial scan-in data, decompression of the scan-in data, launch and capture, compression of the scan-out data, and serial scan-out data.
- *Parallel STIL*– These patterns are used for fast simulation of the launch and capture phases of scan testing. The decompressed test patterns are loaded directly into the scan chains in parallel and bypasses the serial scan-in and scan-out parts of the simulation.

You can write serial or parallel STIL patterns with or without the unified STIL flow. However, to use the unified STIL flow, you must explicitly specify the `-unified_stil_flow` option. The following example writes STIL patterns using the unified STIL flow:

```
TEST-T> write_patterns patterns.stil -format stil \  
-unified_stil_flow
```

For details on generating serial and parallel STIL patterns, see the [Writing ATPG Patterns](#) section.

After you simulate and validate the results of the test procedure using several test patterns, you can skip these patterns in future runs. You can then selectively simulate the launch and capture segments using additional test patterns loaded in parallel.

Script Example for Generating Patterns for DFTMAX Ultra

The following script is an example of a TestMAX ATPG pattern generation session for a chip that uses DFTMAX Ultra compression:

```
### USER INPUTS AND DFTMAX ULTRA OUTPUT FILES ###set  
TOP_MODULE_NAME top_module_nameset NETLIST_FILES1 netlist_files1set  
NETLIST_FILES2 netlist_files2set LIBRARY_FILES1 library_files1set  
LIBRARY_FILES2 library_files2set BUILD_CONSTRAINTS_FILE  
build_constraints_fileset DRC_CONSTRAINTS_FILE drc_constraints_fileset  
STL procedure file_FILE spf_fileset LOG log_filesetenv SYNOPSISYS  
path_to_tool_installation##### BUILD SETTINGS  
#####set_messages -level expert -log $LOG  
-replacereport_version -fullbuild -forceset_faults -pt_credit  
0set_faults -summary verboset_rules N2 warningset_rules B12  
warningset_rules B5 warningset_faults -atpg_effectivenessset_atpg
```

```
-verboset_netlist -redefined_module lastread_netlist
$NETLIST_FILES1read_netlist $NETLIST_FILES2read_netlist
$LIBRARY_FILES1 -libraryread_netlist $LIBRARY_FILES2
-librarysource -echo $BUILD_CONSTRAINTS_FILErun_build_model
$TOP_MODULE_NAME##### DRC SETTINGS #####source
-echo $DRC_CONSTRAINTS_FILEset_faults -model stuckrun_drc
$STL procedure file_FILE##### RUN ATPG
#####add_nofaults -module .*COMPRESSOR.*add_faults
-allrun_atpg -auto_compressionrun_simulation -
remove_padding_patternswrite_patterns ultra.stil format stil
```

Manipulating Patterns for DFTMAX Ultra

You can use the `update_streaming_patterns` command to modify or remove ATPG-generated patterns for use in DFTMAX Ultra compression. In some cases, the order in which you specify this command depends on whether you are using internal or external patterns.

The following topics describe how to use the `update_streaming_patterns` command to manipulate ATPG-generated patterns:

- [Controlling the Peak and Average Power During Shifting](#)
- [Increasing the Maximum Shift Length of Patterns](#)
- [Optimizing Padding Patterns](#)
- [Removing and Reordering Patterns](#)

Controlling the Peak and Average Power During Shifting

You can use the `-load_scan_in` option of the `update_streaming_patterns` command to control the peak and average power during shifting. This option enables you to specify certain scan-in pins to maintain a constant value during the shift operations. For example, if you specify a value of 0 for the test1 scan-in pin, the pattern is modified so that all test1 pins maintain a constant 0 value during load shifting.

You can specify values for as many scan-in pins as required using the Tcl list syntax. The

`-load_scan_in` option reduces overall power consumption during shifting, and it can be used for both internal and external patterns. However, this option also causes some coverage loss and simulation mismatches might occur if you specify scan-in pins connected to OCC chains.

The following example modifies internal patterns for the test_si1 and test_si3 pins after running ATPG:

```
TEST-T> run_atpg -auto
TEST-T> update_streaming_patterns -load_scan_in \ {test_si1 0
test_si3 1}
```

The next example updates external patterns created during ATPG with the specified values of the test_si4 and test_si7 scan-in pins:

```
TEST-T> set_patterns -external pat.stil
TEST-T> update_streaming_patterns -load_scan_in \ {test_si4 1
test_si7 1}
```

Increasing the Maximum Shift Length of Patterns

You can use the `-max_shifts` option of the `update_streaming_patterns` command to specify the maximum shift length, which enables you to increase the size of internal and external patterns from the optimal value set by TestMAX ATPG. You can use this option before an ATPG run so the generated patterns use the specified shift length or you can apply it to external patterns. The `-max_shifts` option prevents overshifting and makes the pattern shift lengths equal across different blocks, which assures correct pattern porting.

You can apply this option in an initial session to increase the shift length of the patterns, then use these same patterns in another session by writing and reading them back again. For subsequent sessions, make sure you set the pattern shift length to the same value set in the previous session. Otherwise, you will see errors and simulation mismatches.

The following example uses the `-max_shifts` option before an initial ATPG run:

```
read_image design.img
update_streaming_patterns -max_shifts 300
run_atpg -auto
write_patterns pat.stil -format stil -serial -replace
```

The following example applies the `-max_shifts` option in a second session using external patterns:

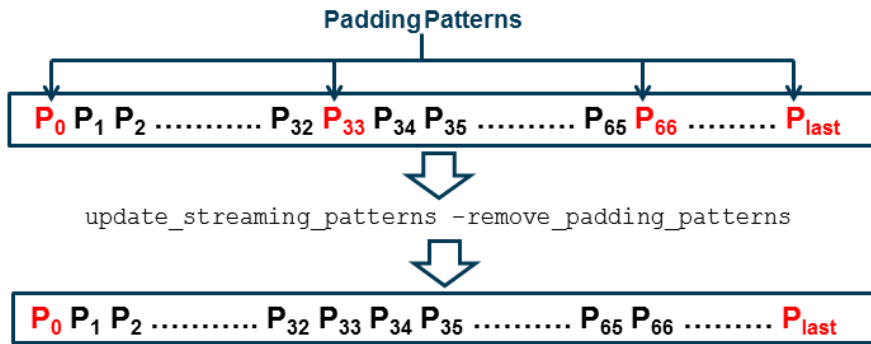
```
read_image design.img
update_streaming_patterns -max_shifts 320
set_patterns -external pat.stil
```

Optimizing Padding Patterns

When you use DFTMAX Ultra compression technology, the MUX control bits for each test pattern are loaded by the previous pattern. During ATPG, patterns are created in groups. For the first pattern in a group, TestMAX ATPG prepends a padding pattern that loads the required MUX control bits. During the later stages of pattern generation, TestMAX ATPG searches for patterns that do not incrementally detect new faults and removes those patterns from the pattern set. This process also introduces padding patterns.

You can use the `-remove_padding_patterns` option of the `update_streaming_patterns` command to optimize padding patterns by removing all padding patterns except for the first and last padding pattern.

Figure 145 Optimizing Padding Patterns



Performing Padding Pattern Optimization

The following commands optimize padding patterns for an internal pattern set generated by ATPG:

```
run_atpg ...
update_streaming_patterns -remove_padding_patterns
write_patterns ...
```

The following commands optimize padding patterns for an external pattern set:

```
set_patterns -external stil_file_name
update_streaming_patterns -remove_padding_patterns
write_patterns ...
```

Removing and Reordering Patterns

Use the `-remove` and `-insert` options of the `update_streaming_patterns` command to remove and reorder individual patterns and blocks of patterns.

To remove one or more patterns, specify a list using the `-remove` option of the `update_streaming_patterns` command.

The following command removes patterns 3 and 9:

```
update_streaming_patterns -remove {3 9}
```

You can specify blocks of patterns by providing the first and last pattern numbers of the block as a sublist inside the pattern removal list. You can mix individual pattern numbers and pattern blocks in the list.

The following command removes patterns 5 through 7:

```
update_streaming_patterns -remove {{5 7}}
```

The following command removes patterns 3, 5 through 7, and 9:

```
update_streaming_patterns -remove {3 {5 7} 9}
```

To remove patterns and reinsert them at a different location in the pattern set, use both the

`-insert` and `-remove` options of the `update_streaming_patterns` command. For each pattern number or block provided in the removal list, specify the pattern number at which the removed patterns should be reinserted in the insertion list. For removed patterns that you do not want reinserted, specify an insertion pattern value of X. Note that all removal pattern numbers are pre-manipulation values.

The following command removes pattern 3, and reinserts patterns 4 through 6 at pattern 0:

```
update_streaming_patterns -remove {3 {4 6}} -insert {X 0}
```

You can issue multiple the `update_streaming_patterns` command using the `-remove` and `-insert` options. Each command resequences the patterns and pattern numbers after performing the specified pattern manipulations. Subsequent `update_streaming_patterns` commands must refer to the resequenced pattern numbers.

You can reorder chain test patterns, stuck-at patterns, and transition fault patterns. However, you cannot perform reordering operations that mix these pattern types. If you mix pattern types, an error is reported, as shown in the following example:

```
Error: Pattern 2 and 13 are not of same type. Reordering is  
possible between patterns of same type only  
Transition patterns are 6 to 99  
Chain test patterns are 1 to 5
```

High Resolution Pattern Flow for DFTMAX Ultra Chain Diagnosis

If a failing part has multiple chain defects, you can create high resolution patterns that can more accurately identify failing scan cells when there are multiple failing scan chains.

The following sections describe the basic steps to this process:

- [Identifying Defective Chains](#)
- [Generating High Resolution Patterns](#)
- [Rerunning Diagnosis](#)
- [Flow Example](#)

Identifying Defective Chains

You need to perform an initial chain diagnostics run to identify a set of defective chains based on the chain test failures reported in the failure log file. To do this, specify the `-streaming_report_chains_only` option of the `run_diagnosis` command, as shown in the following example:

```
run_diagnosis failure_log_file.log -streaming_report_chains_only
chain_fail_report.txt
```

Generating High Resolution Patterns

To generate a set of high resolution patterns that identify the failing flip-flops in the defective chains, apply the `add_chains_masks` command to the entire production pattern set (including the chain test patterns and other logic patterns). You then use these patterns to generate a new set of failure log files that are used to identify the defective flip-flops. The following example shows this process:

Note:

If you mask the chain using the `add_chain_mask` command, the compares during output capture may fail. Use the `add_cell_constraint xx` command to avoid failing.

```
set_patterns -external full_pattern_set.stil
add_chain_masks -filename chain_fail_file.txt -diagnosis -external
write_patterns high_resolution_set.stil -format stil -external
```

Rerunning Diagnosis

Using the newly generated high resolution patterns, you need to retest the failing part and collect the new fail data. You can then rerun diagnostics using the failure log file generated from the high resolution patterns. You don't need to use any specific options in the `run_diagnosis` command for DFTMAX Ultra compression designs, as shown in the following example:

```
run_diagnosis high_res_pat_failure_log_file.log -verbose
```

Flow Example

```
read_image image_file.dat

# Step 1
## Run diagnostics to identify defective chains
run_diagnosis high_res_pat_failure_log_file.log
-streaming_report_chains_only chain_fail_list
set_patterns -delete
```

```
# Step 2
## Read full pattern test file
set_patterns -external full_pattern_set.stil

## Use add_chain_masks command to generate high resolution patterns
add_chain_masks -external -filename chain_fail_list -diagnosis

## Write the patterns
write_patterns high_resolution_set.stil -format stil -external

# Step 3
## Retest the failing part with the high resolution patterns and collect
the fail data

# Step 4
set_patterns -external high_resolution_set.stil

## Main diagnosis run using log file generated using high resolution
patterns
run_diagnosis high_res_pat_failure_log_file.log -verbose
```

Test Validation and VCS Simulation for DFTMAX Ultra Designs

You can perform test pattern validation for a DFTMAX Ultra design using MAX Testbench and then run a VCS simulation to validate the test protocol and test patterns.

For more information, see the [Using MAX Testbench](#) in the *Test Pattern Validation User Guide*.

The validation process for a DFTMAX Ultra design uses a serial STIL file or a parallel STIL file.

Limitations for Using DFTMAX Ultra

The following ATPG requirements and limitations apply to DFTMAX Ultra compression:

- Full-sequential ATPG is not supported
- Path delay fault testing is supported only for fast-sequential ATPG.
- The `write_patterns` command normally writes out a unified STIL file by default, which uses a single STIL file for both serial and parallel simulation. You can perform serial or parallel simulation using the unified STIL flow. However, to use this flow for DFTMAX Ultra designs, you must explicitly specify the `-unified_stil_flow` option to write out the STIL pattern files.

- The following options of the `set_drc` command are not supported:
 - `-lockup_after_compressor` | `-nolockup_after_compressor`
 - `-pipeline_in_compressor` | `-nopipeline_in_compressor`
- The `-per_pin_limit` option of the `set_diagnosis` command is not supported.
- You cannot mix the generation of launch-on-capture (LOC) and launch-on-shift (LOS) patterns in the same session.
- The `analyze_faults`, `analyze_compressors`, and `run_justification` commands are not supported.
- Retention tests are not supported.

28

Troubleshooting

The following sections describe troubleshooting tips and techniques:

- [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](#)
- [Using Pipeline Guidance](#)

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-T> report_primitives -pis
```

```
DRC-T> report_primitives -pos
```

```
DRC-T> report_primitives -pios
```

```
DRC-T> report_primitives -ports
```

To obtain the names of ports for any specific module, use the following command:

```
DRC-T> report_modules module_name -verbose
```

The following example shows a verbose report produced by the `report_modules` command. The names of the pins are listed in the Inputs and Outputs sections.

Verbose Module Report

```
TEST-T> 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:CO )
HADD2S : xor conn=( O:S2 I:A2 I:C1 )
HADD3S : xor conn=( O:S3 I:A3 I:C2 )
HADD0C : and conn=( O:CO I:A0 I:CI )
HADD1C : and conn=( O:C1 I:A1 I:CO )
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 will need to specify 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 TestMAX ATPG 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 [Truth Table Logic Model](#), and the command generates the warning messages shown in [Read Netlist Showing Warnings](#).

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
  
```

Read Netlist Showing Warnings

```
BUILD-T> 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:

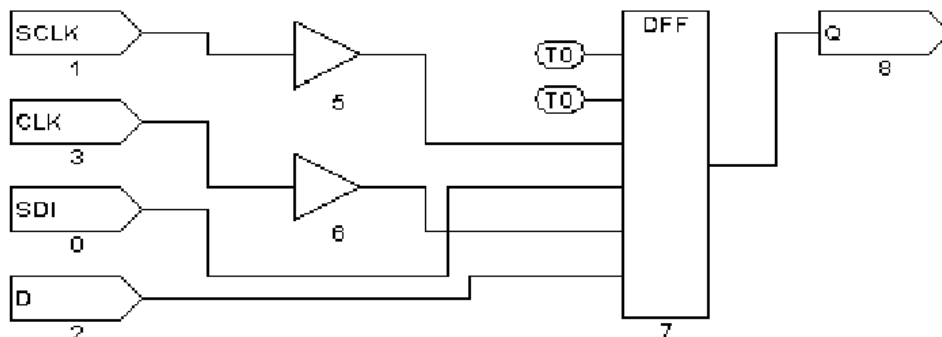
1. Execute the `run_build` command:

```
BUILD-T> 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 the following figure appears, allowing you to examine the ATPG model.

Figure 146 Module Showing Correct Interpretation



Do not be concerned if the schematic shows extra buffers. During the model building process, TestMAX ATPG 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-T> run_drc specfile.spf
```

```
# pause here for edits to DRC file  
TEST-T> drc -force  
DRC-T> run_drc
```

Troubleshooting Netlists

The following tips are for troubleshooting problems TestMAX ATPG might encounter while reading netlists:

- For severe syntax problems, start troubleshooting near the line number indicated by the TestMAX ATPG 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 TestMAX ATPG 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 can be either syntax errors or DRC violations. Syntax errors usually result in a category V (vector rule) violation message, and TestMAX ATPG reports the line number near the violation.

The following sections describe how to troubleshoot STIL procedures:

- [Opening the STL Procedure File](#)
- [STIL load_unload Procedure](#)
- [STIL Shift Procedure](#)
- [STIL test_setup Macro](#)
- [Correcting DRC Violations by Changing the Design](#)

Opening the STL Procedure File

To fix the problem, open the STL procedure file 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 [STIL Procedure Files](#).

A general tip for troubleshooting any of the STL procedure file procedures is to click the ANALYZE button in the GSV toolbar and select the applicable rule violation from the Analyze dialog box. TestMAX ATPG 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](#).

STIL load_unload Procedure

When you analyze DRC violations TestMAX ATPG 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.

The following guidelines are for 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 non-loading 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 STL procedure file.

The following guidelines are 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).

The following example 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.

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!
```

```
}  
}
```

The following example 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.

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;}  
}
```

The following example 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.

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 STL procedure file.

The following rules are 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 TestMAX DFT 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 STL procedure file 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 STL procedure file 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 the [Working With Design Netlists and Libraries](#) section 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, you should review the AN (ATPG untestable), ND (not detected), and PT (possibly detected) faults, and refer to the following sections:

- [Where Are the Faults Located?](#)
- [Why Are the Faults Untestable or Difficult to Test?](#)
- [Using Justification](#)

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-T> report_faults -class an -level 4 256
```

The following example 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.

Fault Report of AN Faults Using the Level Option

```
TEST-T> report_faults -class AN -level 4 256
#faults testcov instance name (type)
-----
22197 91.70% /spec_asic (top_module)
2630 83.00% /spec_asic/born (born)
2435 28.00% /spec_asic/born/fpga2 (fpga2)
788 5.35% /spec_asic/born/fpga2/avge1 (avge)
1647 3.28% /spec_asic/born/fpga2/avge2 (yavge)
5226 0.00% /spec_asic/dac (dac)
5214 0.00% /spec_asic/dac/dual_port (dual_port)
11098 66.46% /spec_asic/video (video)
11098 66.24% /spec_asic/video/decipher (vdp_cyphr)
11027 60.00% /spec_asic/video/decipher/dpreg (dpreg)
426 96.97% /spec_asic/gex (gex)
260 93.89% /spec_asic/gex/fifo (gex_fifo)
799 94.56% /spec_asic/vint (vint)
798 54.29% /spec_asic/vint/vclk_mux (vclk_mux)
1514 94.80% /spec_asic/crtc_1 (crtc)
476 96.79% /spec_asic/crtc/crtc_sub (crtc_sub)
465 94.20% /spec_asic/crtc/crtc_sub/attr (attr)
1004 77.68% /spec_asic/crtc/crap (crap)
```

The report shows that the two major contributors to the high number of AN faults are the following hierarchical blocks:

- `/spec_asic/dac/dual_port` (with 5,214 AN faults and 0.00 percent test coverage)
- `/spec_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-T> analyze_faults -class an
```

The following example 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.

Fault Analysis Summary of AN Faults

```
TEST-T> 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-T> analyze_faults -class an -verbose -max 3
```

You can redirect this report to a file by using the output redirection option:

```
TEST-T> 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-T> analyze_faults /gcc/hclk/U864/B -stuck 0
```

The following example 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.

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](#).

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 TestMAX ATPG is given more CPU time or a higher abort limit and allowed to continue.
- The fault location is uncontrollable.

In the following example, the `run_justification` command is used to confirm that an internal point can be set to both a high and low value.

Using `run_justification`

```
TEST-T> run_justification -set /spec_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-T> run_justification -set /spec_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](#).

Completing an Aborted Bus Analysis

During the DRC analysis, TestMAX ATPG 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.

TestMAX ATPG 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 TestMAX ATPG 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](#).

Using Pipeline Guidance

While debugging a specific violation on a input port of the load compressor you can bypass the automatic tracing by providing user input of your head pipeline structure. This should only be attempted if you suspect there was an error with how initial tracing was done by DRC.

Following are the steps that are followed:

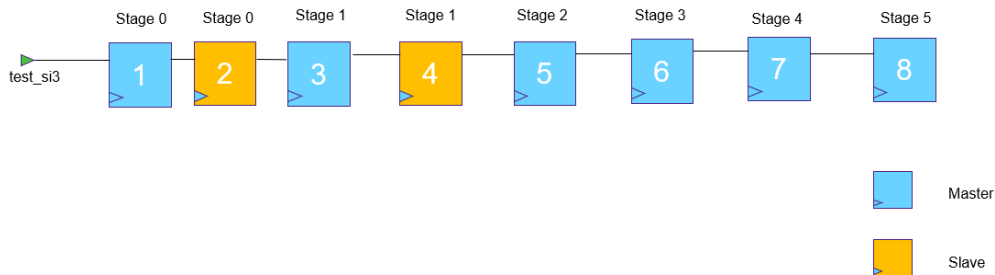
1. [Specifying the Head Pipeline Structures in the SPF](#)
2. [Using `set_drc -pipeline_structures`](#)

Specifying the Head Pipeline Structures in the SPF

No Fork

```
LoadPipelineStages 6
LoadPipelineElements {
Input "test_si1" !{
Stage 0 1 0 "SNPS_PipeHead_test_si6_1/Q" M ;
Stage 0 2 1 "SNPS_PipeHead_test_si6_1_slave/Q" S ;
Stage 1 3 2 "SNPS_PipeHead_test_si6_2/Q" M ;
Stage 1 4 3 "SNPS_PipeHead_test_si6_2_slave/Q" S ;
Stage 2 5 4 "SNPS_PipeHead_test_si6_3/Q" M ;
Stage 3 6 5 "SNPS_PipeHead_test_si6_4/Q" M ;
Stage 4 7 6 "SNPS_PipeHead_test_si6_5/Q" M ;
Stage 5 8 7 "SNPS_PipeHead_test_si6_6/Q" M ;
}
}
```

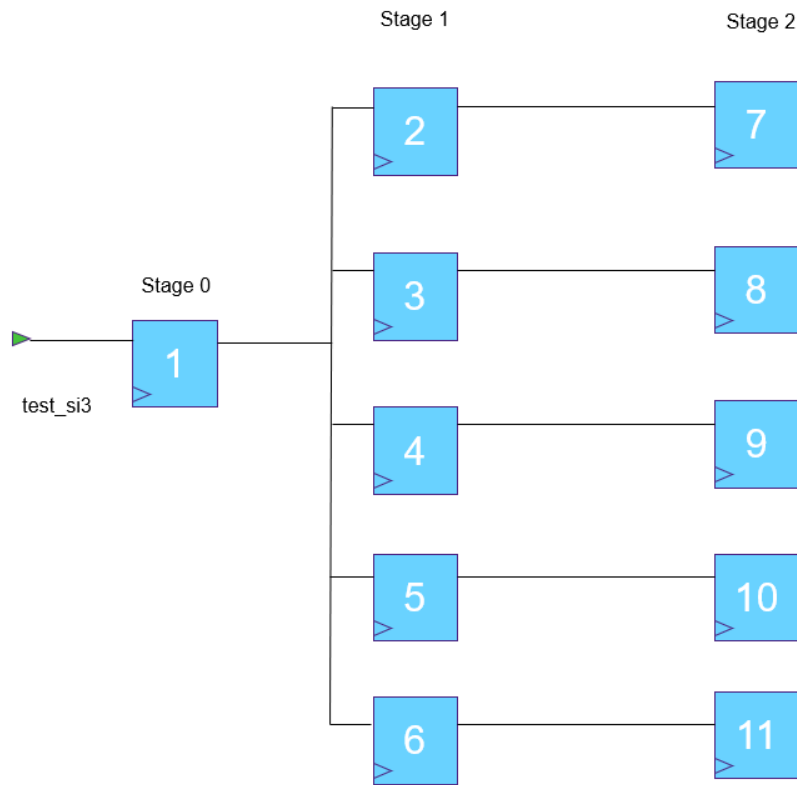
Figure 147 No Fork



Forked Pipelines

```
LoadPipelineStages 3;
LoadPipelineElements{
Input "test_si3"{
Stage 0 1 0 "pipein/reg_3/Q" M;
Stage 1 2 1 "pipelin/reg_3/Q" M;
Stage 1 3 1 "pipe2in/reg_3/Q" M;
Stage 1 4 1 "pipe3in/reg_3/Q" M;
Stage 1 5 1 "pipe4in/reg_3/Q" M;
Stage 1 6 1 "pipe5in/reg_3/Q" M;
Stage 2 7 2 "pipe12in/reg_3/Q" M;
Stage 2 8 3 "pipe22in/reg_3/Q" M;
Stage 2 9 4 "pipe32in/reg_3/Q" M;
Stage 2 10 5 "pipe42in/reg_3/Q" M;
Stage 2 11 6 "pipe52in/reg_3/Q" M;
}
}
```

Figure 148 Forked Pipelines



Using set_drc -pipeline_structures

In order to use head pipeline guidance feature; you must use `set_drc -pipeline_structures` before running `drc` with updated SPF. In order to see more information on pipeline tracing make sure to `set_messages -level expert`.

Pipeline default DRC tracing:

Load compressor pipeline input SI_3b successfully traced forward to gate 1066 (invert=0).

Load pipeline SI_3b (84) = stage 0:1066

Pipeline guidance:

Load pipeline structure for input SI_3b successfully traced forward to gate 1066 (invert=0).

Load pipeline SI_3b (84) = stage 0:1066

Please note that wrong specifications could lead to other R* violation failures or lead to suppressing real issues that would lead to failing ATPG patterns during simulations. This feature should be used for debug only.

29

ATPG FAQ

This section contains the following topics:

- [What is the Difference Between Multicore Processing and Multithreading?](#)
- [How Can I Avoid Generating Patterns With Floating BIDI Ports?](#)
- [How Do I Abbreviate Commands?](#)
- [What Special Characters Are Used in Tcl Mode?](#)
- [What Are Limited Regular Expressions?](#)
- [What are the Compressor Connections in report_scan_chains Output?](#)
- [What are Some Examples of Pin Data?](#)
- [How Do I Use the write_testbench Command to Customize MAX Testbench Output?](#)
- [Validating Simulation Libraries Used For ATPG](#)
- [How Do I Customize Ltran Output for FTDL, TSTL2, or TDL91?](#)
- [How TestMAX ATPG Processes Setup and Hold Violations](#)
- [Interpreting UDP Messages](#)
- [What is the Difference between the add_capture_masks vs add_cell_constraints Commands?](#)
- [JTAG Support](#)
- [Node File Format for Bridging Faults](#)
- [Optimizing Basic Scan Patterns](#)
- [Design and ATPG Usage Tips for Designs with Phase Lock Loops \(PLLs\)](#)
- [Shared Scan-In Designs](#)
- [Creating End-of-Cycle Measures in ATPG Patterns](#)
- [Troubleshooting Pattern Simulation Failures](#)
- [WGL Pattern Generation Options](#)

What is the Difference Between Multicore Processing and Multithreading?

- [Subnet Formats for Diagnosis](#)
- [Handling Escape Characters in Tcl Mode](#)
- [Passing Complex Options to LSF/GRID](#)

What is the Difference Between Multicore Processing and Multithreading?

Answer:

Multicore processing uses two or more cores to execute multiple concurrent processes. Multithreading concurrently executes small, multiple tasks or *threads* based on instructions from a single central controlling scheduler. The following table and figures summarize the differences between multicore processing and the multithreading technology used by TestMAX ATPG II:

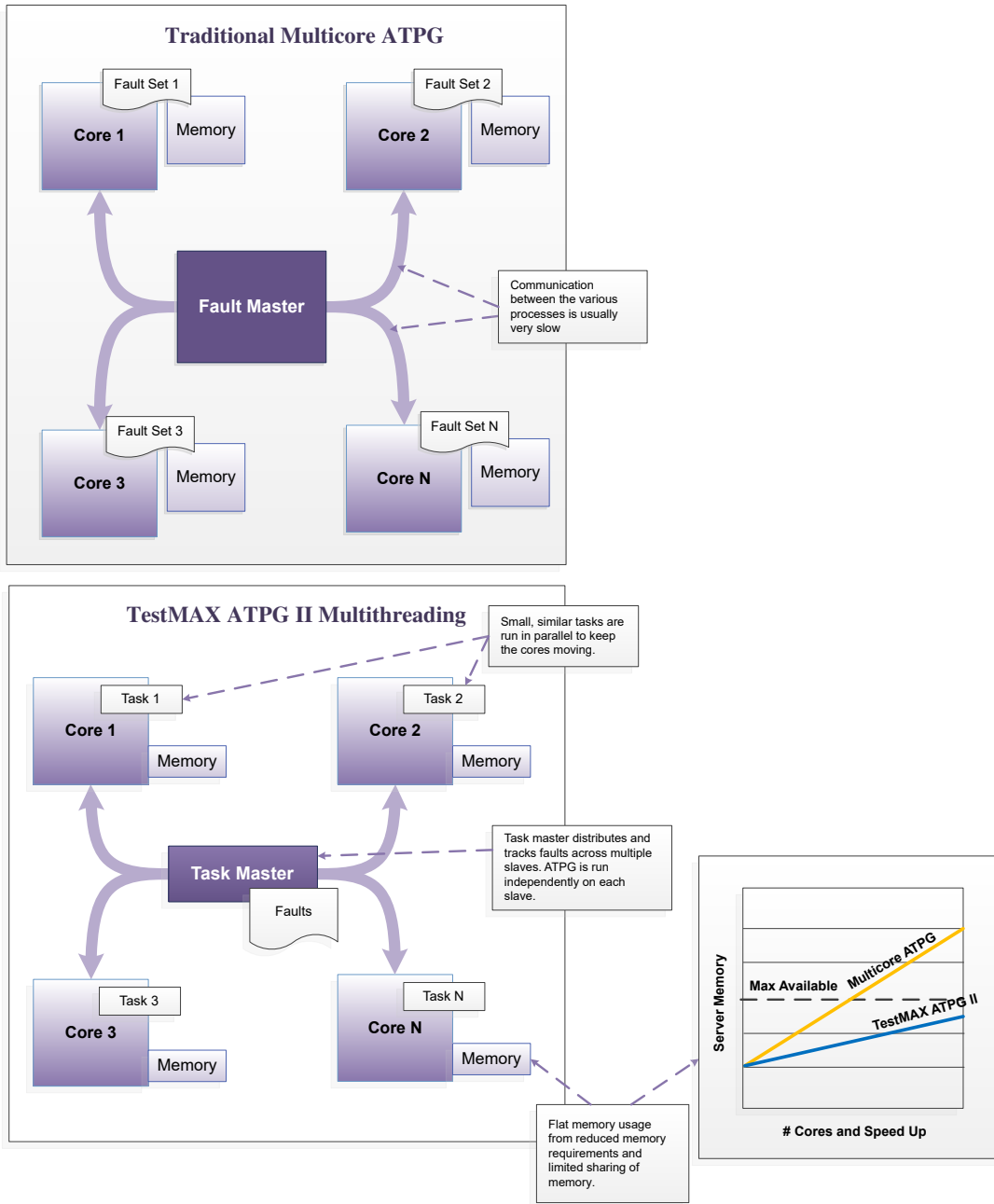
Table: Multicore Processing Versus Multithreading

<i>Multicore Processing</i>	<i>Multithreading</i>
"Heavyweight" – resource intensive	"Lightweight" – takes considerably fewer resources than multicore processing
Process switching requires constant interaction with operating system	Thread switching occurs independent of operating system
Each process executes the same code but uses its own memory and file resources	Threads share the same set of open files and child processes.

Figure: Comparing Multicore Processing to Multithreading

Chapter 29: ATPG FAQ

What is the Difference Between Multicore Processing and Multithreading?



How Can I Avoid Generating Patterns With Floating BIDI Ports?

Answer:

By default, TestMAX ATPG creates patterns that leave bidirectional ports floating -- unless it is necessary to drive the port to detect a target fault. This means the majority of generated patterns contain floating bidirectional ports.

To avoid floating inputs, you need to specify one of the following `set_simulation` commands before pattern generation:.

```
set_simulation -bidi_fill
set_simulation -strong_bidi_fill
set_simulation -weak_bidi_fill
```

How Do I Abbreviate Commands?

Answer:

Some commands and command keywords might be specified with just a few characters. Abbreviation parameters differ depending on whether you are running TestMAX ATPG in Tcl mode or Native mode.

Tcl Mode

In Tcl mode, application commands are specific to TestMAX ATPG. 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, TestMAX ATPG attempts to help you find the correct command.

Example

The command entered in the following example, `report_scan_c`, is ambiguous:

```
> report_scan_c
Error: ambiguous command `report_scan_c' matched 2 commands:
(report_scan_cells, report_scan_chains) (CMD-006).
```

TestMAX ATPG 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
```

Native Mode

In Native mode, the minimum number of characters needed for any specific command or keyword can be found by using the `help command_name` command. The letters shown in uppercase represent the minimum abbreviation allowed. Commands are not case-sensitive.

For example:

```
TEST> help analyze fault
ANalyze Faults < < pin_pathname -Stuck <0|1>
[-Observe gate_id] [-Display] > |
<-Class fault_class>... > >
[-Verbose]
```

In this example, the minimum abbreviation for the `analyze faults` command is `AN F` and the keywords `-observe`, `-display`, `-class`, and `-verbose` require only one character each. In addition, the underscore character can be used between command tokens in place of a space.

The following list shows equivalent forms of the same command: `remove atpg primitives`, `remove_atpg_primitives`, `rem atpg prim`, `rem_atpg_prim`, `rem_a_p`, and `REM_A_P`.

What Special Characters Are Used in Tcl Mode?

Answer:

The characters listed in the following table have special meaning for TestMAX ATPG in Tcl mode in certain contexts:

Character Description

\$	De-references 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.

What Are Limited Regular Expressions?

Answer:

TestMAX ATPG has several commands and options that enable you to specify a regular expression in place of a specific string. The following information shows how to construct a regular expression to achieve your desired string matching goals:

- [Regular Expression Meta-Characters](#)
- [Examples](#)
- [Using Escape Characters with Wildcards and Regular Expressions](#)

Regular Expression Meta-Characters

Regular Expression Description

.	Matches any single character
*	Matches preceding item 0 or more times
+	Matches preceding item 1 or more times
?	Matches preceding item 0 or 1 times
^	Matches start of string (optional)
[a-d]	Matches a single character within a range of characters: a,b,c,d

Regular Expression	Description
[~a-d]	Matches a single character not within a range of characters
\c	Escapes next character 'c', except if that character is within a set of []

Usage Notes:

- A '\' that appears as the first character of a string, escapes the entire string, and none of the remaining characters are considered as meta-characters.
- A '~' must appear as the first character of a [...] range to invert the range match, or it is taken literally.
- A '-' should appear as the first or last character of a [...] range, or it is taken as a range separator.
- A '\t' is not a tab, but the letter 't'.
- If a regular expression is enclosed in matched double quotation marks "..." the quotation marks are removed before being evaluated.
- A caret '^' that appears anywhere except at the start of a string is taken literally as that particular character.
- The `set_build -hierarchical_delimiter` command impacts regular expression matching. You should always know the status of your hierarchical separator character.

Examples

Example	Matches
.*_pad	ABC_pad go[123]_pad pad_pad_pad _pad
..._pad	ABC_pad zyx_pad
a.*z	az abc%xyz az%123%z
[a-z]+\[[0-9]+\]	abc[12] alpha[1] a[123456789]
[a-zA-Z_]+\[[0-9]+\]	A_B[12] Dog_Cat[9]
[a-z]+\[[0-9]+\].*	abc[2]xyz bat[121]bat[121] cat[9]dog
[A-C]+_[_pad]+	ABC_pad ABC_a ABC_p ABC_ppaadd BAA_dad

Example	Matches
[A-C]+_[pad]+	ABC_pad C_p CC_pp
[A-C]?_[pad]?	A_p B_a _p C_
[A-C]*_[pad]+	_pad AAAAA_p BB_aaaa CAB CAB_papa
[a-zA-Z0-9_]*z	abRAcadabra_9z_zzz_abc_y_z z Z9z
[a-z_]*[0-9_]*.+z	abracadabro_z_123_abc_y_z y_200z yz z
[aA].*[zZ]	Az aZ abc%xyz az[123]/z
data[12]	data1 data2
data\[12\]	data[12]

Using Escape Characters With Wildcards and Regular Expressions

A wildcard or regular expression specification used in an escaped identifier is not recognized as wildcard or regular expression. For example, in the following command, the regular expression is not correctly interpreted:

```
add_nofaults -instance \\u_padring_wrap/u_jtag/.*
```

If you want to use a wildcard with an escaped name, you must use the following:

```
add_nofaults -instance *u_padring_wrap/u_jtag/.*
```

What are the Compressor Connections in report_scan_chains Output?

Answer:

When you run the `report_scan_chains` command, there is a column titled "compressor connections," as shown in the following example:

```
TEST-T> report_scan_chains
chain          group          length  input_id  output_id  compressor
connections
-----
11             core_group          24      3816      38758
0/8=test_so7  1/8=test_so7  2/8=test_so7
```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

At first glance, the report appears to only identify the output connections of the chains in load_mode and unload_mode. However, the text identified in purple actually corresponds to output for the `report_compressors` command, as shown in the following example:

```
output port                                chain connection for load_mode=0
  unload_mode=8
-----
test_so7                                  11
-----
output port                                chain connection for load_mode=1
  unload_mode=8
-----
test_so7                                  11
-----
output port                                chain connection for load_mode=2
  unload_mode=8
-----
test_so7                                  11
-----
```

The output of the `report_compressors` command also identifies how each chain physically connections to the decompressor and compressor, as shown in the following example:

name	#outputs	#modes	mode controls	type	#inputs	
ORCA_TOP_U_decompressor_ScanCompression_mode	96	3	Mode 0: test_si7=0 test_si6=0 Mode 1: test_si7=1 test_si6=0 Mode 2: test_si7=0 test_si6=1	load	5	
external port connection	connection			input	output	external chain
test_si1	1				0	0
test_si2	2				1	1
test_si3	3				2	2
test_si4	4				3	3
test_si5	5				4	4
...			---	5	6	
			---	95	96	

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

In this case, there are 5 input data pins. They are numbered 0-4 on the input side of the decompressor. They flow out to ports 0-95 on that block, and hook to chains 1-96, respectively. There are also 2 mode pins and an 8th pin: XTOL_ENABLE.

For each input mode (0-2), the report shows the flow of data from the top-level pin to the internal chains (connection 10 is to chain 11), as shown in the following example:

```

output_id  inv  ports connected to output for mode 0
-----  ---
-----
---
10         no   test_si2

output_id  inv  ports connected to output for mode 1
-----  ---
-----
---
10         no   test_si1

output_id  inv  ports connected to output for mode 2
-----  ---
-----
---
10         no   test_si5
The report for the output compressor is as follows:
name                                     type          #inputs
#outputs  #modes  mode controls
-----  -----  -----
ORCA_TOP_U_compressor_ScanCompression_mode  unload        96
 8          3
Unload compression type = shared
-----
external chain connection      input  output  external port
connection
-----
-----
1
0          test_so1          0
2
1          test_so2          1
3
2          test_so3          2
4
3          test_so4          3
5
4          test_so5          4
6
5          test_so6          5
7
6          test_so7          6

```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```

8                                     7
  7      test_so8
9                                     8
  ---
...
96                                     95
  ---

```

The block in the example has 96 inputs (chains 1-96, numbered 0-95), and these inputs flow to 8 scan outputs. Note in the following example, that chain 11 appears in the input connection side:

```

output_id  inv  chains connected to output
-----
-----
6          no   4 11 14 18 22 27 33 34 38 40 43 46 49 51 54 57 60 62
64 67 71 73 76 79 80 84 87 90 93 94
7          no   5 11 15 19 25 28 32 35 39 42 44 47 50 53 56 58 61 64
67 69 71 75 77 80 83 85 87 90 92 95

```

Note that this data was also in another report for "single observe mode." But it appears again here in this report with all the connections of chain 1:

```

output port          chain connection for load_mode=0
unload_mode=3
-----
test_so7             4 11 18 22 34 43 49 57 67 76 84 90

output port          chain connection for load_mode=0
unload_mode=7
-----
test_so7             4 11 18 22 34 49

output port          chain connection for load_mode=0
unload_mode=8
-----
test_so7             11

output port          chain connection for load_mode=0
unload_mode=15
-----
test_so7             4 11 18 22 34 43 49

output port          chain connection for load_mode=1
unload_mode=3
-----
test_so7             4 11 18 22 34 43 49 57 67 76 84 90

output port          chain connection for load_mode=1
unload_mode=3
-----

```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```

test_so7                4 11 18 22 34 43 49 57 67 76 84 90
output port              chain connection for load_mode=1
  unload_mode=8
-----
test_so7                11
output port              chain connection for load_mode=1
  unload_mode=15
-----
test_so7                4 11 18 22 34 43 49
output port              chain connection for load_mode=2
  unload_mode=3
-----
test_so7                4 11 18 22 34 43 49 57 67 76 84 90
output port              chain connection for load_mode=2
  unload_mode=7
-----
test_so7                4 11 18 22 34 49
output port              chain connection for load_mode=2
  unload_mode=8
-----
test_so7                11
output port              chain connection for load_mode=2
  unload_mode=15
-----
test_so7

```

In the STIL procedure file, this refers to the following chain:

```

ScanChain "11" {
  ScanLength 24;
  ScanEnable "scan_en";
  ScanMasterClock "pclk";
}

```

The following relationships in the STL procedure file are also important:

```

ScanChain "sccompin0" {
  ScanIn "test_si1";
}
ScanChain "sccompin1" {
  ScanIn "test_si2";
}
ScanChain "sccompin2" {
  ScanIn "test_si3";
}
ScanChain "sccompin3" {
  ScanIn "test_si4";
}

```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```
}
ScanChain "sccompin4" {
  ScanIn "test_si5";
}
ScanChain "sccompin5" {
  ScanIn "test_si6";
}
ScanChain "sccompin6" {
  ScanIn "test_si7";
}
ScanChain "sccompin7" {
  ScanIn "test_si8";
}
ScanChain "sccompout0" {
  ScanOut "test_so1";
}
ScanChain "sccompout1" {
  ScanOut "test_so2";
}
ScanChain "sccompout2" {
  ScanOut "test_so3";
}
ScanChain "sccompout3" {
  ScanOut "test_so4";
}
ScanChain "sccompout4" {
  ScanOut "test_so5";
}
ScanChain "sccompout5" {
  ScanOut "test_so6";
}
ScanChain "sccompout6" {
  ScanOut "test_so7";
}
ScanChain "sccompout7" {
  ScanOut "test_so8";
}

load_group {
  "sccompin0";
  "sccompin1";
  "sccompin2";
  "sccompin3";
  "sccompin4";
}
unload_group {
  "sccompout0";
  "sccompout1";
  "sccompout2";
  "sccompout3";
  "sccompout4";
  "sccompout5";
}
```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```

    "sccompout6";
    "sccompout7";
}
mode_group {
    "sccompin5";
    "sccompin6";
}
unload_mode_group0 {
    "sccompin0";
    "sccompin1";
    "sccompin2";
    "sccompin3";
}
unload_mode_group1 {
    "sccompin4";
    "sccompin0";
    "sccompin1";
    "sccompin2";
}
unload_mode_group2 {
    "sccompin3";
    "sccompin4";
    "sccompin0";
    "sccompin1";
}
enable_group {
    "sccompin7";
}

```

The only new information is the enable pin for X-tolerance (masking) being “sccompin7” which is test_si8.

```

CompressorStructures {
  Compressor "ORCA_TOP_U_decompressor_ScanCompression_mode" {
    Mode 0 {
      UnloadModelGroup unload_mode_group0;
      ModeControls {
        "test_si6" = 0;
        "test_si7" = 0;
      }
      Connection 1 "2" "6" "11" "16" "22" "26" "31" "37" "43" "49" "54"
        "59" "64" "69" "76" "79" "85" "90" "95";
    }
  }
}

```

So, load mode 0 connects chain 11 to connection 1. Connection “1” is really test_si2 (“1” coming from sccompin1). This matches report_compressors.

Load mode 1 and 2 also match the following report:

```

Mode 1 {
  UnloadModelGroup unload_mode_group1;
  ModeControls {
    "test_si6" = 0;
    "test_si7" = 1;
  }
}

```


Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```

}
Connection 0 "1" "6" "11" "16" "21" "25" "32" "37" "42" "48" "52"
"56" "62" "66" "72" "78" "83" "89" "93";
}
Mode 2 {
  UnloadModelGroup unload_mode_group2;
  ModeControls {
    "test_si6" = 1;
    "test_si7" = 0;
  }
  Connection 4 "5" "11" "15" "21" "25" "30" "35" "39" "44" "49" "55"
"59" "65" "70" "73" "79" "83" "90" "93";
}

```

The output compressor is similarly described as follows:

```

Compressor "ORCA_TOP_U_compressor_ScanCompression_mode" {
  Mode 0 {
    ModeControls {
      "test_si8" = 0;
    }
    Connection "11" 6 7;
  }
}

```

So, when the xtol enable signal (test_si8) is 0, you can view scan chain 11 at ports 6 and 7, which are really test_so7 and test_so8.

The remaining 49 modes are also defined. Chain 11 in these definitions follows:

```

Mode 2 {
  ModeControls {
    "test_si8" = 1;
    "test_si6" = 0;
    "test_si7" = 0;
    "test_si1" = 0;
    "test_si2" = 0;
    "test_si3" = 0;
    "test_si4" = 1;
  }
  Connection "11" 6;
}

Mode 13 {
  ModeControls {
    "test_si8" = 1;
    "test_si6" = 0;
    "test_si7" = 0;
    "test_si1" = 1;
    "test_si2" = 1;
    "test_si3" = 0;
    "test_si4" = 0;
  }
  Connection "11" 6;
}

Mode 15 {
  ModeControls {

```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 0;
"test_si1" = 1;
"test_si2" = 1;
"test_si3" = 1;
"test_si4" = 0;
}
Connection "11" 6;

Mode 16 {
ModeControls {
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 0;
"test_si1" = 1;
"test_si2" = 1;
"test_si3" = 1;
"test_si4" = 1;
}
Connection "11" 6;

Mode 18 {
ModeControls {
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 1;
"test_si5" = 0;
"test_si1" = 0;
"test_si2" = 0;
"test_si3" = 1;
}
Connection "11" 6;

Mode 29 {
ModeControls {
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 1;
"test_si5" = 1;
"test_si1" = 1;
"test_si2" = 0;
"test_si3" = 0;
}
Connection "11" 6;

Mode 31 {
ModeControls {
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 1;
"test_si5" = 1;
```

Chapter 29: ATPG FAQ

What are the Compressor Connections in report_scan_chains Output?

```
"test_si1" = 1;
"test_si2" = 1;
"test_si3" = 0;
}
Connection "11" 6;

Mode 32 {
ModeControls {
"test_si8" = 1;
"test_si6" = 0;
"test_si7" = 1;
"test_si5" = 1;
"test_si1" = 1;
"test_si2" = 1;
"test_si3" = 1;
}
Connection "11" 6;

Mode 34 {
ModeControls {
"test_si8" = 1;
"test_si6" = 1;
"test_si7" = 0;
"test_si4" = 0;
"test_si5" = 0;
"test_si1" = 0;
"test_si2" = 1;
}
Connection "11" 6;

Mode 45 {
ModeControls {
"test_si8" = 1;
"test_si6" = 1;
"test_si7" = 0;
"test_si4" = 1;
"test_si5" = 1;
"test_si1" = 0;
"test_si2" = 0;
}
Connection "11" 6;

Mode 47 {
ModeControls {
"test_si8" = 1;
"test_si6" = 1;
"test_si7" = 0;
"test_si4" = 1;
"test_si5" = 1;
"test_si1" = 1;
"test_si2" = 0;
}
}
```

```
Connection "11" 6;  
  
Mode 48 {  
  ModeControls {  
    "test_si8" = 1;  
    "test_si6" = 1;  
    "test_si7" = 0;  
    "test_si4" = 1;  
    "test_si5" = 1;  
    "test_si1" = 1;  
    "test_si2" = 1;  
  }  
}  
Connection "11" 6;
```

What are Some Examples of Pin Data?

Answer:

The following examples show the various pin data types and helps you interpret the contents of the data.

You can use the `set_pindata` command or the SETUP button to adjust the pin data displayed in the graphical schematic viewer (GSV). Also, the ANALYZE button automatically selects an appropriate pin data choice when drawing a rule violation.

The simulation waveform viewer (SWV) supports only the following pin data types: Debug Sim Data, Sequential Sim Data, and Test Setup.

Bidi Control Value

You use the `-bidi_control_value` option of the `set_pindata` command to set the simulated values that occur when the `bidi_control` pin is set to its off state (as defined by the `set_drc -bidi_control_pin` command).

Examples:

x

1

Clock Cone

When you specify the `-clock_cone` argument of the `set_pindata` command, a clock port is also specified. The characters displayed near device pins indicate whether the attached net is in the clock or effect cone for that clock. A `C` designation indicates the net is in the clock cone for the selected clock. `E` indicates the net is in the effect cone, `CE` indicates the net is in both, and `N` indicates the net is in neither.

Examples:

C

E

CE

N

Clock On

Examples:

X-X

0-1

X-1

When you specify the `-clock_on` option, the GSV displays two bits separated by a dash (-) for each pin. The first bit is the simulated value that results when all clocks are set to off. All other inputs are set to X or to their constrained values. This is the same as the Clock Off data. It is displayed here to help you debug errors. The second bit is the simulated value that results when a specified clock is set to on. All other inputs are set to X or their constrained values.

Clock Off

Examples:

X

1

0

CX

c1

c0

When you select the `-clock_off` option, the GSV displays the simulated values which occur when all defined clocks are set to their off values and all other inputs are set to X. Any 0 or 1 values are indications of logic with an unblocked combinational path back to a clock port. xs are a sign of logic with no path to a clock port or the path is blocked by gates which have Xs for inputs.

Note that the data displays differently if the `run_drc` command successfully completes. In this case, the clock off data is displayed as `CX`, `C1`, and `C0`. The "C" prefix indicates that the gate is in the clock cone, and is independent from the 0, 1 or X simulated values.

Constraint Data

Examples:

1. `X/-,X/-,X/-`
2. `0/-,0/B,0/-`
3. `X/-,1/B,1/-`
4. `X/-,~01/B,~01/B`

When constraint data is chosen, the data consists of three pairs of characters "T/B,C/B,S/B" where:

T = simulated value due to tied gates

B = fault blockage due to tied gates indicated by T

C = simulated value due to constraints for combinational ATPG:

tied gates, constrained pins, constant value gates

B = fault blockage due to constraints indicated by C

S = simulated value due to constraints during sequential scan ATPG

B = fault blockage due to any constraint indicated by S

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

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

Example 1 `"X/-,X/-,X/-"` indicates no constraints and no blocks.

Example 2 `"0/-,0/B,0/-"` indicates the pin is constrained to 0 due to tied logic, there is also a constraint of 0 during combinational ATPG which contributes to a blockage. During sequential scan ATPG there is a constraint to 0 but no corresponding blockage.

Example 3 `"X/-,1/B,1/-"` indicates a constraint to 1 for both combinational and sequential scan ATPG, but a blockage due to that constraint only for combinational ATPG.

Example 4 `"X/-,~01/B,~01/B"` indicates there is no constraint or blockage due to tied gates but there is a constraint and a blockage for the combinational and sequential scan ATPG

algorithm of not 0 or 1 (~01), in other words X or Z, due to constrained pins and constant value gates.

Debug Sim Data

Examples:

#000-011

00011

#000-011 / 01

#010-#010-0001 / -

The debug sim data displays either a single selected simulation source, or a comparison of two simulation sources. When two sources are shown, the data is separated by a slash, as in "AAA / BBB ". For a Fast Sequential simulation source a scan load is shown by "#", and a dash "-" is used to separate different capture procedures. For full sequential or VCD simulation sources a single continuous stream of characters are shown. A VCD source cannot contain data for all circuit points and the absence of data is shown with a single dash as in "-" or " AAA / - ".

Delay Data

The following list shows all the possible delay data annotations which may show up in the GSV or primitive report:

- no path delay behavior 0->1 rising edge node in path 1->0 falling edge node in path LCC launch and capture clock gate LC launch clock gate LN launch node gate CC capture clock gate CN capture node gate

Error Data

This setting only has meaning if the DRC process stops due to an rules violation with a severity of error. As an alternative to the ANALYZE button you can also set the pin data type to error.

Fault Data

Examples:

1. NO/DS
2. AN/UT

3. --/--
4. ##/##
5. DI/DI

Choosing a pin data type of Fault Data displays the current fault codes for the pin in the order: stuck-at-0 / stuck-at-1. "##" is an indication of a location where a nofault exists. "--" indicates there are not any faults at this pin.

Example 1 indicates the stuck-at-0 has been classified as not-observed (NO) and the stuck-at-1 has been classified as detected-by-simulation (DS).

Example 2 indicates there are no faults associated with this pin.

Example 3 indicates a site where the nofault attribute has been placed to inhibit faults.

Fault Sim Result

Examples:

1. 0
2. 0/1
3. 1->1
4. 0->1/0

Choosing a pin data setting of Fault Sim Results displays the good machine vs. bad machine values for a specified faulty location and selected pattern. A single value indicates there is no difference between the good machine and faulty machine simulation. When there is a difference, the good machine is given first followed by a forward slash and the faulty machine value.

Example 1 indicates no difference between good and faulty machine.

Example 2 indicates a good machine response of 0 and faulty machine response of 1.

Example 3 is a clocked output with no difference between good and faulty machine.

Example 4 is a clocked output in which the good machine goes to 1 but the bad machine goes to 0.

FULL_SEQ_Scoap_data

Examples:

1. 1-1-4
2. *-0-2
3. 2-3-z2-0

If you set the pin data display mode to FULL_SEQ_Scoap_data, the pin data field shows the set of SCOAP controllability and observability numbers for the pin. For a three-state node, the data display format is:

c0.c1.cZ.obs

where c0 is the control-to-0 measure, c1 is the control-to-1 measure, cZ is the control-to-Z measure, and obs is the observability measure. For an ordinary (non three-state) node, the data display format is the same, except that there is no control-to-Z field so you should observe:

c0.c1.obs

Example 1 indicates a control-to-0 and control-to-1 measure depth of 1 and 1, respectively, and an observability measure depth of 4.

Example 2 shows a control-to-0 measure depth of asterisk "*". This is typically found on a net which cannot be controlled to 0 (tied high).

Example 3 shows a tristateable node.

Full Sequential TG Data

Examples:

1. ?xxxx
2. #0xx
3. #x00-0xx

When the pin data setting of full_seq_tg_data is selected, the displayed data is the simulated data from Full-Sequential Test Generator. This data takes the form of 3 events per cycles, unless a sequential capture procedure has been defined, in which case the number of characters can vary greatly. The leading character is a ? when it has not been decided if a load is required, a "#" if a load is required, and omitted if no load is required.

Example 1 indicates that the need for a load is unknown.

Example 2 indicates a single load, followed by a sequence of 0xx.

Example 3 indicates a single load, followed by a capture sequence of x00, followed by a second capture sequence of 0xx without another load.

Good Sim Results

Examples:

1. 010
2. 0
3. 0->X
4. X->X

When you specify the `-good_sim_results` option, the GSV displays the simulated data from the specified ATPG pattern. *Note:* This option is for ATPG patterns only. For functional patterns, use the `-seq_sim_data` option.

The event time of the displayed data can be selected from five settings using the `time` option of the `set_primitive_report` command. The choices are `clock`, `preclock`, `postclock`, `lete`, and `all` (the default). A setting of `all` shows a three-character value of `preclock`, `clock`, `postclock`. Any information that is not available is displayed as a question mark "?".

The data on the output of a scan cell might go to X in the last frame before the scan unload operation, even when the scan cell is not clocked. When this occurs, the X does not propagate further. There are two possible explanations for this situation:

1. In compressed patterns, the X indicates that the scan cell cannot be observed due to compressor effects.
2. If timing exceptions are enabled, the X indicates that the expected value captured by the scan cell was overridden by a timing exception.

Load

Examples:

1. X{}X
2. 1{}1
3. 001{}1
4. X{}0{}1

When the pin data setting of Load is selected, the simulation events from the `load_unload` test procedure are displayed. The curly braces "{}" represent a placeholder for the

shift procedure. The logic values in front of the "{}" are from test vectors defined in the load_unload procedure before the application of Shift. The first logic value after the "{}" is the final simulated value at the end of the shift procedure. Any additional values following this character are from test vectors defined within the load_unload procedure but occurring after the application of the Shift procedure.

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

Example 1 indicates the pin is at an X state during the load_unload procedure.

Example 3 indicates three time events before shift of "001" and after the shift the pin is still a 1.

Example 4 indicates there are two shift procedures and the pin begins at a value of X, is a 0 at the end of the first shift procedure and is a 1 at the end of the second procedure.

Master Observe

Examples:

010

111

When the pin data setting of Master Observe is selected, the simulation events due to the test cycles defined in the master_observe procedure are displayed.

None

No pin data is displayed when the pin data selection is none.

Pattern

Examples:

010

0

0->X

X->X

XXXX0100000011000010000010000000

When you select the `-pattern` option, the GSV displays the simulated data from one of the 32 most recent patterns (from the pattern simulation buffer). You can select all 32 patterns, if needed. *Note:* The contents of the pattern simulation buffer can vary since this buffer gets used during ATPG pattern generation when specifying the `analyze_buses`, `analyze_faults`, `run_justification`, and other commands. To see the values from a specific ATPG pattern use the `-good_sim_results` option.

You can display the event time from among five different settings using the `time` option of the `set_primitive_report` command. The choices are `clock`, `preclock`, `postclock`, `lete`, and `all` (the default). A setting of `all` shows a three-character value of `preclock`, `clock`, `postclock`. Any information that is not available is displayed as a question mark "?".

When fast-sequential patterns are displayed, they will always follow the three-character format for each clock cycle.

Examples:

```
#010
#011 100
#111 100 #100 011
```

The # character indicates a scan chain load. There might be only one scan chain load at the start of a fast-sequential pattern or there might be multiple loads within a single pattern.

SCOAP Data

Examples:

```
2-2-1 0-0-0-0
0-*-2 0-*-0-0
*-*-* 1-1-2-2
*-*-6 3-3-0-0 e) *-0-8 *-0-0-0
```

When the pin data setting of SCOAP is selected, the displayed data is the SCOAP rating value. There are two sets of numbers, the first set consists of three characters of the form "C0-C1-O" and is for combinational ATPG. The the second set consists of four characters of the form "C0-C1-O-D" and is for sequential ATPG.

For the "C0-C1-0" format, each field is the minimum number of scan cells or input ports needed to:

C0 = control the pin to a 0.

C1 = control the pin to a 1.

Chapter 29: ATPG FAQ

What are Some Examples of Pin Data?

O = observe the value at the pin.

For the "C0-C1-0-D" format, each field is the minimum sequential depth necessary to:

C0 = control the pin to a 0.

C1 = control the pin to a 1.

O = observe the value at the pin.

D = sensitize the gate to detect the fault at an observe point.

An asterisk "*" indicates the value exceeds the 254 number program limit for tracking this information.

SDC Case Analysis

Examples:

a) 1/0 b) 0/X c) X/1

When *sdc_case_analysis* is chosen, two values separated by a slash (/) are displayed for each pin. The first value is the value set by the *set_case_analysis* commands in the SDC file. The second value is the value set by the constraints used by TestMAX ATPG DRC analysis.

Example a, "1/0," indicates that the SDC *set_case_analysis* commands set the pin to a logic 1, and the constraints used by TestMAX ATPG DRC set the pin to a logic 0.

Example b, "0/X," indicates that the SDC *set_case_analysis* commands set the pin to a logic 0, and the pin is unconstrained for TestMAX ATPG DRC.

Example c, "X/1," indicates that the pin is unconstrained by SDC *set_case_analysis* commands, and the constraints used by TestMAX ATPG DRC set the pin to a logic 1.

Seq Sim Data

Examples from `run_simulation -sequential`:

1. 0-0-1
2. 010-010-0-0
3. 0-1-00111-0
4. X/0X/0-Z/1Z/1

The `set_simulation -data` command can be used before a `run_simulation -sequential` command to indicate that a range of patterns should be stored during simulation. Then, by setting the pin data selection to Seq Sim Data, the saved patterns

are selected for display. Each pattern translates into one or more simulation events, which are displayed with a dash, "-" as a separator between patterns. When the pattern source is from external functional patterns, the dashes cannot be accurate as event collapsing occurs during pattern reading. If a series of two pattern events do not change inputs, then the redundant one is discarded.

Example 1 indicates 3 tester cycles and 3 events.

Example 2 indicates 4 tester cycles, and 8 events.

Example 4 indicates 2 tester cycles and 4 events. This type of data shows up when you run simulation with faults inserted (for example, `run_simulation -seq pin_pathname 0 | 1`). The value standing before "/" indicates good machine values. In the example shown above, the good values are XX-ZZ. The value standing after "/" indicates faulty machine values. In this case, the faulty values are 00-11.

Examples from `run_simulation` with full-sequential ATPG patterns:

1. #100011-#111100
2. #000#110#110001
3. #01100

The simulator that works with full-sequential ATPG patterns separates the patterns with a dash but does not separate the clock cycles as done by the fast-sequential simulator. For this reason, it can sometimes be difficult to separate the continuous string of characters into clock cycles. The easiest way to do this is to display a clock PI or a clock input to a sequential device so that it is obvious where the clock pulses occur.

Example 1 above represents two patterns, each with a single scan chain load, indicated by the "#" character, and two clock cycles. The signal in question is most likely changing in response to the leading edge of a clock because the changes occur in the second of the three simulation frames that normally make up a single clock cycle. The initial set of three values (100) in the first pattern represents the first clock cycle and the next set of three values (011) represents the second clock cycle. In the second pattern, the first clock cycle is represented by the 111 and the second clock cycle by the 100.

Example 2 represents a single pattern with three separate scan chain loads. A single clock cycle follows each of the first two loads, and two clock cycles follow the third scan chain load. In this example, the signal displayed is likely changing in response to the trailing edge of a clock because the changes occur in the third of the three simulation frames for a clock cycle.

Example 3 represents a single pattern with two clock cycles. However the last simulation frame of the first clock cycle has been merged with the first simulation frame of the last clock cycle; as a result, there are only five simulation values listed. If you examined the clock pulses for this pattern, they would appear as 01010.

Shadow Observe

Examples:

010

111

When the pin data setting of Shadow Observe is selected, the simulation events due to the test cycles defined in the shadow_observe procedure are displayed.

Shift

Examples:

1. xxx

2. 010

3. 000

4. 00010

5. 010/00110

When the pin data setting of Shift is selected, the simulation events due to test cycles defined in the shift procedure are displayed.

Simulation is performed by setting all constrained pins and constant value gates to their appropriate values and then simulating the test cycles in the load_unload before the shift procedure to determine the initial values at the start of the shift procedure. Then the test cycles in the shift procedure are simulated to provide the values which are displayed.

Example 1 indicates the shift procedure contains 3 simulation events and the pin of interest is an X value for all three events.

Example 2 indicates a pin which is clocked during the shift procedure.

Example 5 is a special case which is often associated with JTAG related designs. The first three values before the slash "/" indicate the general case of the shift procedure. The values after the slash indicate an additional (non-general) application of a shift. to obtain this type of shift pattern the load_unload procedure must contain test cycles after the Shift which pulse the shift clock. One or more additional shift operations is possible in which case each would be separated by a forward slash "/".

Stability Patterns

Examples:

1. xxxx/xxx
2. 1111/111
3. 000011/111

By default, the stability patterns are not available unless the `-store_stability` option of the `set_drc` command has been selected before performing a `run_drc` command. If the stability patterns are available, then selecting the Stability Patterns setting of pin data will cause the simulation events due to the test cycles in the `load_unload`, `shift`, and `capture` procedures to be displayed. However, this data is only available for sequential devices that have been classified as stable with constant 1 or constant 0 values.

Example 1 indicates a pin which is at an X value through the `load_unload` (values before "/") and through the `capture` procedure (values after "/").

Example 3 indicates a pin which is a 1 by the end of `load_unload` and remains so through a `capture` clock procedure.

Test Setup

Examples:

1. x
2. 0
3. 1
4. x0111

When the pin data setting of Test Setup is selected, the simulation events due to test cycles defined in the `test_setup` macro are displayed. By default, only the final simulated value is available unless the `set_drc -store_setup` command has been performed before running `drc` checks.

If there is no `test_setup` macro defined but PI constraints exist, then there is an implied `test_setup` macro consisting of one test cycle in which the constrained ports are assigned their constraint values.

Example 2 indicates a logic value of 0 due to `test_setup` macro.

Example 4 indicates 5 simulation events with a final value of 1.

Tie Data

Examples:

X

0

1

Z

When the pin data setting of Tie Data is selected, the logic values resulting from tied logic are displayed. An X indicates that there is no value due to tied logic while a 0, 1, or Z indicates the affect of tied logic at that pin.

How Do I Use the `write_testbench` Command to Customize MAX Testbench Output?

Answer?

MAX Testbench can be configured at several levels. At the top of the MAX Testbench configuration file, you can edit the `set cfg_*` variables to define the various testbench default values, such as the progress message interval time and the simulation time unit. The second half of the configuration file contains a set of editable setup parameters for the VCS/MIT/Cadence simulation script file. A default version of this configuration file is shown in the [following example](#).

You can use the `write_testbench` command or the Write Testbench dialog box in the GUI to specify a customized configuration file. TestMAX ATPG, in this case, invokes a separate translation process to customize the testbench output for Verilog.

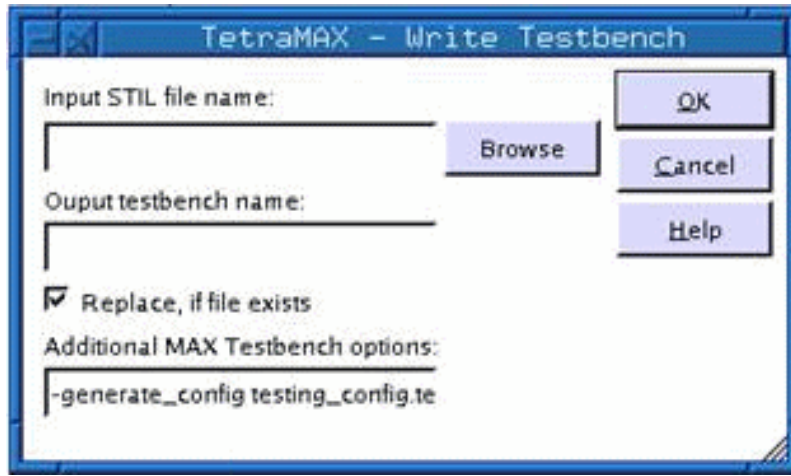
The following steps show you how to use `write_testbench` to customize testbench output for Verilog:

1. Create a local copy of the default Verilog configuration file.

Using the `write_testbench` command:

```
write_testbench -parameters {-generate_config stil2verilog.config}
```

Or, using the Write Testbench dialog box:



There is also an alias, `write_testbench_config`, that creates a configuration file called `stil2verilog.config`. This alias is created from the following command:

```
write_testbench -parameters {-generate_config stil2verilog.config} -
replace
```

2. Edit the default configuration file, `stil2verilog.config`, so that it contains your preferred parameters. (You can also rename this file, for example, "my_custom_config_file".)
3. Create a reference to this custom configuration file when you create patterns using the `write_patterns` command. For example:

```
TEST> write_testbench -input <stil_filename> -output <testbench_name>
-config_file my_custom_config_file -replace
```

Example

The following example shows the default configuration file:

```
##### STIL2VERILOG CONFIGURATION FILE TEMPLATE (go-nogo default values)
#####

# uncomment out the setting statement to use predefined variables

##### variables to define test bench default values

# cfg_patterns_read_interval: specifies the maximum number of patterns
loaded simultaneously in the simulation process
#set cfg_patterns_read_interval 1000
```

```

# cfg_patterns_report_interval: Specifies the interval of the progress
message
#set cfg_patterns_report_interval 5

# cfg_message_verbosity_level: control for a pre-specified set of trace
options
#set cfg_message_verbosity_level 0

# cfg_evcd_file evcd_file: generates an extended-VCD of the simulation
run
#set cfg_evcd_file "evcd_file"

# cfg_diag_file: generates a failures log file compliant with TestMAX
ATPG diagnostics
#set cfg_diag_file "diag_file"

# cfg_serial_timing: generates a delay for parallel scan access to align
parallel load timing with serial load timing
#set cfg_serial_timing 0

# cfg_time_unit: specifies the simulation time unit
#set cfg_time_unit "1ns"

# cfg_time_precision: specifies the simulation time precision
#set cfg_time_precision "1ns"

# cfg_dut_module_name: specifies the DUT module name to be tested
(variable to be used only when the tool asks for it)
#set cfg_dut_module_name "dut_module_name"

# cfg_tb_module_name: specifies the test bench top level module name
#set cfg_tb_module_name "tbench_module_name"

##### variables to generate simulator script

# define_<preprocessor_define>: specifies the preprocessor definitions
for the simulator
#set define_<user_def1> 0
#set define_<user_def2> "TRUE"

# design_files: specifies all source files required to run the
simulation
# Multiple files are specified in double quotation marks separated by
space. E.g.: "file1.v file2.v ..."
# Wild-card character (*) is supported. E.g., "dir1/*.v dir2/*.v"
#set design_files "netlist1.v designs/*.v"

# lib_files: specifies all library source files required to run the
simulation
# Multiple files are specified in double quotation marks separated by
space. E.g.: "file1.v file2.v ..."
# Wild-card character (*) is supported. E.g., "dir1/*.v dir2/*.v"

```

```
#set lib_files "lib1.v libs/*.v"

# vcs_options: specifies the user VCS command line options
#set vcs_options "VCSoption1 VCSoption2"

# nc_options: specifies the user NCSim command line options
#set nc_options "NCOption1 NCOption2"

# mti_options: specifies the user ModelSim command line options
#set mti_options "MTIOption1 MTIOption2"

# xl_options: specifies the user Verilog XL command line options
#set xl_options "XLOption1 XLOption2"

#### TB file formatting section
# cfg_tb_format_extended: specifies whether an extended TB file is need
#set cfg_tb_format_extended 0

#### TB file IDDQ Testing parameters
# cfg_iddq_seed_file: set this parameter when faults are seeded from
an external fault list file and you want to override the default
<tb_module_name>.faults
#set cfg_iddq_seed_file "your_fault_list_file"

# cfg_iddq_verbose: 1 (default) to enable PowerFault verbose report, 0
otherwise
#set cfg_iddq_verbose 1

# cfg_iddq_leaky_status: 1 (default), enable generaton of PowerFault
leaky nodes report in file <tb_name>.leaky, 0 otherwise
#set cfg_iddq_leaky_status 1

# cfg_iddq_seed_faul_model: set the PowerFault fault model, 0 (default)
SA faults, 1 for Bridging faults, for automatic seeding
#set cfg_iddq_seed_faul_model 0

# cfg_iddq_cycle: set the initial counter value for IDDQ strobes
(default to 0)
#set cfg_iddq_cycle 0
```

Note: For a complete list of all customized parameters and their descriptions, see "Customized MAX Testbench Parameters Used in a Configuration File with the write_testbench Command."

Validating Simulation Libraries Used For ATPG

TestMAX ATPG reads simulation libraries in Verilog structural form and dynamically creates an internally derived ATPG model for use during ATPG. Not all models come through cleanly, and it is common to experience many rules violations messages. The following template can assist you in identifying problems with libraries. This template

reads in the library cells, and then creates a test jig design with each of the library cells instantiated once. Then ATPG patterns are generated which should be simulated against the original Verilog library cells.

This validation technique is not foolproof or exhaustive. Because ATPG is used to generate patterns, there are no Xs or Zs applied to inputs. This means any differences in how the derived ATPG model and the original Verilog model handle Xs are not be uncovered.

```
# --- Boilerplate for Library Validation ---
# -----
#
set_messages log tmax.log -replace -double
report_version -full
build -force

set_netlist -nocheck_only_used_udps

#
# --- step #1: Read in all of the library cells.
# Be sure to use "-library" in case no `celldefines
# are used or the 'write_netlist' in step #4 is
# unable to filter out all the library cells.
#
read_netlist lib1/*.v -library -delete -noabort

#
# --- step #2: Define a black box list to identify modules
# not desired in the output. TestMAX ATPG automatically
# drops 1-pin devices from the testbench, but you
# can wish to keep OTHER library modules out of the
# testbench.
#
set_build -reset_box # clear the list
set_build -black_box ad01d0 # this removes 'ad01d0'

#
# --- step #3: Create the in-memory testbench, module named
# AAA using an undocumented switch to set netlist.
#
set_netlist -testbench AAA

# NOTE: Do not use the -testbench option when reading design
# netlist and libraries. The option -testbench is designed to
# instantiate all unused modules into one top-level module
# to create a design that can be used to generate
# test vectors for all the unused cells for library verification
# purposes only. Your netlist usually contains unused library cells
# when the -testbench option is used.
#
# NOTE: Any modules defined with set build -black_box is
# excluded in the resulting testbench for library validation.
```

```
#
# --- step #4: Write the testbench out, excluding the library cells
# by use of the '-stop design_level' option. If all
# goes well, there should be a single module in the
# output and it is the test_jig.
#
write_netlist test_jig.v -top AAA -stop design_level -replace

#
# --- step #5: Build in-memory image of testjig needed for ATPG
# and review any N messages before proceeding to step #6
run_build_model AAA

#
# --- step #6: Define any PI constraints or PI Equivs or Clocks
# you know the testing requires, and which TestMAX ATPG
# can have trouble figuring out. The testing port
# names is <libcell>_<pin>, where the port name
# is constructed of the name of the library cell, an
# underscore, and the pin name of the cell. So pin
# "D" on cell "SDFF", becomes "SDFF_D".
#
remove_pi_equivalences -all # clear the list
#add_pi_equivalences ABC_CP -inv ABC_CN # example differential in

remove_pi_constraints -all # clear the list
#add_pi_constraints 1 XYZ_DS # example constant pin

remove_clocks -all # clear the list
#add_clocks 0 LSSD_A # example clock, offstate=0

#
# --- step #7: First pass at Clock Definitions
# This undocumented command uses the existing list of
# clocks, PI constraints, and PI equivalents, and tries to
# determine additional clocks not defined. The
# -add_clocks option causes the clock list to be
# updated as the command runs. Otherwise it is just
# an informative report.

analyze_clocks -all -add_clocks -command_report -verbose -max 8
# --- Record clocks at this point into a command file.
#
report_clocks -command > define_clocks.cmd

#
# --- step #8: change to test mode
#
run_drc

#
# --- At this point, manual edits to the clock list can be needed.
```

Chapter 29: ATPG FAQ

How Do I Customize Ltran Output for FTDL, TSTL2, or TDL91?

```

# The file 'define_clocks.cmd' usually
# requires edits to change clocks, add clocks, add PI
# constraints, etc., until DRC violations are reduced as much
# as possible. It might not be possible to satisfy the
# C3 rule check on a standalone latch.
#
interrupt #command file paused: F12 or RESUME to continue

#
# --- step #9: generate ATPG patterns
#
set_atpg -capture 4 -abort 100
add_faults -all
run_atpg

#
# --- step #10: supplement patterns with Full-Sequential patterns
#
set_atpg -full_seq_atpg -full_seq_time 0 0 -full_seq_abort 100
run_atpg
report_summaries

#
# --- step #11: save patterns
#
write_patterns testbench.stil -format stil -serial -replace

# All done. Verilog simulation should now be performed using
# 'test_jig.v', 'testbench.v', and the original verilog library
# cells.

```

How Do I Customize Ltran Output for FTDL, TSTL2, or TDL91?

Answer:

The default Ltran configuration files are adequate for most translations. However, there are a number of fields in the configuration files that you can modify to customize the output file. The configurable fields are part of the simulator command. The comments in the Ltran configuration files identify these fields. All of these fields are optional. You can change them into comments using curly braces {}. The fields enable you to specify header information in the output file.

Customizing Ltran Configuration Files

The default Ltran configuration files are in the directory `$(SYNOPSYS)/auxx/syn/ltran`. Before you edit them, ensure that you make a local copy (unless you need to permanently change them).

This section shows you how to customize Ltran configuration files.

Customizing Simulator Format-Specific Controls

The following links describe the simulator format-specific controls:

- [Customizing Ltran Output for FTDL](#)
- [Customizing Ltran Output for TDL91](#)
- [Customizing Ltran Output for TSTL2](#)

Common Ltran Controls

This section describes the Ltran configuration commands generic to all formats.

Most of the Ltran configuration files contain two Ltran commands, `rename_bus_pins` and `header`, which you can use to customize the format of the pattern output files. These commands are embedded in comments by default. You can enable them by deleting the curly braces `{}` surrounding them that make them comments.

The `rename_bus_pins` command has the syntax:

```
rename_bus_pins $bus$vec;
```

The `rename_bus_pins` command flattens bussed signal names. It changes a bus signal name such as `bus[5]` into `bus5`. To control the format of the mapped name, change the `busvec` string. For example:

```
rename_bus_pins $bus_$vec_;
```

maps the signal name `bus[5]` to the signal name `bus_5_`.

To make Ltran place the signal names in a vertical list as comments above their column position in the vectors, use the `header` command. The syntax is:

```
header nn;
```

where `nn` is an integer that specifies how often (in number of lines) the pin header listing should be repeated.

Character Padding

To pad the FTDL, TSTL2 and TDL_91 formats, use the `SCANIN_DEFAULT` parameter in the `SIMULATOR` statement of the Ltran command file. The following example shows how to set the pad state using the `SCANIN_DEFAULT` parameter:

```
SIMULATOR WGL
  SCANIN_DEFAULT = "c",
  ...
;
```


Where "c" is the pad character for short chains. It is either a 1, 0 or caret (^). The caret tells Ltran to use the first non-X state in a short chain as the pad character; it handles the differential input cases.

How TestMAX ATPG Processes Setup and Hold Violations

Setup violations and hold violations are processed in different ways by the TestMAX ATPG simulators. Both types of violations use a worst-case analysis of possible glitches produced by simultaneous changes at the inputs of gates. Because of gate timing, the zero-delay simulation cannot resolve which glitches are impossible. For the purposes of timing exceptions, glitches never come from the outputs of flip-flops.

Setup violations use the following assumption:

Transitions are launched and propagated as usual, but they become X at their destination. After a violation has been turned into an X, the X persists until the next scan load. Most patterns have only a single load, and in this case the X persists until the end of the pattern. Multi load patterns might have the X removed before the end of the pattern. For delay tests using system clock launch, the state of the circuit is assumed to have stabilized after the scan load. For last-shift launch or stuck-at fault tests, transitions created by the last shift of the scan load are considered.

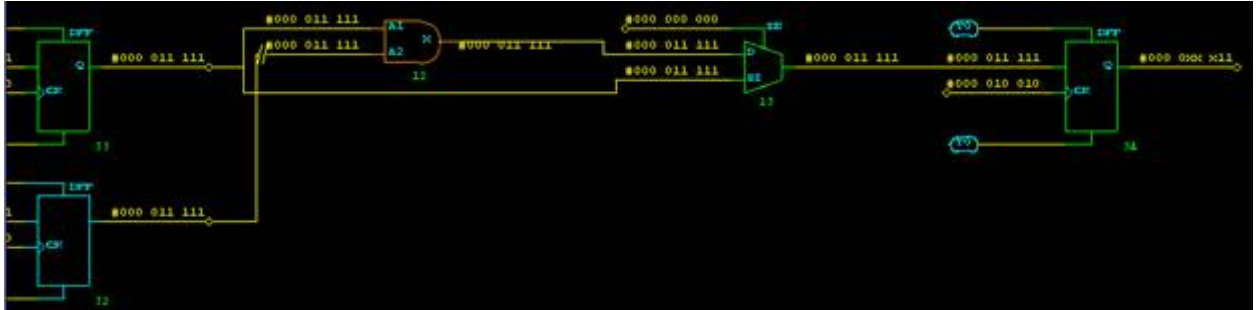
Hold violations use the following assumption: Transitions are launched and propagated as usual, but the capturing register is clocked too late to capture the old value and becomes X. If the capturing register is clocked again, in a later capture cycle of the same pattern, while its data inputs are stable, it clocks in the stable value successfully (unless there is also a setup violation on it).

Violations created by the `set_false_path` and `set_multicycle_path` SDC commands are setup or hold violations or both, as specified by the command and the `set_sdc` setting. Violations created by `set_disable_timing` and `set_case_analysis` (with `set_sdc -case_paths`) are treated as setup violations only.

Example of How TestMAX ATPG Handles an Ambiguous Case

Note in Figure 1, there is a `set_false_path -hold` exception from flip-flop 33 to flip-flop 34, and the path from flip-flop 32 has no exception.

Figure 1: Example With a `set_false_path -hold` Exception



TestMAX ATPG interprets hold exceptions as late clocking of the capturing register, which, in this case, occurs in flip-flop 34. However, there is no exception from flip-flop 32, which applies a controlling value to gate 12. But the controlling value is not removed until the same simulation frame in which the transition from flip-flop 33 occurs. By a strict interpretation, the timing violation should be masked. However, this implies that flip-flop 32 is also clocked late even though it has no exception. As a result, this example is an ambiguous case: Flip-flop 32 would need to be considered as clocked normally for the other paths to which it fans out.

In this case, the transition along the exception path causes the capturing register, flip-flop 34, to be assigned to X for the violating clock cycle. This particular non-violation is difficult to reliably distinguish from actual violations. Any misinterpretation that caused over-optimistic behavior would result in the generation of bad patterns. Therefore, TestMAX ATPG is slightly pessimistic in this case.

Interpreting UDP Messages

Many of the detailed DRC violation messages associated with creating derived ATPG models for UDPs from vendor libraries need additional explanation on how the messages should be interpreted and whether further action is needed.

Variant #1

For example, one such commonly occurring message is something on the order of "Expected <string of chars with 't'> got <non 't'>. Users ask, "How do I interpret the 't', ':', and '.'?"

For the LAT2 UDP table shown below, TestMAX ATPG issues the following text:

```
underspecified UDP (Expected "tx....." got "?x?:?:-")
primitive LAT2 (q, d, gn, ntfy);
output q;
reg q;
input d, gn, ntfy;
```

```

table
# D GN ntfy : Q- : Q+
# --- --- --- ---
? 1 ? : ? : - ; #1
0 0 ? : ? : 0 ; #2
1 0 ? : ? : 1 ; #3
1 x ? : 1 : 1 ; #4
0 x ? : 0 : 0 ; #5
? p ? : ? : - ; #6 "p" includes (0x)
? ? * : ? : X ; #7
endtable
endprimitive

```

In this case the string `Expected "tx...."` contains a 't' meaning a test for 0 or 1 is expected. The periods "." occur for each position for which TestMAX ATPG does not care what the value is after the mismatch is detected, and the colons ':' occur as separators in the same position they hold in the table entries.

The 't' occurs in the first character position, so TestMAX ATPG is expecting the first character of the table entry to test for a 0 or 1. Instead it got "`?x?:?:-`", which can be used to help identify table entry #1 or #6 as the entry corresponding to the violation message. In this case you might need to get the Verilog reference manual out to decipher that the "p" is shorthand for the edge combinations of (01), (0x), (x1), (0z), and (z1). TestMAX ATPG is warning about the (0x) transition, which is similar to a steady state input on GN of "X" because the table entry implies that the output holds state when GN goes from 0 to X regardless of the value of D and Q-.

Let's rewrite table entry #6 and expand it into its variants, skipping those that have edge combinations which include Z.

```

primitive LAT2 (q, d, gn, ntfy);
output q;
reg q;
input d, gn, ntfy;

table
# D GN ntfy : Q- : Q+
# --- --- --- ---
? 1 ? : ? : - ; #1
0 0 ? : ? : 0 ; #2
1 0 ? : ? : 1 ; #3
1 x ? : 1 : 1 ; #4
0 x ? : 0 : 0 ; #5
? (01) ? : ? : - ; #6a expanded
? (x1) ? : ? : - ; #6b expanded
? (0x) ? : ? : - ; #6c expanded
? ? * : ? : X ; #7
endtable
endprimitive

```

This new variant of the UDP table has three lines to replace the original. If you read this with TestMAX ATPG and then perform a run build_model on it, you get the same violation message:

```
underspecified UDP (Expected "tx...." got "?x?:?:-")
```

Note the entry 6c which has the (0x) edge transition for GN and is the closest match to the error message showing "? x ? : ? : -". If the latch enable GN transitions to an X state, the table entry should not indicate a hold state, it would be better if it tested that D and Q- were identical, and if so then the output would be known. You can test this change by expanding entry 6c into two new lines as follows:

```
primitive LAT2 (q, d, gn, ntfy);
  output q;
  reg q;
  input d, gn, ntfy;

  table
    # D GN ntfy : Q- : Q+
    # --- --- --- --- ---
      ? 1 ? : ? : - ; #1
      0 0 ? : ? : 0 ; #2
      1 0 ? : ? : 1 ; #3
      1 x ? : 1 : 1 ; #4
      0 x ? : 0 : 0 ; #5
      ? (01) ? : ? : - ; #6a expanded
      ? (x1) ? : ? : - ; #6b expanded
      0 (0x) ? : 0 : 0 ; #6c1 expanded
      1 (0x) ? : 1 : 1 ; #6c2 expanded
      ? ? * : ? : X ; #7
  endtable
endprimitive
```

This new variant of the UDP table has two lines to replace the original 6c. If you read this with TestMAX ATPG and then perform a run build_model on it, you get a different violation this time:

```
unsupported UDP entry (Entry "??*?:X")
```

The reason the violation is different is because by default TestMAX ATPG shows only the single most significant violation for each module. By curing one violation message you expose the next. To see all of the violation messages at once issue the set_netlist -nocheck_only_used_udps command before reading the file with the modules being tested.

This N21 violation occurs any time a UDP has a notify column entry. There is no gate-level representation for the functionality described by the characters in the notify column and TestMAX ATPG is issuing a warning. Nearly all such models use this column for setting the output to an X as a result of a timing violation. For our ATPG functional model (timingless),

this violation can be ignored. At this point you have eliminated all the warnings that we can.

This example has been useful for demonstrating how to interpret UDP related warning messages and what action to take to try to reduce the warnings. Having led you through the process we'll now suggest some additional changes to the table that you can or cannot have noticed along the way.

The first change is that entry 6a and 6b are essentially redundant to entry 1. An edge transition on GN to a 1 for a level sensitive device is identical to a constant 1 on GN. So we could drop these entries from the table.

The second change is that entries 6c1 and 6c2 are essentially redundant to entries 4 and 5 by a similar argument.

So in the end we could have reduced warnings for this UDP by commenting out the original entry 6 rather than by expanding it into other lines. Each case is different and sometimes an idea for the final solution becomes more obvious by commenting out troublesome lines than by expanding them. Both methods should be considered when troubleshooting UDP messages. The reduced warning UDP is then:

```
primitive LAT2 (q, d, gn, ntfy);
  output q;
  reg q;
  input d, gn, ntfy;

  table
    # D GN ntfy : Q- : Q+
    # --- --- --- --- ---
      ? 1 ? : ? : - ; #1
      0 0 ? : ? : 0 ; #2
      1 0 ? : ? : 1 ; #3
      1 x ? : 1 : 1 ; #4
      0 x ? : 0 : 0 ; #5
      ? ? * : ? : X ; #6
  endtable
endprimitive
```

Variant #2

As another example, a message similar to "Expected <string of chars with 't'> got <string with 0 or 1 or X> is provided.

For the UDP table shown below TestMAX ATPG reports the following message:

```
underspecified UDP (Expected "1xr11?:t:." got "1xr11?:?:X")

primitive FJK (q, j, k, cp, cd, sd, ntfy);
  output q;
  reg q;
```

Chapter 29: ATPG FAQ

Interpreting UDP Messages

```

input j,k, cp, cd, sd, ntfy;

table
# J K CP CD SD ntfy Q- : Q+
# --- --- --- --- --- : --- : ---
    0 0 r 1 1 ? : ? : - ; # 1. hold

    0 1 r 1 1 ? : ? : 0 ; # 2. clocked K
    0 1 r x 1 ? : ? : 0 ; # 3.
    ? ? ? x 1 ? : 0 : 0 ; # 4.

    1 0 r 1 1 ? : ? : 1 ; # 5. clocked J
    1 0 r 1 x ? : ? : 1 ; # 6.
    ? ? ? 1 x ? : 1 : 1 ; # 7.

    1 1 r 1 1 ? : 0 : 1 ; # 8.
    1 1 r 1 1 ? : 1 : 0 ; # 9.

    ? ? f 1 1 ? : ? : - ; # 10.

    0 0 (x1) 1 1 ? : ? : - ; # 11.
    0 1 (x1) 1 1 ? : 0 : 0 ; # 12.
    1 0 (x1) 1 1 ? : 1 : 1 ; # 13.
    0 0 (0x) 1 1 ? : ? : - ; # 14.
    0 1 (0x) 1 1 ? : 0 : 0 ; # 15.
    1 0 (0x) 1 1 ? : 1 : 1 ; # 16.

    * ? ? 1 1 ? : ? : - ; # 17.
    ? * ? 1 1 ? : ? : - ; # 18.

    ? ? ? 0 1 ? : ? : 0 ; # 19. clear
    ? ? ? 1 0 ? : ? : 1 ; # 20. set
    ? ? ? 0 0 ? : ? : 0 ; # 21. clear and set active
    ? ? ? 0 x ? : ? : 0 ; # 22. pessimism

    ? ? (?0) 1 1 ? : ? : - ; # 23. ignore falling clock.
    ? ? (1x) 1 1 ? : ? : - ; # 24.

    ? ? ? (?1) 1 ? : ? : - ; # 25. ignore changes on set and
    ? ? ? 1 (?1) ? : ? : - ; # 26. reset.
    ? ? ? ? ? * : ? : X ; # 27.
endtable
endprimitive

```

Here the message "Expected "1xr11?:t." got "1xr11?:?:X" provides a useful hint. When we try to find the table entry to match the "1xr11?:?:X" we won't be able to do so. Notice that the Q+ entry is an X and the only entry in our UDP table that explicitly sets the next state Q+ to X is entry 27. By Verilog default, all input combinations not explicitly defined by UDP table entries result in outputs set to X. So what TestMAX ATPG is trying to hint at is that a table entry is missing and it expects to see one of the form "1 x r 1 1 ? : t : . ;", where the t is replaced by 0 and 1, and an appropriate value for Q+ has been used of either 0/1/x/- .

Chapter 29: ATPG FAQ
Interpreting UDP Messages

Let us construct two additional entries and add them to the table by expanding the 't' into 0 and 1, and expanding the final output column, shown as a period, into its appropriate values given the input states:

```
# J K CP CD SD ntfy Q : Q+
# --- --- : --- : ---
1 x r 1 1 ? : 0 : 1 ; # A. Q=0, clock J=1
1 x r 1 1 ? : 1 : x ; # B. Q=1, clock K=X
```

When we insert these new entries we don't really need the ones in which the output is X as that is the default but it won't hurt for now. We also don't want to add these to the end of the table after all the entries with wildcard '?' or we won't get a match. We'll add our two new lines after entry #9 as lines A and B.

```
primitive FJK (q, j, k, cp, cd, sd, ntfy);
  output q;
  reg q;
  input j,k, cp, cd, sd, ntfy;

table
# J K CP CD SD ntfy Q- : Q+
# --- --- : --- : ---
  0 0 r 1 1 ? : ? : - ; # 1. hold

  0 1 r 1 1 ? : ? : 0 ; # 2. clocked K
  0 1 r x 1 ? : ? : 0 ; # 3.
  ? ? ? x 1 ? : 0 : 0 ; # 4.

  1 0 r 1 1 ? : ? : 1 ; # 5. clocked J
  1 0 r 1 x ? : ? : 1 ; # 6.
  ? ? ? 1 x ? : 1 : 1 ; # 7.

  1 1 r 1 1 ? : 0 : 1 ; # 8.
  1 1 r 1 1 ? : 1 : 0 ; # 9.

  1 x r 1 1 ? : 0 : 1 ; # A. Q=0, clock J=1
  1 x r 1 1 ? : 1 : x ; # B. Q=1, clock K=X

  ? ? f 1 1 ? : ? : - ; # 10.

  0 0 (x1) 1 1 ? : ? : - ; # 11.
  0 1 (x1) 1 1 ? : 0 : 0 ; # 12.
  1 0 (x1) 1 1 ? : 1 : 1 ; # 13.
  0 0 (0x) 1 1 ? : ? : - ; # 14.
  0 1 (0x) 1 1 ? : 0 : 0 ; # 15.
  1 0 (0x) 1 1 ? : 1 : 1 ; # 16.

  * ? ? 1 1 ? : ? : - ; # 17.
  ? * ? 1 1 ? : ? : - ; # 18.
```

```

? ? ? 0 1 ? : ? : 0 ; # 19. clear
? ? ? 1 0 ? : ? : 1 ; # 20. set
? ? ? 0 0 ? : ? : 0 ; # 21. clear and set active
? ? ? 0 x ? : ? : 0 ; # 22. pessimism

? ? (?0) 1 1 ? : ? : - ; # 23. ignore falling clock.
? ? (1x) 1 1 ? : ? : - ; # 24.

? ? ? (?1) 1 ? : ? : - ; # 25. ignore changes on set and
? ? ? 1 (?1) ? : ? : - ; # 26. reset.
? ? ? ? ? * : ? : X ; # 27.
endtable
endprimitive

```

After adding these new entries TestMAX ATPG now produces a different violation message of:

```
underspecified UDP (Expected "xtrl1....." got "x?r11?:?:X")
```

We have succeeded in eliminating one warning message by adding a table entry which reduces pessimism in the model behavior. By repeatedly analyzing the UDP messages in this manner and adjusting the table we can eventually eliminate all of the warning messages due to missing entries.

Variant #3

As another example, a violation message of N28, `unsupported priority` can occur. This is an indication of incomplete information in the table needed for asynchronous set or clear behavior of the UDP to match the TestMAX ATPG primitive's asynchronous set/clear behavior. N28 violations deal generally with prioritization between asynchronous set, clear, and clocks.

For the UDP table shown below TestMAX ATPG issues an N28 violation:

```

unsupported priority (reset "CD" has no priority over other clocks)

primitive TOGGLE (Q, CP, CD, ntfy);
  output Q;
  reg Q;
  input CP, CD, ntfy;

table
# CP CD ntfy: Q- : Q+
# --- --- --- : --- : --- ;
  (01) 1 ? : 0 : 1 ; # 1. toggle
  (01) 1 ? : 1 : 0 ; # 2. toggle

  0 1 ? : ? : - ; # 3. hold

  ? 0 ? : ? : 0 ; # 4. async clear

```



```

(01) x ? : 1 : 0 ; # 5. reduce pessimism
    0 x ? : 0 : 0 ; # 6. potential clear
    ? ? * : ? : X ; # 7. go to X
endtable
endprimitive

```

What TestMAX ATPG is trying to indicate with the N28 message is that the pin CD has been identified as an asynchronous reset but that it has not been completely described to have priority over "other clocks", or in this case the CP pin. The TestMAX ATPG primitive for a DFF device models the behavior of the asynchronous reset pin to have priority over the clock pin, so the ATPG primitive we wish to use does not exactly match this table.

If you are interested in eliminating the N28 violation look for entries in the table which describe asynchronous set or reset functions. Generally they define the behavior when the "clock" pins are at steady states. You should add an entry that defines the async behavior in the presence of clock events. In our example line "4b" is added to produce the model below:

```

primitive TOGGLE (Q, CP, CD, ntfy);
output Q;
reg Q;
input CP, CD, ntfy;

table
# CP CD ntfy: Q- : Q+
# --- --- : --- : --- ;
(01) 1 ? : 0 : 1 ; # 1. toggle
(01) 1 ? : 1 : 0 ; # 2. toggle

    0 1 ? : ? : - ; # 3. hold

    ? 0 ? : ? : 0 ; # 4. async clear
    * 0 ? : ? : 0 ; # 4b. async clear

(01) x ? : 1 : 0 ; # 5. reduce pessimism
    0 x ? : 0 : 0 ; # 6. potential clear
    ? ? * : ? : X ; # 7. go to X
endtable
endprimitive

```

Line 4 specifies that the output is cleared whenever CD=0 for any steady state value on CP (? = 0, 1, or x). By adding line 4b we also define the behavior that any edge event on CP while CD=0 also produces Q=0. This then fully describes an asynchronous reset behavior with priority over clocks and matches our ATPG primitive. This eliminates the N28 violation.

Variant #4

As another example, a violation message of N23, `inconsistent entry` can occur. This is an indication that two entries in the table define conflicting behavior.

For the UDP table shown below TestMAX ATPG issues the N23 violation:

```
inconsistent UDP (Entry should have clocks off: *?1??:-)

primitive DFF (Q, D, CP, SD, ntfy );
output Q;
input D, CP, SD, ntfy;
reg Q;
table
# D CP SD ntfy: Q- : Q+
# --- --- --- : --- : --- ;
  1 r 1 ? : ? : 1 ; # 1. clock D
  0 r 1 ? : ? : 0 ; # 2.

  ? ? 0 ? : ? : 1 ; # 3. async set
  ? * 0 ? : ? : 1 ; # 4.

  ? 0 1 ? : ? : - ; # 5. hold
  ? (?0) 1 ? : ? : - ; # 6.

  * ? ? ? : ? : - ; # 7. ignore edge
  ? ? (?1) ? : ? : - ; # 8. ignore edge

  1 (0x) 1 ? : 1 : 1 ; # 9. possible clock with Q- = D
  0 (0x) 1 ? : 0 : 0 ; # 10.

  1 r x ? : ? : 1 ; # 11. possible SD
  ? ? x ? : 1 : 1 ; # 12. possible SD, with Q- = 1

  1 (0x) x ? : 1 : 1 ; # 13. possible set, possible clock

  ? ? ? * : ? : X ; # 14. go to X
endtable
endprimitive
```

Reviewing the violation message against the lines in the table we can identify line 7 as the closest match to `"* ? 1 ? : ? : -"`. The difference between the violation message and line 7 is that the violation message shows `SD=1`. In reviewing line 7 we see that this line indicate the output is held for any transition on the D input and for any values of CP and SD, including `SD=0!!!` So TestMAX ATPG is suggesting we change this line to indicate `SD=1` is required.

```
primitive DFF (Q, D, CP, SD, ntfy );
output Q;
input D, CP, SD, ntfy;
reg Q;
```

```

table
# D CP SD ntfy: Q- : Q+
# --- --- --- : --- : --- ;
  1 r 1 ? : ? : 1 ; # 1. clock D
  0 r 1 ? : ? : 0 ; # 2.

  ? ? 0 ? : ? : 1 ; # 3. async set
  ? * 0 ? : ? : 1 ; # 4.

  ? 0 1 ? : ? : - ; # 5. hold
  ? (?0) 1 ? : ? : - ; # 6.

  * ? 1 ? : ? : - ; # 7. ignore edge
  ? ? (?1) ? : ? : - ; # 8. ignore edge

  1 (0x) 1 ? : 1 : 1 ; # 9. possible clock
  0 (0x) 1 ? : 0 : 0 ; # 10.

  1 r x ? : ? : 1 ; # 11. possible SD
  ? ? x ? : 1 : 1 ; # 12.
  1 (0x) x ? : 1 : 1 ; # 13. possible set, possible clock

  ? ? ? * : ? : X ; # 14. go to X
endtable
endprimitive

```

After changing CP=? in line 7 to SD=1 we've eliminated the N23 violation.

Debugging UDP-based Models

The following advice is beneficial if you are debugging violation messages encountered when reading UDP modules:

1. Put the UDP definition into its own file until debugging is completed. By doing so, the only warning or error messages are from the single UDP you are debugging.
2. Enable display of ALL messages in the model, not just the most serious ones by issuing "set netlist -nocheck_only_used_udps" before reading the file containing the UDP.
3. Within TestMAX ATPG define an alias to make the repeated steps easier. For example, if the file containing the UDP is named 'udp.v' then define an alias named 'go' similar to:v

```

alias go clear ; build -f ; read net udp.v -del ; \ run build ; rep
viol -all

```

4. Now type "go" and review the violation messages for guidance on what might need changed. Next, edit your source 'udp.v' file, save the edits and return to TestMAX

What is the Difference between the `add_capture_masks` vs `add_cell_constraints` Commands?

ATPG and type "go" again, or "!!". Repeat this process until you've eliminated as many violation messages as possible.

5. Review the resulting derived ATPG model in the graphical schematic viewer:
 - a. Use the SHOW button and select ALL.
 - b. Compare the gate level functionality of the derived ATPG model with the intended functionality of the truth table.
6. If your UDP table is complex, or results in more gates than you expected then consider using the `set_netlist -noxmodeling` option before reading in the UDP. This avoids trying to explicitly model output=X states. After this is done you can use a `write_netlist` command to get a gate level implementation of the simplified function using ATPG primitives. This might be helpful in trying to understand how to modify your UDP table to produce fewer ATPG gates.

What is the Difference between the `add_capture_masks` vs `add_cell_constraints` Commands?

Answer:

As an example, suppose you had a cell called `my_bad_cell` with a 1 on its D input and an active clock edge. If that cell has a cell constraint of OX, it captures the value of 1. Then, when the scan chain unload values are set by the simulator, it substitutes an X for the existing value of 1. Further, if any other subsequent clock cycles captures a value of 1 in `my_bad_cell`, the captured value of 1 might propagate to other cells.

In these same circumstances, if you placed a capture mask on `my_bad_cell`, it captures a value of X rather than of 1. This means that any cells downstream from `my_bad_cell` also encounter that X value rather than the value of 1 that was on the D input of `my_bad_cell`.

Similarly, if you have an unclocked test pattern, the cell with a capture mask unloads the same value that was loaded into it by the scan chain load. Even in that situation, the cell with a cell constraint of OX unloads an X.

Masking a Scan Cell by Instance Name

You can use the following commands to mask a scan cell by instance name:

- To mask load/unload but allow non-X capture: `add_cell_constraints xx instance_name`
- To mask capture but allow non-X load/unload: `add_capture_masks instance_name`
- To mask both load/unload and capture: `add_cell_constraints xx instance_name` and `add_capture_masks instance_name`

Masking a Nonscan Cell by Instance Name

You can use the following commands to mask a nonscan cell by instance name:

- To mask load but allow non-X capture: `add_capture_masks instance_name -load_only`
- To mask capture but allow non-X load: There is no direct method. The closest approximation is `set_sdc -hold` (with `set_simulation -timing_exceptions_for_stuck_at`, if necessary) and `read_sdc`, with `set_false_path -to instance_name` in the SDC file
- To mask both load and capture: `add_capture_masks instance_name`

JTAG Support

TestMAX ATPG supports many designs variants using 1149.1 Boundary Scan test methods (JTAG). Designs that contain scan chains as well as JTAG can be classified into four major types according to how scan chains are accessed for test:

Type 1 - Core scan chains are independently accessed in parallel through top level ports without the use of the TAP controller.

Type 2 - Core scan chains are independently accessed in parallel after loading a single JTAG instruction during device initialization.

Type 3 - One or more core or boundary scan chains are accessed via TDI/TDO pins after loading a single JTAG instruction during device initialization. This includes variants of: A) access to the boundary scan only; B) access to an individual core scan chain; C) access to two or more core scan chains daisy-chained into one long chain. The most common variant of this is access to `all` core scan chains as a single, long, daisy-chained scan chain using TDI and TDO.

Type 4 - Boundary Scan chains and core scan chains require unique TAP controller instructions which enable access via TDI/TDO in a one-at-time fashion. These instructions must be applied dynamically each time scan chains are loaded or unloaded.

TestMAX ATPG is able to support all four types above.

Common Tasks for Supporting JTAG

Supporting Scan-Through-Tap designs can involve one or more of the following tasks:

- Use of `test_setup` to initialize the TAP controller.
- Use of `load_unload` to step the TAP controller to different states.

- Constraining TCK and TRSTN to their off states during ATPG to keep the TAP controller in a known state.
- Constraining TMS.
- Limiting the clocks used during ATPG.

Initializing TAP Using test_setup

Except for type 1 designs as described previously, nearly all other types require some initialization of the TAP control. This is accomplished by defining test cycles in the test_setup procedure by means of Vector or V{...} statements. An example test_setup procedure which is common for type 2 designs is shown in the following example. This resets the TAP controller and loads one instruction intended to open up parallel access to core scan chains.

```
MacroDefs {
  test_setup {
    W "my_timing" ;
    V { TMS=0; TCK=0; TRSTN=P; CLK=0; } # pulse TAP reset
    V { TRSTN=1; } # reset off, state now RESET
    V { TMS=0; TCK=P; } # move to IDLE
    V { TMS=1; TCK=P; } # move to SELECT-DR
    V { TMS=1; TCK=P; } # move to SELECT-IR
    V { TMS=0; TCK=P; } # move to CAPTURE-IR
    V { TMS=0; TCK=P; } # move to SHIFT-IR

    # load 4-bit instruction
    V { TMS=0; TCK=P; TDI=0; } # SHIFT-IR, bit1 : IREG=0
    V { TMS=0; TCK=P; TDI=0; } # SHIFT-IR, bit2 : IREG=00
    V { TMS=0; TCK=P; TDI=1; } # SHIFT-IR, bit3 : IREG=001
    V { TMS=1; TCK=P; TDI=1; } # move to EXIT1-IR, IREG=0011

    V { TMS=1; TCK=P; } # move to UPDATE IR
    V { TMS=0; TCK=P; } # move to IDLE
    V { TMS=0; TCK=0; } # clock off
  }
}
```

Each of the V {...} statements in this test_setup procedure should be interpreted as a single tester cycle. Start by applying a reset to the TAP controller by pulsing TRSTN. Next, apply various 1/0 combinations to the TMS input while clocking TCK to move about the TAP controller state machine until you are in the SHIFT-IR state. In the SHIFT-IR state, use TDI to march in our desired instruction, in this example it is a 4-bit instruction "0011". This varies by design. On the last IR shift, raise TMS high causing a transition to the EXIT1-IR state and proceed to adjust TMS while clocking TCK until we arrive at the IDLE state.

For most type 2 designs, this initialization sequence enables the parallel access to core scan chains and with some additional constraints on TCK and TRSTN should be sufficient to pass DRC checks.

For type 3 and 4 designs some additional steps are generally added to the end of `test_setup` to move from an IDLE state to a CAPTURE-DR state. This depends on whether the internal scan chain accessed by the IR instruction loaded requires a CAPTURE-DR operation before shifting the data out. An example `test_setup` with some additional test cycles highlighted is shown below:

```
MacroDefs {
  test_setup {
    W "my_timing" ;
    V { TMS=0; TCK=0; TRSTN=P; CLK=0; } # pulse TAP reset
    V { TRSTN=1; } # reset off, state now RESET
    V { TMS=0; TCK=P; } # move to IDLE
    V { TMS=1; TCK=P; } # move to SELECT-DR
    V { TMS=1; TCK=P; } # move to SELECT-IR
    V { TMS=0; TCK=P; } # move to CAPTURE-IR
    V { TMS=0; TCK=P; } # move to SHIFT-IR

    # load 4-bit instruction
    V { TMS=0; TCK=P; TDI=0; } # SHIFT-IR, bit1 : IREG=0
    V { TMS=0; TCK=P; TDI=0; } # SHIFT-IR, bit2 : IREG=00
    V { TMS=0; TCK=P; TDI=1; } # SHIFT-IR, bit3 : IREG=001
    V { TMS=1; TCK=P; TDI=1; } # move to EXIT1-IR, IREG=0011

    V { TMS=1; TCK=P; } # move to UPDATE IR
    V { TMS=0; TCK=P; } # move to IDLE
    V { TMS=1; TCK=P; } # move to SELECT DR      V { TMS=0; TCK=P; } #
move to CAPTURE DR      V { TMS=0; TCK=0; } # clock off
  }
}
```

Keeping the TAP Controller from Changing State

For type 2, 3, and 4 designs, after working hard to define a `test_setup` procedure to get the TAP controller into just the "right" state it is now important to keep the ATPG algorithm from using TCK or TRSTN which could disturb this state by defining PI constraints. This can be done in a number of ways, but the most convenient of which is to issue the command lines:

```
add_pi_constraints 0 TCK

add_pi_constraints 1 TRSTN
```

When to Constrain TMS

For some designs, constraining TCK and TRSTN is not sufficient and DRC checks fail with scan blockages (rule = S1). Depending upon the design and the design of the TAP controller logic it can also be necessary to constrain the TMS port to a constant value. If you experience scan chain blockages during DRC you might want to define a PI constraint on TMS and try again. Generally TMS is constrained to a 1 but both states might need to be tried.

Controlling TAP using load_unload

For type 3 and type 4 designs, the shifting of scan chains is done through the TDI/TDO ports of the TAP controller and involve clocking of TCK with TMS=0. The majority of scan chain shifting for load/unload is done with the TAP controller in the SHIFT-DR state or sometimes the SHIFT-IR state. For proper TAP controller operation it is necessary to exit this state on the last shift and this requires using TMS=1 for the final shift. To accomplish this last shift with TMS=1 we use a special form of the load_unload procedure where the final scan out measure occurs *outside* of the Shift statement:

```
load_unload
  V { TMS=0; TCK=0; TRSTN=1; CLK=0; RESETB=1; SCAN_EN=1;}
  Shift {
    V { TDI=#; TDO=#; TCK=P; }
  }
  V { TMS=1; TDI=#; TDO=#; TCK=P; } # move to EXIT1-DR
}
```

In the previous example all of the scan chain shifts are done within the Shift statement when TMS=0 except for the final shift which is done with TMS=1. TestMAX ATPG supports scan shifts and measures outside of the Shift statement and you can define more complicated procedures if they are required. For example, the following load_unload procedure not only does the final shift with TMS=1 but also transitions the TAP controller from the SHIFT-DR state through the EXIT1-DR to UPDATE-DR to SELECT-DR to CAPTURE-DR.

```
load_unload
  V { TMS=0; TCK=0; TRSTN=1; CLK=0; RESETB=1; SCAN_EN=1;}
  Shift {
    V { TDI=#; TDO=#; TCK=P; }
  }
  V { TMS=1; TDI=#; TDO=#; TCK=P; } # move to EXIT1-DR
  V { TMS=1; TCK=P; } # move to UPDATE-DR
  V { TMS=1; TCK=P; } # move to SELECT-DR
  V { TMS=0; TCK=P; } # move to CAPTURE-DR
}
```

Accessing Internal Scan Chains Through the TAP

For type 4 designs, multiple internal scan chains are accessed one at a time by loading different TAP instructions. Support of this type of serial access to multiple internal scan chains requires use of special multiple scan group syntax. For more information and examples see: Supporting Multiple Scan Groups in STIL

Limiting Clocks during ATPG

For type 3 and 4 designs, the transition from load_unload to capture and back to load_unload must be done with great care so that proper movement around the TAP controller state diagram is achieved. For example, when accessing the boundary scan register (BSR) as a scan chain a TAP instruction is loaded to enable access, then ATPG wishes to load the BSR by shifting in known values using the "load_unload" procedure. After this loading occurs, the ATPG algorithm applies a "capture_XXXX" procedure in normal functional mode before shifting out the BSR. Without any restrictions, the ATPG algorithm randomly uses any defined clock which has not been constrained to off, as well as a non-clocking capture where no clocks are used. When the BSR is being used, this is disastrous. To avoid this problem TestMAX ATPG provides for restricting clocks used during the capture procedures and it is often the case that the clock needs to be restricted so that only TCK is used. This is accomplished by the `set_drc -clock TCK` command:

Use of this command not only limits the clocks used to only TCK, it also ensures that TCK is `always` used and there are no non-clocking capture procedures attempted. So if DRC should fail with a scan chain block, you might want to try limiting the clock used during "capture" to just TCK and see if the situation improves.

If a design uses scan-through-tap of Type 4 (for example, the TAP controller must be stepped around a precise flow during shift and capture), then you must use the `set_drc -clock TCK` command.

The use restrictions for the `-clock TCK` argument are as follows:

- Only TCK is used; no other clocks can be used.
- Only one application of TCK is allowed between shifts.
- A chain test using our current algorithm always occurs because a chain test wants no clock and this restriction requires TCK.

If you are unaware of these ATPG restrictions, you might have a design in which the core clocks are separate from TCK and are not usable during ATPG. It might be that 90% of the logic cannot be accessed because TestMAX ATPG cannot activate clocks other than TCK.

As an alternative, the `set_drc -controller_clock TCKoption` provides some relief from these restrictions. This setting blindly applies a pulse of TCK during the last capture of each pattern, but does not otherwise use this clock. This ensures that one (and only one)

pulse of TCK occurs for each pattern, which keeps the TAP controller synchronized as it travels around its state machine.

The `set_drc -controller_clock TCK` command supports designs where a scan-through-tap is used with separate core clocks.

The `set_drc -seq_capture` command selects the special Full-Sequential capture procedure defined in the DRC file for Full-Sequential ATPG. Otherwise, by default, the Full-Sequential algorithm uses the same capture procedures as the Basic-Scan and Fast-Sequential ATPG algorithms.

Limitations of `set_drc -controller_clock tck` are as follows:

- TestMAX ATPG cannot create a chain test.
- **Caution:** there is absolutely *no checking* in which it is safe to apply the TCK pulse in parallel with other clocks. This is the designer's responsibility.

TAP controllers with no reset pin

If your TAP controller has no asynchronous reset pin then it is impossible to change the state machine from "XXXX" to a non-X state by simulation of TCK clocks. To overcome this, issue the following command before performing DRC:

```
set_drc -initialize_dfa_dlat random
```

This causes a random initialization to 0 or 1 of all DFF and DLAT devices in the design, and allows the DRC analysis to process the test_setup vectors with the JTAG state machine in a non-X state. *Note:* This random initialization is not part of any patterns produced so that you are responsible for establishing similar initial conditions in any simulator in which you verify patterns.

Node File Format for Bridging Faults

The node file is read with an `add_faults -node_file` command when the fault model is set to bridging. This file can be in two forms: Star-RCXT coupling capacitance report, or node file containing pairs of nets as described in "Node File Format" below.

Star-RCXT Format

An unmodified Star-RCXT coupling capacitance report can be used for a node file; for example:

```
* % coupling victim aggressor
24 1.03e-15 io_c0[5] io_c7[4]
23.9 2.58e-15 io_c2[3] io_if/io_decode1/G7055
23.5 1.32e-15 io_if/\io_c8_decoded[13] io_c5[5]
```

See "Bridging Fault ATPG " in the *TestMAX ATPG User Guide* for information on using Star-RCXT to generate a coupling capacitance report.

Node File Format

The node file contains a list of node pairs used to create bridging faults. It can be provided compressed in gzip format. TestMAX ATPG has no facility to generate such a list, so you are expected to supply it.

Each node pair must be on separate lines. The first two entries indicate the bridge locations of the nodes for a bridging fault. The bridge location must be a valid instance input pin, instance output pin, or a recognized net name. Otherwise, the node pair is ignored. A summary message prints when TestMAX ATPG reads the file, indicating how many node pairs were ignored, as well as the line number of the first occurrence. Lines that have no entries or that begin with a double slash ("/") are ignored. Otherwise, any line that has fewer than two entries results in an error condition, reading stops, and a parsing error message prints the line number.

By default (that is, when only the bridge locations are specified), all four bridging faults associated with a node pair are added to the fault list. The faults added are:

```
Location_A with ba0 and location_B as aggressor at 0  
Location_A with ba1 and location_B as aggressor at 1  
Location_B with ba0 and location_A as aggressor at 0  
Location_B with ba1 and location_A as aggressor at 1
```

When a third field is used, it must be a valid bridging fault selection or a parsing error is issued and reading terminates. Any entries used after the third field is ignored.

The node pair line has the following format:

```
<bridge_locationA> <bridge_locationB> [wand | wor | adom | bdom | comp | ba0 | ba1 |  
rba0 | rba1]
```

Where:

wand

Indicates that the bridging fault exhibits a wired AND effect.

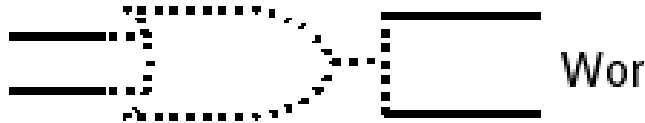


These faults are added:

- A with ba0 and B as aggressor at 0
- B with ba0 and A as aggressor at 0

wor

Indicates that the bridging faults exhibits a wired OR effect.



These faults are added:

- A with ba1 and B as aggressor at 1
- B with ba1 and A as aggressor at 1

adom

Indicates that the bridging faults associated with a dominant first node are added to the fault list.

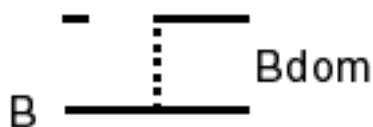


This includes the ba0 and ba1 faults for the node pair with bridge_locationA being the aggressor node.

- B with ba0 and A as aggressor at 0
- B with ba1 and A as aggressor at 1

bdom

Indicates that the bridging faults associated with a dominant second node are added to the fault list. Bdom is identical to Adom but with B taking the place of A.



This includes the ba0 and ba1 faults for the node pair with bridge_locationB being the aggressor node.

- A with ba0 and B as aggressor at 0
- A with ba1 and B as aggressor at 1

comp

Indicates that a composite set of all four faults associated with a node pair are added to the fault list. It is the same as default behavior. Comp type is equivalent to cumulate Adom and Bdom type for the same node pair. It is also equivalent to cumulate Wand and Wor type or BA0, BA1, RBA0 and RBA1 type for the same node pair.

- A with ba0 and B as aggressor at 0
- A with ba1 and B as aggressor at 1
- B with ba0 and A as aggressor at 0
- B with ba1 and A as aggressor at 1

BA0

A single ba0 fault is added to A, and B is the aggressor at 0.

BA1

A single ba1 fault is added to A, and B is the aggressor at 1.

RBA0

A single ba0 fault is added to B, and A is the aggressor at 0.

RBA1

A single ba1 fault is added to B, and A is the aggressor at 1.

Optimizing Basic Scan Patterns

You can use the `-optimize_patterns` option of the `run_atpg` command to produce a compact set of patterns with high test coverage. This option enables you to use a single `run_atpg` command instead of iterating multiple `run_atpg` commands and manually adjusting various parameters.

When the `-optimize_patterns` option is set, TestMAX ATPG monitors the ATPG process and dynamically adjusts the internal algorithms to generate a compact pattern set. The trade-off is a longer runtime. All manually specified `run_atpg` settings, such as abort limits, minimum detects, and merge limits, are ignored during this operation. However, these settings are restored after pattern optimization is completed.

Note that the `-optimize_patterns` option generates two-clock ATPG patterns as basic scan patterns. But they are stored, read, and simulated as fast-sequential patterns. As a result, a fault simulation that uses two-clock ATPG patterns usually takes longer than the original ATPG run.

The `-optimize_patterns` option of the `run_atpg` command will work with the `-chain_test`, `-coverage`, and `-patterns` options of the `set_atpg` command. This option also works with all power aware options of the `set_atpg` command. However, the power aware options might impact the effectiveness of the pattern optimization process.

The `-optimize_patterns` option is useful during a final TestMAX ATPG run when you want to optimize the pattern count. It generates a lower number of patterns and produces similar test coverage compared to a single `run_atpg -auto_compression` command. You cannot use the `-optimize_patterns` option with any additional `run_atpg` options.

You should use the `run_atpg -auto_compression` command for general pattern generation purposes, such as initial test coverage estimates, writing patterns for verification, analyzing the effects of various options, and obtaining good test coverage and pattern count without increased runtimes. For details on using the `-auto_compression` option, see Using Automatic Mode to Generate Optimized Patterns.

Note the following limitations when using the `-optimize_patterns` option:

- Multiple `run_atpg` commands are supported, but pattern optimization can only be specified one time.
- A learned recipe is not saved.
- Fast-Sequential and Full-Sequential ATPG modes are not supported.
- Be aware that unlike the `run_atpg -auto_compression` command, specifying `set_atpg -capture_cycle number` will not enable Fast-Sequential ATPG during the pattern optimization process. To run Fast-Sequential top-off ATPG, it must be done as an extra step. For example:

```
run_atpg -optimize_patterns
set_atpg -capture 4
run_atpg -auto fast_sequential
```

- Only stuck-at and transition fault models are supported.
- Distributed APTG is not supported.

Design and ATPG Usage Tips for Designs with Phase Lock Loops (PLLs)

It is very common for designs to have one and sometimes more than one PLL devices used as clock generators. Here are some design and ATPG setup techniques that can be generically applied to designs containing PLLs.

Design Considerations:

- Plan on bypassing the PLL as a clock source in ATPG test mode. The most common method for doing this is to place a MUX on the clock output of the PLL under the control of a test mode pin and in test mode to provide the core clock from a direct top level input.
- Some PLLs are designed with either a power down control or a test mode control. If they exist, use them by placing the PLL into either a power down mode or test mode during ATPG test. Consult with your library supplier for their preferred mode if there are choices. In the absence of a preference you should choose a powered down mode. On the tester, this can help reduce the power consumption which would otherwise occur if the PLL is trying to acquire lock and we've broken the clock feedback for ATPG purposes.
- If the PLL does not have a power down or test mode, then consider disabling its clock `input` during ATPG test mode in addition to the MUX around its clock output. If the PLL has no power down mode then the next best thing is to make sure it sees no transitions on its input clock during the application of ATPG patterns on the tester. In addition, some simulation models behave better if the input clock is shut off during ATPG mode. This can be accomplished as easily as adding an AND gate in the clock input path, disabled when in ATPG test mode.

ATPG Tool Considerations:

- PLL modules are typically modeled with behavioral techniques and TestMAX ATPG does not support behavioral modeling outside of its RAM/ROM syntax. It might be necessary to black box the PLL model. This can be done before building the in-memory image by use of the `-black_box` option of the `set_build` command. For example:

```
set_build -black_box PLL_500MHZ
```

where "PLL_500MHZ" is the module name to be black boxed.

- The MUX on the PLL output must have its select line held at a constant value. This generally involves defining a PI constraint, but on circuits with JTAG/TAP this can involve sequencing the TAP controller in the `test_setup` procedure to archive a constant value on the select line.

Shared Scan-In Designs

TestMAX ATPG only supports the patterns formats for WGL and STIL. Note that shared scan-in designs are not supported if they are Adaptive Scan designs.

Release 2002.09 (and older) of TestMAX DFT does not write correct STL procedure file-formatted files for such designs. Before running DRC on a shared scan-in design, you need to modify its STIL procedure file.

Here is an example of an STIL procedure file for a shared scan-in design that passes DRC.

Assume the design has:

- 10 scan chains. Three scan chains (C8, C9, and C10) share the same scan input port.
- clocks CLKA and CLKB (off state is 0). These clocks pulse during scan chain shifting.
- TEST_MODE enable signal to put the chip in test mode.
- SCAN_ENABLE enable signal to start scan chain shifting.
- RESET signal that resets the chip to its active 1 state.

For this example, the minimum STL procedure file information needed to pass is as follows:

```
STIL;

ScanStructures {
ScanChain c1 { ScanIn "si1"; ScanOut "so1"; }
ScanChain c2 { ScanIn "si2"; ScanOut "so2"; }
ScanChain c3 { ScanIn "si3"; ScanOut "so3"; }
ScanChain c4 { ScanIn "si4"; ScanOut "so4"; }
ScanChain c5 { ScanIn "si5"; ScanOut "so5"; }
ScanChain c6 { ScanIn "si6"; ScanOut "so6"; }
ScanChain c7 { ScanIn "si7"; ScanOut "so7"; }
ScanChain c8 { ScanIn "si8"; ScanOut "so8"; }
ScanChain c9 { ScanIn "si8"; ScanOut "so9"; }
ScanChain c10 { ScanIn "si8"; ScanOut "so10"; }
}

Procedures {
"load_unload" {
V { "CLKA"=0; "CLKB"=0; "RESET"=0; "TEST_MODE"=1; "SCAN_ENABLE"=1; }
Shift {
V { _si=\r8 #; _so=\r10 #; "CLKA"=P; "CLKB"=P; }
}
}
}

MacroDefs {
```



```
"test_setup" {  
V { TEST_MODE=1; RESET=0; CLKA=0; CLKB=0; SCAN_ENABLE=0; }  
}  
}
```

Creating End-of-Cycle Measures in ATPG Patterns

The TestMAX ATPG combinational ATPG algorithm is based on a `preclock` measure of scan outputs and regular design outputs. This `preclock` measure requires a fundamental event order within a tester cycle of:

- Force inputs
- Measure outputs
- Pulse capture clocks (optional)

This `preclock` measure has been chosen because it enables superior ATPG pattern generation performance without compromising on pattern count or tester cycle count.

Many ASIC vendors and users prefer to have patterns with an event order using `postclock` or End-of-Cycle measures. A `postclock` measure seems to be a more comfortable form because it matches the event order of most functional patterns and is perhaps easier to debug.

Many ASIC vendors claim that they can only accept `postclock` measure format. It is rare to find an ASIC tester which does not support the `preclock` measure. More often than not it is a software translation limitation rather than a tester limitation. The fundamental event order for a `postclock` measure cycle is:

1. Force inputs
2. Pulse capture clocks (optional)
3. Measure outputs

The TestMAX ATPG combinational ATPG algorithm will not produce this `postclock` form of patterns. However, the `postclock` style of patterns can be created using some post processing techniques applied during the `write_patterns` command.

Drawbacks of Using End-of-Cycle Measures

Here are some drawbacks of creating End-of-Cycle style ATPG patterns:

- The internal pattern format is in preclock format and attempting to compare internal patterns to an external form in STIL, Verilog, VHDL, and so forth. is more difficult.
- At least one additional tester cycle is needed for every ATPG pattern. This additional cycle is placed in the load_unload procedure and performs a scan chain pre-measure before the Shift procedure.
- Capture Clock procedures cannot be condensed into a single tester cycle and must be defined with a minimum of 2 tester cycles. The first cycle performs a force PI, measure PO, and the second cycle performs an optional clock pulse.

In general terms, the cost of implementing the End-of-Cycle measure is two additional tester cycles for every ATPG pattern generated. There is no increase or decrease to overall test coverage or the number of ATPG patterns produced by choosing End-of-Cycle measures over preclock measure. This can or cannot be significant, depending upon your budget for test cycles or tester time.

Requirements Needed to Produce End-of-Cycle Measures

To create End-of-Cycle style ATPG patterns with the `write_patterns` command the following setup steps are required:

1. The DRC procedure file must contain a timing definition block and the time at which outputs and scan outputs are measured must be defined to occur at the end of a test cycle, after any potential clock pulses.
2. All capture procedures must be defined using two or more test cycles and the event order must be:

```
cycle 1: force PI's, measure PO's cycle 2: mask PO's, pulse clocks
```
3. The load_unload procedure must pre-measure the first scan chain output before the first scan shift is performed.

In this case, you are still measuring outputs before a clock. You do not change the fundamental event order which must continue to be: 1) force PI's, 2) measure PO's, 3) pulse clocks; make sure that relative to a single tester cycle timing, the measures occur after any clock pulses. For example, if you define tester timing for a 100nS period in which PI's are forced at offset zero, a clock is pulsed from 50 to 70ns and outputs are measured at 99ns, then your "capture_XXX" procedures produce a 2-cycle timing of:

```
time action                               cycle
-----
000 force PI's                             1
```

Chapter 29: ATPG FAQ

Troubleshooting Pattern Simulation Failures

050	assert clock (but inhibited)	1
070	remove clock	1
099	measure PO's	1
100	force PI's (no change needed)	2
150	assert clock	2
170	remove clock	2
199	measure PO's (masked)	2

The fundamental event order is still that of the preclock timing under which the ATPG patterns are generated but the per cycle timing is such that measures are performed at the end of a tester cycle.

Troubleshooting Pattern Simulation Failures

The following sections provide troubleshooting guidance when you experience mismatches during simulation of ATPG generated patterns.:

- [Your ATPG Patterns are Failing: What Next?](#)
- [Interpreting the Simulation Failure Messages](#)
- [Isolating a Failing Pattern to Assist in Troubleshooting](#)
- [Eliminating a Few Failing Patterns from a Larger Set](#)
- [Locating the Target Fault Site for the Failing Pattern](#)
- [Isolating a Fault List to Assist in Troubleshooting](#)
- [Interpreting the report_patterns Command](#)
- [Viewing Pattern Data in the Graphical Schematic Viewer](#)
- [Using the analyze_simulation_data Command](#)

Note the following:

If you need to regenerate patterns while debugging them, you should adjust the tester cycle period to a value that is easy to work with and to troubleshoot. For example, a tester cycle period of 1000ns is easier to work with than one of 240ns. If you have multiple timing sets defined, try to make them all have identical tester cycle periods.

The parallel Verilog testbench does not present the same simulation times for the capture procedures as the serial testbench. If you want to have the parallel scan load have the identical timing as the serial scan load would take, then define `tmax_serial_timing` during the Verilog compilation/simulation. This might be done by placing a ``define` statement within the Verilog testbench, or by adding a `+define+tmax_serial_timing` argument to the Verilog command line options.

Your ATPG Patterns are Failing: What Next?

The following troubleshooting tips are presented in a suggested order of investigation and attack. However, you should read all of the suggestions and choose an order of exploration that makes sense for your particular design and failure mechanism.

1. What timing mode are you using in the simulator?

Most users simulate 5 to 10 patterns using serial scan loading with full timing and the balance of the patterns using parallel scan loading in *zero delay* or *typical* timing mode. If you are using unit delay timing you should try zero delay. If zero delay is failing then try typical timing. If that fails try full annotated timing. To make this process more efficient see the topic at the end of this topic on isolating a failing pattern.

2. Are the chain tests failing?

Unless you have explicitly suppressed the creation of the chain test by use of `set atpg -chain off` then the first pattern (pattern 0) in the pattern block is a chain test pattern. By default, this pattern shifts a repeating value of 0011... into each scan chain, there is no capture clock in this pattern so the values are not disturbed, and the same values (adjusted for scan-in to scan out inversions) is expected to shift out again during the scan unload of pattern 1. If you experience simulation mismatches on pattern 0 (which occurs during pattern 1 scan loading) then you have a fundamental problem with your design because it cannot successfully shift a bit from scan input to scan output. Look for a clock timing problem in the vicinity of the scan cell which fails. You might find one of the alternative scan chain patterns such as "1000" easier to debug, these are selected by the `-chain_test` option of the `set_atpg` command before generating patterns. See the topic at the end of this topic for interpreting failure messages and translating them into scan cell instance names.

Another possibility when the scan chain tests fail is that you are using a pattern translator from one of the four native pattern formats created by TestMAX ATPG (STIL, WGL, VHDL) and your pattern translator is introducing an error. This can happen when the pattern translator was created for a different ATPG tool and then used with the TestMAX ATPG output. If you are using WGL through a translator to other formats carefully study the inversion control as well as the bidirectional port mapping controls of the `set_wgl` command. Also check the FAQ section of this online help for vendor specific WGL setup and configuration advice.

Stick to basics and work on getting the chain tests to pass before moving on to failures in other patterns.

3. Are you observing setup/hold or other timing errors?

Does your simulator indicate any setup or hold or other timing violation before the ATPG pattern mismatch? If so, you can have some timing problems to correct that are outside of the scope of ATPG patterns. Investigate the areas experiencing the timing problems and

correct them. The `add_capture_mask` command can be helpful for disabling the capture of any expected values at state elements with setup/hold timing problems. If the state element is also a part of a scan chain, then the `add_cell_constraints` command can be used to mask observed values and control loaded values into that state element.

If you are experiencing simulation timing problems then you might benefit from switching to zero delay simulation mode. This is not always successful, though, and having ATPG pattern mismatches in zero delay mode is not conclusive proof of bad patterns.

If you feel comfortable, you can also temporarily edit the SDF timing annotation file to change the setup or hold limit to zero or a very small number and then re-simulate. If these simulations now pass, this is an indication that the ATPG simulation mismatch is directly linked to a timing problem. The ATPG patterns show mismatches until the timing problems are corrected.

4. Has your ATPG library been validated?

Massive simulation mismatches can often be a sign of a bad ATPG model.

Unless you have successfully used your current library before, you should suspect the ATPG models in use do not match the simulation library models. Even if you have used the library before, you might be using different library cells with the current design than with the previous design. When an ATPG model produces a different expected answer (other than X) than the simulation model, result in ATPG patterns which fail in simulation.

5. Is TestMAX ATPG producing a bad pattern?

One sanity check that you can perform to check TestMAX ATPG patterns is to run a good machine simulation on the patterns both with and without the `-sequential` option.

```
set_patterns -external saved_pattern_file  
  
run_simulation  
  
run_simulation -sequential
```

Here is some background on the `run_simulation` command. The first form uses the same simulation engine used during Basic-Scan and Fast-Sequential pattern generation to check the ATPG algorithms generated patterns. So a person would not expect there to be any simulation mismatches reported by the `run_simulation` command for patterns created with the same version of TestMAX ATPG and restored to the same conditions under which the original patterns were generated. If you see mismatches, please provide a testcase, as there might be a bug involved.

The second form of the command, with the `-sequential` option actually uses a different simulation engine whose primary intended function is the simulation of non-ATPG functional patterns. This simulation engine has many limitations, and it is not uncommon for it to report mismatches when it is used on ATPG patterns. However, if use of the

`-sequential` option does show a difference, then it is best to have Synopsys evaluate whether that difference is expected or a bug.

In summary, mismatches using `run_simulation` are not expected and considered a high probability indicator of a bug. In contrast, mismatches using the `-sequential` option are often normal and only occasionally an indication of a problem; however, it is best to submit a testcase to Synopsys for review.

If there are no mismatches reported by either form of the `run_simulation` command, this does not rule out the possibility that there is a TestMAX ATPG bug. If you have exhausted all reasonable possibilities then it is time to send in a testcase and the referenced simulation libraries, timing files, control files, and so forth.

Important Note: Do not add the `run_simulation` step to your generic command scripts, as it is not a necessary step in the standard ATPG flow. Performing a `run_simulation` command at the end of an ATPG run wastes CPU time because the simulation has already been done during the vector generation process; there is no need to repeat this simulation. Use of the `run_simulation` command is only recommended when debugging patterns is necessary or when working with functional patterns.

6. Are you getting massive failures or just a few patterns failing?

If just a few patterns are failing you can gain some useful clues about those patterns by performing a `report_patterns -all -types` command. Check to see whether the patterns that fail are associated with the same clock or whether they are of a particular type, such as Basic Scan with `clock_on_measures` (COM) for example.

Finding a pattern to the failures can give you some potential workarounds. If you find the clock-on measures are failing you could return to DRC mode and disable them and then regenerate patterns. For some clocks, such as asynchronous resets, you could try constraining them to an off value.

If there are just a few patterns failing you could extract and eliminate them from the pattern set by making use of the `-reorder` option of the `write_patterns` command. This option takes as an argument a file containing a numeric list which defines both the order and pattern numbers of the patterns to be written. It is a convenient method for dropping selected patterns from the pattern output. *Note*, however, that you cannot drop or reorder patterns from within a range of Full-Sequential patterns. This is because Full-Sequential patterns assume the design is left at the simulation state caused by the prior Full-Sequential pattern. Any dropping or reordering of Full-Sequential patterns would lead to simulation failures. and so is not allowed.

7. Are you using parallel patterns? What was the shift count?

Using the `-parallel n_shifts` with a value of 1 or more when writing patterns assists in loading the nonscan devices to known states by serially simulating the last N shifts of every parallel scan chain load. This value is automatically calculated and included in the STIL pattern file. You can overwrite this value using the `write_testbench` command,

or the `stil2verilog` command using a configuration file, or on the VCS compilation command line using the predefine options `+tmax_parallel=N`, as documented in the *Test Pattern Validation User Guide*.

8. Are your failures isolated to a few scan cells?

Are you getting the same few scan cell locations failing over and over again? If so, then you might wish to go back and use the `add_cell_constraints OX` command to mask off the observe value at that cell and generate new patterns. There is a drop in test coverage but your new ATPG patterns created can then pass in simulation.

If you are using Fast-Sequential or Full-Sequential ATPG along with a cell constraint of X, or XX, you might wish to consider using the `add_capture_masks` as well. The cell constraint causes the cell to be loaded to X, but the capture mask is necessary to ensure the cell remains at X for a pattern where multiple capture clocks might be applied.

9. Did you have DRC violation warnings that would indicate that patterns might fail?

Did you have any N20 violations when reading the library or building the design? If so, there is a risk that the Verilog simulation model predicts an X when the ATPG model predicts a non-X. This rarely causes a simulation mismatch but you might be in the unlucky 2% of N20 violations that are the root cause of the simulation mismatch. It might be worthwhile to perform a library validation if your library cells. A mismatch during library validation can identify a potential cause of simulation mismatches in your design.

The presence of certain DRC rule violations, such as C1, C5-C14, and S29 for the Basic-Scan or Fast-Sequential ATPG algorithms, and C22 and C25 for the Full-Sequential algorithm, can cause TestMAX ATPG to create patterns that fail in simulation. These warnings should be carefully investigated/corrected before starting ATPG. If you have not corrected them, an additional warning is issued at the start of ATPG pattern generation.

There is a `-mask` option of the `set_rules` command you can use which attempts to increase the chances of patterns successfully simulating in exchange for potentially lower test coverage. Generally this masking is unnecessary for the Basic-Scan and Fast-Sequential ATPG algorithms.

Have you ignored any V18 or V20 violations? If so, there is a risk that this has caused simulation failures. Consult the online help for the full text of V18 and V20 violations and the risks involved.

Interpreting the Simulation Failure Messages

When a parallel compare fails the bit number listed is the bit in the chain relative to the scan chain output port and the numbering scheme starts from zero. So, bit 0 is the bit that is connected directly to the scan output port, bit 1 is the bit one shift clock away from the scan output, and so forth. So the scan cell bit position can also be thought of as the number of shifts required to move the scan cell data to the scan output.

```

XTB: Reading test data file "/path/to/mxtb_usf.dat"
XTB: Total patterns number 4
XTB: Starting parallel simulation of 4 patterns
>>> Error during scan pattern 2 (detected during parallel unload of
pattern 1)
>>>   At T=840.00 ns, V=9, exp=1, got=0, chain c0, pin SO, scan cell
10
>>>   At T=840.00 ns, V=9, exp=0, got=1, chain c0, pin SO, scan cell
11
    
```

To find out the corresponding scan element to the scan cell bit that has the mismatch you use the `report_scan_cells` command and specify the scan chain name and the bit number. The report lists the instance pathname of the scan cell and if you specify the `-pins` option also lists the pin name and inversion information for the scan cell.

```

TEST> report_scan_cells c1 -pins
chain  cell type      inv gate#      instance_name (type)
-----
c1     0   MASTER          IN 147      reg4/r (N_LATCH)
      input          I 147      reg4/r/D (N_LATCH)
      output         N 147      reg4/r/Q (N_LATCH)
c1     1   MASTER          IN 145      reg3/r (N_LATCH)
      DSLAVE         IN 146      reg4/lat1 (P_LATCH)
      input          I 145      reg3/r/D (N_LATCH)
      output         N 146      reg4/lat1/Q (P_LATCH)
c1     2   MASTER          NI 143      reg2/r (N_LATCH)
      SCANTLA        IN 144      reg3/lat1 (P_LATCH)
      input          N 143      reg2/r/D (N_LATCH)
      output         N 144      reg3/lat1/Q (P_LATCH)
    
```

Many users prefer to write the complete list of scan cells into a file for ease of reference:

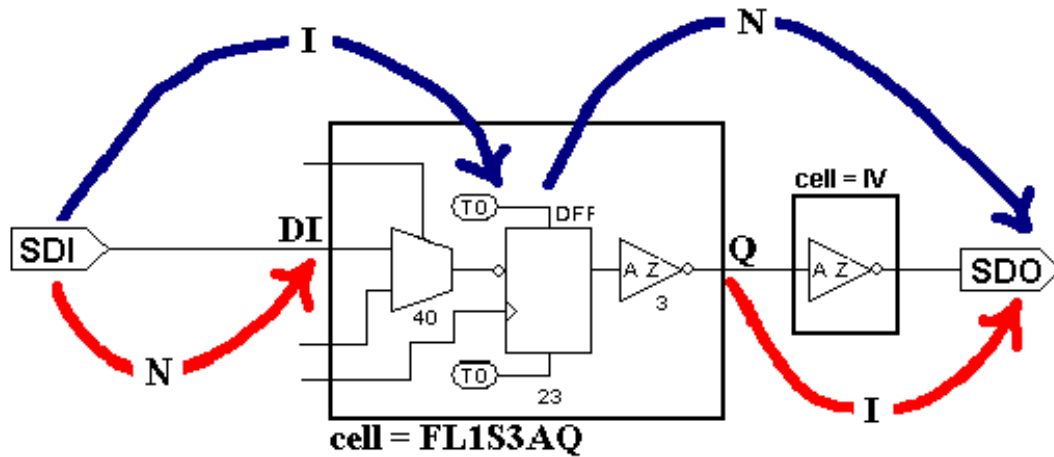
```

TEST> report_scan_cells -all -reverse -verbose > SCAN_CELLS.rpt
    
```

The inversion information is often important to understand if you intend to correctly relate the pattern data to simulated values from the pattern.

The two character inversion code of `IN` for the master cell indicates the inversion to the internal sequential modeling element which has been identified as the master by TestMAX ATPG. The first character is an `I` if there is an inversion from the scan chain input to the sequential device identified as the master and an `N` if there is no inversion. The second character conveys similar information about the inversion from the master's stored value to the scan chain output. The master is most often *inside* of the library cell the user normally sees so the cells pin inversion information is also necessary.

The single character `I` or `N` listed for the input pin conveys the inversion from the scan input port to the library cell's "DI" pin. In the previous example the `N` indicates there is no inversion. Likewise the single character `I` for the output pin "Q" indicates there is an inversion between the value seen on "Q" and the scan output port.



The previous diagram shows in simplified form how a relationship of I_N for the master cell and N and I for the cell's input and output pin affects how data needs to be interpreted. In this diagram, the SDI pin represents the top level design scan in port and SDO represents the top level design's scan out port. There is no inversion in the scan path between SDI port and the library cells "DI" input pin. However, within the library cell there is inversion both going into and coming out of the sequential element. An inverter exists between the library cell's output and the top level SDO scan output port, causing an additional data inversion to be considered.

```

XTB: Starting parallel simulation of 4 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=100.00 ns, V=2)
>>> Error during scan pattern 2 (detected during parallel unload of
pattern 1)
>>> At T=840.00 ns, V=9, exp=1, got=0, chain c0, pin SO, scan cell
10
>>> At T=840.00 ns, V=9, exp=0, got=1, chain c0, pin SO, scan cell
11
XTB: Simulation of 4 patterns completed with 2 errors (time: 1700.00 ns,
cycles: 17)
    
```

Since this is a scan chain unload mismatch the error message indicates expected value at the scan output was expected 1, but got 0. You need translate this into "expected 0", got "1" at the Q pin because of the inversion indicated by the "I" on the output pin in the report scan cells output.

If you find yourself in the middle of a simulator debug session you should keep in mind the four different possible inversion arches shown by the previous diagram. If you are investigating the stored value within the sequential model of the DFF in the simulator then the top-left arch tells you the relation between the scan in data and the data to be found in the state element. The top-right arch tells you whether that value is inverted by the time

it appears at the SDO output. The lower-left and lower-right arch are used when you are looking at the "DI and "Q" pins of the library cell in the simulator.

Isolating a Failing Pattern to Assist in Troubleshooting

Step #1: Get the patterns back in. If you have TestMAX ATPG up and running then no problem, you are done. If you don't you'll need to re-establish the same environment as was used to create the patterns. A typical sequence is:

```
read_libraries

read_design

run_build

add_clocks, PI constraints, PI equivs

run_drc <original.spf> # with original STIL procedure file
```

You can essentially rerun any command file you had with the exception that instead of:

```
add_faults -all

run_atpg -auto_compression
```

You'll do:

```
set_patterns -external <file_you_saved_patterns_in>
```

Not all of the pattern format that TestMAX ATPG can create can be read back in. Synopsys provides the Ltran tool as a way to produce TSTL2, FTDL, and TDL91 formats. If you are using one of those formats it is best to write a binary or STIL pattern file along with the creation of the FTDL/TSTL2/TDL91 patterns. Otherwise, you'll have nothing to read back in.

Use binary formats whenever possible to read patterns into TestMAX ATPG. Other pattern formats such as WGL and STIL have limited features to store all data about the patterns. For example, when you read a STIL or WGL pattern file back, a fast-sequential pattern might be interpreted as a full-sequential pattern.

Step #2: Now that the patterns have been read into TestMAX ATPG again you can write them out. For example, say that your failing pattern is pattern 412 and you want to write out that pattern plus one on either side in STIL format.

```
write_patterns pat412.v -format stil -first 411 -last 413
```

Your output file includes a test_setup procedure, if one was defined in our procedures file, along with patterns 411, 412, and 413.

Eliminating a Few Failing Patterns from a Larger Set

As an example, suppose you have 1000 patterns and want to eliminate patterns 103, 412, and 720 from this set.

Step #1: Get the patterns back in. If you have TestMAX ATPG up and running with original patterns then proceed to step #2. If you don't you'll need to reestablish the same environment as was used to create the patterns. You should essentially re-run any command file you had with the exception that instead of:

```
add_faults -all  
  
run_atpg # or some other variant
```

You'll do:

```
set_patterns -external <file_you_saved_patterns_in>
```

Step #2: Generate a list of patterns vs. pattern type:

```
report_patterns -external -all -type > reorder.dat
```

If your patterns were in the internal pattern buffer just substitute '-internal' for '-external' in the example command above.

Step #3: Edit the 'reorder.dat' file and delete or comment out the lines corresponding to patterns 103, 412, and 720 as well as the table headings in this report.

Step #4: Write out new pattern file using the `-reorder` option. The edited file causes the undesired patterns to be dropped as the patterns are written out.

```
write_patterns pat.wgl -external -format wgl -reorder reorder.dat
```

Locating the Target Fault Site for the Failing Pattern

Suppose you would like to know which fault sites are trying to be tested for the specific scan cell bit which shows a simulation mismatch. You can use the TestMAX ATPG diagnostic capability to help you figure this out.

Step #1: Translate the mismatch messages from the simulator into the failure data file format needed for the `run_diagnosis` command.

```
# 0.00 ns : Begin test setup  
# 900.00 ns : Begin patterns, first pattern = 0  
>>> Error during scan pattern 2 (detected during parallel unload of  
pattern 1)  
>>> At T=840.00 ns, V=9, exp=1, got=0, chain c0, pin S0, scan cell  
10  
5 c4 2 (exp=1, got=0) # pin SRC63, scan cell 2, T= 6595.00 ns
```

You can cut and paste the error text into a file. This is because testbench produces error messages in which the unwanted lines are commented out by the leading '#', and the remaining lines just happen to match the format needed for TestMAX ATPG diagnostics! As a minimum, our file would need this:

```
#pattern output bit
 5 c4 1
 5 c4 2
```

But we could just as easily inserted a more verbose group of lines:

```
# ERROR during scan pattern 5 (detected during load of pattern 6)
 5 c4 1 (exp=1, got=0) # pin SRC63, scan cell 1, T= 6495.00 ns
 5 c4 2 (exp=1, got=0) # pin SRC63, scan cell 2, T= 6595.00 ns
```

The first column is the failing pattern number. The second column is either the name of the scan output pin, or in the case of a design with multiple scan groups it is the name of the scan chain. The third column is the scan cell position. The remaining columns are not required and treated as comment text.

We'll save this text into a file called "failures.dat".

Step #2: Read in the original pattern file into TestMAX ATPG and use the `run_diagnosis` command:

```
set_patterns -external pat.bin # our original patterns
run_diagnosis failures.dat # our failure file data
```

The result of this operation is that TestMAX ATPG attempts to identify all of the possible locations of faults that would produce a failure on the tester on these two patterns at these two bit positions. These are also places we'll want to investigate in a simulation that fails.

```
TEST> run_diagnosis fail.dat
Diagnosis summary for failure file fail.dat
#failing_patterns=2, #defects=1, #unexplained_fails=0
-----
Fault candidates for defect 1: #failing_patterns_explained=1
Warning: Fault candidates cause passing patterns to fail.
-----
Explained pattern list:
 5
-----
val code pin_pathname (module_name)
-----
sa1 DS u2/U34/Z (NR3L)
sa1 -- u2/q1n_sig_reg_0/D1 (FL1S2AQ)
sa0 DS u2/q1n_sig_reg_0/SD (FL1S2AQ)
```

So this has identified three different physical but logically identical fault sites that are being tested by one of the failing patterns. This might be helpful information.

However, a word of caution -- this technique requires that all failures be provided starting from the first failure encountered. You can't just randomly pick a failure from your simulation data and present it to TestMAX ATPG via a failure file. You must present all of the reported failures in sequence up to and including the failure you are interested in.

Isolating a Fault List to Assist in Troubleshooting

Suppose you would like to know which faults are being tested by a particular ATPG pattern. How would you figure this out? Let's use an example in which a failure is occurring on pattern 46. Here are the steps necessary to create a fault list for those patterns.

```
set_patterns -external saved_pattern_file # reload our patterns
write_patterns pat_0_45.bin -external -format bin -last 45
write_patterns pat_46.bin -external -format bin -first 46 -last 46
```

You've just created two pattern files, one with the patterns up to but not including the pattern of interest, and the other with our pattern of interest (pattern = 46).

```
set_patterns -external pat_0_45.bin
add_faults -all # or restore custom fault list
run_atpg # to regrade faults
write_faults faults_left.dat -class an -class nd -uncollapse -rep
```

You've just created a fault list of all the faults NOT detected by the first block of patterns.

```
remove_faults -all
set_patterns -external pat_46.bin
read_faults faults_left.dat
run_atpg
write_fault faults_detect_46.dat -class dt -class pt -collapsed
```

Now you've got a fault list file that contains exactly the faults detected by pattern 46. We wrote the "collapsed" fault list but you could have also save the "uncollapsed" list which would include all primary faults and equivalent fault sites detected.

Interpreting the `report_patterns` Command

There are times when you are just unsure as to whether the patterns in the external format you are simulating match the patterns originally created by TestMAX ATPG. If you wish to compare the internal form of the patterns to the external form you are using the `report_patterns` command. This lists in the transcript the internal pattern generated without any translations to an external form. Don't forget that if the simulation message indicates that the failure occurred during the 'load' of pattern 6 that the ATPG pattern number to review is $6 - 1 = 5$. This is because the scan unload of pattern 5 occurs simultaneously with the scan load of pattern 6.

```
TEST> report_patterns 5 -chain c4
Pattern 5 (basic_scan)
Time 0: load c4 = 111
Time 1: force_all_pis = 0000101111 00101111
Time 2: measure_all_pos = 1011101110 00110101
Time 3: pulse clocks clk1 (0) clk2 (1) clk3 (2) clk4 (3)
Time 4: apply procedure master_observe (ID=0) 1 times
Time 5: unload c4 = 011
```

There are a couple of problems with using this information. The first is that your design probably has more than 18 input pins, 18 outputs, and a scan chain slightly longer than 3 bits. But we need to fit this onto a page and so had to pick a small example.

The second problem is that you need a reference for which bit in the "force_all_pis" and "measure_all_pos" is which. This can be determined by using the `report_primitives -pis -pios` command for the inputs, and the `report_primitives -pos -pios` command for the outputs. The data presented for the `force_all_pis/measure_all_pos` data corresponds left-to-right with the corresponding `report_primitives` command output from top to bottom.

The scan data is presented left-to-right in the order in which bits are shifted into the scan chain for "loads" (111), or shifted out for "unloads" (011).

The previous example shows a six-event sequence for a basic-scan pattern. Time 0 involves loading scan chain c4 and involves shifting of three bits into scan chain c4. Time 1 applies values to all top level inputs. Time 2 measures all top level outputs. Time 3 applies a pulse to four different clocks (which must have been declared as PI equivalent). Time 4 is used to apply a master observe procedure. Finally at Time 5 the expected data from the prior events is unloaded from chain c4. This again involves shifting three bits out of the scan chain.

Viewing Pattern Data in the Graphical Schematic Viewer

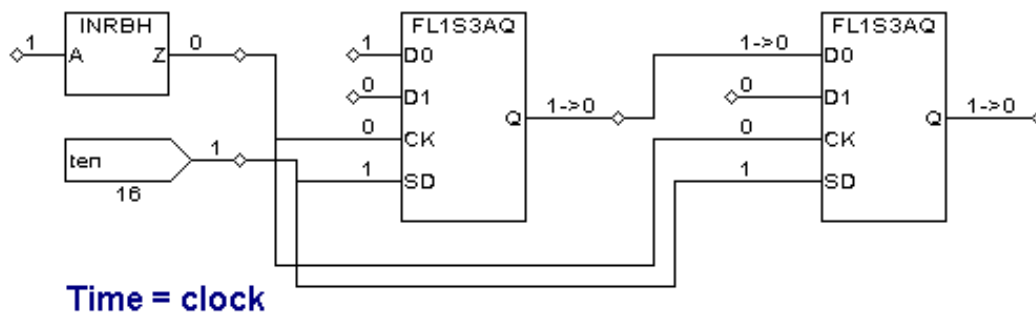
Well, the textual report of the pattern 5 data for chain c4 is pretty dry but fortunately TestMAX ATPG also supports a graphical view. To visually see the pattern data, we have two mechanisms that might be used:

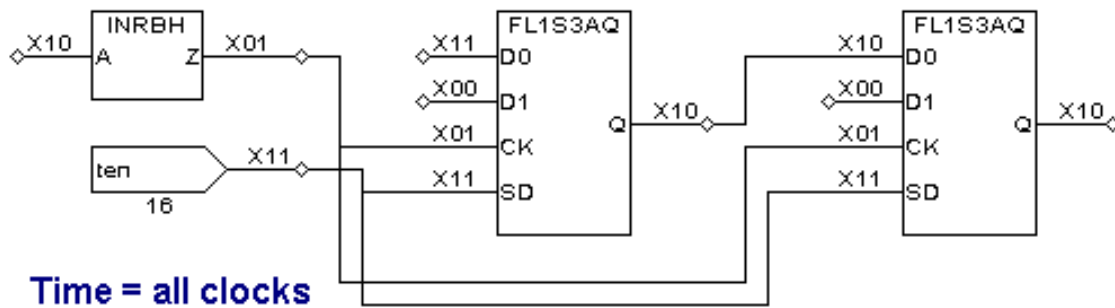
- selecting pindata of "good machine"
- using the `analyze_simulation_data` command

We recommend using `analyze_simulation_data` because it can display all forms of patterns: Basic-Scan, Fast-Sequential, or Full-Sequential. Selecting `pindata = good machine` works fine for Basic-Scan patterns, and, by default, shows the preclock and postclock simulation time.

To begin we should display the appropriate gates in the GSV window. Next we either issue the `analyze_simulation_data 5 -fast` command or use the SETUP button to select a pin data type of "good machine" with a pattern number of 5. The following example shows the good machine pindata for pattern 5 on chain c4, bit 1. This is instance = u1/q1_sig_reg_0 as well as bit 2, instance = u1/q1_sig_reg2_2

An important thing to remember about the "good machine" data is that there are five forms of the data display for the time in the cycle. There is the `time=clock` (default), `time=preclock`, `time=postclock`, `time= all`, and `time=LETE`. The various forms are discussed below.





For time=clock, the values shown on the schematic represent the simulation results with the capture clock active and the state elements at their previous states. This is the default display for Good Sim Data until you adjust it. Think of this as a simulation snapshot at the instant the clocks go active, but before any data-in to data-out changes of DFF's or DLAT's have occurred.

For time=preclock, the values shown represent the simulation results with the capture clock OFF and the state elements at their states from the last scan load. When the pattern is a Basic-Scan pattern, many gates/pins in the design do not have a calculated preclock value and the value is shown as either 'x' or '-' or '?'.

For time=postclock, the values shown represent the simulation results with the capture clock OFF and the state elements at their newly captured states.

For time=LETE, the values shown on the schematic represent the simulation results with the capture clock on and the leading edge and level sensitive state elements at their new value to be used by a trailing edge state element.

For time=all, the values shown on the schematic represent the pre-, active-, and post-clock times as three characters. A question mark "?" represents data not available. This form is probably the more natural representation of data for those familiar with logic simulators. The command to display this type of time value is `set_primitive_report -time all`. Do not trust the pre-clock time value. TestMAX ATPG does not simulate the pre-clock value for all gates in the design — only those gates where there is a need. Generally an "X" value is used as the pre-clock value for all other gates. For clocks and reset values a non-X value is used. This makes ATPG much faster but debugging somewhat harder and potentially more confusing.

Using the analyze_simulation_data Command

Perhaps a better presentation of the pattern data in the GSV is created using the `analyze_simulation_data` command. This command shows the pattern data for both Basic-Scan and Fast-Sequential patterns in a event sequence format that uses "#" for

scan loads, and separates each "capture_XXX" procedure with a dash "-". To try this new option do:

```
# add gates of interest to schematic view
analyze_simulation_data -fast_sequential 5
# where 5 is failing
set_pindata debug_sim_data
# click REFRESH to update schematic
```

The `analyze_simulation_data` command can also be used to read in a Value Change Dump (VCD) data file collected during Verilog simulation and to display the values graphically, or side-by-side with Fast-Sequential or Full-Sequential expected values.

WGL Pattern Generation Options

The following sections explain the various WGL pattern generation options:

- [Creating LSI-Compatible WGL Patterns](#)
- [Creating NEC-Compatible WGL Patterns](#)
- [WGL Scan Chain Padding](#)
- [WGL Scan Chain Definitions](#)
- [Macro Usage in WGL](#)
- [Grouping Bidirectional Port Data in WGL](#)
- [Controlling Port Data Order in WGL](#)
- [Specifying Windowed Measures in WGL](#)
- [Delayed Input Force Timing and Force Prior in WGL](#)
- [Balancing Vector and Scan Statements in WGL](#)
- [Mapping Bidirectional Ports Within Vector Statements in WGL](#)
- [Mapping Bidirectional Ports Within Scan Statements in WGL](#)
- [Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL](#)
- [Selecting Scan Chain Inversion Reference in WGL](#)
- [Effect of CELLDEFINE in WGL](#)
- [Ambiguity of the Master Cell in WGL](#)

Creating LSI-Compatible WGL Patterns

To produce LSI-compatible WGL output you need to use the [set_drc](#), [set_buses](#), [set_simulation](#), and [set_wgl](#) commands, as shown in the following example:

```
set_drc -nomulti_captures_per_load

set_buses -external_z x

set_simulation -xclock_gives_xout

set_rules c13 error

set_rules z4 error

set_wgl -nolast_scan

set_wgl -scan_map keep

set_wgl -pre_measured

set_wgl -inversion_reference master

set_wgl -chain_list shift

set_wgl -nomacro -nopad -nogroup_bidis

set_wgl -bidi_map { 0x 0- 1x 1- xx x- z0 -0 z1 -1 zx -x zz -z }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see "[Defining the Shift Procedure](#)."
- Scan Chain names defined in the STIL procedure file must not contain spaces or other white space. For example, use "chain_1" instead of "chain 1".
- You must define the end-of-cycle timing, as follows:
 1. The timing block must define the end-of-cycle measure. For more information, see "[Creating End-of-Cycle Measures in ATPG Patterns](#)."
 2. The load_unload procedure must use pre-measure scan outputs. For more information, see "[Defining the load_unload Procedure](#)."
- You can use the ReflectIO protocol. However, unless all bidirectional pins are fully controlled, you should avoid this protocol since it can create patterns which fail in simulation and might contain contention when all BIDI pins are not controlled.

For a design with bidirectional ports, the ReflectIO protocol causes each capture_XXX procedure to use the reflectIO style of syntax. For example, you can define all clocks and then issue the `set_drc -bidi_control_pin` command followed by a `write_drc` command to create a template STIL procedure file. Then modify the capture_XXX procedures to appear similar to the following 3-cycle protocol:

```
capture_CLK {
    W _default_WFT_;
    V { _pi=\r15 # ; _po=\j \r44 % ; } # force PI, TN=1
    V { TN=0; _io=\r32 Z ; _po=\j \r44 X ; } # disable bidis
    V { _io=\m \r32 % ; CLK=P; } # reflect bidis, pulse CLK
}
```

- All capture_XXX procedures for clocks must have the same number of tester cycles, V{...} constructs. If you use a three cycle capture for 'CLK', then you must also use a three-cycle capture for 'RST', 'CLK2', and so forth. This includes the non-clocking capture procedure named `capture`.
- Use a [test_setup procedure](#) to initialize all input pins to a known value in the first test cycle. Initialize bidirectional pins to Z.
- If inputs are applied with a delay on the tester, then the Timing block of the STIL DRC procedure file should include a "ForcePrior" or "P" character at time offset zero of each cycle before applying the required value within that cycle. This generates a V6 warning during DRC which will have to be ignored. There is an example of ForcePrior at the end of topic: [Controlling Pin Timing in STIL](#)
- You can use only one timing block.
- Use the `-order_pins` option of the `write_patterns` command when writing WGL patterns.
- Do not use the `-measure_forced_bidis` option of the `write_patterns` command when writing WGL patterns
- Contact LSI for the latest advice and application notes concerning the use of TestMAX ATPG.

See Also

[set_wgl](#)

[set_drc](#)

[set_simulation](#)

[set_contention](#)

[write_drc_file](#)

[write_patterns](#)

[End-of-Cycle Measures and Load_Unload](#)

[End-of-Cycle Measures and Timing](#)

[End-of-Cycle Measures and Capture Procedures](#)

Creating NEC-Compatible WGL Patterns

To produce NEC-compatible WGL output, you need to use both the [set_simulation](#) and [set_wgl](#) commands, as shown in the following example:

```
set_simulation -strong_bidi_fill

set_wgl -nomacro

set_wgl -nopad

set_wgl -notester_ready

set_wgl -inversion_reference master

set_wgl -scan_map dash

set_wgl -bidi_map { 0x 0- 1x 1- xx x- z0 -0 z1 -1 zx -x zz -z -x -- z-
-- }
```

Note the following:

- Scan shifts must use a single tester cycle. For more information, see "[Defining the Shift Procedure.](#)"
- You must define the end-of-cycle timing, as follows:
 1. The timing block must define the end-of-cycle measure. For more information, see "[Creating End-of-Cycle Measures in ATPG Patterns.](#)"
 2. The load_unload procedure must use pre-measure scan outputs. For more information, see "[Defining the load_unload Procedure.](#)"
 3. The clock capture procedures must use the two-cycle end-of-cycle measure format. For more information, see "[Defining Capture Procedures in STIL.](#)"
- You must explicitly initialize bidirectional ports to non-Z values in the load_unload procedure.
- Use the test_setup procedure to eliminate uninitialized ports at T=0. For more information, see "[Defining the test_setup Procedure.](#)"

- Use the `test_setup` procedure to eliminate floating ports at T=0.
- Do not use the `-measure_forced_bidis` option of the `write_patterns` command when writing WGL patterns.
- Use the WGL to ALB to Verilog translation path. Other paths, such as WGL to ALB to CPT, have not been validated to work.

See Also

[set_wgl](#)

[set_simulation](#)

[set_contention](#)

[write_patterns](#)

[End-of-Cycle Measures and Load_Unload](#)

[End-of-Cycle Measures and Timing](#)

[End-of-Cycle Measures and Capture Procedures](#)

WGL Scan Chain Padding

When a design has more than one scan chain and the scan chains are not all the same length then you have the option of causing the WGL patterns to be written so that all scan load and unload data is the same length (`set_wgl -pad`) or is only the length of the scan chain (`set_wgl -nopad`). The default is not to pad, and this is preferred by most vendors.

When padding is enabled, the pad value can be any one of 0, 1, or X and you select which by the `-pad_character` option of the `write_patterns` command when the WGL patterns are written. The default when padding is enabled, is to pad with a zero. *Note*, however, that when padding is enabled and a particular pad character is chosen that this will have no effect on the padding used for the chain test patterns. The padding for chain test patterns is always the continuation of the repeating string 0011.

The first example shows a portion of the WGL SCANSTATE block for a design with three scan chains of length 2, 3, and 8 bits where padding is disabled.

```
# scan chain padding disabled
scanstate
  c1L0 := c1G(11);
  c2L1 := c2G(011);
  c3L2 := c3G(00110011);
  c1E3 := c1G(00);
  c2E4 := c2G(100);
  c3E5 := c3G(11001100);
```

The second example shows the same data with scan chain padding enabled and a pad character of X used so that it is easier to see where the padding occurs. For scan load strings the padding occurs on the left (first shifted in) for all shorter chains. For scan unload strings the padding occurs on the right (last shifted out).

```
# scan chain padding enabled with pad = X
scanstate
  c1L0 := c1G(XXXXXX11);
  c2L1 := c2G(XXXXX011);
  c3L2 := c3G(00110011);
  c1E3 := c1G(00XXXXXX);
  c2E4 := c2G(100XXXXXX);
  c3E5 := c3G(11001100);
```

See Also

[set_wgl](#)

[set_buses](#)

WGL Scan Chain Definitions

By convention, the `scanchain` block in WGL defines the instances in the physical sequence of each scan chain, starting at the scan input, and traversing to the scan output. The number of instances in the scan chain matches the number of bits called for in the `scanstate` block for loading or observing from the scan chain.

On some designs, generally those with JTAG used during ATPG, the final scan chain shift is done outside of the scan loop. This translates into the "scan()" vector being shortened by one bit and an additional vector() or more being added to the procedure to handle the final shift outside of the scan statement. Now most WGL translators require that the number of bits defined in the `scanchain` block match the physical length of the scan chain. However, a few require that the number of bits match the length of data to be loaded by the "scan()" statements. The `-chain_list` option controls how the scan chain is listed in the `scanchain` block. The default is `all` which causes all instances in the scan chain to be included in the defining list. Optionally specifying `shift` causes the list to match only those bits loaded by the "scan()" statements.

The first examples shows the default `scanchain` block for a design with two scan chains of 5 and 4 bits.

```
# set_wgl -chain_list all
scanchain
  chain1 ["si1", "A4", !, "A3", "A2", "A1", "A0", "so1" ];
  chain2 ["si2", "B3", "B2", "B1", "B0", !, "so2" ];
end
```

The second example shows the same scanchain block when the final shift of the scan chain is done outside of the Shift procedure and a selection of -chain_list shift is used. The final instance in each scan chain "A1", and "B1" have been omitted from the scan chain definitions.

```
# set_wgl -chain_list shift
scanchain
  chain1 ["si1", "A4", !, "A3", "A2", "A1", "so1" ];
  chain2 ["si2", "B3", "B2", "B1", !, "so2" ];
end
```

See Also

[set_wgl](#)

[set_buses](#)

Macro Usage in WGL

WGL supports the definition of macros. Macros can be used to represent commonly repeated sequences and the use of macros can lead to more compact WGL pattern files. TestMAX ATPG will write WGL using macros if the [set_wgl](#) -macro option has been used. Most vendors do not support macros as this requires a more complex WGL reader and so the TestMAX ATPG default is not to use macros.

When macros are enabled, TestMAX ATPG adds various macro definitions to the WGL pattern file. The following example is a macro for a capture procedure for the port CLK. There will generally be a macro for each procedure in the DRC file.

```
# an example macro definition
macro capture_CLK (SDI3_I, SDO1_I, D0_I, D2_I, CLK, RSTB, SDI,
  INC, SCAN_9, SDI3_O, SDO1_O, D0_O, D2_O, P, SDO, CO)
  vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
    @INC @SCAN_9 X X X X XX XX X ];
  vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I @CLK @RSTB @SDI
    @INC @SCAN_9 @SDI3_O @SDO1_O @D0_O @D2_O @P
    @SDO @CO ];
  vector(tp1) := [ @SDI3_I @SDO1_I @D0_I @D2_I 1 @RSTB @SDI
    @INC @SCAN_9 X X X X XX XX X ];
endmacro
```

The first following example shows a segment from a WGL PATTERN block which does not use macros and the second example is the same information using macros.

```
# example patterns without macros
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
  "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
  "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
```

```

vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 X X X X X X X X X ];
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];

{ scan_test }
{ pattern 0 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
output [c1:c1U0], output [c2:c2U1], output [c3:c3U2],
input [c1:c1L0], input [c2:c2L1], input [c3:c3L2];
{ capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 1 X X X X X X X X X ];
vector(tp1) := [ - Z - - 0 0 0 0 0 1 Z 0 Z Z Z 0 1 0 0 ];

{ pattern 1 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
output [c1:c1U3], output [c2:c2U4], output [c3:c3U5],
input [c1:c1L3], input
[c2:c2L4], input [c3:c3L5];
{ capture_CLK }
vector(tp1) := [ Z Z 0 Z 0 1 1 1 0 0 X X X X X X X X X ];
vector(tp1) := [ - - 0 - 0 1 1 1 0 0 Z Z X Z Z 0 1 0 1 ];
vector(tp1) := [ Z Z 0 Z 1 1 1 1 0 0 X X X X X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ X X X X 0 1 X X 0 0 X X X X X X X X X ];
vector(tp1) := [ X Z X X 0 1 X X 0 1 X X X X X X X X X ];
scan(tp1) := [ - - X X 1 1 - - 0 1 - - X X X X - - X ],
output [c1:c1U6], output [c2:c2U7], output [c3:c3U8],
input [c1:c1L6], input
[c2:c2L7], input [c3:c3L8];
capture_RSTB }
vector(tp1) := [ Z Z Z Z 0 1 1 1 1 1 X X X X X X X X X ];
vector(tp1) := [ - Z Z Z 0 0 1 1 1 1 Z 0 1 0 Z 0 0 0 0 ];

# example patterns using macros
pattern_group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I, "CLK",
"RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O, "SDO1":O,
"D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
test_setup

{ scan_test }
{ pattern 0 }
load_unload(c1U0, c2U1, c3U2, c1L0, c2L1, c3L2)
capture_RSTB(-, Z, -, -, 0, 1, 00, 0, 1, Z, 0, Z, Z, Z0, 10, 0)

```



```

{ pattern 1 }
load_unload(c1U3, c2U4, c3U5, c1L3, c2L4, c3L5)
capture_CLK(-, -, 0, -, 0, 1, 11, 0, 0, Z, Z, X, Z, Z0, 10, 1)

{ pattern 2 }
load_unload(c1U6, c2U7, c3U8, c1L6, c2L7, c3L8)
capture_RSTB(-, Z, Z, Z, 0, 1, 11, 1, 1, Z, 0, 1, 0, Z0, 00, 0)

```

See Also

[set_wgl](#)

[set_buses](#)

Grouping Bidirectional Port Data in WGL

In WGL patterns a bidirectional port appears as two characters, one for the force input value and another for the measure output value. These two characters can appear side by side (grouped), or in independent locations within the data (split columns). The `set_wgl -group_bidis` command causes the two characters to appear as a single column of two characters, with the first representing the input action and the second representing the output action. The default is to present the bidirectional port data as two separate columns.

The first following example uses grouped bidis and in this example there are four bidirectional ports which appear as the first four columns of each `vector()` statement. The characters "ZX" indicate a force of Z (no force) and a measure of X (mask measure).

```

# example patterns using grouped bidis
pattern group_ALL ("SDI3", "SDO1", "D0", "D2", "CLK",
  "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "P[0]",
  "P[1]", "SDO[2]", "SDO[3]", "CO")
{ test_setup }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 0 0 0 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 0 0 0 X X X X X ];

```

In the second following example split bidis are used. Notice that the pattern data no longer has any two character columns. The port order list now lists each bidirectional port twice and follows each by either `:I` or `:O` to indicate direction. The two parts of the bidirectional port data do not appear as adjacent data in the vector, they can appear at any position.

```

#example patterns using split bidis
pattern group_ALL ("SDI3":I, "SDO1":I, "D0":I, "D2":I,
  "CLK", "RSTB", "SDI[1]", "SDI[2]", "INC", "SCAN", "SDI3":O,
  "SDO1":O, "D0":O, "D2":O, "P[0]", "P[1]", "SDO[2]", "SDO[3]",
  "CO")
{ test_setup }
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 X X X X X X X X X ];

```

```
vector(tp1) := [ Z Z Z Z 0 0 0 0 0 0 0 X X X X X X X X X ];  
vector(tp1) := [ Z Z Z Z 0 1 0 0 0 0 0 X X X X X X X X X ];
```

See Also

[set_wgl](#)

[set_buses](#)

Controlling Port Data Order in WGL

The default pin data order of the WGL pattern data follows the order in which the ports are defined in the design's top module. By changing the order of the ports in the top module you can affect the order of the WGL data.

There is also the `-order_pins` option of the `write_patterns` command. Use of this option causes the ports to occur in the order: inputs, bidis, and outputs. Within each grouping the port data order matches the order the ports are defined in the design's top module.

For a top-level design with port order:

```
module TOP (I1,B1,O1,O2,O4,O3,B3,B2,I3,I2);
```

the following two examples illustrate the difference in data order.

```
# default port order using grouped bidis  
pattern group_ALL ("I1", "B1", "O1", "O2", "O4", "O3", "B3",  
"B2", "I3", "I2")  
{ test_setup }  
vector(tp1) := [ 0 ZX X X X X ZX ZX 0 0 ];  
vector(tp1) := [ 0 ZX 1 1 1 1 ZX ZX 0 0 ];  
vector(tp1) := [ 1 0X 1 1 1 1 0X 0X 1 1 ];
```

```
# port order using ORDER_PINS option  
pattern group_ALL ("I1", "I3", "I2", "B1", "B3", "B2", "O1",  
"O2", "O4", "O3")  
{ test_setup }  
vector(tp1) := [ 0 0 0 ZX ZX ZX X X X X ];  
vector(tp1) := [ 0 0 0 ZX ZX ZX 1 1 1 1 ];  
vector(tp1) := [ 1 1 1 0X 0X 0X 1 1 1 1 ];
```

See Also

[set_wgl](#)

[set_buses](#)

Specifying Windowed Measures in WGL

The default WGL patterns written will define timing which performs a strobed measure (single time measure) when outputs are to be measured. If your tester supports window measure (measure over a continuous range of time) and you would like to have a windowed measure, this type of measure can be created. This time you do not use any `set_wgl` options, but instead make edits to the `Timing` block of the DRC procedure file. Note that these edits must be made before performing the `run_drc` command and before generating ATPG patterns.

The following example illustrates a window measure for the symbolic group `out_ports` defined elsewhere in the DRC file. The STIL language specifies that the uppercase {H,L,T,X} characters indicate a strobed measure, and the lowercase characters {h,l,t,x} call for a window measure. In this specific example the ports associated with the symbolic group `out_ports` is continuously measured for high/low/tristate values between an offset of 450 nS and 490 nS from the beginning of the tester cycle. The `'490ns' x;` text specifies the window measure is turned off at this time and is text which is not needed for a strobed measure.

```
Timing {
  WaveformTable "WINDOW_COMPARE" {
    Period '1000ns';
    Waveforms {
      clocks { P { '0ns' D; '500ns' U; '600ns' D; } }
      input_ports { 01Z { '0ns' D/U/Z; } }
      out_ports { X { '0ns' X; } }
      out_ports { HLT { '0ns' X; '450ns' h/l/t; '490ns' X; } }
      bidi_ports { X { '0ns' X; } }
      bidi_ports { 01Z { '0ns' D/U/Z; } }
      bidi_ports { HLT { '0ns' X; '450ns' H/L/T; } }
    }
  }
}
```

See Also

[set_wgl](#)

[set_buses](#)

Delayed Input Force Timing and Force Prior in WGL

It is a common requirement when running the pattern timing to require that one or more pins have their inputs applied at some delayed offset from the beginning of the tester cycle. This is another adjustment that is made in the `Timing` block of the DRC file rather than with a `set_wgl` command. In the following example the symbolic pin group `input_grp2` has its pattern data applied at an offset of 5ns into the tester cycle.

What is the value on the pins of group `input_grp2` from the start of the cycle to offset 5ns? The answer is that the value is undefined unless you specify some value in the timing block such as 0, 1, X, or perhaps Z. What if you just want the port to continue the value from the previous tester cycle? In WGL as well as STIL there is a "Force Prior" concept which indicates the value is to be whatever was previously assigned.

To cause the WGL output to call for a Force Prior, edit the `Timingblock` of the DRC file before performing a `run_drc` command and before generating any ATPG patterns and add the "P" character to the beginning of the timing definition for those inputs which are applied after a delay. Note that this use of the "P" waveform character will produce a V6 warning which you can ignore. In the following example, the symbolic pin group `input_grp2` calls for the Force Prior value.

```
WaveformTable "FORCE_PRIOR_EXAMPLE" {
  Period '1000ns';
  Waveforms {
    CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
    CLOCK { 01ZN { '0ns' D/U/Z/X; } }
    RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
    RESETB { 01ZN { '0ns' D/U/Z/X; } }
    input_grp1 { 01ZN { '0ns' D/U/Z/X; } }
    input_grp2 { 0 { '0ns' P; '5ns' D; } }
    input_grp2 { 1 { '0ns' P; '5ns' U; } }
    input_grp2 { Z { '0ns' P; '5ns' Z; } }
    out_ports { HLTX { '0ns' X; '490ns' H/L/T/X; } }
    bidi_ports { 01ZN { '0ns' Z; '20ns' D/U/Z/X; } }
    bidi_ports { X { '0ns' X; } }
    bidi_ports { HLT { '0ns' X; '490ns' H/L/T; } }
  }
} # end FORCE_PRIOR_EXAMPLE
```

See Also

[set_wgl](#)

[set_buses](#)

Balancing Vector and Scan Statements in WGL

By default, the last event in the WGL pattern file is a scan chain unload to observe the measure values of the final capture clock. This corresponds to a `scan()` statement in the WGL file. Some vendors require that the final event in the WGL pattern file be a `vector()` statement to ensure that clocks are off and to provide a symmetric order where the scan statements are always followed by an identical number of vector statements. You can cause the final events in the WGL file to be vector statements by using the `set_wgl -nolast_scan` option to change the default behavior.

The first following example shows the default final pattern where the last event is a scan() statement. The second example shows the effect of using `-nolast_scan`.

```
#example made with -last_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture }
vector(tp1) := [ -Z -0 -0 -1 0 1 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
end
```

```
#example made with -nolast_scan
{ pattern 26 }
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U78], output [c2:c2U79], output [c3:c3U80],
input [c1:c1L78], input [c2:c2L79], input [c3:c3L80];
{ capture_CLK }
vector(tp1) := [ -Z -0 -0 -1 0 1 1 1 1 1 Z 1 0 0 0 ];
{ load_unload }
vector(tp1) := [ X- X- X- X- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ X- -- X- X- 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],
output [c1:c1U81], output [c2:c2U82], output [c3:c3U83],
input [c1:c1L81], input [c2:c2L82], input [c3:c3L83];
{ nocapture }
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X -X 0 1 X X 0 0 X X X X X ];
end
```

See Also

[set_wgl](#)

[set_buses](#)

Mapping Bidirectional Ports Within Vector Statements in WGL

You've seen an example earlier of how TestMAX ATPG supports creating WGL patterns with bidirectional port data represented as either a single column of two characters (grouped) or as two columns of single characters (non-grouped or split). In addition to this choice in grouping there is also the ability to change or map the characters used. Not every vendor agrees on what the WGL character representation should be for bidirectional port data so TestMAX ATPG has been designed to provide flexibility by use of the `set_wgl -bidi_map` option.

The syntax for this option is: `set_wgl -bidi_map <from> <to>`

There are 9 mappings that can be adjusted: 3 for which the bidirectional port is an input, 4 for which the bidirectional port is an output, and 2 for when the bidirectional port is a scan input or scan output. This argument can be repeated on the same command line or across multiple commands to specify more than one mapping. If the same from designator is repeated then the later one will replace the earlier ones.

The from designator is a two-character string that represents the TestMAX ATPG internal data. The to designator is a two-character string that specifies the characters which will appear in the WGL pattern output in place of this internal representation.

Definition of TestMAX ATPG Internal Representation = "from"

```
from
====
0x : force 0, no measure
1x : force 1, no measure
xx : force unknown, no measure

z0 : no force, measure 0
z1 : no force, measure 1
zx : no force, no measure
zz : no force, measure Z

-x : bidi is in scan input mode
-z : bidi is in scan output mode
```

The preceding table defines all the legal combinations available for the `from` portion of the mapping option. Any other combination is illegal. The `to` designator is also made up of characters `0/1/x/z/-` but the mapping is checked to ensure that you are not destroying the intent of the data or masking measures that would affect the test coverage reported. As an example of a mapping the following table represents a commonly requested map in which one of the bidirectional characters is always a dash:

A common mapping

```
from : to
==== : ==
```

```
0x : 0- # force 0, no measure
1x : 1- # force 1, no measure
xx : x- # force unknown, no measure
z0 : -0 # force Z, measure 0
z1 : -1 # force Z, measure 1
zx : -x # force Z, no measure
zz : -z # force Z, measure Z
-x  : -- # bidi is a scan input
z-  : -- # bidi is a scan output
```

With the exception of the {zz, -z} mapping above, this table represents the default mapping.

The `set_wgl` command which would implement the previous table is:

```
BUILD> set_wgl -bidi_map { 0x 0- 1x 1- xx x- z0 -0 \
                          z1 -1  zx -x  zz -z  x --  z- -- }
```

Note in the previous example that you can specify the `-bidi_map` option only one time, and the parameters must be in a list structure. Alternatively, you can repeat the entire command line for each entry, as shown in the following example:

```
set_wgl -bidi_map {0x 0-}
set_wgl -bidi_map {1x 1-}
set_wgl -bidi_map {xx x- }
set_wgl -bidi_map {z0 -0 }
set_wgl -bidi_map {z1 -1}
set_wgl -bidi_map {zx -x}
set_wgl -bidi_map {zz -z}
set_wgl -bidi_map {x --}
set_wgl -bidi_map {z- --}
```

Note: Not all mappings are allowed. For example, you cannot map the dash for scan input or scan output to any other character. Also, you can map "zz" to "-z", but you cannot map "zz" to "z-". because of the loss of measure and to unambiguously read back in the WGL which is written out. The "zz"->"z-" mapping still indicates a measure must be performed but a "zz"->"z-" mapping could be confused with a "zx"->"z-" mapping which generally is interpreted to mean there is no force and no measure.

Note: The ability to use some bidi mappings is affected by whether the tester can measure Z values or not. If the tester can measure Z values then the default setting of `set_buses -external_z Z` should be used and the WGL patterns can contain both ZZ and ZX data (no force, measure Z and no force, no measure). If the tester cannot measure Z values or you want to generate patterns for which no Z-measure is needed you would set the `set_buses -external_z X` option before generating patterns. This would result in WGL patterns with "ZX" data for bidirectional pins but no "ZZ". If "ZZ" does not appear in the WGL you can define a bidi map of "ZX"->"Z-" or "ZX"->"z-" which you could not do if the Z measure were enabled and "ZZ" were possibly present.

Note: Most vendors do not support a simultaneous force and measure on the same port in the same cycle. With that in mind you should not use the `-measure_forced_bidis` option of the `write_patterns` command as this allows for a simultaneous force and measure whenever possible.

To report the current bidirectional map settings use the `report_settings wgl` command. The output is similar to the following example and the mapping will appear as a series of (from,to) settings.

```
wgl = macro_usage=off, nopad=on, scan_map=dash
      group_bidis=off, inversion_reference=master, tester_ready=on
      bidi_map=(Z0,-0) (Z1,-1) (0X,0-) (1X,1-) (XX,X-) (ZX,-X) (ZZ,-Z) (Z-,--)
```

As an example of how the `vector()` statement data changes for bidirectional ports the first following example shows some pattern data with four bidirectional pins (grouped as single column of two characters each) where the mapping is identical to the TestMAX ATPG internal representation. The second example uses a common mapping in which the bidirectional character pair always has one character represented as a dash.

An example where the mapping matches TestMAX ATPG internal representation.

```
{ pattern 1 }
{ load_unload }
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ 0X 1X XX ZX 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ 0X 1X XX ZX 1 1 - X 0 1 X X X - X ],
output [c1:c1U0], input [c1:c1L1]];
{ capture_CLK }
vector(tp1) := [ ZX ZX ZX ZX 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ Z0 Z1 ZX ZZ 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ ZX ZX ZX ZX 1 1 0 1 0 1 X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ ZX ZX ZX 0X 0 1 X X 0 1 X X X X X ];
scan(tp1) := [ ZX ZX ZX 0X 1 1 - X 0 1 X X X - X ],
output [c1:c1U1], input [c1:c1L2]];
{ capture_CLK }
vector(tp1) := [ 0X 0X 0X 0X 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0X 1X Z0 ZZ 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0X 1X ZX ZX 1 1 1 1 1 0 X X X X X ];
```

The same patterns after defining a mapping of:

```
(0x,0-) (1x,1-), (xx,x-), (z0,-0), (z1,-1), (zx,-x), (zz,-z)
```

```
{ pattern 1 }
{ load_unload }
```



```

vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ 0- 1- X- Z- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ 0- 1- X- Z- 1 1 - X 0 1 X X X - X ],
  output [c1:c1U0], input [c1:c1L1]];
{ capture_CLK }
vector(tp1) := [ -X -X -X -X 0 1 0 1 0 1 X X X X X ];
vector(tp1) := [ -0 -1 -X -Z 0 1 0 1 0 1 Z 0 0 1 0 ];
vector(tp1) := [ -X -X -X -X 1 1 0 1 0 1 X X X X X ];

{ pattern 2 }
{ load_unload }
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 0 X X X X X ];
vector(tp1) := [ -X -X -X 0- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -X -X -X 0- 1 1 - X 0 1 X X X - X ],
  output [c1:c1U1], input [c1:c1L2]];
{ capture_CLK }
vector(tp1) := [ 0- 0- 0- 0- 0 1 1 1 1 0 X X X X X ];
vector(tp1) := [ 0- 1- -0 -Z 0 1 1 1 1 0 Z 0 1 0 0 ];
vector(tp1) := [ 0- 1- -X -X 1 1 1 1 1 0 X X X X X ];

```

See Also

[set_wgl](#)

[set_buses](#)

Mapping Bidirectional Ports Within Scan Statements in WGL

The vector() statements in WGL correspond to the application of tester cycles. The scan() statements correspond to the serial loading and unloading of scan chains. The various vendor rules for character mapping of the vector() statements cannot be the same as for the scan() statement and so TestMAX ATPG supports the `set_wgl -scan_map` option to allow somewhat independent control of characters in the scan() statement. The available choices for scan mapping are: dash, bidi, keep, and none. The default is dash.

The following examples show some of the variations of `-scan_map`. The patterns are for a design with three scan chains and the first bidirectional port is a scan input and the second bidirectional port is a scan output.

For a setting of `dash`, every scan input and output position in the scan() statement contains a dash, and all bidirectional ports acting as a scan input or output contain a double dash.

```

# set_wgl -scan_map dash

vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];
  scan(tp1) := [ -- -- X- X- 1 1 - - 0 1 X X - - X ],

```

For a setting of `bidi`, every scan input and output position in the `scan()` statement contains a dash, and any bidirectional port acting as a scan input or output follows the mapping defined by the `-bidi_map` options. For the following example, assume a BIDI mapping of `(-x,-)` for scan inputs, and `(z-,z-)` for scan outputs.

```
# set_wgl -scan_map bidi -bidi_map {-x --} -bidi_map {z- z- }  
  
vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];  
scan(tp1) := [ -- Z- X- X- 1 1 - - 0 1 X X - - X ],
```

For a setting of `keep`, every scan input and output position in the `scan()` statement keeps the same characters as from the previous `vector()` statement in the `load_unload` procedure, including any scan inputs or outputs on bidirectional ports. It is important that the `load_unload` procedure have at least one `vector()` statement before the Shift procedure in order for a selection of `keep` to work properly.

```
# set_wgl -scan_map keep  
  
vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];  
scan(tp1) := [ 0- -X X- X- 1 1 X X 0 1 X X X X X ],
```

For a setting of `none`, every scan input and output position in the `scan()` statement contains a dash, and any bidirectional port acting as a scan input or output uses the TestMAX ATPG internal representation of `"-X"` for input and `"Z-"` for output.

```
# set_wgl -scan_map none  
  
vector(tp1) := [ 0- -X X- X- 0 1 X X 0 1 X X X X X ];  
scan(tp1) := [ -X Z- X- X- 1 1 - - 0 1 X X - - X ],
```

See Also

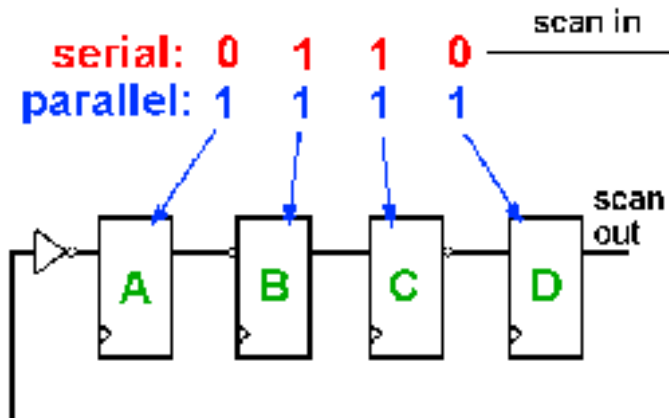
[set_wgl](#)

[set_buses](#)

Adjusting Pattern Data for Serial Versus Parallel Interpretation in WGL

The scan load data in the WGL patterns can be represented in two different ways, depending upon the reference point required by your WGL pattern translation tool. The `set_wgl -tester_ready` setting selects a data format that is ready to serially shift into the device without further processing for scan cell inversions. The `-set_wgl -notester_ready` option selects a data format that is ready to parallel load directly into the scan cells without further processing for inversions.

In the following figure, if you desire to have all devices A,B,C, and D loaded with 1's after a scan load, and your WGL translation application expects the data in parallel (-notester_ready) format, then the WGL scan data must be written as all 1's. However, if your WGL translation application expects the data in serial format (-tester_ready), then the WGL scan data must be adjusted for internal inversions that it passes through before being shifted into place. As you can see, the data is not the same: "1111" vs. "0110". So it is very important to know which data format your WGL translation application is expecting. The parallel format is the more popular, so if you do not know you should try the -notester_ready option first.



Note that both the serial and parallel load formats are sensitive to the referencing scheme for determining inversion if the final WGL translator is doing a parallel-form to serial form translation or a serial-form to parallel-form translation.

One additional variant of WGL output is needed if the WGL is to be interpreted for a parallel simulation and the end-of-cycle protocol is used. This end of cycle protocol results in a scan output pre-measure before beginning the "scan()" statement for the balance of the scan load/unload. The expected scan output vector needs to be shifted by the single bit of the pre-measure. To accomplish this, use the -pre_measured option instead of the -notester_ready option.

See Also

[set_wgl](#)

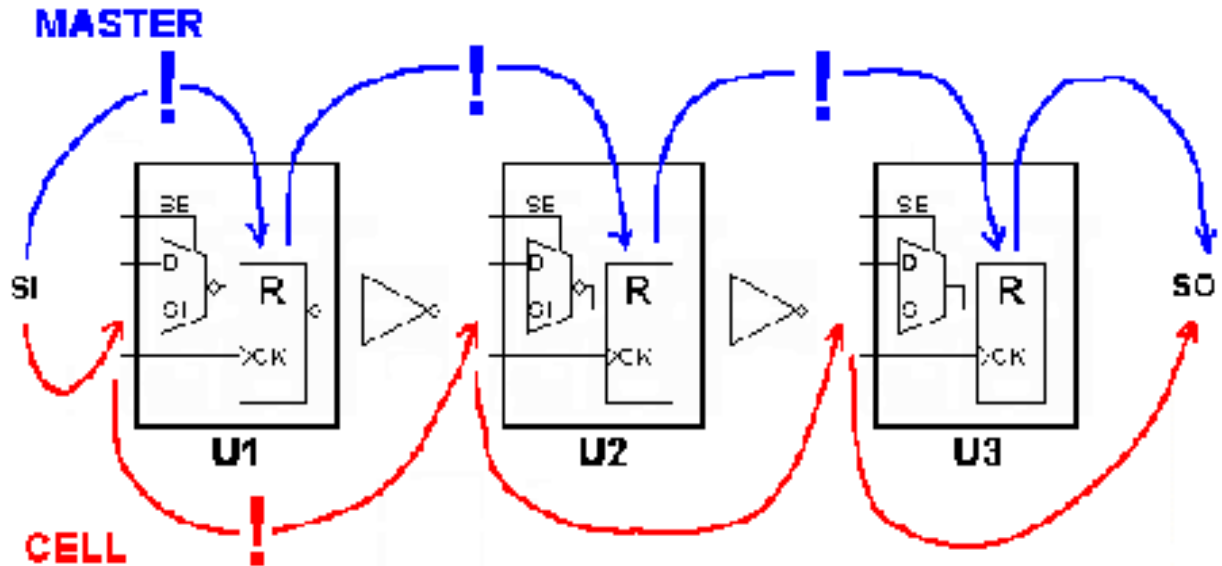
[set_buses](#)

Selecting Scan Chain Inversion Reference in WGL

The `scanchain` block of the WGL pattern file defines each scan chain in physical order from input port to output port. When an inversion exists between positions in the scan order, and exclamation mark "!" is inserted to indicate an inversion of the data has

occurred between the two positions. This inversion information is crucial for the correct translation of the scan chain load and unload data by the WGL-to-simulator or WGL-to-tester tools supported by your vendor.

More than one interpretation of the reference scheme for calculating inversions exists and so TestMAX ATPG offers options of master, cell, and omit for the `set_wgl -inversion_reference` command.



The previous diagram can provide some insight into the two different referencing schemes for inversion markers. When TestMAX ATPG calculates the inversion markers for a setting of `master` the reference point begins at the scan input pin, and then looks at whether the data is inverted from that point to the actual sequential simulation primitive functioning as the "master cell" where the value is stored. This is often a Verilog UDP level underneath the vendor's library cell. For a library cell with only one sequential element there is only one answer but for a library cell with two or more sequential elements, the answer might be ambiguous. As shown in the diagram, for an inversion reference of `master` there are inversions between the scan input and U1, between U1 and U2, and between U2 and U3. The corresponding WGL scanchain definition is shown in the following example.

```
# set_wgl -inversion_reference master

scanchain
  cl ["si", !, "U1/R", !, "U2/R", !, "U3/R", "so" ];
end
```

When TestMAX ATPG calculates the inversion markers for a setting of `cell` it begins at the scan in pin and then determines whether an inversion of the data occurs relative to the scan input pin of each library cell. This reference is used by some WGL translators in forming the FORCE/RELEASE statements needed for a parallel Verilog simulation.

The location of the inversion markers is unambiguous and not affected by which cell is classified as the "master" cell by TestMAX ATPG during DRC. Using an inversion reference of cell and the preceding diagram, there is an inversion only between the scan input of cell U1 and the scan input of U2. The corresponding WGL scanchain definition is shown in the following example:

```
# set_wgl -inversion_reference cell

scanchain
  cl ["si", "U1/R", "!", "U2/R", "U3/R", "so" ];
end
```

Sometimes, no matter which inversion reference you select the external WGL translator seems to come up with patterns that mismatch in simulation. If the simulation environment serially processes scan load information, then there is one more inversion reference that might be of use and that is the `omit` option. This option leaves out all inversion markers. By combining both the `-inversion_reference omit` and `-tester_ready` options, TestMAX ATPG produces scan load/unload data that is preprocessed for inversions and is ready to shift into the device unchanged, and omits the inversion markers so the external WGL translator is misled into thinking that no data adjustments for inversion are needed. The corresponding WGL scanchain data when `omit` is used is shown in the following example:

```
# set_wgl -inversion_reference omit

scanchain
  cl ["si", "U1/R", "U2/R", "U3/R", "so" ];
end
```

See Also

[set_wgl](#)

[set_buses](#)

Effect of CELLDEFINE in WGL

The previous examples showed the effect of different choices of inversion reference on the placement of the inversion markers "!" in the scanchain definition block. Another item which affects the scanchain block is the presence or absence of the ``celldefine` compiler directive in the definition of the library model. Consider the following two examples:

```
# Verilog library module without celldefine
module SDFE (Q, CLK, SE, D, SI);
  input CLK, SE, D, SI;
  output Q;
  uMUX M (di, SE, D, SI);
  uDFFQ R (Q, CLK, di);
endmodule
```

```
endmodule

# WGL scanchain shows instance "R"
scanchain
  cl ["si", "U1/R", !, "U2/R", "U3/R", "so" ];
end

# Verilog library module with celldefine
`celldefine
module Sdff (Q, CLK, SE, D, SI);
  input CLK, SE, D, SI;
  output Q;
  uMUX M (di, SE, D, SI);
  uDffQ R (Q, CLK, di);
endmodule
`endcelldefine

# scanchain instances have no "R"
scanchain
  cl ["si", "U1", !, "U2", "U3", "so" ];
end
```

In the first example the Verilog module definition was not defined inside a ``celldefine/`endcelldefine` pair. The resulting WGL scanchain definition shows instance pathnames that include the `R` of the `uDffQ` device.

In the second example the Verilog module definition was within a ``celldefine/`endcelldefine` pair. The resulting WGL scanchain definition does not include the instance references beneath the `Sdff` module.

Note: Reading a netlist with the `-library` option has the same effect as enclosing the module with ``celldefine/`endcelldefine` pair and is yet another way to affect the WGL output.

See Also

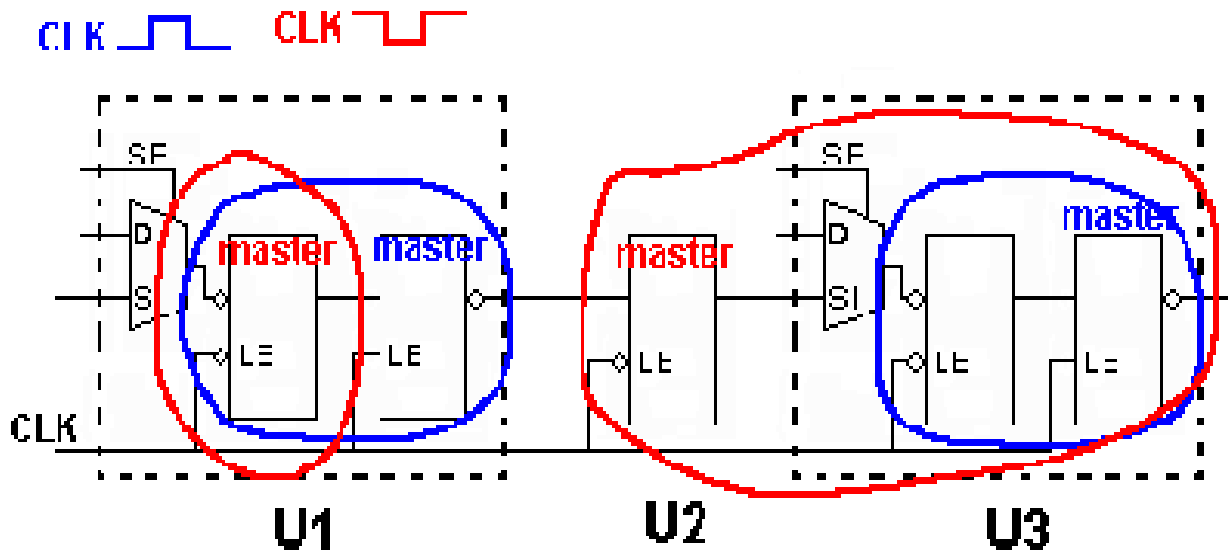
[set_wgl](#)

[set_buses](#)

Ambiguity of the Master Cell in WGL

The diagram below provides one simple example of the potential for ambiguity when using an inversion reference of master. In this example some DFF functions are created with a library cell using two latches. TestMAX ATPG defines the "master" based on which sequential device in a scan chain shifts first due to the leading edge of the defined shift clocks. So with the CLK port defined as active high, the "master" becomes the second LATCH in U1 and U3, with U2 acting as a lockup latch. If the polarity of CLK is reversed,

then the first latch in U1 is classified as the master and the lockup latch is classified as the master for cell U3! Both polarities of CLK generates ATPG patterns but most likely only one resulting WGL inversion set is correct.



See Also

[set_wgl](#)

[set_buses](#)

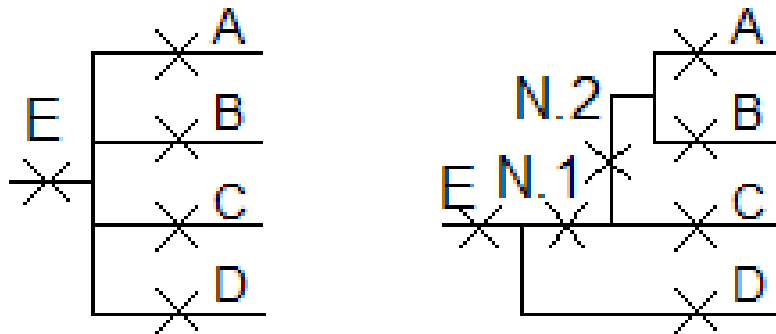
Subnet Formats for Diagnosis

This topic describes the formats used for defining the net structure, including subnets for diagnosis, and also for defining subnet stuck-open faults for the diagnosis ranking flow.

Net Topology Definition

The figure shows an example of a net with a single driver and several fanouts.

Figure 1 Logical (a) and Abstracted Physical (b) View of a Net



A B

The example net in the figure is displayed in the following example as an ASCII representation of the abstracted net topology in a format accepted by TestMAX ATPG:

```
.net
<net_driver># net driver of this net - pin E
<fanout_pin1> # id=1: first fanout branch - pin A
<fanout_pin2># id=2: second fanout branch - pin B
<fanout_pin3># id=3: third fanout branch - pin C
.subnets
1 2 # a defect in subnet N.2 affects 1st (A) and 2nd (B) branches
1 2 3 # a defect in subnet N.1 affects 1st (A), 2nd (B) and 3rd (C)
branches
```

As shown in the previous example, a net is completely defined by two sections: `.net` and `.subnets`.

The `.net` section defines the net driver name, which is the hierarchical pinpath name to a valid pin. This section defines any fanout branches according to their hierarchical pinpath names. There should be one fanout pinpath name per line. To maintain a compact format, each fanout branch in `.subnets` section is referred to using an id. Each fanout branch id is assigned based on the order it is defined. For instance, the id of the first fanout branch A is equal to 1 since it is defined first. The id of fanout branch B is 2, and so forth.

The `.subnets` section defines a list of subnets (one per line) and their fanout branches. The subnets are uniquely defined by a set of ids of the affected branches. For instance, subnet N.2 has pins A (of id 1) and B (of id 2) as fanout branches. Therefore, it is defined as "1 2".

When TestMAX ATPG reads the subnet ASCII file, it assigns an id for each subnets based on the order of the definition. These ids are used for reporting purposes. For instance, the subnet with id 1 in the previous example is N.2.

Subnets Fault Format

The fault list file is different than the net structure file, although both files contain similar sections.

To completely describe a subnet fault, the .net and .subnets sections need to be defined. The .net section contains the description of the net driver pinpath name, and the .subnets section contains the list of the subnets to rank for the previously defined net driver. Each subnet is identified by its id, and one id should be present per line.

An example of fault file format is shown below:

If the following subnets are present in memory:

```
net=CORE/\I_ENC/I_CSC/r189/U2_2/S #subnets=3
```

```
net=CORE/\I_ENC/I_CSC/U421/Y #subnets=4
```

Then, to rank each subnet of each net driver, the subnet fault file should appear as follows:

```
.net
CORE/\I_ENC/I_CSC/r189/U2_2/S
.subnets
1
2
3
.net
CORE/\I_ENC/I_CSC/U421/Y
.subnets
1
2
3
4
```

The subnet fault format is flexible enough so that it is not necessarily to specify the .subnets section. In this case, all subnets of this driver is added in the fault list. Furthermore, only a subset of a particular subnet could be specified and ranked.

For example, say the following subnet fault file is read in:

```
.net
CORE/\I_ENC/I_CSC/U411/Y
.net
CORE/\I_ENC/I_CSC/U421/Y
.subnets
1
3
.net
CORE/\I_ENC/I_CSC/r189/U2_2/S
```

The fault list includes all subnets of net drivers CORE/\I_ENC/I_CSC/r189/U2_2/Sand CORE/\I_ENC/I_CSC/U411/Y, as well as subnets 1 and 3 for net driver CORE/\I_ENC/I_CSC/U421/Y.

Handling Escape Characters in Tcl Mode

In Tcl mode, a backslash character (\) specified at the end of a line represents a line continuation. Any single backslash specified in the middle of a word escapes the character following it.

The following sections describe the various methods for handling escape characters:

- [Using Escape Characters with Wildcards and Regular Expressions](#)
- [Specifying Escaped Names for a List Argument](#)
- [Specifying Escaped Names for a String Argument](#)

Using Escape Characters with Wildcards and Regular Expressions

A wildcard or regular expression used in an escaped identifier is not recognized as wildcard or regular expression. For example, in the following command, the regular expression is not correctly interpreted:

```
add_nofaults -instance \\u_padring_wrap/u_jtag/.*
```

If you want to use a wildcard with an escaped name, you must use the following:

```
add_nofaults -instance *u_padring_wrap/u_jtag/.*
```

Specifying Escaped Names for a List Argument

The following examples show the various methods for specifying escaped names for a list argument:

```
add_faults {{\abccdef/hij/U1/A}}
add_faults {\abccdef/hij/U1/A}
add_faults {abccdef/hij/U1/A}
add_faults [list {\abccdef/hij/U1/A} ]
add_faults [list \abccdef/hij/U1/A ]
add_faults [list abccdef/hij/U1/A ]
```

Specifying Escaped Names for a String Argument

The following examples show the various methods for specifying escaped names for a string argument:

```
add_nofaults {\\abccdef/hij/U1/A}
add_nofaults {\abccdef/hij/U1/A}
add_nofaults {abccdef/hij/U1/A}
```

Passing Complex Options to LSF/GRID

The `-options` choice of the `add_distributed_processors` command can be used for passing most simple options for the `bsub` and `qsub` executables. However, parsing of the options has some limitations. For example, you cannot pass a value enclosed in double quotation marks.

For passing more complex options, please use this procedure:

1. Create a wrapper script called "bsub_wrapper" and make it an executable file; for example,

```
#!/bin/csh -f /path/to/real/bsub -R"sunos5_8 && maxmem>512" $*
```

2. In a TestMAX ATPG command file, include an `add_distributed_processors` command similar to this example:

```
add_distributed_processors -lsf /path/to/bsub_wrapper -nslaves 4 \
-options "-q hw-atpg -u kaiser.soze@mysterychip.com"
```

You can pass complex options to `qsub` with the previous example in a similar fashion: replace "lsf" with "grd" and the path to `bsub` with the path to `qsub`.

30

Scripts

This section provides links to the following scripts:

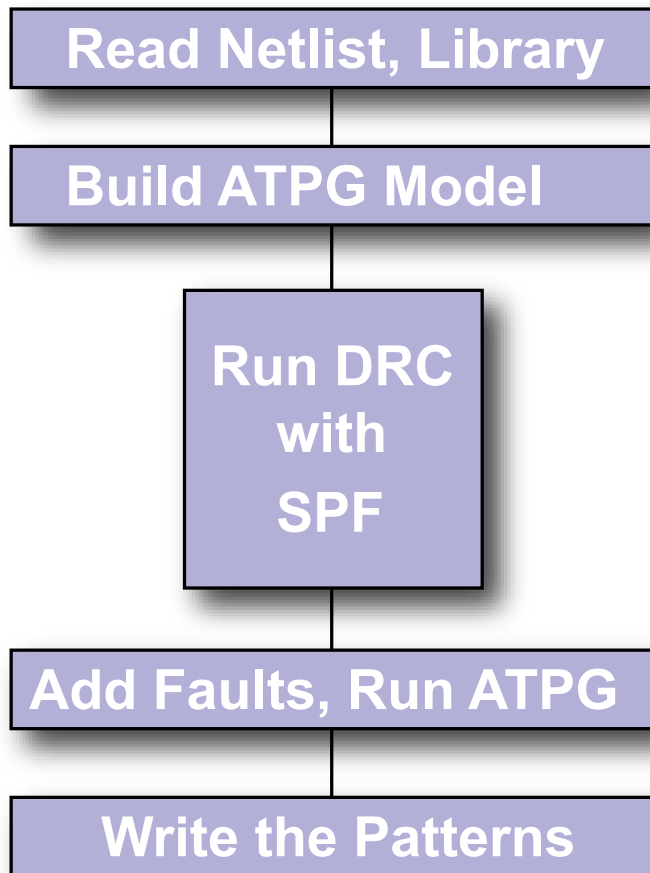
ATPG

- [Basic ATPG Run](#)
- [Basic TestMAX ATPG Run](#)
- [ATPG Run Without SPF](#)
- [Bridging Faults](#)
- [Cell-Aware ATPG](#)
- [Dynamic Bridging ATPG](#)
- [Low Power ATPG](#)
- [Multicore ATPG;](#)
- [Scan-Through Tap ATPG](#)
- [Transition Fault ATPG](#)
- [Transition Fault ATPG Using LOES](#)

General

- [Basic TestMAX ATPG Diagnosis Run](#)
- [Distributed Processing Fault Simulation](#)
- [DFTMAX What-If Analysis](#)
- [Fault Coverage of Combined ATPG and JTAG Vectors](#)
- [Generating DFMAX Ultra High Resolution Patterns](#)
- [Generating Patterns for DFTMAX Ultra](#)
- [IDDQ Bridging](#)
- [Slack-Based Testing](#)

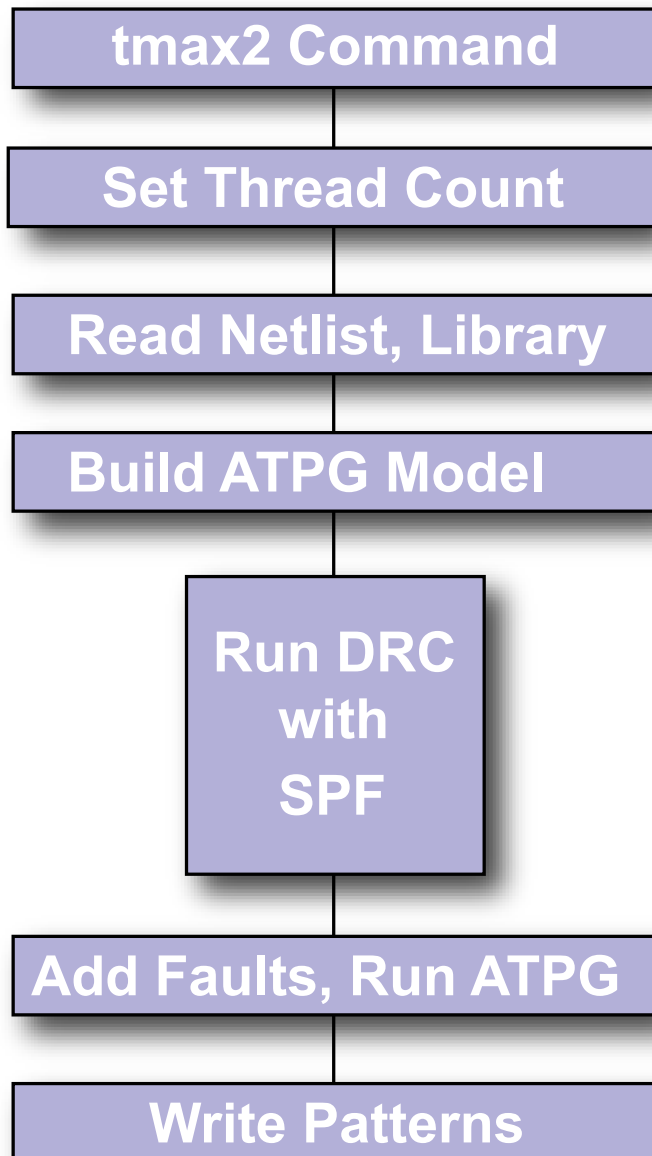
Basic ATPG Run Script



```
set_messages -log mylog -replace
# Read in the netlist library and
# Verilog library
read_netlist -library library/*.v
read_netlist design.v
# Build the ATPG Model
run_build_model DESIGN_TOP
# Set up and run DRC
set_drc stil_procedures.spf
run_drc
# Add faults and run ATPG
add_faults -all
run_atpg -auto
# Write the patterns
write_patterns DESIGN.stil -format STIL \
-replace
write_patterns DESIGN.bin -replace
```

For more information, see [Basic ATPG Flow](#)

Basic TestMAX ATPG Run



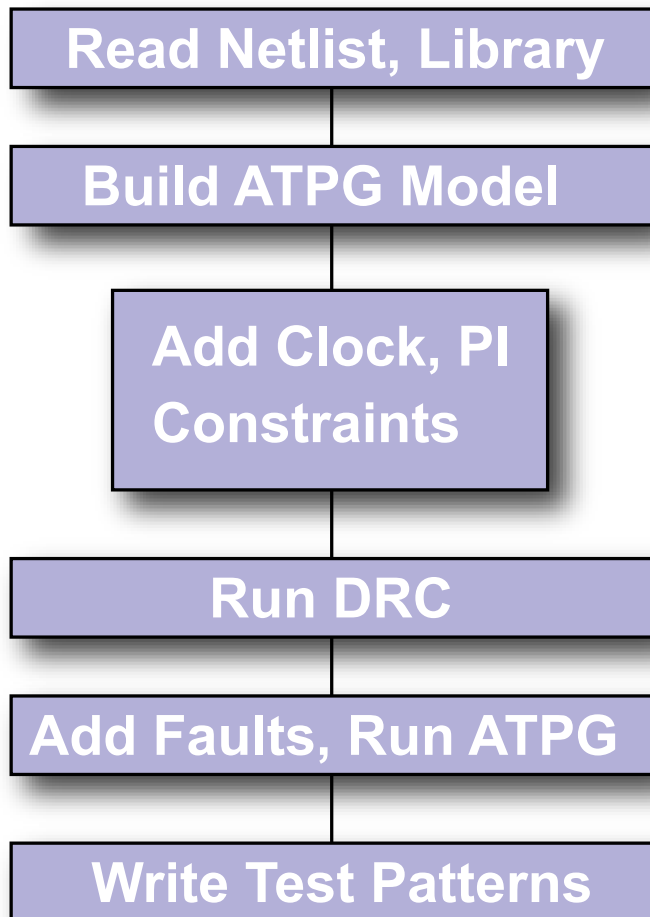
```
# In TestMAX ATPG, expert and  
# verbose messaging is  
recommended.  
set_messages -level expert
```

Chapter 30: Scripts

Basic TestMAX ATPG Run

```
set_atpg -verbose
# TestMAX ATPG Uses 8 threads by
# default. Use these settings to
# change
# the number of threads. Make sure
# that both set_atpg and set_
simulation
# use the same thread count.
# set_atpg -num_threads 8
# set_simulation -num_threads 8
# Read in the netlist library and
# Verilog library
read_netlist -library library/*.v
read_netlist design.v
# Build the ATPG Model
run_build_model DESIGN_TOP
#Set up and run DRC
set_drc stil_procedures.spf
run_drc
# Add faults and run ATPG
add_faults -all
run_atpg -auto
# Write the patterns
write_patterns DESIGN.stil -format STIL -replace
write_patterns DESIGN.bin -replace
```

ATPG Run No SPF



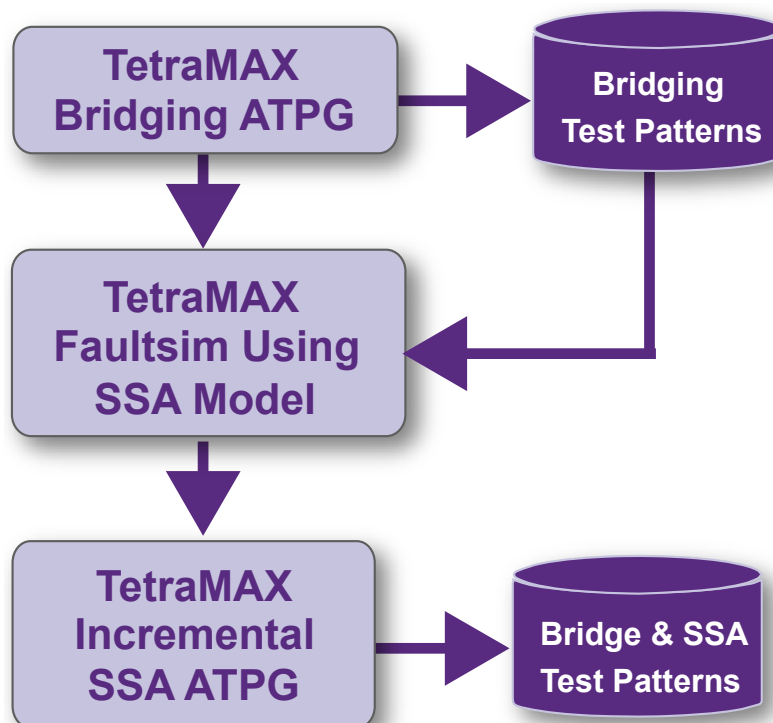
```
set_messages -log mylog -replace
# Read in the netlist library and
# Verilog library
read_netlist -library library/*.v
read_netlist design.v
# Build ATPG Model
run_build_model DESIGN_TOP
# Add Clocks and PI Constraints
add_clocks 0{clk}-shift \
-timing{100 50 80 40}
add_clocks 1{rst}-timing{100 50 80
40}
add_scan_enable 1 test_se
add_scan_chains c1 test_sis data_out
add_pi_constraints 1 test_mode
# Set up and run DRC
```



```
set_drc stil_procedures.spf
run_drc
# Add faults and run ATPG
add_faults -all
run_atpg -auto
# Write the patterns
write_patterns DESIGN.stil -format STIL \
-replace
write_patterns DESIGN.bin -replace
```

For more information, see [Basic ATPG Flow](#).

Bridging Fault ATPG



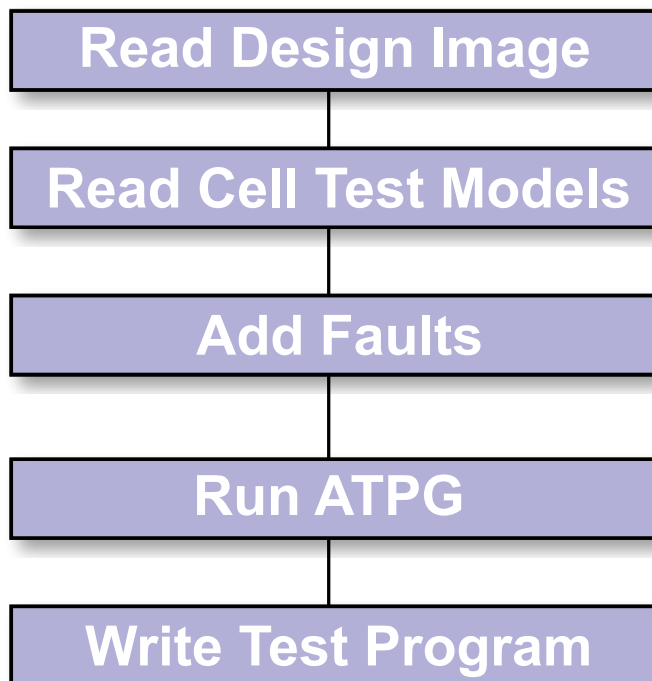
```
set_faults -model bridging
add_faults -node_file coupling.txt
// Optionally set drive strength
// set atpg \
// -optimize_drive_strength
run_atpg -auto
write_patterns bridging.pat \
-format bin
remove_faults -all
set_faults -model stuck
```

```
add_faults -all
set_patterns -external bridging.pat
run_fault_sim
set_patterns -internal
run_atpg -auto
set_patterns -external bridging.pat \
-append
write_patterns -bridge_ssa.pat \
-format ...
```

Note:

The stuck-at fault model can detect many bridges. So fault simulate stuck patterns against bridging. You can also try the opposite flow: Run ATPG with bridging faults and write out bridging patterns. Then run fault simulation of bridging patterns using stuck-at faults. Then, tophoff with additional stuck-at faults for any remaining undetected stuck-at faults. A similar scheme can be used to combine patterns from other fault models (e.g. transition).

Cell-Aware ATPG



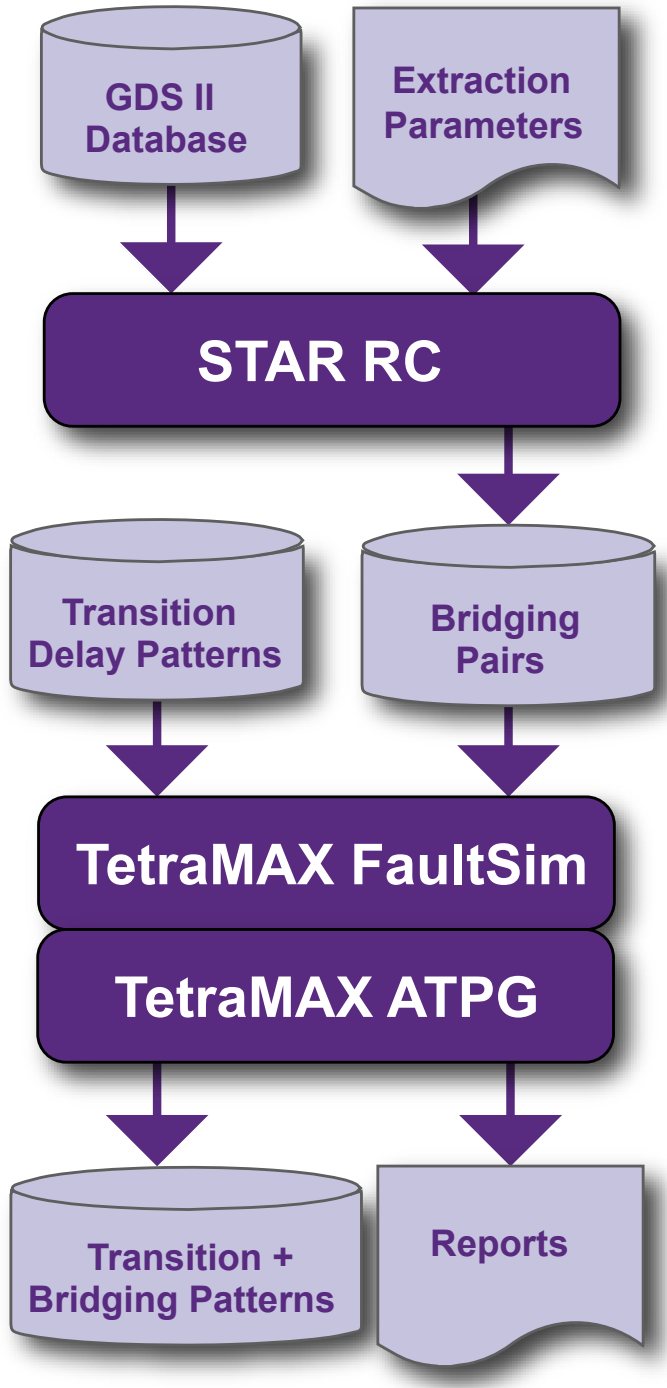
```
# Read the design image
read_image ./design/leon3mp.img
# Read CTMs
```

Chapter 30: Scripts

Cell-Aware ATPG

```
read_cell_model ./CTM/*.CTM
# Add Faults
add_faults -all -cell_aware
# Run ATPG
run_atpg -optimize
# Write the patterns
write_patterns \
./patterns/cell_aware_patterns.bin -replace
```

Dynamic Bridging Fault ATPG



Chapter 30: Scripts

Dynamic Bridging Fault ATPG

```

# read netlist and libraries,
# build, run drc
read_netlist design.v -delete
run_build_model design
run_drc design.spf
# set fault model to dynamic
bridging
set_faults -model dynamic
bridging
# read in fault list
add_fault -node ...
# run atpg
run_atpg -auto_compression
# write out the bridging
patterns
write_patterns dyn_bridge_pat.bin \\
-format binary -replace
# Fault simulate dynamic bridge patterns with stuck-at
# faults. This is intended to reduce the set of patterns
# by not generating patterns for stuck-at faults
# detected by the dynamic bridging patterns
remove_faults -all
set_faults -model stuck
add_faults -all
# read in dynamic bridging pattern
set_patterns external dyn_bridge_pat.bin
# fault simulate
run_fault_sim
# generate additional stuck-at patterns
set_patterns internal
run_atpg -auto_compression

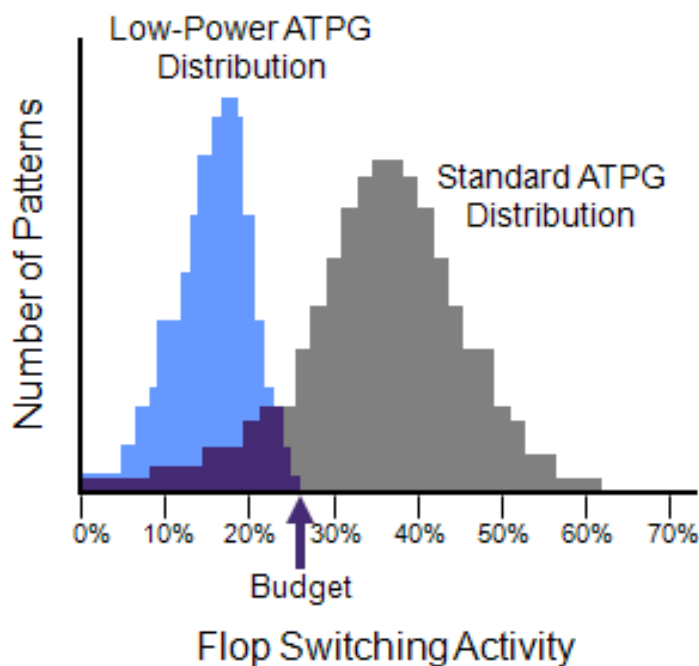
```

Note the following:

- ATPG attempts to launch a transition along the victim while holding the aggressor at a static value
- Before running ATPG with `run_atpg` a fault list should have been created
- Basic-scan (launch on last shift), Two Clocks, and Fast-Sequential ATPG modes are supported
- Fault simulation supported as usual with `run_fault_sim` command
- Dynamic Bridging ATPG and Fault Simulation do not support Full-Sequential mode
- Strength-based pattern generation (similar to what exists for the static bridging fault model) is not supported Node file format: pair of bridge locations on each line, separated by a space
- An unmodified coupling capacitance report generated from Synopsys Star-RCXT could be also used

- More info on node file, see the “Node File Format for Bridging Pairs” topic in TestMAX ATPG Online Help.
- Fault list file should not include clocks and asynchronous set or reset signals, because proper detection status cannot be guaranteed for these faults
- For more information on dynamic bridging, see [Running the Dynamic Bridging Fault ATPG Flow](#).

Low Power ATPG

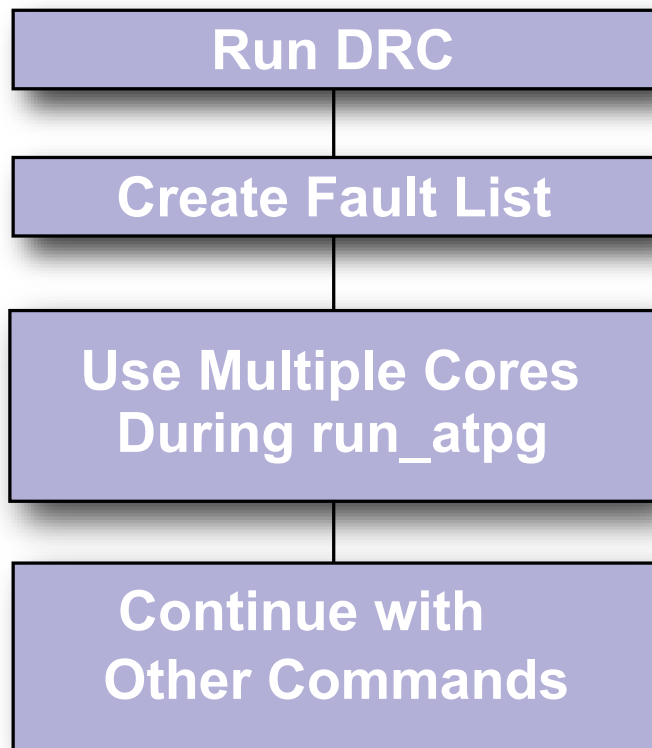


```
read_netlist -lib $my_lib.v
read_netlist $my_design.v
run_build $my_design
run_drc $my_drc_file.spf
report_clock -gating
set_atpg -fill adjacent
set_atpg -power_budget 25
add_faults -all
run_atpg -auto
report_power -per_pattern -percentage
```

Notes:

- TestMAX ATPG performs low-power fill during ATPG
 - `set_atpg -fill adjacent`
- Reduces Peak Power during Capture
 - Accepts designer-specified switching activity budget
 - Simple, highly automated flow eliminates need for manual ad hoc solutions
 - Produces compact pattern set
 - Enables testing at mission-mode power levels
- For more information, see [Power-Aware ATPG](#).

Multicore ATPG



```
# Perform DRC and enter TEST mode  
run_drc top_level.spf
```

Chapter 30: Scripts

Multicore ATPG

```

# select the fault model and create
# the fault list
set_faults -model transition
# Other ATPG settings here
...
# Use two cores during run_atpg
set_atpg -num_processes 2
run_atpg -auto
# Continue with other commands
after run_atpg
...
// Multicore Usage - Farm Multi-Host (LSF)
bsub -R "span[hosts=1]" -n 4 \
tmax_multicore_batch.csh
// Ensure all slots are reserved on a single host
// 4 slots are allocated for ATPG run
read_netlist Libs/*.v -delete -library -noabort
run_build_model top_level
run_drc top_level.spf
set_faults -model stuck
...
add_faults -all
set_atpg -num_processes 4
// Multicore Usage - Farm Multi-Host (GRID)
qsub -l cputype=amd64,\
mem_free=16G,mem_avail=16G,\
cpus_used=4,model=AMD2800 \
tmax_multicore_batch.csh
// 4 slots are allocated for ATPG run
read_netlist Libs/*.v -delete -library -noabort
run_build_model top_level
run_drc top_level.spf
set_faults -model stuck...
add_faults -all
set_atpg -num_processes 4

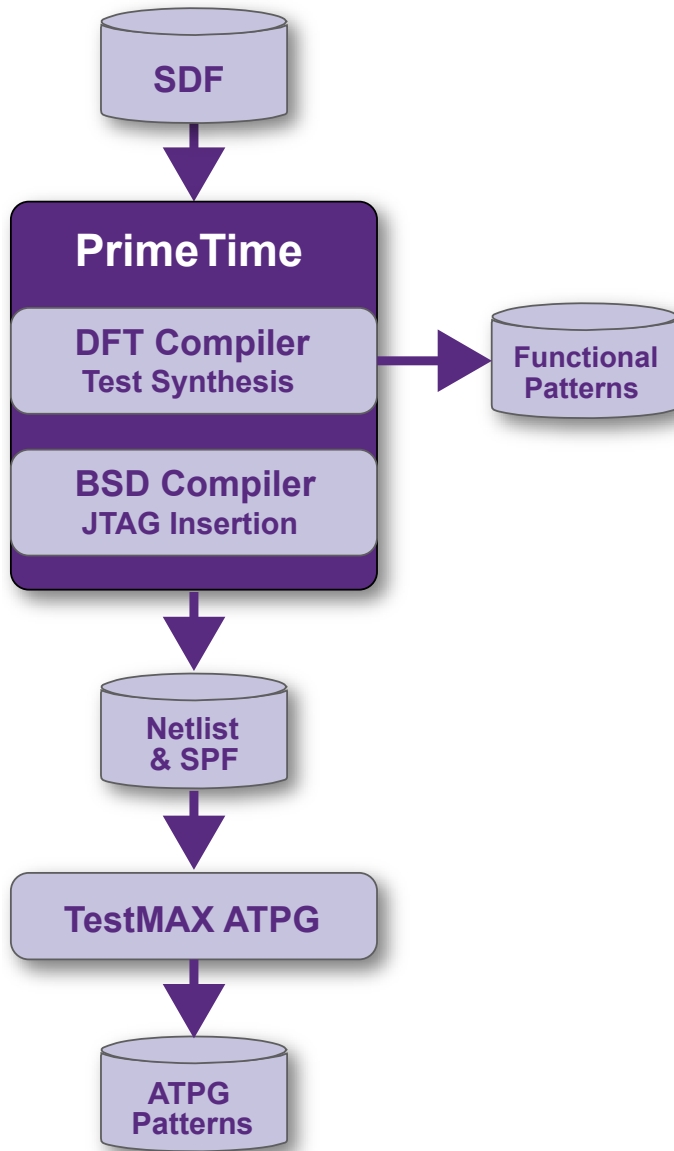
```

Notes:

- Allows you to parallelize and improve ATPG runtime.
- Uses up to the number of user-specified cores.
- Better memory utilization (~50% increase per core) compared to distributed (~100% increase per slave)
- Good runtime scalability
- Very predictable QoR
- Communication via shared memory instead of files and sockets
- Simple user model: `set_atpg -num_processes 2`

- All ATPG engines: basic-scan, two-clock, fast-seq, full-seq
- For more information, see [Running Multicore ATPG](#)

Scan-Through-TAP ATPG Flow



```
set_messages -log STT_tmax_  
atpg.log -replace  
read_netlist ./tmax_  
lib/class.v
```

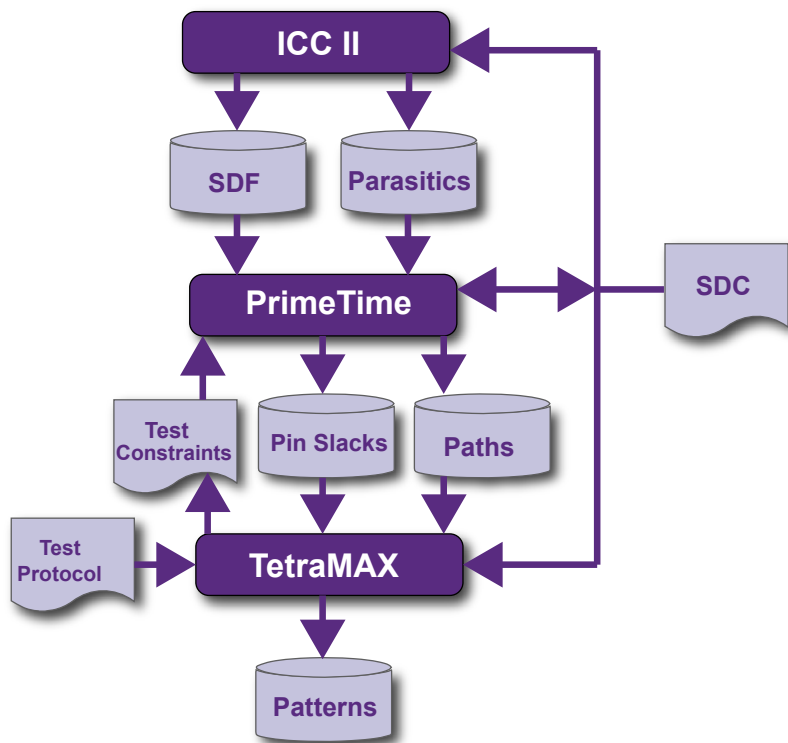
Chapter 30: Scripts
Transition Delay Fault ATPG

```

read_netlist $output_dir/TOP_
bsd.v
run_build TOP
// only allow captures with
tck clock
add_clock 0 TAP_TCK
set_drc -clock TAP_TCK -init 0
//use spf generated from BSDC
run_drc $output_dir/stt_
bsd.spf
add_faults -all
run_atpg -auto
// write out fault list and ATPG patterns
write_faults $output_dir/top_stt.faults -all -replace
write_patterns $output_dir/top_stt.stil -format stil -replace

```

Transition Delay Fault ATPG



```

// last_shift launch:
read_netlist lib.v -lib
read_netlist test.v
run_build top
set_delay -launch_cycle \
last_shift

```

Chapter 30: Scripts

Transition Delay Fault ATPG Using LOES Timing

```

set_fault -model transition
set_drc test.spf
read_sdc <FILE_NAME>
set_delay -nopi_changes
set_delay -nopo_measures
add_po_mask -all
set_delay -common_launch_capture_
clock
set_delay -allow_multiple_common_clocks
#optional# set_delay -slow_equivalence
#optional# set_delay -nodisturb
add_pi_constraints 0 scan_en
run_drc
run_atpg -auto
write_pattern patp.stil -format stil -parallel
write_pattern pats.stil -format stil -serial
exit
// system_clock launch:
read_netlist lib.v -lib
read_netlist test.v
run_build top
set_delay -launch_cycle system_clock
set_drc test.spf
set_fault -model transition
read_sdc <file_name>
set_delay -nopi_changes
set_delay -nopo_measures
add_po_mask -all
set_delay -common_launch_capture_clock
set_delay -allow_multiple_common_clocks
#optional# set_delay -slow_equivalence
#optional# set_delay -nodisturb
add_pi_constraints 0 scan_en
run_drc
run_atpg -auto
write_pattern patp.stil -format stil -parallel
write_pattern pats.stil -format stil -serial
exit

```

For more information, see [Transition Delay Fault ATPG](#).

Transition Delay Fault ATPG Using LOES Timing

```

// Typical Flow for Using Launch-On
// Extra Shift Mode
set_delay -launch_cycle extra_shift
# Other delay settings are the same for LOES and LOC
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loes.spf -patternexec Internal_scan
set_faults -model transition

```

Chapter 30: Scripts

Transition Delay Fault ATPG Using LOES Timing

```
run_atpg -auto
write_patterns design_with_loes.stil -format stil
write_faults design_with_loes.faults -all
drc -force
# Prepare to change the LOSPipelineEnable constraint value
remove_pi_constraints -all
set_delay -launch_cycle system_clock
set_delay -common_launch_capture_clock -nopi_changes
add_po_masks -all
run_drc design_with_loc.spf -patternexec Internal_scan
set_faults -model transition
#Use -retain_code so the redundant faults do not need to be
identified again
read_faults design_with_loes.faults -retain_code
# Many faults that are AU for LOES can be detected by LOC
update_faults -reset_au
```

For more information, see [Transition Delay Fault ATPG Timing Modes -- LOES](#).

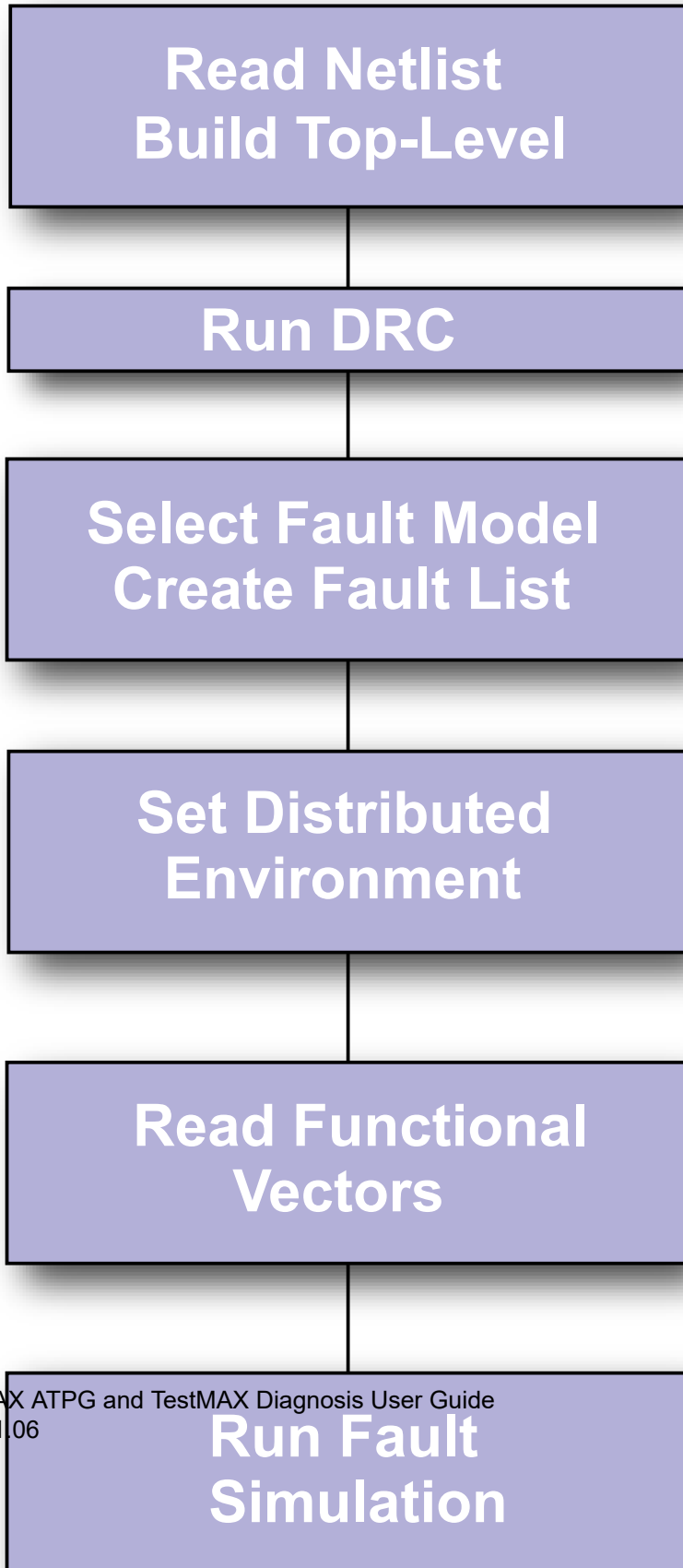
Basic TestMAX ATPG Diagnosis Run

Chapter 30: Scripts

Basic TestMAX ATPG Diagnosis Run

```
# Read the image used to
# generate the pattern for
# diagnostics:
read_image \
Results/design1_img.gz
# Use the class-based candidate
# organization in the diagnosis
# report.
set_diagnosis -organization \
class
set_diagnosis -fault_type all
# TestMAX ATPG thread settings
set_atpg -num_threads 8
set_simulation -num_threads 8
# Set the pattern file, using the
# binary pattern set if available, but
# only to diagnose pattern-based failure
# data. STIL (or WGL) patterns are
# required to diagnose cycle-based
# failure data, and may be used to
# diagnose pattern-based failure data.
set_patterns -external patterns.bin
# Perform diagnosis on the failure file
# and generate data for Yield Explorer.
# Multiple runs using different failure files
# with the same pattern may be
# run in sequence:
run_diagnosis datalogs/fail1.log
write_ydf results1/design1_diagnosis.ydf -replace
run_diagnosis datalogs/fail2.log
write_ydf results2/design1_diagnosis.ydf -append
run_diagnosis datalogs/fail3.log
write_ydf results3/design1_diagnosis.ydf -append
```

Distributed Processing Fault Simulation Flow



```

read_netlist libs.v -delete -library
read_netlist scan_design.v
run_build <top_level >
run_drc top_level.spf
set_fault -model stuck
add_faults -all
set_distributed \
-work_dir /shared/workdir
add_distributed_processors "hosta hostb"
set_pattern -external func_pat.bin.gz
run_fault_sim -distributed -sequential
write_fault final.flt -all -collapsed \
-compress gzip -replace

```

DFTMAX What-If Analysis

```

set_messages -log \
scancompress_analysis.log -replace
# read libraries and netlist
read_netlist libs/tmax_libs/*.v -library
read_netlist design.v
run_build top
# run drc in regular scan mode
run_drc scan.spf
set_faults -model stuck
add_faults -all
analyze_compressor -num_inputs 15 \
-num_scanouts 15 -num_chains 180

```

DFTMAX Ultra High Resolution Pattern Flow

```

read_image image_file.dat
## Step 1
## Run diagnostics to identify defective
## chains. For faster run time, read
## chain test patterns only. To
## generate a separate set
## containing chain test patterns
## use run_atpg -only_chain_test
set_patterns -external chain.stil
run_diagnosis \
high_res_pat_failure_log_file.log \
-streaming_report_chains_only \
chain_fail_list
set_patterns -delete
## Step 2
## Read full pattern test file
set_patterns -external full_pattern_set.stil
## Use add_chain_masks command to
## generate high resolution patterns

```


Chapter 30: Scripts

Fault Coverage of Combined ATPG and JTAG Test Vectors

```

add_chain_masks -external -filename chain_fail_list \
-diagnosis
## Write the patterns
write_patterns high_resolution_set.stil -format stil \
-external
## Step 3
## Retest the failing part with the high resolution
## patterns and collect the fail data
## Step 4
set_patterns -external high_resolution_set.stil
## Main diagnosis run using log file generated using
## high resolution patterns
run_diagnosis high_res_pat_failure_log_file.log -verbose

```

Note:

If you mask the chain using the `add_chain_mask` command, the compares during output capture may fail. Use the `add_cell_constraint xx` command to avoid failing.

Fault Coverage of Combined ATPG and JTAG Test Vectors

```

//Generate the JTAG test vectors in BSD Compiler:
create_bsd_patterns
write_test -f verilog -out jtag_tb.v
// Edit the testbench to add the eVCD dumpvar
// command in the initial block:
initial begin
_failed = 0;
/* Generate VCDE */
// 'ifdef vcde_out
// extended VCD, see IEEE Verilog 1364-2001
$display("// %t : opening Extended VCD output file", $time);
$dumpsports( design.design_inst, "sim_vcde.out");
// 'endif
// Run VCS simulation as follows:
Command: vcs -R -l vcs.log ver_files.v jtag_tb.v
+define+GATES+ +notimingchecks+ -y models -v libs/my_io.v -v
libs/my_techlib.v +libext+.v+
// Step 2: Run ATPG for regular scan patterns
# Read in libraries and top-level design with bsd inserted
read_netlist core.v
read_netlist io.v
read_netlist design_w_bsd.v
run_build
# Run drc for atpg patterns
run_drc mydesign.spf
# Obtain the coverage of the core, excluding IEEE 1149.1 logic
add_nofaults ORCA_BSR_top_inst
add_nofaults ORCA_DW_tap_inst
add_faults -all

```

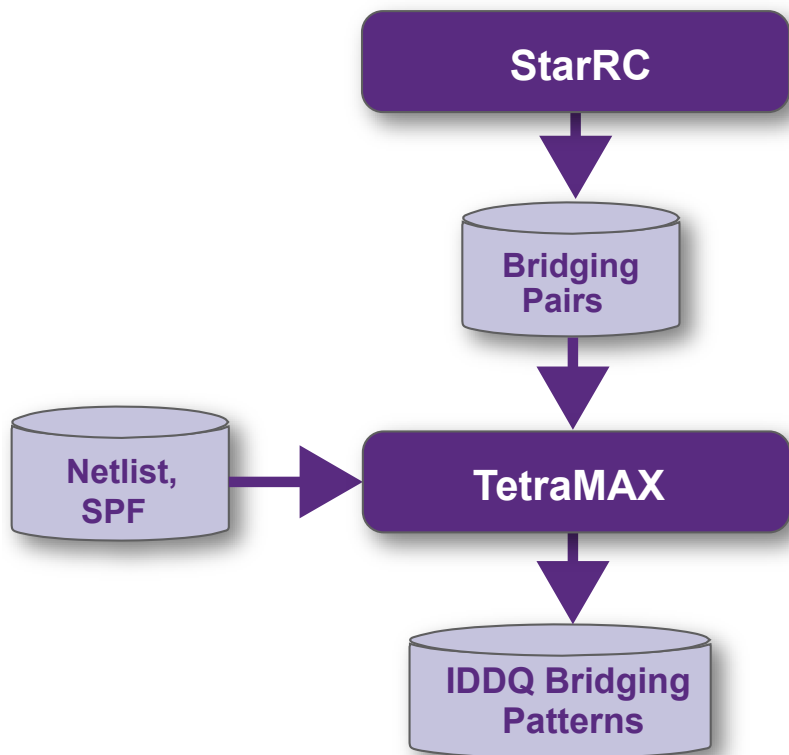
```
# Generate ATPG patterns
run_atpg -auto
# Write out atpg patterns and faultlist
write_faults atpg.faults.gz -compress gzip -all -replace
write_patterns atpg.pats.gz -format bin -compress gzip -replace
// Step 3: Read in external bsd patterns & run fault_sim
# Read in top-level design and run drc without a .spf file
read_netlist {core.v io.v design_w_bsd.v}
run_build
run_drc
# read in external patterns with appropriate strobe options
set_patterns -delete -external sim_vcde.out
-strobe_period {100 ns} -strobe_offset {95 ns}
# run seq simulation to verify there are no mismatches
run_sim -sequential
# Read in atpg faultlist and add in the jtag faults
read_faults atpg_faults.gz -retain
add_faults ORCA_BSR_top_inst
add_faults ORCA_DW_tap_inst
# Fault simulate the jtag patterns
run_fault_sim -sequential
report_summaries
```

Generating Patterns for DFTMAX Ultra

```
### USER INPUTS AND DFTMAX ULTRA OUTPUT FILES ###
set TOP_MODULE_NAME top_module_name
set NETLIST_FILES1 netlist_files1
set NETLIST_FILES2 netlist_files2
set LIBRARY_FILES1 library_files1
set LIBRARY_FILES2 library_files2
set BUILD_CONSTRAINTS_FILE build_constraints_file
set DRC_CONSTRAINTS_FILE drc_constraints_file
set STL_procedure_file_FILE spf_file
set LOG log_file
setenv SYNOPSIS path_to_tool_installation
##### BUILD SETTINGS #####
set_messages -level expert -log $LOG -replace
report_version -full
build -force
set_faults -pt_credit 0
set_faults -summary verbose
set_rules N2 warning
set_rules B12 warning
set_rules B5 warning
set_faults -atpg_effectiveness
set_atpg -verbose
set_netlist -redefined_module last
read_netlist $NETLIST_FILES1
read_netlist $NETLIST_FILES2
read_netlist $LIBRARY_FILES1 -library
```

```
read_netlist $LIBRARY_FILES2 -library
source -echo $BUILD_CONSTRAINTS_FILE
run_build_model $TOP_MODULE_NAME
##### DRC SETTINGS #####
source -echo $DRC_CONSTRAINTS_FILE
set_faults -model stuck
run_drc $STL procedure file_FILE
##### RUN ATPG #####
add_nofaults -module .*COMPRESSOR.*
add_faults -all
run_atpg -auto_compression
run_simulation -remove_padding_patterns
write_patterns ultra.stil format stil
```

IDDQ Bridging Flow



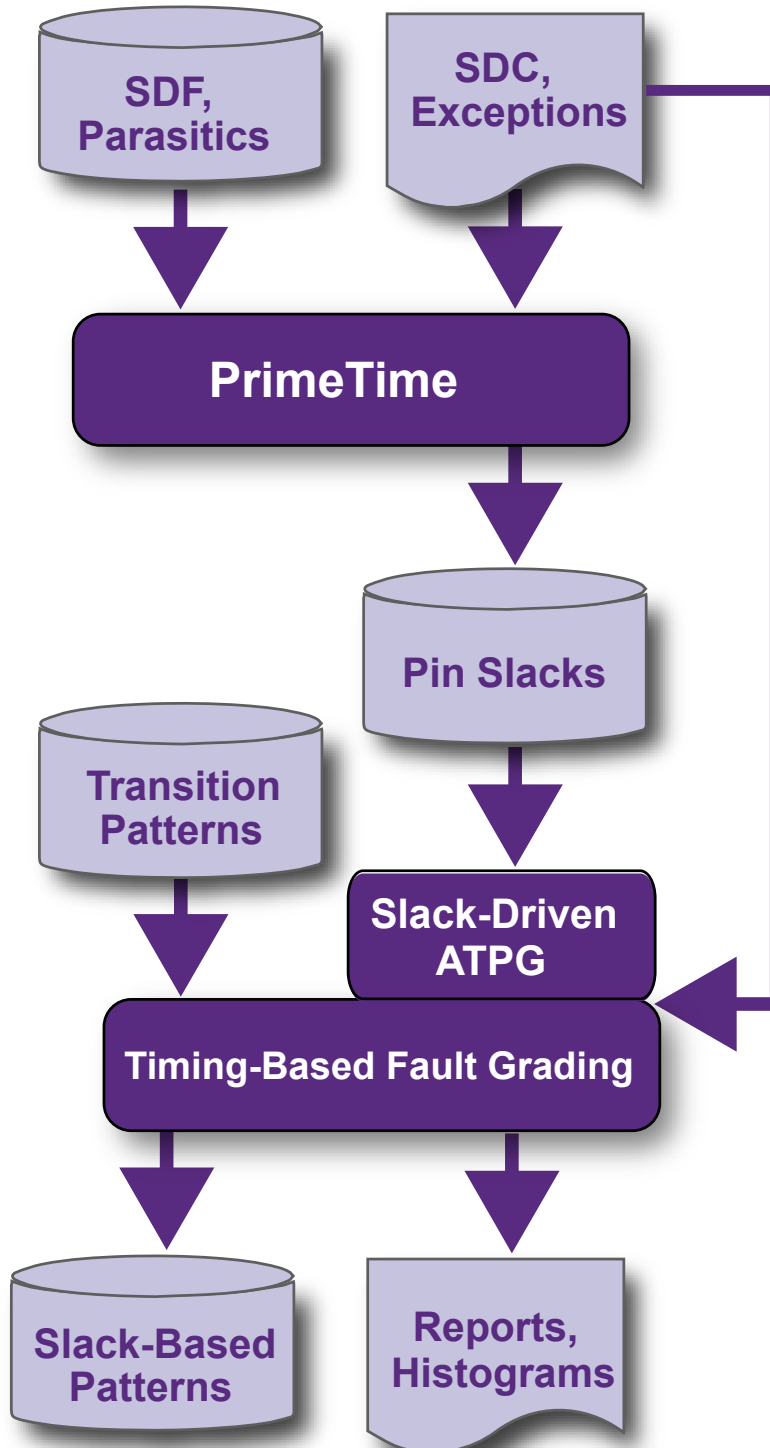
```
# Enable IDDQ Bridging fault model
set_fault -model iddq_bridging
# Read bridging pairs
add_fault -node_file bridging_
pair.txt
# Run ATPG
```

Chapter 30: Scripts

IDDQ Bridging Flow

```
run_atpg -auto
# Write patterns
write_patterns iddq_bridging.stil
\
-format stil
```

Slack-Based Testing



```
// PrimeTime:
...
set_timing_save_pin_arrival_and_slack
true
update_timing
report_global_slack -max -nosplit
global_slack_file
...
// TestMAX ATPG:
read_netlist CORE.v -library
read_netlist DESIGN.v
run_build_model TOP
add_pi_constraint 0 scan_en
run_drc DESIGN.spf
set_faults -model transition
set_delay -launch system
read_timing global_slack_file
set_delay -max_delta_per_fault 0.5
set_delay -max_tmgn 2.5
run_atpg -auto
# report commands
```

Note:

Because of additional requirements placed on the on-chip clocking controller, launch-on shift mode has some major drawbacks when used with internal clocking. However, launch-on extra shift mode provides many of the advantages of launch-on shift mode, without placing an extra burden on the on-chip clocking controller.

For more information, see [Transition Delay Fault ATPG Timing Modes](#).

31

Validating Test Patterns

This section describes the Synopsys tools you can use to validate generated test patterns. This includes MAX Testbench, which validates STIL patterns created from TestMAX ATPG, and PowerFault, which validates IDDQ patterns created from TestMAX ATPG.

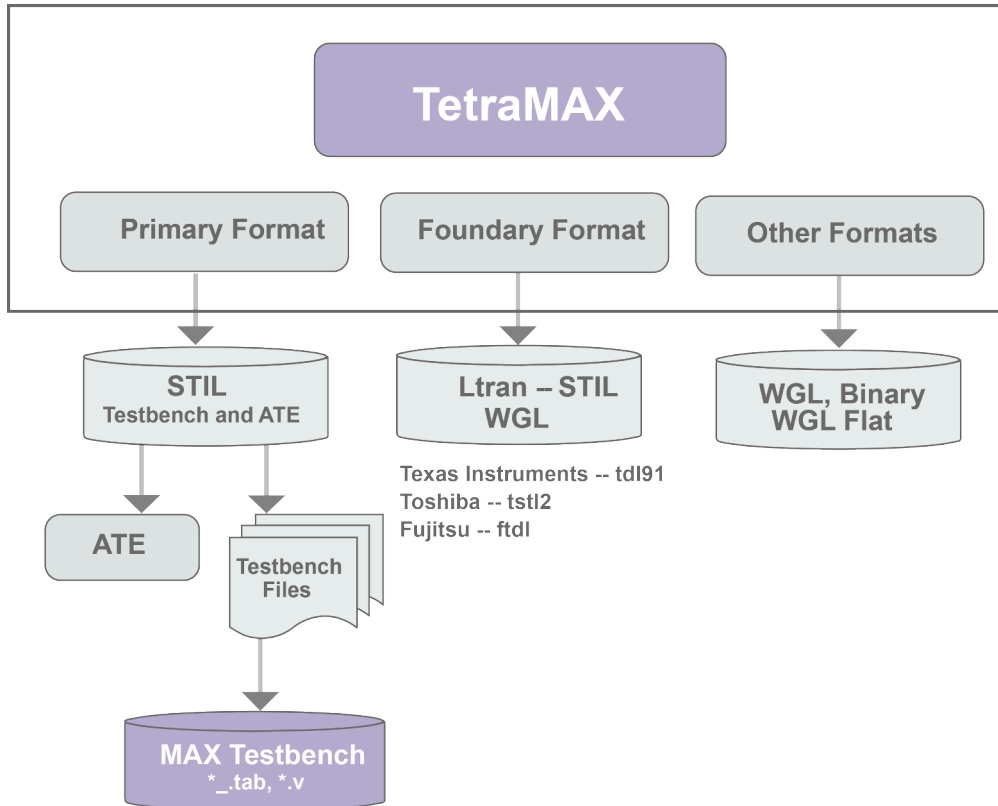
The following sections provide an introduction to test pattern validation:

- [TestMAX ATPG Pattern Format Overview](#)
- [Writing STIL Patterns](#)
- [Design to Test Validation Flow](#)

TestMAX ATPG Pattern Format Overview

The following figure shows an overview of the TestMAX ATPG pattern formats.

Figure 149 TestMAX ATPG Pattern Formats



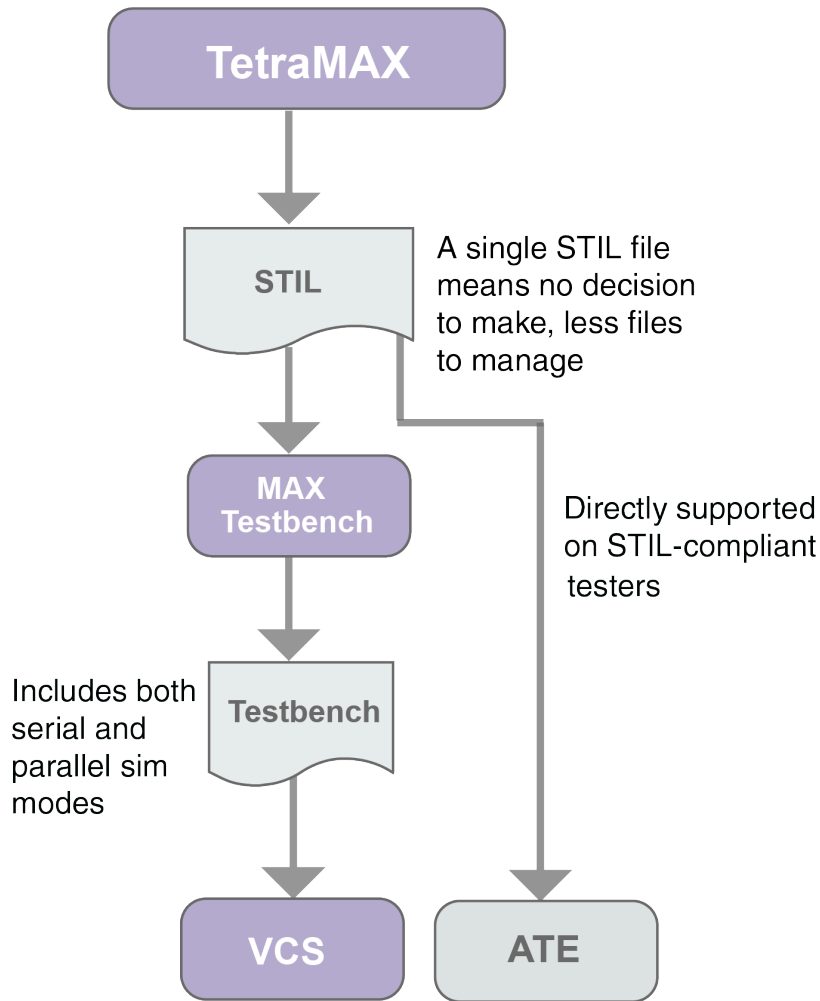
Writing STIL Patterns

TestMAX ATPG creates unified STIL patterns by default. This simplifies the validation flow considerably because only a single STIL file is required to support all simulation modes (you do not need to write both serial and a parallel formats).

You can use unified STIL patterns in MAX Testbench. It is based only on the actual STIL file targeted for the tester.

You can use a single unified STIL pattern file to perform all types of simulation, including parallel and mixed serial and parallel.

Figure 150 Unified STIL Pattern Validation Flow
Unified STIL Pattern Flow



The `write_patterns` command includes several options that enable TestMAX ATPG to produce a variety of pattern formats.

The `-format stil` option of the `write_patterns` command writes patterns in the proposed IEEE 1450.1-2005 Standard Test Interface Language (STIL) for Digital Test Vectors format. For more information on the proposed IEEE 1450.1-2005 STIL for Digital Test Vectors format (extension to the 1450.0-1999 standard), see Appendix E STIL Language Format in the TestMAX ATPG User Guide. This format can be both written and

read. However, only a subset of the language written by TestMAX ATPG is supported for reading back in.

The `-format stil99` option of the `write_patterns` command writes patterns in the official IEEE-1450.0 Standard Test Interface Language (STIL) for Digital Test Vectors format. This format can be both written and read, but only the subset of the language written by TestMAX ATPG is supported for reading back in.

You must use a 1450.0-compliant DRC procedure as input when to write output in `stil99` format.

The syntax generated when using the `-format stil` option is part of the proposed IEEE 1450.1-2005 extensions to STIL 1450-1999.

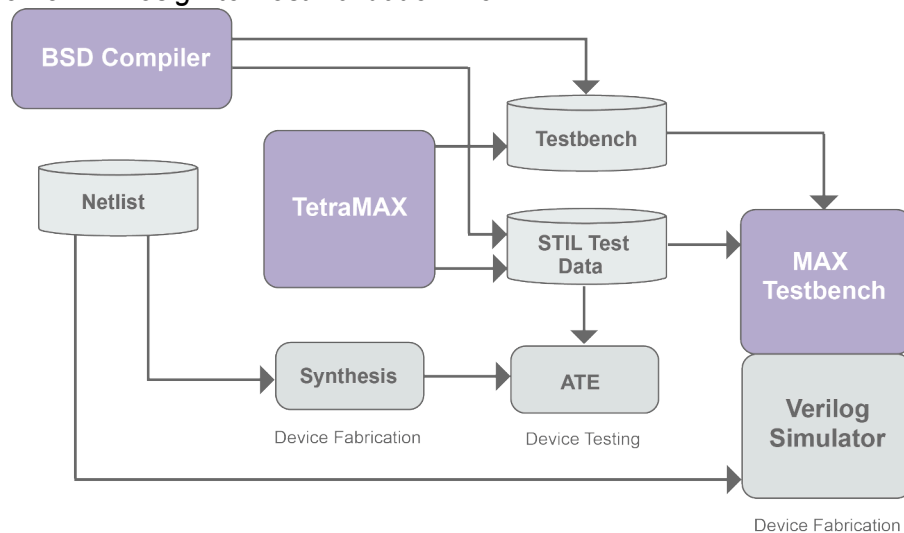
If you use the `-format stil` or `stil99` options, TestMAX ATPG generates a STIL file with a name in the filename `<pfile>.<ext>` in which you specified `write_patterns pfile.<ext>`.

When you use the `-format stil` or `-format stil99` options, you can also use the `-serial` or `-parallel` options to specify TestMAX ATPG to write patterns in serial (expanded) or parallel form. See the description of the `write_patterns` command in TestMAX ATPG Help for detailed information on using these options.

Design to Test Validation Flow

The following figure shows the validation flow using MAX Testbench. In this flow, test simulation and manufactured-device testing use the same STIL-format test data files.

Figure 151 Design-to-Test Validation Flow



When you run the Verilog simulation, MAX Testbench applies STIL-formatted test data as stimulus to the design and validates the design's response against the STIL-specified expected data. The simulation results ensure both the logical operation and timing sensitivity of the final STIL test patterns generated by TestMAX ATPG.

MAX Testbench validates the simulated device response against the timed output response defined by STIL. For windowed data, it confirms that the output response is stable within the windowed time region.

32

Using MAX Testbench

MAX Testbench is a pattern validation tool that converts TestMAX ATPG STIL test vectors for physical device testers into Verilog simulation vectors.

The following sections describe how to use MAX Testbench:

- [Overview](#)
- [Running MAX Testbench](#)
- [write_testbench Command Syntax](#)
- [MAX Testbench Command-Line Parameters Used With the write_testbench Command](#)
- [stil2Verilog Command Syntax](#)
- [Configuring MAX Testbench](#)
- [MAX Testbench Error Messages and Warnings](#)
- [Troubleshooting MAX Testbench](#)
- [Debugging Parallel Simulation Failures Using Combined Pattern Validation](#)

Overview

MAX Testbench simulates and validates STIL test patterns used in an ATE environment. These patterns are used in an ATE environment.

MAX Testbench reads a STIL file generated from TestMAX ATPG, interprets its protocol, applies its test stimulus to the DUT, and checks the responses against the expected data specified in the STIL file. MAX Testbench is considered a genuine pattern validator because it uses the actual TestMAX ATPG STIL file used by the ATE as an input to test the DU.

MAX Testbench supports all STIL data generated by TestMAX ATPG, including:

- All simulation mechanisms (serial, parallel and mixed serial/parallel)
- All type of faults (SAF, TF, DFs, IDDQ and bridging)
- All types of ATPG (Basic ATPG, Fast and Full Sequential)

- STIL from BSDC
- All existing DFT structures (e.g., normal scan, adaptive scan, PLL including on-chip clocking, shadow registers, differential pads, lockup latches, shared scan-in ...)

MAX Testbench does not support DBIST/XDBIST or core integration.

Adaptive scan designs run in parallel mode only when translating from a parallel STIL format written from TestMAX ATPG. Likewise, for serial mode, adaptive scan designs run only when translating from a serial STIL format written from TestMAX ATPG.

Installation

The command setup and usage for MAX Testbench is as follows:

```
alias stil2Verilog 'setenv SYNOPSIS /install_area/latest; $SYNOPSIS/  
platform/syn/bin/stil2Verilog'
```

Then execute the following:

```
stil2Verilog -help
```

Obtaining Help

To access help information, specify the `-help` option on the tool command line. This command will print the description of all options.

There is no specific man page for each error or warning. The messages that are printed if errors occur are clear enough to enable you to adjust the command line to continue.

See Also

- [Writing ATPG Patterns](#)

Running MAX Testbench

You can run the MAX Testbench using either the `write_testbench` command or the `stil2Verilog` command. The `write_testbench` command enables you to run MAX Testbench without leaving the TestMAX ATPG environment, and the `stil2Verilog` command is a standalone executable.

The MAX Testbench flow consists of the following basic steps:

1. Use TestMAX ATPG to write a STIL pattern file.

```
TEST-T> write_patterns STIL_pat_file -format STIL
```

For details on using the `write_patterns` command, see [Writing ATPG Patterns](#).

2. Specify the `write_testbench` or `stil2Verilog` command using the STIL pattern file generated from the `write_patterns` command.

Examples:

```
% write_testbench -input stil_pattern_file.stil \ -output  
Verilog_testbench.v
```

```
% stil2Verilog stil_pattern_file.stil Verilog_testbench.v
```

Two files are generated:

- The first file is the Verilog principal file, which uses the following convention:
Verilog_Testbench_filename.v.
- The second generated file is a data file named
Verilog_Testbench_filename.dat.

An example of the output printed after running the `stil2Verilog` command is as follows:

```
##### # # STIL2VERILOG  
# # # Copyright (c) 2007-2014 SYNOPSYS INC. ALL RIGHTS RESERVED  
# # #####  
maxtb> Parsing command line... maxtb> Checking for feature  
license... maxtb> Parsing STIL file "comp_usf.stil" ... .. STIL  
version 1.0 ( Design 2005) ... .. Building test model ... ..  
Signals ... .. SignalGroups ... .. Timing ... .. ScanStructures :  
"1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "sccompin0" "sccompin1"  
"sccompout0" "sccompout1" "sccompout2" "sccompout3" "sccompin2"  
"sccompin3" ... .. PatternBurst "ScanCompression_mode" ... ..  
PatternExec "ScanCompression_mode" ... .. ClockStructures  
"ScanCompression_mode": pll_controller ... ..  
CompressorStructures : "test_U_decompressor_ScanCompression_mode"  
"test_U_compressor_ScanCompression_mode" ... .. Procedures  
"ScanCompression_mode": "multiclock_capture" "allclock_capture"  
"allclock_launch" "allclock_launch_capture" "load_unload" ... ..  
MacroDefs "ScanCompression_mode": "test_setup" ... .. Pattern  
block "_pattern_" ... .. Pattern block "_pattern_ref_clk0" ...  
maxtb> Info: Event ForceOff (Z) interpreted as CompareUnknown (X)  
in the event waves of WFT "_multiclock_capture_WFT_" containing  
both compare and force types (I-007) maxtb> STIL file successfully  
interpreted (PatternExec: ""ScanCompression_mode"). maxtb> Total  
test patterns to process 21 maxtb> Detected a Scan Compression mode.  
maxtb> Test data file "comp_usf.dat" generated successfully. maxtb>  
Test bench file "comp_usf.v" generated successfully. maxtb> Info  
(I-007) occurred 2 times, use -verbose to see all occurrences. maxtb>  
Memory usage: 6.9 Mbytes. CPU usage: 0.079 seconds. maxtb> End.
```

3. Run the simulation.

Invoke the VCS simulator using the following command line:

```
% vcs Verilog_testbench_file design_netlist \ -v design_library
```

Note the following:

- When running zero-delay simulations, you must use the `+delay_mode_zero` and `+tetramax` arguments.

See Also

- [Configuring MAX Testbench](#)
- [Predefined Verilog Options](#)

write_testbench Command Syntax

The syntax for the `write_testbench` command is as follows:

```
write_testbench -input [stil_filename | {-split_in \
{list_of_stil_files_for_split_in\}}] -output testbench_name
[-generic_testbench] [-patterns_only] [-replace] [-config_file
config_filename] [-parameters {list_of_parameters}]
```

The options are described as follows:

-input [*stil_filename* | {-split_in \ {*list_of_stil_files_for_split_in*\}}]

The *stil_filename* argument specifies the path name of the previous TestMAX ATPG-generated STIL file requested by the equivalent Verilog testbench. You can use a previously generated STIL file as input. This file can originate from either the current session or from an older session using the `write_patterns` command.

The following syntax is used for specifying split STIL pattern files as input (note that backslashes are required to escape the extra set of curly braces):

```
{-split_in \{list_of_stil_files_for_split_in\}}
```

The following example shows how to specify a set of split STIL pattern files:

```
write_testbench -input {-split_in \{patterns_0.stil
patterns_1.stil\}} -output pat_mxtb
```

-output *testbench_name*

Specifies the names used for the generated Verilog testbench output files. Files are created using the naming convention `<testbench_name>.v` and `<testbench_name>.dat`.

-generic_testbench

Provides special memory allocation for runtime programmability. Used in the first pass of the runtime programmability flow, this option is required because the Verilog 95 and 2001 formats use static memory allocation to enable buffers and arrays to store and manipulate .dat file information. For more information on using this command, see [Runtime Programmability](#).

-patterns_only

Used in the second pass, or later, run of the runtime programmability flow, this option initiates a light processing task that merges the new test data in the test data file. This option also enables additional internal instructions to be generated for the special test data file. For more information on using this command, see [Runtime Programmability](#).

-replace

Forces the new output files to replace any existing output files. The default is to not allow a replacement.

-config_file *config_filename*

Specifies the name of a configuration file that contains a list of customized options to the MAX Testbench command line. See "Customized MAX Testbench Parameters Used in a Configuration File with the write_testbench Command" for a complete list of options that can be used in the configuration file. You can use a configuration file template located at \$SYNOPTSYS/auxx/syn/ltran.

-parameters {*list_of_parameters*}

Enables you to specify additional options to the MAX Testbench command line. See "MAX Testbench Command-Line Parameters Used with the write_testbench Command" for a complete list of parameters you can use with the `-parameters` option.

If you use the `-parameters` option, make sure it is the last specified argument in the command line, otherwise you might encounter some Tcl UI conversion limitations.

A usage example for this option is as follows:

```
write_testbench -parameters { -v_file \"design_file_names\" -  
v_lib \"library_file_names\" -tb_module module_name -config_file  
config1}
```


Note the following:

- All the parameters must be specified using the Tcl syntax required in the TMAX shell. For example: `-parameters {param1 param2 -param3 \"param4\"}`
- quotation marks must have a backslash, as required by Tcl syntax, to be interpreted correctly and passed directly to the MAX Testbench command line.
- Parameters specified within a `-parameters {}` list are order-dependent. They are parsed in the order in which they are specified, and are transmitted directly to the MAX Testbench command line. These parameters must follow the order and syntax required for the MAX Testbench command line.

MAX Testbench Command-Line Parameters Used With the write_testbench Command

You can use the `-parameters` option of the `write_testbench` command to specify a list of customized configuration parameters for running MAX Testbench in the TestMAX ATPG environment.

The syntax for the `-parameters` option is as follows:

```
write_testbench -parameters {list_of_parameters}
```

A usage example for this option is as follows:

```
write_testbench -parameters { -v_file \"design_file_names\" -v_lib
\"library_file_names\" -tb_module module_name -config_file config1}
```

Note the following:

All the parameters must be specified using the Tcl syntax required in the TMAX shell. For example: `-parameters {param1 param2 -param3 \"param4\"}`

Quotation marks must have a backslash, as required by Tcl syntax, to be interpreted correctly and passed directly to the MAX Testbench command line.

Parameters specified within a `-parameters {}` list are order-dependent. They are parsed in the order in which they are specified, and are transmitted directly to the MAX Testbench command line. These parameters must follow the order and syntax required for the MAX Testbench command line.

The parameters you can specify are as follows:

```
-config_file TB_config_file
-first d
-generate_config config_file_template
```

Chapter 32: Using MAX Testbench

MAX Testbench Command-Line Parameters Used With the write_testbench Command

```

-generic_testbench
-help [msg_code]
-last d
-log log_file
-parallel
-patterns_only
-replace
-report
-run_mode (go-nogo) | diagnosis
-sdf_file sdf_file_name
-serial
-ser_only
-shell
-sim_script [ [vcs] | [mti] | [nc] | [xl] ]
-split_in {1.stil, 2.stil...} | "dir1/*.stil" testbench_name
-split_out pat_intervstil_filetestbench_name
-tb_format (v95) | v01 | sv
-tb_module module_name
-verbose
-version
-v_file "design_file_names"
-v_lib "library_file_names"
-verdi

```

Note that all options can be abbreviated. For example, you can abbreviate the `-generate_config`, as `-generate`, or `-gen`.

```
-config_file TB_config_file
```

MAX Testbench can be configured at several levels. At the top of the MAX Testbench configuration file, you can edit the `set cfg_*` variables to define the various testbench defaults, such as the progress message interval time and the

simulation time unit. The second half of the configuration file contains a set of editable setup parameters for the VCS/MIT/Cadence simulation script file.. The `TB_config_file` parameter specifies the name of the configuration file used to set up the testbench at generation time. See [Example of the Configuration Template](#).

`-first d`

Specifies the first pattern number that TestMAX ATPG writes. The default is to begin with pattern 0. For Full-Sequential patterns, this option might cause simulation mismatches.

`-generate_config config_file_template`

MAX Testbench can generate a configuration file template that you can edit and modify. The `config_file_template` parameter specifies the path where the configuration file template is written.

`-generic_testbench`

Provides special memory allocation for runtime programmability. Used in the first pass of the runtime programmability flow, this option is required because the Verilog 95 and 2001 formats use static memory allocation to enable buffers and arrays to store and manipulate .dat file information. For more information on using this option, see "[MAX Testbench Runtime Programmability](#)."

`-help [msg_code]`

Shows all possible options, displays the complete `stil2verilog` syntax, and exits. If `msg_code` is specified, then prints the help page corresponding to that code `msg_code` syntax: '1-letter'-'3-digit code' where letter can be 'E', 'W' or 'I' and the 3-digit code must correspond to a valid code in the range [000-999] For example: E-001, W-010

`-last d`

Specifies the last pattern number for the patterns to be written. The default is to end with the last available pattern.

`-log`

Generates a log file.

`-parallel`

This option specifies the parallel load mode for simulation, which is the default.

`-patterns_only`

Used in the second pass, or later, run of the runtime programmability flow, this option initiates a light processing task that merges the new test data in the test data file. This option also enables additional internal instructions to be generated

for the special test data file. For more information on using this option, see "[MAX Testbench Runtime Programmability](#)."

`-replace`

Forces MAX Testbench to overwrite the testbench files, the configuration file template, and simulation script.

`-report`

Displays the configuration setting and test pattern information. It has the following parameters (note that multiple parameters can be specified if separated by commas):

`all` — displays all the information (default in verbose mode)

`config` — displays the configuration setting

`dft` — displays DFT structure information

`drc` — displays DRC warnings

`flow` — displays STIL pattern flow

`macro` — displays macro information

`nb_patterns` — displays the total number of patterns to be executed

`proc` — displays procedure information

`sig` — displays all the signal information

`sig_groups` — displays all the signal groups information

`wft` — displays waveformtable information

`-run_mode go-nogo | diagnosis`

Allows the targeting of either "go-nogo" mode or diagnosis mode; "go-nogo" mode is the default. For details, see [Setting the Run Mode](#).

`-sdf_file sdf_file_name`

Specifies the SDF file name used for back annotation.

`-serial`

The serial option states that a serial scan simulation is required. The default simulation scan load is parallel. The same behavior can be obtained by using the `+define+tmx_serial` compiler directive to force the simulation of all patterns to be serial. If `+tmx_serial=N` is used, MAX Testbench forces serial simulation of the first `N` patterns, and then starts parallel simulation of the remaining patterns

`-ser_only`

Allows a reduction in the size of the testbench and limits the simulation to serial only mode.

`-shell`

Runs MAX Testbench in shell mode. (Note that this option is not yet supported.)

`-sim_script vcs | mti | nc | xl`

The `-sim_script` option specifies a simulation script to be generated together with the testbench file. You also must provide the `v_file` and `v_lib` options. Note that only VCS scripts are supported; the other simulator scripts that are generated conform to the simulator script generated by TMAX (`write_patterns` command). The argument specifies the target simulator:

`vcs` — VCS simulator command shell script

`mti` — ModelSim simulator command shell script

`xl` — Cadence XL simulator command shell script

`nc` — Cadence NCVerilog simulator command shell script

Note the specification of several arguments at the same time to target all of the simulators is supported as repetitive entries "`-sim_script vcs -sim_script mti -sim_script xl`"

The output name of the generated script file is:

`<name_of_testbench_file>_<simulator>.sh.`

`-split_in {1.stil, 2.stil...} | "dir1/*.*stil" testbench_name`

Specifies that MAX Testbench uses split STIL files based on either a detailed list of STIL files or a generic list description using the wildcard (*) symbol. In the generic list format, the files are recognized in alphabetical order. For more information on this option, see [Using Split STIL Pattern Files](#).

`-split_out pat_interval stil_file testbench_name`

Specifies that MAX Testbench splits STIL files. The `pat_interval` argument specifies the maximum number of patterns that a given .dat file will contain. For more information on this option, see [Splitting Large STIL Files](#).

`-tb_format v95 | v01 | sv`

Specifies the test bench format applied to the `-tbench_file>` specification. The default is `v95`. The options and formats include:

`v95` Verilog 1995

v01 Verilog 2001

sv SystemVerilog

-tb_module *module_name*

Specifies the module name for the top-level module of the Verilog testbench.

-verbose

Activates verbose mode.

-version

Prints the stil2verilog banner, including the version.

-v_file "*design_file_names*"

Specifies design netlist source files (the DUT description) required to run the simulation. This option is required when using the `sim_script` option. Wildcard characters are supported. Note that `design_file_name1` and `design_file_nameN` must be separated with spaces. Quotation marks are required for more than one file name.

-v_lib "*library_file_names*"

Specifies the library file (the DUT related technology library) required to run the simulation. This option is required when using `sim_script` option. Note that `library_file_name1` and `library_file_nameN` must be separated with spaces. Quotation marks required for more than one file name. Supports wildcard characters for easy simulation script generation.

-verdi

Allows to automatically generate a precomputed waveform signal window.

MAX TestBench generates the following files:

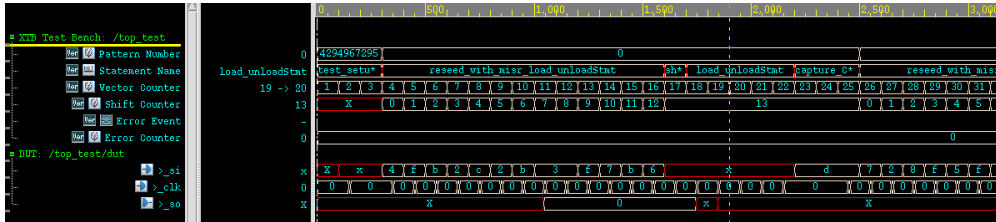
1. `<tb_name>_verdi.play`: TCL Verdi play script
2. `<tb_name>_verdi.signals`: Verdi waveform signal file.

The simulation generates the `<tb_name>.fsdb` file

For example when using `write_testbench -input pat.stil -output xtb -parameters { -verdi }` the output reports the following:

```
maxtb> Test data file "xtb.dat" generated successfully.
maxtb> Test bench file "xtb.v" generated successfully.
maxtb> Verdi TCL script file "xtb_verdi.play" generated
      successfully.
maxtb> Verdi waveform signal file "xtb_verdi.signals" generated
      successfully.
```

```
maxtb> VCS options to generate the FSDB file: "+define+txmax_fsdb
-lca -kdb -debug_access+all"
maxtb> Verdi command line: "verdi -ssf xtb.fsdb -play
xtb_verdi.play"
```



Note:

The `>_si`, `>_clk` and `>_so` are virtual buses that can be expanded.

See Also

- [write_testbench](#)
- [How Do I Use the write_testbench Command to Customize MAX Testbench Output?](#)

stil2Verilog Command Syntax

The basic syntax for the `stil2Verilog` command is as follows:

```
stil2Verilog [pattern_file] [tbench_file] [options]
```

The syntax descriptions are as follows:

`pattern_file`

Specifies the ATPG-generated STIL pattern file used as input. This file must be specified, except when the `-split_in` option is used.

`tbench_file`

Specifies the name of the testbench file to generate. When the `tb_file_name` is specified, a `.v` extension is added when generated the protocol file, and a `.dat` extension is used when generating the test data file. You should use only the root name with the command line, for example, `stil2erilogpat.stil tbench`, that generates `tbench.v` and `tbench.dat` files in the current working directory.

This argument is optional when the `-generate_config` or `-report` options are specified.

Other optional arguments can be specified, as shown in the following syntax. The defaults are shown in bold enclosed between parentheses.

```
-config_file TB_config_file
```

Chapter 32: Using MAX Testbench stil2Verilog Command Syntax

```
-first d
-force_enhanced_debug
-generate_config config_file_template
-generic_testbench
-help [msg_code]
-last d
-log log_file
-parallel
-patterns_only
-replace
-report
-run_mode (go-nogo) | diagnosis
-sdf_file sdf_file_name
-serial
-ser_only
-sim_script <= [ [vcs] | [mti] | [nc] | [ xl] ]
-split_in { 1.stil, 2.stil... } | { dir1 /*.stil } testbench_name
-split_out pat_intervstil_filetestbench_name
-tb_format <= (v95) | v01 | sv
-tb_module module_name
-verbose
-version
-v_file { design_file_names }
-v_lib { library_file_names }
-verdi
```


The descriptions for the optional syntax items are as follows:

`-config_file TB_config_file`

MAX Testbench can be configured at several levels. At the top of the MAX Testbench configuration file, you can edit the `set cfg_*` variables to define the various testbench defaults, such as the progress message interval time and the simulation time unit. The second half of the configuration file contains a set of editable setup parameters for the VCS/MIT/Cadence simulation script file. The `TB_config_file` parameter specifies the name of the configuration file used to set up the testbench at generation time. See [Example of the Configuration Template](#).

`-first d`

Specifies the first pattern number that TestMAX ATPG writes. The default is to begin with pattern 0. For Full-Sequential patterns, this option might cause simulation mismatches.

`-force_enhanced_debug`

Forces MAX Testbench to halt if any errors are encountered when processing the parallel strobe data (PSD) file. The default is to not force MAX Testbench to stop. For more information on the PSD file, see [Understanding the PSD File](#).

`-generate_config config_file_template`

MAX Testbench can generate a configuration file template that you can edit and modify. The `config_file_template` parameter specifies the path where the configuration file template is written.

`-generic_testbench (or -streaming_patterns)`

Generates a generic testbench that can load future test patterns (.dat files) without recompiling. For more information on using this command, see [Runtime Programmability](#).

`-help [msg_code]`

Shows all possible options, and the complete `stil2Verilog` syntax and exits. If `msg_code` is specified, then prints the help page corresponding to that code `msg_code` syntax: '1-letter'-'3-digit code' where letter can be 'E', 'W' or 'I' and the 3-digit code must correspond to a valid code in the range [000-999] For example: E-001, W-010

`-last d`

Specifies the last pattern number for the patterns to be written. The default is to end with the last available pattern.

`-log`

Generates a log file.

`-parallel`

Specifies the parallel load mode for simulation, which is the default.

`-patterns_only`

Generates test patterns only (.dat file) to be used with an existing equivalent testbench (.v file). For more information on using this command, see [Runtime Programmability](#).

`-replace`

Forces MAX Testbench to overwrite the testbench files, the configuration file template, and simulation script.

`-report`

Displays the configuration setting and test pattern information. It has the following parameters (note that multiple parameters can be specified if separated by commas):

`all` — displays all the information (default in verbose mode)

`config` — displays the configuration setting

`dft` — displays DFT structure information

`drc` — displays DRC warnings

`flow` — displays STIL pattern flow

`macro` — displays macro information

`nb_patterns` — displays the total number of patterns to be executed

`proc` — displays procedure information

`sig` — displays all the signal information

`sig_groups` — displays all the signal groups information

`wft` — displays WaveformTable information

`-run_mode go-nogo | diagnosis`

Allows the targeting of either Go-nogo mode (the default) or diagnosis mode. For details, see [Setting the Run Mode](#).

`-sdf_file sdf_file_name`

Specifies the SDF file name used for back annotation.

`-serial`

Specifies the serial load mode simulation. The default simulation scan load is parallel. The same behavior can be obtained by using the `+define +tmax_serial` compiler directive to force the simulation of all patterns to be serial. If `+tmax_serial=N` is used, MAX Testbench forces serial simulation of the first `N` patterns, and then starts parallel simulation of the remaining patterns

`-ser_only`

Generates the testbench file for serial load mode only. This allows a reduction in the size of the testbench and speeds up the simulation.

`-shell`

Runs the tool in shell mode.

`-sim_script vcs | mti | nc | xl`

The `sim_script <simulator>` option specifies a simulation script to be generated together with the testbench file. You also must provide the `v_file` and `v_lib` options. Note that only VCS scripts are supported; the other simulator scripts that are generated conform to the simulator script generated by TMAX (`write_patterns` command). The argument specifies the target simulator:

- `vcs` — VCS simulator command shell script
- `mti` — ModelSim simulator command shell script
- `xl` — Cadence XL simulator command shell script
- `nc` — Cadence NCVerilog simulator command shell script

Note the specification of several arguments at the same time to target all of the simulators is supported as repetitive entries "`-sim_script vcs -sim_script mti -sim_script xl`"

The output name of the generated script file is:

`<name_of_testbench_file>_<simulator>.sh.`

`-split_in { 1.stil, 2.stil... } | { dir1 /*.stil }`

Specifies MAX Testbench to use split STIL files based on either a detailed list of STIL files or a generic list description using the wildcard (*) symbol. In the generic list format, the files are recognized in alphabetical order. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket, as shown in the following example:

```
stil2Verilog -split_in { bill.patt.stil.ts_and_chain
  bill.patt_0.stil bill.patt_1.stil bill.patt_2.stil
```

```
bill.patt_3.stil bill.patt_4.stil bill.patt_5.stil  
bill.patt_6.stil bill.patt_7.stil bill.patt_8.stil  
bill.patt_9.stil bill.patt_10.stil } bill.pat_stil.v -replace
```

Note that you can also specify multiple files in the configuration file. For more information on this option, see [Using Split STIL Pattern Files](#).

`-split_out pat_intervalstil_file`

Specifies MAX Testbench to split STIL files. The `pat_interval` argument specifies the maximum number of patterns that a given .dat file will contain. For more information on this option, see [Splitting Large STIL Files](#).

`-tb_format v95 | v01 | sv`

Specifies the testbench format applied to the `tbench_file` specification. The default is `v95`, and is currently the only supported option. Formats:

`v95` — Verilog 1995

`v01` — Verilog 2001

`sv` — SystemVerilog

`-tb_module module_name`

Specifies the module name for the top-level module of the Verilog testbench.

`-verbose`

Activates verbose mode.

`-version`

Prints the stil2Verilog banner, including the version.

`-v_file { design_file_names }`

Specifies design netlist source files (the DUT description) required to run the simulation. It is required when using the `sim_script` option. Wild characters are supported. Note that `design_file_name1` and `design_file_nameN` must be separated with spaces. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket (you can also specify multiple files in the configuration file).

`-v_lib { library_file_names }`

Specifies the library file (the DUT related technology library) required to run the simulation. This option is required when using `sim_script` option. Note that `library_file_name1` and `library_file_nameN` must be separated with

spaces. Multiple file names must be enclosed in curly brackets with spaces on both sides of each bracket, as shown in the following example:

```
stil2Verilog pats.stil maxtb -replace -v_lib { lib1.v lib2.v }
```

Note that you can also specify multiple files in the configuration file. Wildcard characters are supported for simulation script generation.

`-verdi`

Allows to automatically generate a precomputed waveform signal window.

stil2verilog generates the following files:

1. `<tb_name>_verdi.play`: TCL Verdi play script
2. `<tb_name>_verdi.signals`: Verdi waveform signal file.

The simulation generates the `<tb_name>.fsdb` file.

Setting the Run Mode

There are two basic run modes you can set when starting MAX Testbench using the `stil2Verilog` command: Go-nogo and Diagnosis.

The Go-nogo mode is set using the `-run_mode go-nogo` option. In this mode, MAX Testbench does the following:

- Sets the verbosity level to 0 (equivalent to using `+define+tmax_msg=0` at VCS compilation time)
- Makes the testbench reporting the beginning of each 5 patterns (equivalent to using `+define+tmax_rpt=5` at VCS compilation time)
- Initializes the file name for the collection of diagnostics failures to `<testbench_name>.diag`.

The Diagnosis mode is set using the `-run_mode diagnosis` option. In this mode, MAX Testbench saves the mismatches in the `<testbench_name>.diag` file in a pattern-based format compatible with the TestMAX ATPG `run_diagnosis` command.

For example, the mismatches are recorded in the following manner:

```
30 test_so2 10 (exp=0, got=1) // chain , V=313, T=31240.00 ns
30 test_so3 10 (exp=0, got=1) // chain , V=313, T=31240.00 ns
30 test_so4 10 (exp=0, got=1) // chain , V=313, T=31240.00 ns
```

These failures can be used by the TestMAX ATPG diagnostics to identify the failing scan chain. You can print a report using the command `run_diagnosis -only_report_failures`.

The failures log file name default can be changed at the time the simulation is executed by using the following compiler directive:

```
% vcs ... +define+tmax_diag_file=\"<file_name>\"
```

The default can also be changed at the time the testbench is generated using the configuration file parameter `cfg_diag_file`.

See Also

- [Understanding the Failures File](#)
- [Using the Failures File](#)

Configuring MAX Testbench

You can specify options for running MAX Testbench using either a configuration file or a set of predefined simulator script options. The following table describes the MAX Testbench configuration options.

Table 9 MAX Testbench Configuration Options

Configuration type	Configuration file option	Simulator predefined option
Predefined Verilog code included in the simulator script. Specifies the initial number of serial (flattened scan) vectors.	<p>Syntax:</p> <pre>set define_u ser_def N</pre> <p>Example:</p> <pre>set define_t max_ser_i al 0</pre>	<p>Syntax:</p> <pre>:</pre> <pre>+tmax _ser_i al OR tmax_ ser_i l=N</pre> <p>Example:</p> <pre>+defi ne+tm ax_se rial= 0</pre>

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Predefined Verilog code included in the simulator script. Specifies the parallel scan access with <i>N</i> serial vectors.	<p>Syntax:</p> <pre>set define_u +tmax ser_def _para N llel=</pre> <p>Example:</p> <pre>set define_t max_para +defi llel 0 ne+tm</pre>	<p>Syntax:</p> <pre>:</pre> <p>Example:</p> <pre>Examp le: ax_pa ralle l=0</pre>
Predefined Verilog code included in the simulator script. Specifies the number of patterns to simulate.	<p>Syntax:</p> <pre>set define_u +tmax ser_def _n_pa N ttern</pre> <p>Example:</p> <pre>set define_t max_n_pa +defi ttern_si ne+tm m 10 ax_n_ patte rn_si m=10</pre>	<p>Syntax:</p> <pre>:</pre> <p>Example:</p> <pre>Examp le: ax_n_ patte rn_si m=10</pre>
Predefined Verilog code included in the simulator script. Generates a delay (a "dead period") for parallel scan access to align parallel load timing with serial load timing.	<p>Syntax:</p> <pre>set define_t +defi max_seri ne+tm al_timin ax_se rial_ g tmin</pre> <p>Example:</p> <pre>set define_t max_seri al_timin g</pre>	<p>Syntax:</p> <pre>:</pre> <p>Example:</p> <pre>g</pre>

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Sets the top-level module.	Syntax: <pre>set tb_modul e_name "new_nam e" Example: set tb_modul e_name "top1"</pre>	N/A
Sets the TestMAX ATPG DRC severity level. The <code>drcw_severity</code> option requires two parameters: <i>rule_name</i> : TestMAX ATPG rule name (the wild-card character '*' is supported) <i>severity</i> : severity level ("ignore" "warning" "error")	Syntax: <pre>set drcw_sev erity rule_nam e severity Example: set drcw_sev erity C11 warning</pre>	N/A
Extends the size optimization and generates an extended testbench. To create a compact testbench, set this option to 1.	Syntax: <pre>set cfg_tb_f ormat_ex tended N Example: set cfg_tb_f ormat_ex tended 1</pre>	N/A

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Specifies the maximum number of patterns that can be simultaneously loaded during the simulation.	Syntax: <pre>set cfg_patt erns_rea d_interv al N</pre> Example: <pre>set cfg_patt erns_rea d_interv al 1</pre>	N/A
Specifies the interval for reporting a simulation progress message (0 is disabled; <i>N</i> is every <i>M</i> th pattern reported) .	Syntax: <pre>set cfg_patt erns_rep ort_inte rval N</pre> Example: <pre>set cfg_patt erns_rep ort_inte rval 3</pre>	Syntax: : +tmax _rpt= N Exam ple: +tmax _rpt= 3
Defines the verbosity output level for MAX Testbench:	Syntax: <pre>set cfg_mess age_verb osity_le vel N</pre> Example: <pre>set cfg_mess age_verb osity_le vel 2</pre>	Syntax: : +tmax _msg= N Exam ple: +tmax _msg= 3
0 , 1 , 2 , 3 and 4 .		

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Generates an extended VCD file of the simulation run	<p>Syntax:</p> <pre>set cfg_evcd _file "evcd_fi le"</pre> <p>Example:</p> <pre>set cfg_evcd _file "run1.ev cd"</pre>	N/A
Changes the default name of the failure log file when the simulation is executed. This option overrides the failure log file name specified in the testbench file and affects the simulation runtime.	<p>Syntax:</p> <pre>set cfg_diag _file "diag_fi le"</pre> <p>Example:</p> <pre>set cfg_diag _file "failure _2"</pre>	<p>Syntax:</p> <pre>:</pre> <pre>+tmax _diag _file ="dia g_fil e"</pre> <p>Examp le:</p> <pre>+tmax _diag _file =" failu re_1"</pre>
Displays annotation statements in the main pattern block during simulation: 0-disabled, 1-pattern adjacent statements only, 2-all annotation statements.	<p>Syntax:</p> <pre>set cfg_disp lay_ann_ stmts number</pre> <p>Example:</p> <pre>set cfg_disp lay_ann_ stmts 1</pre>	N/A

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Configures a miscompare in pattern-based ($N=1$) format or cycle-based ($N=2$) format	N/A	Syntax: : +tmax _diag =N Example: +tmax _diag =2
Sets the patterns per PSD partition file for the CPV flow. The default is 1000.	Syntax:	N/A
	<pre>set cfg_nb_p atterns_ per_psd_ file N</pre>	
	Example:	
	<pre>set cfg_nb_p atterns_ per_psd_ file 10</pre>	
Specifies the simulation time unit (time scale)	Syntax:	N/A
	<pre>set cfg_time _unit "N"</pre>	
	Example:	
	<pre>set cfg_time _unit "1ps"</pre>	

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
Specifies the simulation time precision in units	<p>Syntax:</p> <pre>set cfg_time _precision "unit"</pre> <p>Example:</p> <pre>set cfg_time _precision "1ns"</pre>	N/A
Defines the DUT Module name. This option is not required unless prompted by the tool.	<p>Syntax:</p> <pre>set cfg_dut_ module_n ame "name"</pre> <p>Example:</p> <pre>set cfg_dut_ module_n ame "module_ 1"</pre>	N/A
Delays (or advances, if <i>N</i> is set to a negative value) the release time of the parallel shift starting at the beginning of the next cycle. The default is 0, which means the release time is applied to the end of the current cycle. A negative delay is not supported when using a DFTMAX serializer design with a clock controller. The units must be included in the specification.	<p>Syntax:</p> <pre>set cfg_para lled_rel ease_tim e N</pre> <p>Example:</p> <pre>set cfg_para lled_rel ease_tim e 5.00ns</pre>	N/A

Table 9 MAX Testbench Configuration Options (Continued)

Configuration type	Configur ation file option	Simul ator predef ined option
<p>Reports the instance name of the failing cells during the simulation of a parallel-formatted STIL file. To enable the report, you must set the boolean variable to '1'. The default, 0, turns off this reporting. This feature affects simulation memory consumption.</p>	<p>Syntax:</p> <pre>set cfg_parallel_stil_report_cell_name N</pre> <p>Example:</p> <pre>set cfg_parallel_stil_report_cell_name 1</pre>	<p>N/A</p>
<p>Reverts the order in which all ports created using the <code>add_net_connections</code> command are connected to the design under test (DUT). This option is enabled when set to 1 (the default is 0).</p>	<p>Syntax:</p> <pre>set cfg_add_net_connection_revert_order N</pre> <p>Example:</p> <pre>set cfg_add_net_connection_revert_order 1</pre>	<p>N/A</p>
<p>Reverses the name of the specified port name connected to the DUT. This configuration option takes precedence over the <code>cfg_add_net_connection_revert_order</code> command.</p>	<p>Syntax:</p> <pre>set cfg_reverse_bus_order "port_name"</pre> <p>Example:</p> <pre>set cfg_reverse_bus_order "t"</pre>	<p>N/A</p>

Note the following:

- The "Configuration File Option" column describes variables for the configuration file defined by the `-config_file file_name` option of the `stil2Verilog` command.
- The "Simulator Predefined Option" column describes options used in a simulator script or defined in configuration file specified by the `-config_file file_name` option of the `stil2Verilog` command (see the following example).

Use the following compiler directive to change the time of the simulation execution.

```
% vcs ... +define+tmx_serial=1
```

- In the first three rows of Table 1, the `define_user_def` syntax is used for user-defined simulator variables. This syntax is also used for variables that are hard-coded into the testbench file, such as `tmx_serial` and `tmx_parallel`.

You can change the default for the `define_user_def` option when the testbench is generated using the `-sim_script vcs|mti|xl|nc` option and the `-config_file file_name` option. To change the default, specify the `"set define_tmx_serial=1"` option in the configuration file.

Understanding the Failures File

When you set the `-run_mode diagnosis` option of the `stil2Verilog` command, MAX Testbench prints all miscompare messages to a file used with the `run_diagnosis` command for diagnostics. The format of this file is dependent of the pattern type (legacy scan, DFTMAX compression, or serializer), the simulation mode (serial or parallel), and the STIL type (dual or unified).

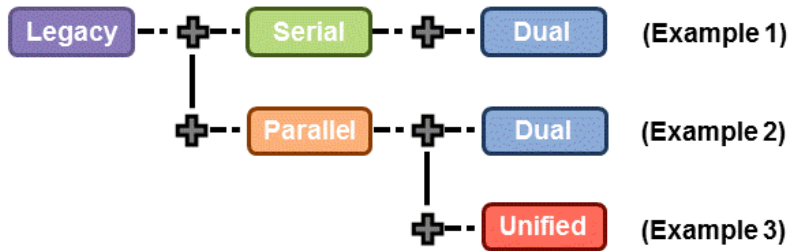
The following sections describe the relationship of the failures formats for each pattern type:

- [Legacy Scan Failures](#)
- [DFTMAX Compression Failures](#)
- [Serializer Scan Failures](#)

MAX Testbench and Legacy Scan Failures

In legacy scan, given the serial/parallel and dual/unified types, the failure formats are the same. A failure contains the cycle count of the failure (`v=`), the expected data (`exp=`), the data captured (`got=`), the chain name (`chain`), the scan output pin name (`pin`), and the scan cell position (`scan cell`). The following figure describes the relationship of the failures for legacy scan.

Figure 1 Relationship of Failures Format for Legacy Scan



The following examples are reports for the same failure printed during the simulation of the patterns.

Example 1

```
Error during scan pattern 32 (detected during unload of pattern 31)
At T=49240.00 ns, V=493, exp=0, got=1, chain 4, pin test_so4, scan cell
10
```

Example 2

```
Error during scan pattern 32 (detected during parallel unload of pattern
31)
At T=16240.00 ns, V=163, exp=0, got=1, chain 4, pin test_so4, scan cell
10
```

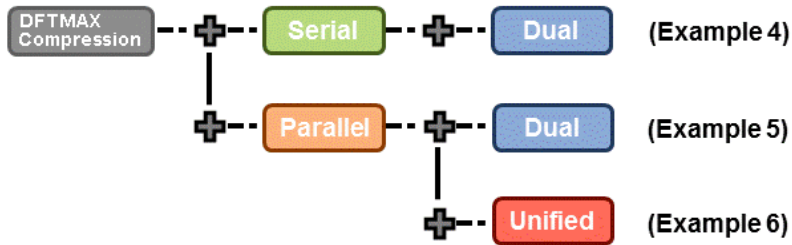
Example 3

```
Error during scan pattern 32 (detected during parallel unload of pattern
31)
At T=16240.00 ns, V=163, exp=0, got=1, chain 4, pin test_so4, scan cell
10
```

MAX Testbench and DFTMAX Compression Failures

The failure formats are different when using DFTMAX compression. A failure contains the cycle count of the failure (*v=*), the expected data (*exp=*), the data captured (*got=*), the chain name (*chain*) only for dual STIL flow parallel, the scan output pin name (*pin*) for dual STIL flow serial mode and unified STIL flow parallel mode. The pin information for dual STIL flow for parallel mode is the pin pathname of the failing scan cell output. The report also contains the scan cell position (*scan cell*).

Figure 2 Relationship of Failures Format for DFTMAX Compression



The following examples are reports for the same failure printed during the simulation of the patterns.

Example 4

```
Error during scan pattern 31 (detected during unload of pattern 30)
At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so2, scan cell 10
At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so3, scan cell 10
At T=31240.00 ns, V=313, exp=0, got=1, chain , pin test_so4, scan cell 10
```

Example 5

```
Error during scan pattern 31 (detected during parallel unload of pattern
30)
At T=15740.00 ns, V=158, exp=0, got=1, chain 10, pin
snps_micro.mic0.pc0.prog_counter_q_reg[11] .QN, scan cell 10
```

In the case of dual STIL flow parallel mode for DFTMAX compression patterns, MAX Testbench, reports the failing scan chain and failing scan cell position. But, for performance reasons, the scan cell instance name for the failing position is not reported. However, it does report the scan cell instance name with position 0 for the failing scan chain.

Example 6

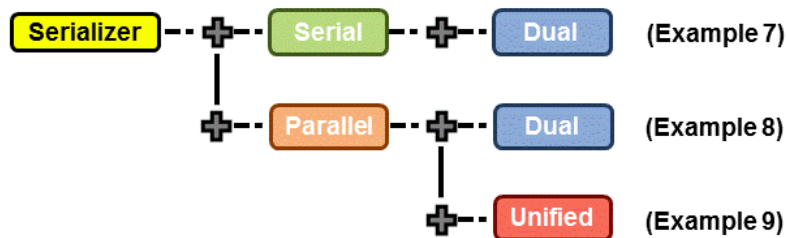
```
Error during scan pattern 31 (detected during parallel unload of pattern
30)
Error during scan pattern 31 (detected during parallel unload of pattern
30)
At T=15740.00 ns, V=158, exp=0, got=1, pin test_so3, scan cell 10
Error during scan pattern 31 (detected during parallel unload of pattern
30)
At T=15740.00 ns, V=158, exp=0, got=1, pin test_so4, scan cell 10
```


In the case of Unified STIL flow parallel mode for DFTMAX compression patterns, MAX Testbench reports the failing scan cell position only. The failing scan chain name and the failing scan cell instance name are not provided. You can use TestMAX ATPG diagnostics to retrieve the failing scan chain name.

MAX Testbench and Serializer Scan Failures

The following figure describes the relationship of serializer scan failures.

Figure 3 Relationship of Failures Format for Serializer



Example 7

Error during scan pattern 5 (detected during unload of pattern 4)

```
At T=28340.00 ns, V=284, exp=0, got=1, chain , pin test_sol, scan cell 2,
serializer index 1
```

```
At T=28440.00 ns, V=285, exp=0, got=1, chain , pin test_sol, scan cell 2,
serializer index 2
```

```
At T=28540.00 ns, V=286, exp=1, got=0, chain , pin test_sol, scan cell 2,
serializer index 3
```

In the case of the dual STIL flow parallel mode for serializer patterns, MAX Testbench reports the failing scan chain and failing scan cell position. But, for performance reasons, the scan cell instance name for the failing position is not reported. However, it does report the scan cell instance name of position 0 for the failing scan chain.

Example 8

Error during scan pattern 5 (detected during parallel unload of pattern 4)

```
At T=6640.00 ns, V=67, exp=1, got=0, chain 1, pin
snps_micro.mic0.alu0.accu_q_reg[4] .Q, scan cell 2
```

In the case of unified STIL flow parallel mode for serializer patterns, MAX Testbench reports the failing scan cell position only. The failing scan chain and the failing scan cell

instance name are not provided. The failing scan chain name could be retrieved using the diagnostics in TestMAX ATPG.

Example 9

```
Error during scan pattern 5 (detected during unload of pattern 4)

At T=28340.00 ns, V=284, exp=0, got=1, chain , pin test_sol, scan cell 2,
serializer index 1

At T=28440.00 ns, V=285, exp=0, got=1, chain , pin test_sol, scan cell 2,
serializer index 2

At T=28540.00 ns, V=286, exp=1, got=0, chain , pin test_sol, scan cell 2,
serializer index 3
```

Using the Failures File

You can configure and use the failures file printed by MAX Testbench for diagnosis. To use this file, you need to set the `+tmax_diag` option.

By default, the diagnosis file name is `<tbenchname>.diag`. The default names of the diagnosis file when the `-split_out` option is used are `<tbenchname>_0.diag`, `<tbenchname>_1.diag`, and so forth, for the different partitions. You can change the default using the `+tmax_diag_file` option.

The setting `+tmax_diag=1` reports the pattern-based failure format. The setting `+tmax_diag=2` reports the cycle-based failure format.

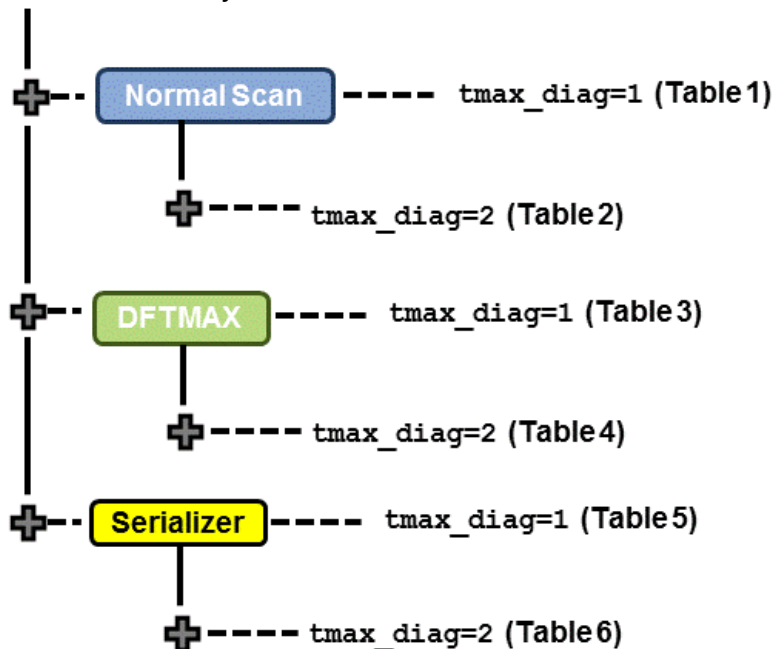
Note the following limitations:

- You cannot run the diagnosis directly if all the partitions are simulated sequentially. This is because the failures are created in separate failure log files. Before running the diagnosis, you must manually append the failure log files into a single file.
- You cannot run the diagnosis if the entire partitions are simulated sequentially and the cycle-based format is used (`+tmax_diag=2`). This is because the recorded cycles are reset for each partition simulation.

Both settings offer a way to generate a failure log file that can be used for a diagnostic if a fault is injected in a circuit and its effect simulated. You can also use these settings to validate the detection of a fault by TestMAX ATPG diagnostics. In addition, they can be used for per-cycle pattern masking or for TestMAX ATPG diagnostics to find the failing scan chain and cell for a unified STIL flow miscompare.

The following figure summarizes the formats and applications possible for failures printed using the `+tmax_diag` option.

Figure 152 Summary of Uses for Failures File



The format names and their descriptions are as follows:

- *Format A* = <pat_num> <pin_name> <shift_cycle> (exp=%b, got=%b)
- *Format B* = <pat_num><chain_name> <cell_index> (exp=%b, got=%b)
- *Format C* = <pat_num><pin_name> (exp=%b, got=%b)
- *Format S* = <pat_num><pat_num> <pin_name> <unload_shift_cycle> <shift_position> (exp=%b, got=%b)
- *Format D* = C <pin_name> <vect_nbr> (exp=<exp state>, got=<got state>)

Note the following:

- The USF and DSF serial simulation modes have the same format and capability. Thus, only the USF parallel is present in the tables. The USF serial is not displayed in the tables.
- The cycle-based format is printed only for serial simulation. This is because the simulation in parallel has less cycles than serial simulation. Thus, the cycles reported by parallel simulation are not valid. If +tmax_diag=2 is used for a parallel simulation mode, the simulation is not stopped, but the testbench automatically changes the +tmax_diag setting to 1. A warning message is also printed in the simulation log. Then, as shown in the following tables, the following statement is printed for all parallel simulation DSF and USF modes: "Not Supported."

The following tables describe the failures file format and their usage in detail.

Figure 153 Failures Format

		Mode
		Unified STIL Parallel
Failure Format For:	Shift	Not Supported
	Capture	Not Supported
Good For Diagnostics		No
Good for Masking		No

Figure 154 Simulation Failures Format and Usage

		Mode
		Unified STIL Parallel
Failure Format for:	Shift	Not Supported
	Capture	Not Supported
Good for Diagnostics		No
Good for Masking		No

Figure 155 Failures Format and Usage for DFTMAX Compression

		Mode
		Unified STIL Parallel
Failure Format for:	Shift	A
	Capture	C
Good for Diagnostics		Yes
Good for Masking		Yes

* Failures are usable for TestMAX ATPG diagnostics provided that the command `set_diagnosis -dftmax_chain_format` is used

Figure 156 Simulation Failures Format and Usage for DFTMAX Compression

		Mode
		Unified STIL Parallel
Failure Format for:	Shift	Not Supported
	Capture	Not Supported
Good for Diagnostics		No
Good for Masking		No

Figure 157 Failures Format and Usage for Serializer

		Mode
		Unified STIL Parallel
Failure Format for:	Shift	S
	Capture	C
Good for Diagnostics		Yes
Good for Masking		Yes

* If the `set_diagnosis -dftmax_chain_format` command is specified, failures can be used for TestMAX ATPG diagnostics.

Figure 158 MAX Testbench Simulation Failures Format and Their Usage for Serializer

		Mode
		Unified STIL Parallel
Failure Format for:	Shift	Not Supported
	Capture	Not Supported
Good for Diagnostics		No
Good for Masking		No

See Also

- [Diagnosing Manufacturing Test Failures](#)

Using Split STIL Pattern Files

You can use the `-split_in` option of the `stil2Verilog` command to specify the use of split STIL pattern files. This option has two different formats:

- The `-split_in { 1.stil, 2.stil... }` format uses split STIL files based on a detailed list of STIL files.
- The `-split_in { dir1/*.stil }` format uses split STIL files based on a generic list description using the wildcard (*) symbol.

Note the following:

- The input STIL files from both the detailed list format and the generic list format are assumed to belong to the same pattern set (split patterns of the same original patterns). Multiple files must be specified within curly brackets, with a space before and after each bracket.
- The input STIL files all have the same test protocol (procedures, signals, WFTs, etc). The only difference between these STIL files is the content of the "Pattern" block, which contains test data. Max Testbench takes the first STIL file it encounters as a representative to the other STIL files and extracts and interprets the protocol information from it.
- You must ensure that the input STIL files correspond to the same split patterns. You must also avoid any form of mixing with other STIL files in the list (using the detailed list format) or mixing within the directory (using the generic list format).

Execution Flow for `-split_in` Option

When the `-split_in` option is specified, the testbench is generated using a single execution. One testbench (.v) file is generated for all STIL files. The number of .dat files directly correlates to the number of input STIL files.

The following example shows a MAX Testbench report:

```
maxtb> Parsing STL procedure file "pat1.stil" ...
maxtb> Parsing STIL data file "pat1.stil, pat2.stil, pat3.stil..."
maxtb> STIL file successfully interpreted (PatternExec: "").
maxtb> Detected a Normal Scan mode.
maxtb> Test bench files " xtb_tbench.v", "xtb_tbench1.dat"...
"xtb_tbench3.dat" generated successfully.
maxtb> Test data file mapping :
pat1.stil ?? xtb_tbench1.dat (patterns <X1> to <Y1>)
pat2.stil ?? xtb_tbench2.dat (patterns <X2> to <Y2>)
pat3.stil ?? xtb_tbench3.dat (patterns <X3> to <Y3>)
```

The header of each .dat file identifies the STIL partition that was used to generate it, as shown in the following example line:

```
// Generated from original STIL file : ./pat1.stil
```

Using this information, you can link various simulations to the original STIL partitions, regardless of the order of the STIL files specified by the `-split_in` option. You can also combine the existing `-sim_script` option with the `-split_in` option to generate a validation script that enables automatic management of the validation step when using different simulation modes.

See Also

- [Reading Multiple Pattern Files](#)

Splitting Large STIL Files

You can use the `-split_out` option of the `stil2Verilog` command to specify MAX Testbench to split large STIL files. For example, for a STIL file with ten patterns, the following command generates one testbench file and three .dat files:

```
stil2Verilog -split_out 4 mypat.stil my_tb
```

The first .dat file contains four patterns (0 to 3), the second .dat file contains four patterns (#4 to #7), and the third .dat file contains two patterns (patterns #8 and #9).

The splitting process is based on a user-specified interval. Therefore, you should avoid splitting between two interdependent patterns.

The following sections describe how to split large STIL files:

- [Why Split Large STIL Files?](#)
- [Executing the Partition Process](#)
- [Example Test](#)

Why Split Large STIL Files?

The ability to split STIL files is useful for two situations:

- When the number of patterns in a .dat file is so large that it cannot be simulated because it exceeds the system memory capacity. For example, to simulate two million patterns, the size of the .dat file contains 24 million lines, which corresponds to all instructions for all patterns. In this case, the simulator (VCS) runs out of memory before completing the simulation.
- Even if the system memory can accommodate the entire simulation, the excessive memory consumption drastically impacts the performance of the simulation. This can occur when the use of memory swapping and memory resources prevent other applications from using that machine.

When it splits the STIL files, MAX Testbench can resolve a completely blocked simulation, or optimize the memory and runtime simulation. This capability also allows MAX Testbench to serially run a set of patterns as if these patterns were split from TestMAX ATPG in different STIL files.

Executing the Partition Process

You use the `-split_out` option to define the maximum number of patterns to include in each partition. Based on your specification, MAX Testbench generates a testbench (.v) file and a set of partitioned data (.dat) files from a single STIL file.

When splitting large STIL files, MAX Testbench uses the following equation to determine the number of partitions (or .dat files) to create:

$$\text{Number of Partitions} = \frac{\text{Total Number of Patterns}}{\text{Number of Patterns in a Partition}}$$

The partitioning process is as follows:

1. The first partition (partition 0) starts as normal and stops at the execution of the last pattern of the partition.
2. The second partition (partition 1) starts by reproducing the test_setup macro and the Condition statement to restore the context of the last pattern of the first partition (partition 0).

The second partition contains a duplication of the last pattern of the previous partition, except that all unload states are masked. The strobe of the states corresponds to the second-to-last pattern of the previous partition. This strobe is ensured by the first partition, so you do not need to replicate it. All subsequent partitions follow the architecture of the second partition.

3. Use VCS to create a simulation executable for MAX Testbench, then use the simulation executable and the +tmax_part=partition_number option to simulate each partition, as shown in the following example:

```
simv +tmax_part=0 simv +tmax_part=1 simv +tmax_part=2
```

Example Test

Note the following example test:

```
./simv +tmax_part=0 | tee run_vcs_par_usf_split_simv0.log
./simv +tmax_part=1 | tee run_vcs_par_usf_split_simv1.log

./simv +tmax_part=0 | tee run_vcs_par_usf_split_simv0.log
Chronologic VCS simulator copyright 1991-2013
Contains Synopsys proprietary information.
#####
MAX TB
Test Protocol File generated from original file
"pattn/pattn_comp_USF_par.stil"
STIL file version: 1.0
Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime
options
#####

XTB: Reading partition 0 (test data
file /TEST_split/pattn/pattn_comp_USF_par_split_0.dat)
XTB: Enabling Enhanced Debug Mode. Using mode 1 (conditional parallel
strobe).
XTB: Starting parallel simulation of 6 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 10 (T=1700.00 ns, V=18)
XTB: Begin parallel scan load for pattern 10, unload 2 (T=2000.00 ns,
V=21)
XTB: Begin parallel scan load for pattern 5 (T=1700.00 ns, V=18)
```

```
XTB: Simulation of 6 patterns completed with 0 mismatches (0 internal mismatches) (time: 2000.00 ns, cycles: 20)
```

```
V C S   S i m u l a t i o n   R e p o r t  
Time: 2000000 ps
```

```
./simv +tmax_part=1 | tee run_vcs_par_usf_split_simv1.log
```

```
Chronologic VCS simulator copyright 1991-2013
```

```
Contains Synopsys proprietary information.
```

```
#####
```

```
MAX TB
```

```
Test Protocol File generated from original file
```

```
"pattn/pattn_comp_USF_par.stil"
```

```
STIL file version: 1.0
```

```
Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime options
```

```
#####
```

```
XTB: Reading partition 1 (test data
```

```
file /TEST_split/pattn/pattn_comp_USF_par_split_1.dat)
```

```
XTB: Enabling Enhanced Debug Mode. Using mode 1 (conditional parallel strobe).
```

```
XTB: Starting parallel simulation of 6 patterns
```

```
XTB: Using 0 serial shifts
```

```
XTB: Begin parallel scan load for pattern 5 (T=200.00 ns, V=3)
```

```
XTB: Begin parallel scan load for pattern 10 (T=1700.00 ns, V=18)
```

```
XTB: Simulation of 6 patterns completed with 0 mismatches (0 internal mismatches) (time: 2200.00 ns, cycles: 22)
```

```
V C S   S i m u l a t i o n   R e p o r t  
Time: 2200000 ps
```

Controlling the Timing of a Parallel Check/Assert Event

When a vector is applied for a parallel-load operation, the state of all the scan elements must be examined so it can be compared to the expected unload state. Subsequently, the next state must be forced to implement the load operation. This operation is performed as a single event:

- It compares the current state across all elements
- It forces the next state on all elements (during the same time in the Verilog simulation).

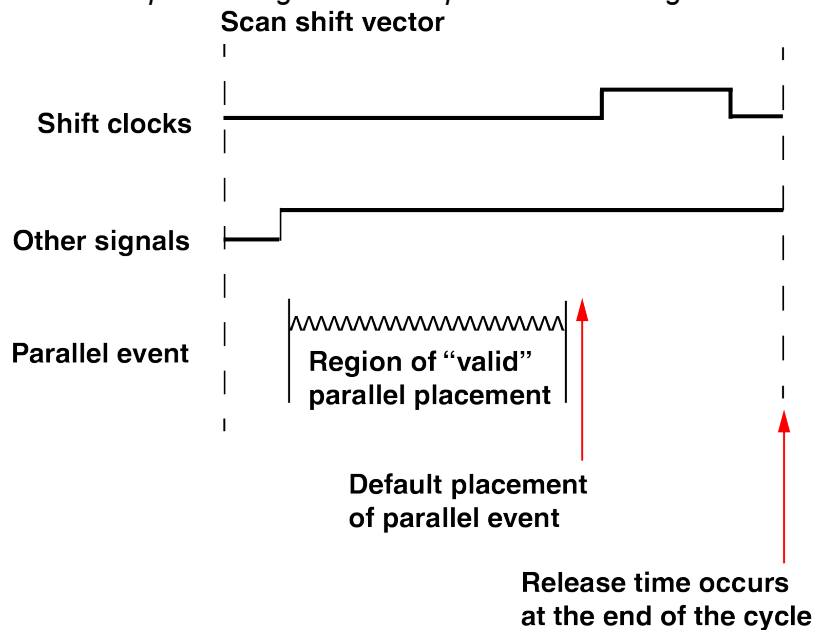
The placement of this event is restricted within the vector period. It should not occur before the "scan" mode has stabilized. If not, it might sample scan states too early, and generate miscompares because the scan state has not stabilized. Also, the event should not occur after the first clock pulse into the scan elements, as the force operation will not have set the proper value to load with this clock event. See the following figure.

Note that the default parallel operation will place this event a single (1) time increment before the first defined clock pulse in the shift vector. This is the latest possible time, which allows the device scan state to stabilize on the scan elements.

However, some situations might forcing the next state at this time might violate the setup time. This occurs particularly in cases in which setup timing checks have been defined on scan elements. When setup violations have occurred, the simulation may generate X values on the flip-flops and affect the simulation response for the next unload operation. In circumstances such as this, you may need to override the default placement of the parallel event and specify a different time for this event. You can set this time using the following configuration command:

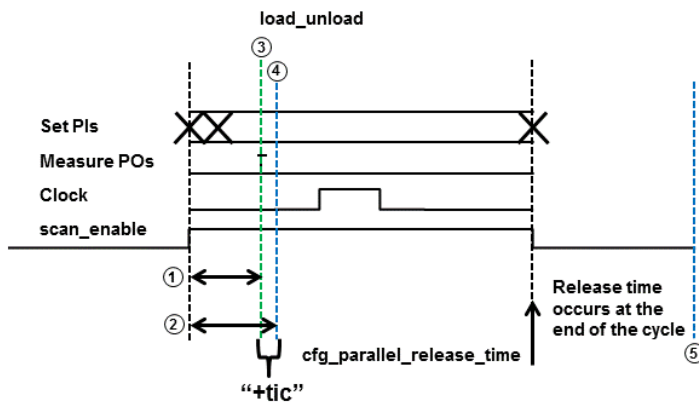
```
cfg_parallel_release_time time_value
```

Figure 159 Implementing the Load Operation as a Single Event



The timing for parallel load simulation differs from a serial load simulation when the data is driven directly on the flip-flops. The following figure shows how the parallel load MAX Testbench works in terms of force, release, and strobe times.

Figure 160 Timing For Parallel Load MAX Testbench



- ① The time of the scan-enable to the first output measure performed for scan-outs only
- ② The time of the scan-enable to the first clock of each scan element performed for every scan cell. The input to the scan cell in scan mode must be stable before the effective edge of the clock pulse, and if the scan enable affect that stability, it needs to be checked as a critical path.
- ③ Parallel placement strobe time check
- ④ Parallel Force Event occurs "+tic" after strobe
- ⑤ Release time can be controlled using the "cfg_parallel_release_time" config file option.

There is a limitation in which the delayed parallel_release_time cannot be displayed in parallel simulation waveforms. If you need to see the parallel release time, you can edit your testbench file and see the results in the simulation log file only.

In the testbench, look for release_scalls followed by #PRTIME and add the following two lines for the \$display statement(s):

```
always /* ParallelShiftMode */ @(release_scalls) begin
#PRTIME;
$display("`XTB: initiating release at T=%0t,    (using %0d serial
  shifts)", $time, SSHIFTS);
$display("`XTB: initiating release at T=%0t,    for pattern %0d (using %0d
  serial shifts)", $time, cur_pat, SSHIFTS);
```

The difference between these two \$display statements is that the second statement prints the current pattern number. See the example in the following figure, which shows the different results produced by these two statements in the output simulation log file.

Chapter 32: Using MAX Testbench Configuring MAX Testbench

Figure 161 Example Testbench Edits

```

1124 end
1125 end
1126 else begin
1127   shr_0(dlarg, valargs, 0, 1);
1128 end
1129 end
1130 end
1131 endtask
1132
1133 always # ParallelismMode ? (@(force_sca) begin
1134   force { CHAIN10, CELL10 } = LOD1;
1135   force { CHAIN11, CELL11 } = LOD2;
1136   force { CHAIN12, CELL12 } = LOD3;
1137   force { CHAIN13, CELL13 } = LOD4;
1138   force { CHAIN14, CELL14 } = LOD5;
1139   force { CHAIN15, CELL15 } = LOD6;
1140   force { CHAIN16, CELL16 } = LOD7;
1141   force { CHAIN17, CELL17 } = LOD8;
1142   force { CHAIN18, CELL18 } = LOD9;
1143   force { CHAIN19, CELL19 } = LOD10;
1144   force { CHAIN20, CELL20 } = LOD11;
1145   force { CHAIN21, CELL21 } = LOD12;
1146 end
1147
1148 always # ParallelismMode ? (@(release_sca) begin
1149   #RTIME;
1150   $display{ "jv-XTB: initiating release at T=%0t (using %0d serial shifts)", $time, $SHIFTS};
1151   release { CHAIN10, CELL10 } = LOD1;
1152   release { CHAIN11, CELL11 } = LOD2;
1153   release { CHAIN12, CELL12 } = LOD3;
1154   release { CHAIN13, CELL13 } = LOD4;
1155   release { CHAIN14, CELL14 } = LOD5;
1156   release { CHAIN15, CELL15 } = LOD6;
1157   release { CHAIN16, CELL16 } = LOD7;
1158   release { CHAIN17, CELL17 } = LOD8;
1159   release { CHAIN18, CELL18 } = LOD9;
1160   release { CHAIN19, CELL19 } = LOD10;
1161   release { CHAIN20, CELL20 } = LOD11;
1162   release { CHAIN21, CELL21 } = LOD12;
1163 end
1164
1165 task load_unload;
1166   input reg [50:0] idlg;
1167   input reg [2:0] width;
1168   input reg [0:1] valarg;
1169   begin
1170     if (verbose == 2) $display{ "XTB: Starting proc load_unload... T=%0t V=%0d", $time, v_c};
1171     if (stb_sim_model == 0 && ser_pat == 0 && cur_pat == (ser_pat+first_pat)) begin
1172       $display{ "XTB: Switching into Parallel simulation mode at pattern %0d (using %0d serial shifts)", $time, $SHIFTS};
1173       stb_sim_model = 1;
1174     end
1175     if (cur_pat == prev_pat) begin
1176       loads = 1;
1177     end
1178     prev_pat = cur_pat;
1179     if (cur_pat % 4 == 0)
1180       $display{ "XTB: Begin %0s scan load for pattern %0d (T=%0t, V=%0d)", stb_sim_model,
1181         cur_pat, $time, v_c};
1182   end
1183 end
  
```

Figure 162 Example of Simulation Log File Edits

```

59 XTB: Starting proc load_unload... T=200000.00 ps, V=3
60 XTB: Begin parallel scan load for pattern 0 (T=200000.00 ps, V=3)
61 XTB: Starting V# 3 at time 200000.00 ps
62 XTB: (parallel) shift at 300000.00 ps
63 XTB: Starting V# 4 at time 300000.00 ps
64 - jv-XTB: initiating release at T=400000.00 ps (using 0 serial shifts)
65 + jv-XTB: initiating release at T=400000.00 ps for pattern 0 (using 0 serial shifts)
66 XTB: Processed statement: load_unload$1mt
67 XTB: Starting proc capture... T=400000.00 ps, V=5
68 XTB: Starting V# 5 at time 400000.00 ps
69 XTB: Starting V# 6 at time 500000.00 ps
70 XTB: Processed statement: capture$1mt
71 XTB: Processed statement: Self-force$1
72 XTB: Processed statement: InPat
73 XTB: Starting proc load_unload... T=600000.00 ps, V=7
74 XTB: Starting V# 7 at time 600000.00 ps
75 XTB: (parallel) shift at 700000.00 ps
76 XTB: Starting V# 8 at time 700000.00 ps
77 - jv-XTB: initiating release at T=800000.00 ps (using 0 serial shifts)
78 + jv-XTB: initiating release at T=800000.00 ps for pattern 1 (using 0 serial shifts)
79 XTB: Processed statement: load_unload$1mt
80 XTB: Starting proc capture: ckl_1... T=800000.00 ps, V=9
81 XTB: Starting V# 9 at time 800000.00 ps
82 XTB: Starting V# 10 at time 900000.00 ps
83 XTB: Starting proc capture: ckl_2... T=800000.00 ps, V=5
84 XTB: Processed statement: capture_ckl$1mt
85 XTB: Processed statement: InPat
86 XTB: Starting proc load_unload... T=1100000.00 ps, V=12
87 XTB: Starting V# 12 at time 1100000.00 ps
88 XTB: (parallel) shift at 1200000.00 ps
89 XTB: Starting V# 13 at time 1200000.00 ps
90 - jv-XTB: initiating release at T=1300000.00 ps (using 0 serial shifts)
91 + jv-XTB: initiating release at T=1300000.00 ps for pattern 2 (using 0 serial shifts)
92 XTB: Processed statement: load_unload$1mt
93 XTB: Starting proc capture: ckl_1... T=1300000.00 ps, V=14
94 XTB: Starting V# 14 at time 1300000.00 ps
  
```

The following figure shows an example with a 25 ns delay. Currently, adding the \$display statement is the only way to validate that the delay is honored.

Figure 163 Example with 25 ns Delay

```
xterm <@snpsexp5>
-Wh,-whole-archi... /global/apps3/vcs_2016.06-SP1/linux64/lib/libvcsuccli.so -Wh,-no-whole-archi...
/global/apps3/vcs_2016.06-SP1/linux64/lib/vcs_save_restore_new.o -ldl -lc -lm -lpthread \
-ldl
../simv up to date
Command: ./simv +tetramax +libext+.v+ -a xtb_par.log +tmax_msg=4
Chronologic VCS simulator copyright 1991-2016
Contains Synopsys proprietary information.
Compiler version L-2016.06-SP1_Full164; Runtime version L-2016.06-SP1_Full164; Oct  7 17:29 2016
*****
MAX TB Version L-2016.03-SP4-VAL
Test Protocol File generated from original file "pats/pr_pat.stil"
STIL file version: 1.0
Enhanced Runtime Version; use <sim_exec> +tmax_help for available runtime options
*****

XTB: Setting runtime option "tmax_msg" to 4.
XTB: Starting parallel simulation of 25 patterns
XTB: Using 0 serial shifts
XTB: Processed statement: WFTStmt
XTB: Processed statement: ConditionStmt
XTB: Starting macro test_setup..., T=0,00 ns, V=1
XTB: Starting V# 1 at time 0,00 ns
XTB: Starting V# 2 at time 100,00 ns
XTB: Processed statement: test_setupStmt
XTB: Processed statement: SetForceSI
XTB: Processed statement: SetPat
XTB: Starting proc load_unload..., T=200,00 ns, V=3
XTB: Begin parallel scan load for pattern 0 (T=200,00 ns, V=3)
XTB: Starting V# 3 at time 200,00 ns
XTB: (parallel) shift, at 300,00 ns
XTB: Starting V# 4 at time 300,00 ns
XTB: Processed statement: load_unloadStmt
XTB: Starting proc multicl... T=400,00 ns, V=5
XTB: Starting V# 5 at time 400,00 ns
jv-XTB: initiating release at T=425,00 ns, (using 0 serial shifts)
jv-XTB: initiating release at T=425,00 ns, for pattern 0 (using 0 serial shifts)
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=500,00 ns, V=6
XTB: Starting V# 6 at time 500,00 ns
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=600,00 ns, V=7
XTB: Starting V# 7 at time 600,00 ns
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=700,00 ns, V=8
XTB: Starting V# 8 at time 700,00 ns
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=800,00 ns, V=9
XTB: Starting V# 9 at time 800,00 ns
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=900,00 ns, V=10
XTB: Starting V# 10 at time 900,00 ns
XTB: Processed statement: multicl...Stmt
XTB: Starting proc multicl... T=1000,00 ns, V=11
*****
```

For more information, see [Defining the load_unload Procedure](#).

Using MAX Testbench to Report Failing Scan Cells

You can use MAX Testbench to report the instance names of failing scan cells to help you debug mismatches during a parallel simulation. MAX Testbench reads the VCS log file with the basic simulation failure log file information (the scan chain names and the cell IDs) and reports the instance names of the failing scan cells.

This flow is ideal for large designs because it significantly reduces the VCS compilation time and memory requirements, and the memory required to run a VCS simulation.

Note: The PSD file is still required in the unified STIL flow for scan compression.

This topic includes the following sections:

- [Flow Overview](#)
- [Flow Example](#)

Flow Overview

The flow for reporting failing scan cells includes two separate MAX Testbench runs:

- In the first run, MAX Testbench creates a testbench used in a parallel VCS simulation. The following example shows the command used in the first MAX Testbench run:

```
stil2verilog pats.stil maxtb -config_file xtb.cfg -replace
```

The output log file from the VCS simulation contains basic diagnosis information, such as the vectors, scan chain names, cell IDs, and scan cell names.

- In the second run, MAX Testbench uses the simulation failure log file information from the VCS simulation log file and the original STIL data to produce a log file containing a list of failing scan cells. The command for the second MAX Testbench run includes the new

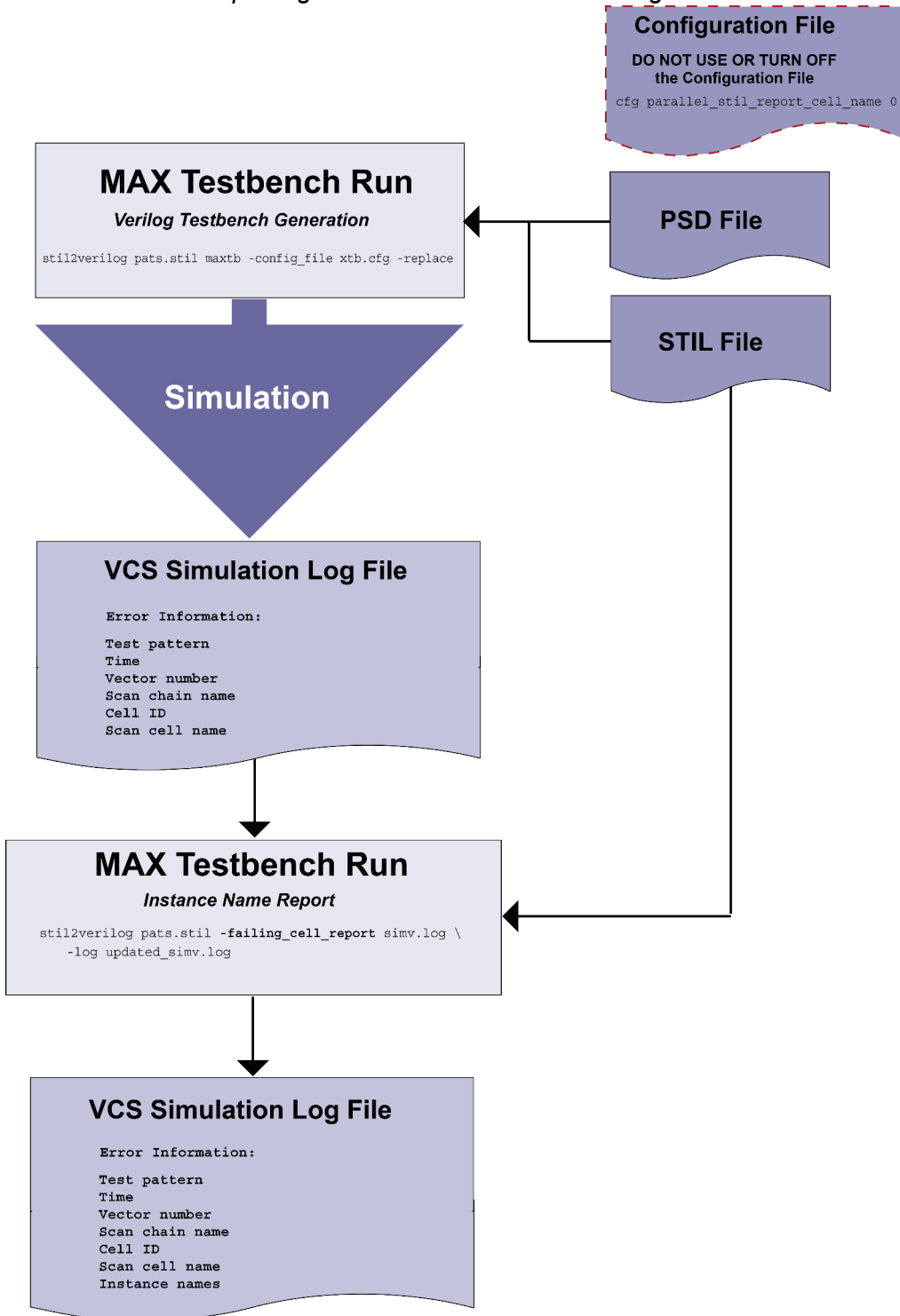
`-failing_cell_report` option, which specifies the VCS simulation log file, and the

`-log` option, which specifies the name of the log file containing the failing scan cells.

```
stil2verilog pats.stil -failing_cell_report simv.log \ -log  
new_simv.log
```

The following figure shows the flow.

Figure 164 Flow for Reporting the Instance Names of Failing Scan Cells



Flow Example

The following example shows the process for reporting the instance names of failing scan cells.

First MAX Testbench Run:

```
stil2verilog pats.stil maxtb -config_file xtb.cfg -replace
```

VCS Simulation Log (*simv.log*):

```
XTB: Starting parallel simulation of 31 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=6200.00 ns, V=63)
XTB: Begin parallel scan load for pattern 10 (T=12200.00 ns, V=123)
XTB: Begin parallel scan load for pattern 15 (T=18200.00 ns, V=183)
XTB: Begin parallel scan load for pattern 20 (T=24200.00 ns, V=243)
XTB: Begin parallel scan load for pattern 25 (T=30200.00 ns, V=303)
>>> Error during scan pattern 26 (detected from parallel unload of
pattern 25)
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 0
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 7, scan cell 2
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 3
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 6
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 7, scan cell 7
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 8
>>> Error during scan pattern 26 (detected from parallel unload of
pattern 25)
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 14, scan cell 1
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 3
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 7
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 14, scan cell 10
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 11
>>> Error during scan pattern 26 (detected from parallel unload of
pattern 25)
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 2
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 3
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 4
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 5
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 7
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 9
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 11
>>> Error during scan pattern 26 (detected from parallel unload of
pattern 25)
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 28, scan cell 6
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 28, scan cell 7
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 28, scan cell 10
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 28, scan cell 11
>>> Error during scan pattern 26 (detected from parallel unload of
pattern 25)
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 3
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 35, scan cell 5
```

```
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 8
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 11
XTB: Begin parallel scan load for pattern 30 (T=36200.00 ns, V=363)
XTB: Simulation of 31 patterns completed with 26 mismatches (time:
  37400.00 ns, cycles: 374)
```

Second MAX Testbench Run:

```
stil2verilog pats.stil -failing_cell_report simv.log -log new_simv.log
```

MAX Testbench Log (new_simv.log):

```
STIL2VERILOG
Copyright (c) 2007 - 2016 Synopsys, Inc.
This software and the associated documentation are proprietary to
Synopsys,
Inc. This software may only be used in accordance with the terms and
conditions
of a written license agreement with Synopsys, Inc. All other use,
reproduction,
or distribution of this software is strictly prohibited.
maxtb> Parsing command line...
maxtb> Checking for feature license...
maxtb> Parsing STIL file "pats.stil" ...
... STIL version 1.0 ( Design 2005) ...
... Building test model ...
... Signals ...
... SignalGroups ...
... Timing ...
... ScanStructures : "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
"11" "12" "13" "14" "15" "16" "17" "18" "19" "20" "21" "22"
"23" "24" "25" "26" "27" "28" "29" "30" "31" "32" "33"
"34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44" "45"
"46" "47" "48" "49" "50" "51" "52" "53" "54" "55" "56" "57"
"58" "59" "60" "61" "62" "63" "64" "65" "66" "67" "68" "69"
"70" "71" "72" "sccompin0" "sccompin1" "sccompin2" "sccompout0"
"sccompout1" "sccompout2" "sccompout3" "sccompout4" "sccompout5"
"sccompin3" "sccompin4" "sccompin5" ...
... PatternBurst "ScanCompression_mode" ...
... PatternExec "ScanCompression_mode" ...
... ClockStructures "ScanCompression_mode": occ_ctrl ...
... CompressorStructures :
"des_unit_U_decompressor_ScanCompression_mode"
"des_unit_U_compressor_ScanCompression_mode" ...
... Procedures "ScanCompression_mode": "internal_load_unload"
"multiclock_capture" "allclock_capture" "allclock_launch"
"allclock_launch_capture" "load_unload" ...
... MacroDefs "ScanCompression_mode": "test_setup" ...
... Pattern block "_pattern_" ...

maxtb> STIL file successfully interpreted (PatternExec:
  ""ScanCompression_mode"" ).
maxtb> Total test patterns to process 51
```

```
maxtb> Detected an X-Tolerant Scan Compression mode.
maxtb> Parsing simulation log file "simv.log"
maxtb> Report Failing Cells:
>>> Error during scan pattern 26
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 0, cell
name U_CORE.dd_d.data2_reg_2_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 7, scan cell 2, cell
name U_CORE.dd_d.data2_reg_0_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 3, cell
name U_CORE.dd_d.data1_reg_7_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 6, cell
name U_CORE.dd_d.data1_reg_4_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 7, scan cell 7, cell
name U_CORE.dd_d.data1_reg_3_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 7, scan cell 8, cell
name U_CORE.dd_d.data1_reg_2_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 14, scan cell 1, cell
name U_CORE.dd_d.data3_reg_7_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 3, cell
name U_CORE.dd_d.data3_reg_5_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 7, cell
name U_CORE.dd_d.data3_reg_1_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 14, scan cell 10, cell
name U_CORE.dd_d.data2_reg_6_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 14, scan cell 11, cell
name U_CORE.dd_d.data2_reg_5_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 2, cell
name U_CORE.dd_d.data5_reg_4_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 3, cell
name U_CORE.dd_d.data5_reg_3_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 4, cell
name U_CORE.dd_d.data5_reg_2_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 5, cell
name U_CORE.dd_d.data5_reg_1_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 7, cell
name U_CORE.dd_d.data4_reg_7_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 9, cell
name U_CORE.dd_d.data4_reg_5_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 21, scan cell 11, cell
name U_CORE.dd_d.data4_reg_3_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 28, scan cell 6, cell
name U_CORE.dd_d.data6_reg_6_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 28, scan cell 7, cell
name U_CORE.dd_d.data6_reg_5_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 28, scan cell 10, cell
name U_CORE.dd_d.data6_reg_2_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 28, scan cell 11, cell
name U_CORE.dd_d.data6_reg_1_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 3, cell
name U_CORE.dd_d.o_data_reg_7_
>>> At T=31540.00 ns, V=316, exp=0, got=1, chain 35, scan cell 5, cell
name U_CORE.dd_d.o_data_reg_5_
```

```
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 8, cell
name U_CORE.dd_d.o_data_reg_2_
>>> At T=31540.00 ns, V=316, exp=1, got=0, chain 35, scan cell 11, cell
name U_CORE.dd_d.data7_reg_7_
```

Note that the text highlighted in red represents the instance name of the failing cell.

MAX Testbench Runtime Programmability

MAX Testbench supports a runtime programmability flow that enables you to specify a series of runtime simulation options that use the same compiled executable in different modes.

For example, you can compile a single executable using one or more runtime options, such as `+tmax_msg`, `+tmax_rpt`, `+tmax_serial`, `+tmax_parallel`, `+tmax_n_pattern_sim`, and `+tmax_test_data_file`. You can then specify any of these options at runtime using the same executable.

You can also use a set of options to change test patterns. For example, if you want to write out patterns with different chain tests. The flow for using split patterns is different than the flow for regular patterns. For details, see [Runtime Programmability for Patterns](#).

The following sections describe how to configure and execute runtime programmability in MAX Testbench:

- [Basic Runtime Programmability Simulation Flow](#)
- [Runtime Programmability for Patterns](#)
- [Example: Using Runtime Predefined VCS Options](#)
- [MAX Testbench Runtime Programmability Limitations](#)

See Also

- [Configuring MAX Testbench](#)
- [Predefined Verilog Options](#)

Basic Runtime Programmability Simulation Flow

The basic simulation flow for runtime programmability is as follows:

1. Generate a STIL-based testbench. For details, see [Running MAX Testbench](#).
2. Configure the compile-time options, as needed.
3. Compile the testbench, design, and libraries, and produce a single default simulation executable. You only need to compile the executable one time, using minimal configuration.

4. Run the simulation, for example:

```
<sim_exec> +<runtime_option>
```

Note that you can use any of the following runtime options:

- `tmax_msg`
- `tmax_rpt`
- `tmax_serial`
- `tmax_parallel`
- `tmax_n_pattern_sim`
- `tmax_test_data_file`

For details on these options, see the [MAX Testbench Configuration](#) section.

5. If you encounter a new behavior, or need a new report or test patterns, specify the appropriate runtime option and rerun the simulation without recompiling the executable. For example:

```
<simv_exec> <+tmax_test_data_file="myfile.dat">
```

In the previous example, `myfile.dat` is the newly generated data (.dat) file to be used with the existing testbench file.

Note the following:

- If you specify the `tmax_serial` option at compile time and the `+parallel` option at runtime, the resulting simulation is a parallel simulation.
- The `msg` and `rpt` options affect the simulation report by providing different verbosity levels. Their defaults are 0 and 5, respectively. Setting up values different than these values, either at compile-time or runtime, is automatically reported by the testbench at simulation time 0. The runtime options override their compilation-time counterparts.
- The `n_pattern_sim` option overrides the equivalent `tmax_n_pattern_sim` option, if the latter option is specified. Otherwise, it overrides the default initial set of patterns (the entire set in the STIL file, or the set generated by Max Testbench using the `-first` and `-last` options).

Runtime Programmability for Patterns

You can use the `-generic_testbench` and `-patterns_only` options with the

`write_testbench` or `stil2Verilog` commands to configure runtime programmability for patterns.

Do not confuse the use of regular patterns and the use of split patterns for runtime programmability. You cannot simultaneously use the `-generic_testbench` and `-patterns_only` options for split patterns. See [Using Split Patterns](#) for details.

The following sections describe how to use runtime programmability for patterns:

- [Using the `-generic_testbench` Option](#)
- [Using the `-patterns_only` Option](#)
- [Executing the Flow](#)
- [Using Split Patterns](#)

Using the `-generic_testbench` Option

The `-generic_testbench` option, used in the first pass of the flow, provides special memory allocation for runtime programmability. This is required because the Verilog 95 and 2001 formats use static memory allocation to enable buffers and arrays to store and manipulate `.dat` information. This type of data storage cannot be handled by a standard `.dat` file. Also, it is expected that `.dat` files will continue to expand as they store an increasing number of vectors and atomic instructions.

The `-generic_testbench` option runs a task that detects the loading of the `.dat` file, and then allocates an additional memory margin. If, at some point, the data exceeds this allocated capacity, an error message, such as the following, will appear.

```
XTB Error: size of test data file <file_name>.dat exceeding
testbench memory allocation. Exiting...
(recompile using -pvalue+design1_test.tb_part.MDEPTH=<###>).
```

As indicated in the message, you will need to recompile the testbench using the suggested Verilog parameter to adjust the memory allocation.

Using the `-patterns_only` Option

The `-patterns_only` option is used for a second pass, or later, run. It initiates a light processing task that merges the new test data. This option also enables additional internal instructions to be generated for the special `.dat` file. For example, it includes a computation of the capacity for later usage by the testbench for memory management.

If you are running an updated pattern file, and have specified the `-pattern_only` option, you will see the following message:

```
XTB: Setting test data file to "<file_name>.dat" (at runtime).
Running simulation with new database...
```

Executing the Flow

The flow for runtime programmability for patterns is as follows:

1. Generate the testbench in generic mode using the first available STIL file. For example:

```
write_testbench -input pats.stil -output runtime_1 \
-replace -parameter {-generic_testbench \
-log mxtb.log -verbose}
Executing 'stil2Verilog'...
```

2. Compile and simulate this testbench (along with other required source and library files).
3. When a new pattern set is required, generate a new STIL file, while keeping the same STIL procedure file for the DRC (same test protocol).
4. Rerun MAX TestBench against the newly generated STIL file to generate only new the test data file, as shown in the following example:

```
write_testbench -input pats_new.stil -output runtime_2 \
-replace -parameter { -patterns_only -log mxtb_2.log \
-verbose}
```

5. Attach the newly generated .dat file to the simulation executable and rerun the simulation (without recompilation), as shown in the following example:

```
simv +tmax_test_data_file="<new_pattern_filename>.dat"
Command: ./simv +tmax_test_data_file=runtime_2.dat
#####
MAX TB Version H-2013.03
Test Protocol File generated from original file " pats_
new.stil"
STIL file version: 1.0
#####
XTB: Setting test data file to "runtime_2.dat" (at runtime).
Running simulation with new database...
XTB: Starting parallel simulation of 5 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Simulation of 5 patterns completed with 0 errors (time:
2700.00 ns, cycles: 27)
V C S S i m u l a t i o n R e p o r t
```

6. Repeat steps 3 to 5, as needed, to include a new STIL file.

Using Split Patterns

The following examples show how to split patterns for runtime programmability.

This example uses the stil2Verilog command:

```
stil2Verilog input_stil_file_name output_testbench_name \
-tb_module < > -split_out 32 -generic -replace \
-log translation.log
```

The next example uses the `write_testbench` command:

```
write_testbench -input input_stil_file_name -out output_testbench_
name \
-parameters {-split_out 32 -tb_module < > -generic \
-log mxtb.log}
```

The next set of examples show the process of splitting pattern files using the `write_patterns` command and a series of `write_testbench` commands. Note that you do not need to use the `-patterns_only` option to create the first split file. In this case, the first split file is created using the `-generic` option in the first `write_testbench` command of the command sequence.

```
write_patterns ./pattern/top_scan.stil -format stil -replace \
-split 5
write_testbench -input ./pattern/top_scan_0.stil
-output ./pattern/top_scan_maxtb -replace \
-parameter {-generic -log mxtb_generic_split_0.log \
-verbose }
write_testbench -input ./pattern/top_scan_1.stil \
-output ./pattern/top_scan_maxtb_1 -replace \
-parameter {-patterns_only -log mxtb_split_1.log \
-verbose }
write_testbench -input ./pattern/top_scan_2.stil \
-output ./pattern/top_scan_maxtb_2 -replace \
-parameter {-patterns_only -log mxtb_split_2.log \
-verbose }
write_testbench -input ./pattern/top_scan_3.stil \
-output ./pattern/top_scan_maxtb_3 -replace \
-parameter {-patterns_only -log mxtb_split_3.log \
-verbose }
write_testbench -input ./pattern/top_scan_4.stil \
-output ./pattern/top_scan_maxtb_4 -replace \
-parameter {-patterns_only -log mxtb_split_4.log \
-verbose }
write_testbench -input ./pattern/top_scan_5.stil \
-output ./pattern/top_scan_maxtb_5 -replace \
-parameter {-patterns_only -log mxtb_split_5.log \
-verbose }
```

Example: Using Runtime Predefined VCS Options

The following example shows how to use runtime predefined VCS options:

```
%> ./simv_usf +tmax_msg=3 +tmax_n_pattern_sim=1 +tmax_rpt=3
#####
MAX TB Version H-2013.03
Test Protocol File generated from original file "runtime.stil"
STIL file version: 1.0
#####
XTB: Setting runtime option "tmax_n_pattern_sim" to 1.
XTB: User requesting simulating patterns 0 to 1
```



```

XTB: Setting runtime option "tmax_msg" to 3.
XTB: Setting runtime option "tmax_rpt" to 3.
XTB: Starting parallel simulation of 2 patterns
XTB: Using 0 serial shifts
XTB: Processed statement: WFTStmt
XTB: Processed statement: ConditionStmt
XTB: Starting macro test_setup..., T=0.00 ns, V=1
XTB: Processed statement: test_setupStmt
XTB: Processed statement: SetPat
XTB: Starting proc load_unload..., T=200.00 ns, V=3
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: (parallel) shift, at 300.00 ns
XTB: Processed statement: load_unloadStmt
XTB: Starting proc capture..., T=400.00 ns, V=5
XTB: Processed statement: captureStmt
XTB: Processed statement: IncPat
XTB: Starting proc load_unload..., T=500.00 ns, V=6
XTB: (parallel) shift, at 600.00 ns
XTB: Processed statement: load_unloadStmt
XTB: Starting proc capture_clk..., T=700.00 ns, V=8
XTB: Processed statement: capture_clkStmt
XTB: Processed statement: IncPat
XTB: Simulation of 2 patterns completed with 0 error (time: 1000.00 ns,
cycles: 10)
V C S   S i m u l a t i o n   R e p o r t

```

MAX Testbench Runtime Programmability Limitations

The following limitations apply to runtime programmability in MAX Testbench:

- The following runtime options are not supported: `tmax_vcde`, `tmax_serial_timing`, `tmax_diag_file`, `tmax_diag`.
- You cannot change between the `+delay_mode_zero`, `+typdelays`, `+mindelays`, and `+maxdelays` options.
- You cannot use a different `test_setup` procedure at runtime.
- The width of a variable must remain constant.
- The STIL procedure file cannot be changed before generating second-pass patterns.
- Compile-time switches cannot be changed.
- You cannot use `$dumpvars` statements.
- You must use the same version of VCS throughout the entire runtime programmability process.

MAX Testbench Support for IDDQ Testing

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 is 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.

You can use the following methods in MAX Testbench to configure the IDDQ testing:

- [Compile-Time Options for IDDQ](#)
- [IDDQ Configuration File Settings](#)
- [Generating a VCS Simulation Script](#)

See Also

- [Generating IDDQ Test Patterns](#)

Compile-Time Options for IDDQ

MAX Testbench has two compile-time options that support IDDQ testing and are specified at the command line when starting a simulation. Note that these compile-time options cannot be specified in the configuration file:

- `tmax_iddq`

This option enables IDDQ testing during PowerFault simulation. The default behavior is not to use the IDDQ test mode. The following example enables IDDQ testing from the VCS command line:

```
% vcs ... +define+tmax_iddq
```

- `tmax_iddq_seed_mode=<0|1|2>`

This option changes the fault seeding for IDDQ testing to one of three modes:

- 0 for automatic seeding (default)
- 1 for seeding from a fault file only
- 2 for both automatic seeding and file seeding

When the seeding mode is set to 1 or 2, the testbench assumes the existence of a fault list file (or its symbolic link) in the current directory named `tb_module_name.faults`. If this file is not found, the simulation stops and an error is issued.

You can override the default fault list name in the configuration file (see the next section).

See Also

- [Predefined Verilog Options](#)

IDDQ Configuration File Settings

You can make several IDDQ test-related specifications in a dedicated subsection of the configuration file. Note that there are no command-line equivalences to these settings since they are testbench file-specific commands.

`cfg_iddq_seed_file` *fault_list_file*

This parameter overrides the default `tb_module_name.faults` file when faults are seeded from an external fault list file. The default `tb_module_name` file in Max Testbench is `DUT_name_test`.

The following example specifies faults seeded from a file called `my_dut_test`:

```
set cfg_iddq_seed_file my_dut_test
```

`cfg_iddq_verbose` 0 | 1

This parameter enables or disables the PowerFault verbose report. The default is 1, which enables the verbose report. Specify a value of 0 to disable the verbose report.

The following example disables the PowerFault verbose report:

```
set cfg_iddq_verbose 0
```

You can use the `+define+tmax_msg=4` simulation option to report file names that are used during the simulation process.

`cfg_iddq_leaky_status` 0 | 1

This parameter enables or disables the PowerFault leaky nodes report printed in the `tb_name.leaky` file. The default is 1, which enables the leaky nodes report. Specify a value of 0 to disable this report.

The following example disables the PowerFault leaky nodes report:

```
set cfg_iddq_leaky_status 0
```

`cfg_iddq_seed_faul_model` 0 | 1

This parameter specifies the PowerFault fault model used for external fault seeding. The default is 0, which specifies SA faults. Specify a value of 1 for bridging faults.

The following example specifies bridging faults for automatic seeding:

```
set cfg_iddq_seed_faul_model 1
```

`cfg_iddq_cycle` *value*

Use this parameter to set the initial counter value for IDDQ strobcs. The default is 0.

The following example sets the initial counter value to 1:

```
set cfg_iddq_cycle 1
```

Configuring MAX Testbench

Generating a VCS Simulation Script

You can use MAX Testbench to generate a script that sets up required information for IDDQ test simulation. This information is required to enable the PLI access option functions (+acc), the path to the archive PowerFault PLI library (`libiddq_vcs.a`), and the path to the PLI function interface (`iddq_vcs.tab`).

Note that automatic simulation script generation for IDDQ testing is limited to the VCS simulator only.

The following example is a basic script generated by MAX Testbench using the `-sim_script` option (without using any available parameters from the configuration file) when IDDQ test mode is enabled:

```
#!/bin/sh
LIB_FILES="my_lib.v ${IDDQ_HOME}/lib/libiddq_vcs.a
-P${IDDQ_HOME}/lib/iddq_vcs.tab"
DEFINES=""
OPTIONS="+tetramax +acc+2"
NETLIST_FILES="my_netlist.v"
TBENCH_FILE="new_i021_s1_s.v"
SIMULATOR="vcs"
${SIMULATOR} -R ${DEFINES} ${OPTIONS} ${TBENCH_FILE} ${NETLIST_FILES}
${LIB_FILES}
SIMSTATUS=$?
if [ ${SIMSTATUS} -ne 0 ]
then echo "WARNING: simulation command returned error status
${SIMSTATUS}"; exit ${SIMSTATUS};
fi
```

Note the following:

- When generating the script, MAX Testbench assumes that the `IDDQ_HOME` environment variable points to the location of an existing PowerFault PLI.
- You must have a valid Test-IDDQ license to run the PowerFault PLI.

How MAX Testbench Works

The Verilog writer for MAX Testbench is essentially an algorithm that browses the data structure and retrieves the appropriate information according to the order and the form determined by the Verilog testbench template.

MAX Testbench does not parse the netlist file. It retrieves the DUT interface (its hierarchical name and its primary I/O) from the STIL file. Therefore, it is the responsibility of the STIL provider (TestMAX ATPG) to make sure that this interface corresponds effectively to the one described in the netlist. The testbench file (test protocol) contains all the details of the STIL file, whereas the test data file translates the execution part (Pattern blocks).

Figure 4 Relationship of Files in MAX Testbench Flow

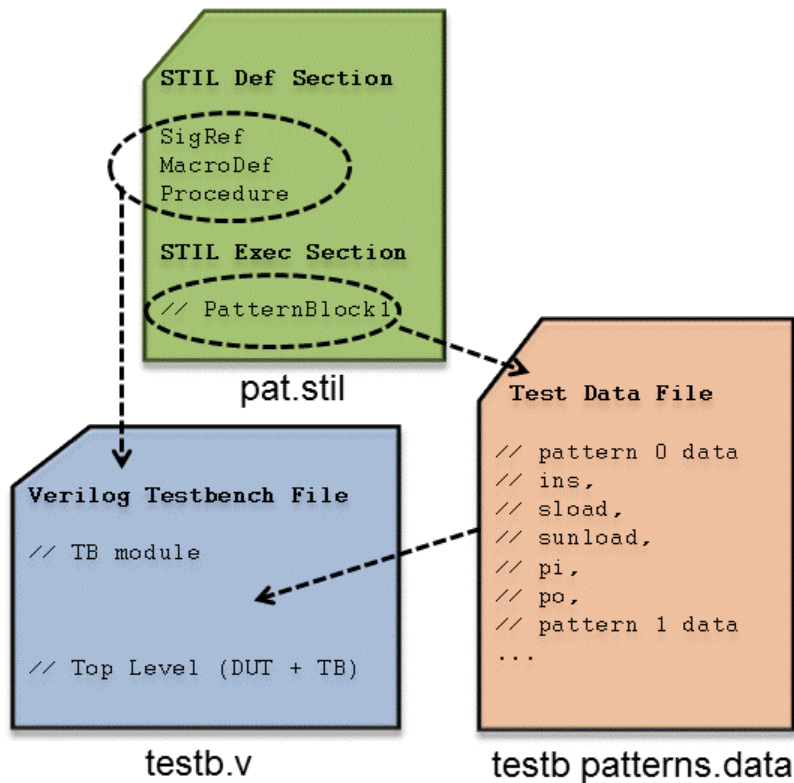
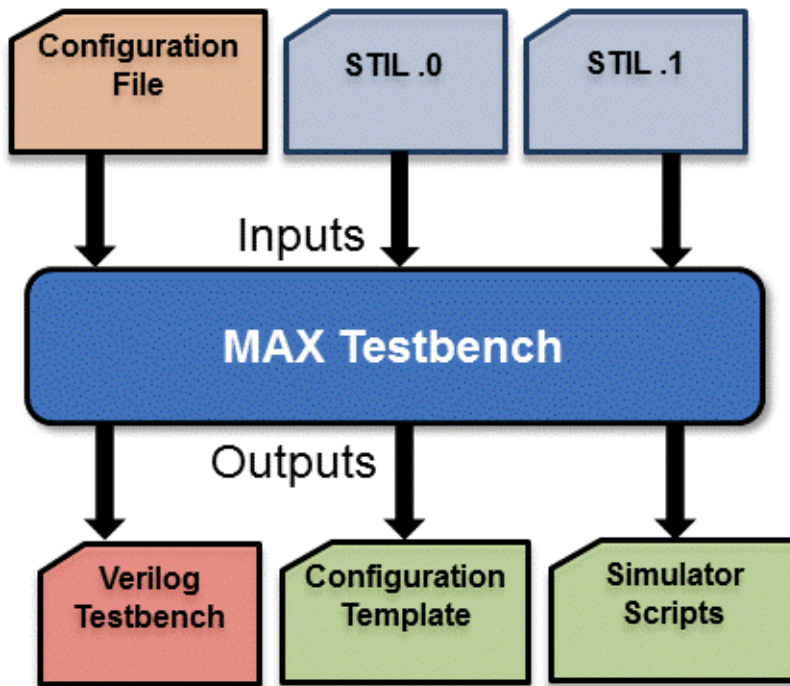


Figure 5 MAX Testbench Flow



For more information, see [Editing the STIL Procedure File](#).

Predefined Verilog Options

The following table describes a set of predefined Verilog options. When specified on the VCS compilation line, these options must be preceded by the `+define` statement.

Table 10 Predefined Verilog Options

Verilog Option	Description
-----------------------	--------------------

<code>+tmax_help</code>	Used with the <code>simv</code> executable, this option reports the available runtime options, which are:
<code>+tmax_n_pattern_sim</code>	
<code>+tmax_serial</code>	
<code>+tmax_parallel</code>	
<code>+tmax_msg</code>	
<code>+tmax_rpt</code>	
<code>+tmax_test_setup_only_once</code>	
<code>+tmax_test_data_file</code>	

Table 10 Predefined Verilog Options (Continued)

Verilog Option	Description
<code>+tmax_parallel=el=N</code>	Forces the parallel-load simulation of all scan data, with <i>N</i> bits extracted and serially simulated. This option overrides the behavior of a testbench written by MAX Testbench with the <code>-serial</code> option and overrides the value of <i>N</i> of a testbench written by MAX Testbench with the <code>-parallel</code> option. If <i>N</i> is not specified, it is processed as zero (meaning all bits are parallel-loaded).
<code>+tmax_serial=N</code>	Forces the serial simulation of the first <i>N</i> patterns for a testbench written by MAX Testbench with only the <code>-parallel</code> option. This option then starts the parallel simulation of the remaining patterns using the parallel parameters specified when the testbench was written.
<code>+tmax_rpt=N</code>	Specifies the interval of the progress message
<code>+tmax_msg=N</code>	Control for a pre-specified set of trace options
<code>+tmax_vcde</code>	Generates an extended VCD of the simulation run
<code>+tmax_serial_timg</code>	Generates a delay (a "dead period") for parallel scan access.
<code>+tmax_test_setup_only_on</code>	Simulates the <code>test_setup</code> macro only one time when using split patterns with MAX Testbench. This option is useful when you are using multiple STIL pattern files and want to avoid multiple simulations of the <code>test_setup</code> macro. It can be used for both compile time and runtime during a simulation.
<code>+tmax_usf_debug_robe_mode</code>	Reports various levels of details of simulation runtime miscompare messages for scan compression technology. For complete information on using this option, see " Debug Modes for Simulation Compare Messages ."

The `+tmax_rpt` option controls the generation of a statement on entry to every TestMAX ATPG pattern unit during the simulation. This statement is printed during the simulation run, and provides an indication of progress during the simulation run. This progress statement has two forms, depending on whether the next scan operation is executed in serial or parallel fashion:

```
Starting Serial Execution of TestMAX ATPG pattern N, time NNN, V #NN
Starting Parallel Execution of TestMAX ATPG pattern N, time NNN, V #NN
```

```
Starting Serial Execution of TestMAX ATPG pattern N (load N), time NNN, V  
#NN Starting Parallel Execution of TestMAX ATPG pattern N (load N), time  
NNN, V #NN
```

By default, the pattern reporting interval is set to every 5 patterns. This value can be changed by specifying the interval value to the `+tmax_rpt` option. For instance, `+define +tmax_rpt=1` on the VCS compile line generates a message for each TestMAX ATPG pattern executed. All pattern reporting messages can be disabled by setting `+define +tmax_rpt=0`.

The `+tmax_msg` option controls a pre-defined set of trace options, using the values 1 through 4 to specify tracing, where '1' provides the least amount of trace information and '4' traces everything. These values activate the trace options as follows:

- 0 — disables all tracing (except progress reports with `+tmax_rpt`)
- 1 — traces entry to each Procedure and Macro call
- 2 — adds tracing of WaveformTable changes
- 3 — adds tracing of Labels
- 4 — adds tracing of Vectors

The `+tmax_msg` option is set to 0 by default.

These two options `+tmax_rpt` and `+tmax_msg` provide a single control of tracing information, established as the simulation environment is started. By editing the testbench file, additional options can be specified during the simulation run.

The option `+tmax_evcd` supports generation of an extended VCD file for the instance of the design under test (dut). The name of this file is "sim_vcde.out". The option `+tmax_serial_timing` causes an interval of no events to be generated for each parallel scan access operation. This period aligns the overall simulation time of parallel scan access with the same time required for a normal serial shift operation. This "dead period" is described in "Parallel Scan Access". By default, this dead period is not present and the parallel scan access simulation occupies a single cycle period for the entire scan operation. For designs that can accept this dead period, this option facilitates coordinating times between parallel and serial simulations, and facilitates identifying the physical runtime of a pattern set with parallel scan access operation present. Some designs might not support this dead period, for instance certain styles of PLL models might lose synchronization for intervals without clock events present. These designs should not use this option.

The `+tmax_diag` option controls the generation of miscompare messages formatted for TestMAX ATPG diagnostics during the simulation.

For more information, see [Configuring MAX Testbench](#).

MAX Testbench Limitations

The following limitations apply when using MAX Testbench:

- MAX Testbench does not support DBIST/XDBIST, or core integration. XDBIST and CoreTest are EOL (End-Of-Life) tools.
- For script generation, predefined options are supported only for a VCS script.

See Also

- [Runtime Programmability Limitations](#)

Example of the Configuration Template

You can specify the following command to generate a template containing the options described in Table 1:

```
stil2Verilog -generate_config TB_config_file
```

Example 1 Configuration Template Example

```
## STIL2VERILOG CONFIGURATION FILE TEMPLATE (go-nogo default values) ##

# uncomment out the setting statement to use predefined variables
# the "set cfg_*" variables only affect the testbench definition

# cfg_patterns_read_interval: specifies the maximum number of patterns
loaded simultaneously in the simulation process

#set cfg_patterns_read_interval 1000

# cfg_patterns_report_interval: Specifies the interval of the progress
message

#set cfg_patterns_report_interval 5

# cfg_message_verbosity_level: control for a prespecified set of trace
options

#set cfg_message_verbosity_level 0

# cfg_evcd_file evcd_file: generates an extended-VCD of the simulation
run

#set cfg_evcd_file "evcd_file"
```

Chapter 32: Using MAX Testbench Configuring MAX Testbench

```
# cfg_diag_file: generates a failures log file compliant with TestMAX
ATPG diagnostics. This overrides the name in the tb file.

#set cfg_diag_file "diag_file"

# cfg_serial_timing: generates a delay for parallel scan access to align
parallel

# load timing with serial load timing

#set cfg_serial_timing 0

# cfg_time_unit: specifies the simulation time unit

#set cfg_time_unit "1ns"

# cfg_time_precision: specifies the simulation time precision

#set cfg_time_precision "1ns"

# cfg_dut_module_name: specifies the DUT module name to be tested
(variable to be used only when the tool asks for it)

#set cfg_dut_module_name "dut_module_name"

### TB file formatting section

# cfg_tb_format_extended: specifies whether an extended TB file is needed

#set cfg_tb_format_extended 0

# set drcw_severity <rule_name> <severity>

# The command "drcw_severity" needs two mandatory parameters:

# - <rule_name>: TestMAX ATPG rule name (wild-card character '*' is
supported)

# - <severity>: severity level ("ignore"|"warning"|"error")

#set drcw_severity C11 warning

### variables affecting only the simulator script generation

# define_<preprocessor_define>: specifies the preprocessor definitions
for the simulator

#set define_<user_def1> 0

#set define_<user_def2> "TRUE"
```

```
#design_files: specifies all source files required to run the simulation
#set design_files "netlist1.v netlist2.v"

# lib_files: specifies all library source files required to run the
simulation
#set lib_files "lib1.v lib2.v"

# vcs_options: specifies the user VCS command line options
#set vcs_options "VCSoption1 VCSoption2"

# nc_options: specifies the user NCSim command line options
#set nc_options "NCOption1 NCOption2"

# mti_options: specifies the user ModelSim command line options
#set mti_options "MTIOption1 MTIOption2"

# xl_options: specifies the user Verilog XL command line options
#set xl_options "XLOption1 XLOption2"
```

An example configuration file is shown in the following example.

Example 2 Example Configuration File

```
##### STIL2VERILOG CONFIGURATION FILE #####

# Specifies the maximum number of patterns
# loaded simultaneously in the simulation process
set cfg_patterns_read_interval 1000

# Specifies the interval of the progress message
set cfg_patterns_report_interval 5

# Control for a prespecified set of trace options
set cfg_message_verbosity_level 3

# Generates a failures log file compliant with
# TestMAX ATPG diagnostics
set cfg_diag_file "diag_file"
```

Chapter 32: Using MAX Testbench MAX Testbench Error Messages and Warnings

```
# Specifies the DUT module name to be tested
#set cfg_dut_module_name "dut_module_name"

# Specifies all source files required to run the simulation
#set design_files "netlist1.v netlist2.v"

# other configurations...
```

To assign a value to a configuration parameter, use the following syntax:

```
set config_parameter_name value
```

A comment line must begin with the pound symbol (#).

See Also

- [Runtime Programmability](#)
- [Predefined Verilog Options](#)

MAX Testbench Error Messages and Warnings

The following sections list and describe the various error messages and warnings associated with MAX Testbench:

- [Error Message Descriptions](#)
- [Warning Message Descriptions](#)
- [Informational Message Descriptions](#)

You can access a detailed description for a particular message by specifying either of the following commands:

```
stil2Verilog -help [message_code]
```

or

```
write_testbench -help [message_code]
```

Error Message Descriptions

The following table describes all MAX Testbench error messages:

Table 11 Error Message Descriptions

Error Message	Description	What Next
E		
E-001- No license found for this site	The license file specified in the Synopsys installation does not contain a valid license for this site.	Check the <code>SYNOPSYS</code> environment variable or contact Synopsys to get a valid license.
E-002- No threads associated with the first PatternExec	The tool automatically searches for the first PatternExec statement in the specified STIL file. Its name is displayed in the verbose mode execution. This message occurs when the STIL interpretation process failed to retrieve any execution threads corresponding to the detected PatternExec statement.	Check the validity of the STIL file and its first PatternExec statement.
E-003 - Multiple PatList found, not fully supported yet (only one at a time or in parallel but with PLL like patterns)	The PatList statement is not yet fully supported. The tool only supports for now only simple PatList representations, like the PLL like patterns.	Generate a STIL that uses the supported PatList syntax and patterns .
E-006- Cannot recover signal <name> from the STIL structures, last label <name>	Respective signal cannot be found in the Signals list of the STIL file.	Check the STIL file syntax
E-007- Unsupported event %s in wave of cluster "%c" of signal %s in WFT "%s"	The tool currently does not support the following event types: WeakDown, WeakUp, CompareLowWindow, CompareHighWindow, CompareOffWindow, CompareValidWindow, LogicLow, LogicHigh, LogicZ, Marker, ForcePrior	Generate a STIL that uses only the supported event types

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-008- The event waves of cluster <name> of signal <name> in WFT <name> have incompatible types (force and compare simultaneously, not yet supported)	The cluster of reported signal contains both force and compare event waves simultaneously. The tool does not support this yet .	Generate a STIL that does not use this type of event waves in the WaveForm description
E-010- Can't find definition for <name> in the STIL structures	The specified Procedure or Macro cannot be found in the STIL structures. That can be caused by an incomplete STIL file.	Check the syntax of the STIL file
E-011- Too many signal references in the Equivalent statement %s, not yet supported	The tool only supports one to one equivalences for now and the input STIL file contains Equivalent statements with multiple signal specifications.	Generate a STIL that contains only one to one equivalences
E-013- Invalid Equivalent statement <location>	The tool only supports one to one equivalences for now and the specified. Equivalent statement does not respect this rule.	Generate a STIL that contains correct Equivalent statements
E-014- Loop Data statement in <name> not yet supported	Only the simple Loop statement is currently supported. The Loop Data is not yet supported.	Generate a STIL that does not contain Loop Data
E-015 - The requested help page does not exist	A message code was specified that does not correspond to an existing help page.	Check the correctness of the message code
E-017- Duplicate definition for <name>	There is more than one definition for a specified Procedure/Macro in the input STIL file. This represents a bad STIL syntax and should be corrected.	Check the syntax of the input STIL file
E-018- Multiple specification of -log option	The command line -log option has been specified more than one time. Only one specification is allowed to avoid confusion.	Check and edit the command line to have a single -log specification
E-019- Missing "log" option value	The command line -log option has an mandatory argument that specifies the name of the file which is used to write the transcription of the tool execution. This argument is absent.	Check and edit the command line to add a file name as argument for -log

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-021- Error during the consistency checking of the command line parameters and options	The error message indicates which parameter/option is cone timerned.	Modify the command line according to the error message. Check the user documentation for more details
E-023- cannot write file <file_name> as it already exists, specify -replace if you want to overwrite it	When the tool is about to generate a file it checks if the respective file name already exists on disk. In this case, to avoid accidental lost of user important data the tool asks the user for a confirmation, more specific the user has to provide the -replace option in the command line to confirm that this is the desired behavior.	If the overwriting of the respective file is desired then add the -replace option in the command line
E-024- Ambiguous option <name>, can match multiple options like <enum>	The specified command line option match more than one command line option. The command line processing allows for incomplete option name specifications, but a minimal specification is required to avoid ambiguity.	Edit the command line and clearly specify your options to avoid ambiguity
E-025- <file/directory_name> No such file or directory	The specified file(s) or folder(s) cannot be found on disk. This usually is caused by a wrong specification of the design/library files generated from the command line or from the config file.	Specify correct file/folder names
E-028- <value> is not a valid cfg_time_unit or cfg_time_precision value (Valid integer value are 1, 10 and 100. Units of measurement are s, ms, us, ns, ps and fs)	Specified value for cfg_time_unit or cfg_time_precision is invalid. This usually occurs in the config file consistency checking process.	Edit the invalid values with correct ones
E-029- It is illegal to set the time precision larger than the time unit	Value specified for time precision is too big.	Specify a lower value for time precision, lower or equal with the time unit
E-030- Cannot generate Verilog testbench neither for serial nor for parallel load mode...	Specified testbench generation mode is not possible with the given STIL file. This might happen when you specify the parallel_only or serial_only configuration.	Specify a different simulation mode

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-031- Cannot open <file_name> file.	Specified file name is not accessible. It can be a config file name, a log file name, design file name, library file name, test data file, protocol file, and so forth.	Check the existence, the location, or the permission of the specified file
E-032- Error during the consistency checking of config_file data	The error message indicates which config file field is affected.	Modify the config file according to the error message
E-033- Error reading Tcl file <file_name> at line <#>. Only comments and variable settings allowed	The config file only supports a limited Tcl syntax, such as variable settings, comments and empty lines.	Modify the config file by removing the unsupported syntax.
E-035- Cannot retrieve DUT module name in STIL file. Set the "cfg_dut_module_name" in the config file to avoid the problem	The tool automatically extracts the DUT module name from the specified STIL file.	Use a config file to specify it by setting the cfg_dut_module_name parameter. A template config file can be generated using the -generate_config option
E-036- Detected an unsupported multi-vector Shift construct.	The tool detected a STIL Shift block that includes multiple Vector statements – some of which are not consuming data without a pound (#) sign .	Make sure the vectors are not intended to be post-amble (or preamble) vectors that need to be defined after (or before) the Shift block. If so, correct the STIL file accordingly. If not, contact Synopsys support.
E-037- Detected an unsupported multi-vector Loop construct.	The tool detected a STIL Loop block that includes multiple Vector statements.	USF Parallel simulation is not supported for STIL files using these type of constructs.
E-038- Cannot process MISR outputs. Theratio between the number of compressors and the number of SERDES MISR outputs is not supported. Parallel simulation may fail	The tool detected a situation in which it can't determine the assignment between the compressor outputs and the SERDES MISR output	If possible, use a number of compressors that can divide with the number of SERDES MISR outputs. The simulation may fail otherwise.

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-039- Shift statement can only be called from Procedures	Shift statements are only supported when they are called inside a Procedure.	Generate a STIL file that respects this syntax
E-040 - Wrong values for -first and/or -last options	The first and last options need to be positive integers and in increasing order (last > first). First and last must both be less than max_patterns.	Set the appropriate values.
E-041 - Parallel simulation mode for loop block within procedure "proc"	Parallel simulation for a STIL file with a loop block consuming scan data within a load_unload procedure is not supported.	Regenerate a "serial_only" STIL version from TestMAX ATPG or use the -ser_only MAXTestbench option (in case of USF STIL) to generate the appropriate testbench and run the simulation in serial mode.
E-042 - Error during the consistency checking of the input STIL file	Identifies a missing structure or field in the STIL file.	Add the missing structure or field in the input STIL file.
E-043 - Enhanced Debug Mode for Combined Pattern Validation (EDCPV)	Due to some consistency checks, EDCPV mode cannot be activated. As a result, the generated testbench cannot pinpoint the exact failing scan cell in parallel simulation mode.	Refer to the requirements described in " Debugging Parallel Simulation Failures Using Combined Pattern Validation. "
E-044 -Detected an invalid multibit scan cell. Simulation cannot be performed in parallel mode	MAX Testbench detected multibit scan cells that are incorrectly described. In this case, parallel mode simulation is not possible, since the respective scan cell cannot be correctly identified in the design.	Check the input STIL file and the TestMAX ATPG parameters for errors.

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-045 - Cannot find integer variable <string> used in Loop	The variable specified in the Loop statement is incorrect. The Loop statement accepts an integer variable declared in the Variables block or an integer expression. The support for integer expressions is limited to simple divisor operations. Integer expressions must be delimited by quotes. For example, use the following: <code>Loop 'cnt/4'</code> Where <code>cnt</code> is the name of the variable. If the variable name is surrounded by double quotes, it must be correctly represented in the expression.	Fix the STIL protocol so it uses the correct syntax.
E-046 - Invalid number of connections in the ShiftPowerControllersStructures block for the scan chain	STIL analysis shows that the ShiftPowerControllerStructures block defines a number of connections bigger than the number of cells in the scan chain.	Please fix the ShiftPowerControllerStructures block definition in the STIL file.
E-047 - Internal computation failure due invalid order of ScanChainGroups in the STIL file	STIL analysis shows that the ScanChainGroups are not described in the order of the Compressors	Please fix the order in the STIL file.
E-048 - Internal computation failure due invalid number of scan inputs/outputs of SeqCompressor	STIL analysis shows that the ScanChainGroup is incorrectly described (or missing)	Please fix the order in the STIL file.
E-049 - I(ATE Protocol Error) Detected STIL value (%s) different from requested UI value(%s)	Enabling protocol-based TB using two conflicting setting values (STIL annotation and UI <code>-ate_prot</code> option))	Please, review requested protocol and either align both values or set only one single value.
E-050 - Internal computation failure due to invalid patterns length in the STIL File	STIL analysis shows that the length of patterns in pattern block are different.	Please fix the STIL file to have equal length of patterns

Table 11 Error Message Descriptions (Continued)

Error Message	Description	What Next
E-051 - PiP for Normal Scan mode not supported without CellType information	STIL analysis shows that the celltype information is missing for the current configuration, which is required for Parallel simulation.	Please Use "-cellnames type" in <code>write_patterns</code> to generate STIL with required information.
E-052 - Scanned input(s) are missing from LoadPipelineElements structure	STIL analysis shows that description of scan input from the signal groups is missing in the LoadPipelineElements structure.	Please Use "-usf2 " in <code>write_patterns</code> to generate STIL with required information.
E-053 - min_ate_clock_cycle missing from STIL history block	STIL analysis shows that the min_ate_clock_cycle required for XLBIST pattern parallel simulation is not present in the STIL history block.	Check the input STIL file.
E-054 - Generic testbench not supported for Sequential DFT Type or serial only simulation mode in xtb2.	Generic testbench not supported for Sequential DFT Type or serial only simulation mode in xtb2.	Please do not use <code>-generic_testbench</code> in <code>stil2verilog</code> call
E-999 - The tool has just encountered an internal error		Submit a test case that reproduces the problem to the Synopsys Support Center at: http://solvnet.synopsys.com/EnterACall .

Warning Message Descriptions

The following table lists all MAX Testbench warning messages and their descriptions.

Table 12 MAX Testbench Warning Messages

Warning Message	Description	What Next
<p>w-000: Failed to initialize error file <file_name>, no STIL syntax error messages are available</p>	<p>This message occurs when the reported error filename is invalid, does not exist or the user does not have access rights to it. This does not affect the tool execution, but the eventual STIL syntax error messages will not be displayed.</p>	<p>If this is not the expected behavior, then check the file path and the SYNOPSIS environment variable.</p>
<p>w-001: Multiple assignments for signal <name> (old value <value>), proceeding with <value>, last label <name></p>	<p>This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing. This message is displayed only in verbose mode.</p>	<p>Check the STIL file if this is not the expected behavior.</p>
<p>w-002: Multiple assignments for signal <name> in signal group <name>, proceeding with <value>, last label <name></p>	<p>This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing. This message is displayed only in verbose mode.</p>	<p>Check the STIL file if this is not the expected behavior.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>w-003: Multiple assignments for inout signal <name> in signal group <name> without a WFCMap specified (<values>), last label <name></p>	<p>This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing.</p>	<p>Check the STIL file if this is not the expected behavior.</p>
<p>w-004: Insufficient data for signal group <name>, ignoring signal <name></p>	<p>This message occurs for signal groups when the length of the data assigned to it is less than the length of the signal group itself. In this case the signals for which there is no data to be assigned are ignored. This is usually caused by an incorrect STIL.</p>	<p>Check the STIL file if this is not the expected behavior.</p>
<p>w-005: Multiple assignments for sig <name>, proceeding with <value></p>	<p>This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), if there was needed a WFCMap specification, and the name of the last Label observed during processing. This message is displayed only in verbose mode.</p>	<p>Check the STIL file if this is not the expected behavior.</p>
<p>w-006: Cannot build testbench in parallel load mode (no scan chains found)</p>	<p>This message occurs when the tool did not detect any scan chains in the input STIL file. Without the full description of the scan chains a parallel load mode testbench cannot be generated.</p>	<p>Check the STIL file syntax or regenerate the STIL file using the latest versions of TestMAX DFT and TestMAX ATPG.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-007: SYNOPSISYS and SYNOPSISYS_TMAX environment variables have different values, SYNOPSISYS_TMAX is considered in this case	This message occurs then both SYNOPSISYS and SYNOPSISYS_TMAX environment variables are specified but with different values. In this case the values specified by the SYNOPSISYS_TMAX environment variable is considered.	If this is not the desired behavior, specify the correct environment variables.
w-008: Failed to retrieve WFC <wfc> of signal <name> from WFT <name>, processing its string value, last label <name>	This message occurs when a signal is assigned a WFC that is not described in the current WFT. In this case the tool will try to interpret the WFC behavior using its string value instead of the WFT. This message is displayed only in verbose mode.	Check the STIL file if this is not the expected behavior.
w-009: Failed to retrieve WFC <wfc> for signal <name> of group <name> in WFT <name>, processing its string value, last label <name>	This message occurs when a signal inside a signal group is assigned a WFC that is not described in the current WFT. In this case the tool will try to interpret the WFC behavior using its string value instead of the WFT. This message is displayed only in verbose mode when the concerned signal is of type Pseudo.	Check the STIL file if this is not the expected behavior.
w-010: Cannot build testbench in parallel load mode (no cells specified in <name> scan chain)	This message occurs when the tool did not detect any scan cells in the respective scan chain. Without the full description of the scan chains a parallel load mode testbench cannot be generated.	Check the STIL file syntax or regenerate it using the latest versions of TestMAX DFT and TestMAX ATPG.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-011: Multiple assignments for signal <name> in Vector stmt, proceeding with <value>, last label <name>	This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), and the name of the last Label observed during processing.	Check the STIL file if this is not the expected behavior.
w-012: Cannot generate simulation script file (DUT module name missing)	This message occurs when the tool was not able to automatically detect the name of the DUT module and a simulation script is requested. In this case the script file will not be generated.	Specify the DUT module name using the command line or the configuration file.
w-013: NETLIST_FILE S variable in the simulation script file is empty (design files missing)	This message occurs as a simulation script have been requested but no design files have been specified, neither using the command line -v_file option nor the design_files variable in the configuration file. In this case the script file is not completed.	Specify the design files by editing the generated simulation script file.
w-014: LIB_FILES variable in the simulation script file is empty (library files missing)	This message occurs as a simulation script have been requested but no library files have been specified, neither using the command line -v_file option nor the lib_files variable in the configuration file. In this case the script file is not completed.	Specify the library files by editing the generated simulation script file.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-015: Parallel option ignored as <code>-serial_only</code> testbench requested	When a <code>serial_only</code> testbench is requested then, as expected, all the parallel options are ignored. The user is warned to avoid any confusion.	If this is not the expected behavior, change the testbench generation mode.
w-017: Detected BIST Structures are not supported and might generate errors	The tool detected BIST constructs in the input STIL file. These constructs are not supported and might generate errors if used during execution or simulation.	Generate a STIL file that does not contain BIST constructs.
w-018: Specified time precision <code><value></code> too large. This can cause errors during simulation	The value specified for <code>cfg_time_precision</code> in the configuration file is too large.	Edit the configuration file and change the value accordingly.
w-019: Parallel <code>nshift</code> parameter not supported for scan compression designs. Ignored.	In the case of scan compression designs, the tool can generate a testbench for parallel load mode simulation with <code>nshift</code> only when the input STIL file supports the Unified STIL flow.	Regenerate the STIL file using the default mode of the <code>write_patterns</code> command.
w-020: <code><name></code> parameter not yet supported (ignored)	Certain parameters enumerated in the configuration file example are not yet supported.	A full list of the supported parameters can be found in the user guide. If specified, these parameters are ignored.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-021: Test bench module name already defined in command line. "cfg_tb_module_name" variable in the configuration file ignored	The testbench module name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification.
w-022: Design files already defined in command line. "design_files" variable in the configuration file ignored	The design file name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification.
w-023: Library files already defined in command line. "lib_files" variable in the configuration file ignored	The library file name can be specified both in command line and in the configuration file. If both specified, then the command line specification has priority and so the configuration file specification is ignored.	If this is not the expected behavior, then remove the command line specification.
w-024: Unknown <name> variable (ignored)	The reported variable name is not part of the configuration file syntax.	To find the correct syntax of this file you can generate a configuration file template using the <code>-generate_config</code> option.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-025: Configuratio n file <file_name> does not contain any variable setting	The specified input configuration file does not contain any variable settings.	Check the configuration file content or path if that is not the expected behavior.
w-026: Invalid load/unload chains or groups of ctlCompresso r <name>	The ctlCompressor block is not valid because the load/unload chains or groups are not correct (i.e.: some scan chains are specified in the groups but are undefined or empty). Since the ctlCompressor block is wrong, it is not possible to run a parallel simulation from a serial formatted STIL file.	Check the STIL file and rerun TestMAX DFT or TestMAX ATPG, if necessary.
w-030: Detected Serial Only test patterns, the generated testbench can only be run in serial simulation mode	This occurs either when the user intentionally requested a serial only testbench or when the provided STIL file does not contain enough information to allow a parallel load mode simulation also.	Check the STIL file, TestMAX ATPG script and the options of the <code>write_patterns</code> command and the DFT script used with DFT compiler, and make sure that this is the desired behavior.
w-031: Detected Parallel Only test patterns, the generated testbench can only be run in parallel simulation mode	This message occurs when the provided STIL file contains pure parallel patterns, specially formatted for a parallel simulation. These patterns can't be simulated serially.	Check the STIL file, TestMAX ATPG script and the options of the <code>write_patterns</code> command and the DFT script used with DFT compiler and make sure that this is the desired behavior.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-032: Parallel nshift parameter too small (minimum <value> serial shift required)	This message occurs when the user specifies a parallel nshift parameter too small. A wrong nshift parameter value might cause the simulation to fail.	Change the parallel nshift parameter using the -parallel command line option of MAX TestBench or the -parallel option of the write_patterns command of TestMAX ATPG.
w-033: Unified STIL Flow for Serializer is not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support Unified STIL Flow mode for Serializer architecture.	Contact Synopsys support for the next available release supporting Unified STIL Flow mode for Serializer.
w-034: Unified STIL Flow for multiple shifts load/unload protocol not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support Unified STIL Flow mode for multiple shifts load/unload protocol.	Contact Synopsys support for the next available release supporting Unified STIL Flow mode for multiple shifts load/unload protocol.
w-035: Parallel load mode simulation of multi bit cells not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support parallel load mode simulation of multi bit cells.	Contact Synopsys support for the next available release supporting parallel load mode simulation of multibit cells.

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
w-036: Scan cell with multiple input ports not yet supported: parallel load mode simulation might fail	The current version of MAX Testbench does not support scan cell with multiple input ports. Since the tool cannot force all the specified input ports, parallel load mode simulation might fail.	Contact Synopsys support for the next available release supporting parallel load mode simulation of multiple inputs.
w-037: Unified STIL Flow for Sequential Compression is not yet supported. Mode forced to serial only simulation	The current version of MAX Testbench does not support the Unified STIL Flow mode for Sequential Compression architecture.	Contact Synopsys support for the next available release supporting Unified STIL Flow mode for Sequential Compression.
w-038: Testbench data file requiring very large memory, automatically using/updating -split_out to <value>	MAX Testbench has detected that the testbench data file size required a memory buffer larger than the one supported currently by Verilog 1995 (the default testbench output). To avoid a Verilog simulation failure, the pattern data has been written out in multiple .dat files; each file will contain a maximum number of patterns specified by the <code>-split_out</code> value. A mapping with all the created partitions is reported at the end of MAX Testbench execution. Use this map to simulate the desired partition. For example, <code>simv +tmax_part=0</code>	

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-039: Delayed release time (cfg_parallel_release_time) set in configuration file <> ignored (valid only for DSF parallel STILs).</p>	<p>The configuration option <code>cfg_parallel_release_time</code> is not supported for a USF STIL, nor for a serial-only STIL file</p>	<p>No action required. This message is a notification that the set value is not recognized by MAX Testbench.</p>
<p>W-040: Unified STIL Flow for Scalable Adaptive Scan is not yet supported. Mode forced to serial only simulation.</p>	<p>The current version of MAX Testbench does not support the Unified STIL Flow mode for Scalable Adaptive Scan architecture.</p>	<p>Contact Synopsys support for the next available release supporting the Unified STIL Flow mode for Scalable Adaptive Scan.</p>
<p>W-041: Disabling the Enhanced Debug Mode for Unified STIL Flow (EDUSF).</p>	<p>Due to consistency checks, EDUSF mode cannot be activated. The generated testbench will not be able to pinpoint the exact failing scan cell in parallel simulation mode.</p>	
<p>W-042: Pattern-based failure data format in serial load mode simulation is not compliant with the TestMAX ATPG diagnosis tool.</p>	<p>The pattern-based failure data format of DFTMAX Ultra Chain Test in serial load mode simulation is not compliant with the TestMAX ATPG diagnosis tool.</p>	<p>Use a cycle-based failure data format in serial load mode simulation for DFTMAX Ultra Chain Test in serial load mode simulation. Contact Synopsys support for the next available release with the full support of pattern-based failure data format.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-043: Testbench data file requiring very large memory, simulation might fail. Please regenerate using <code>-split_out</code> with a max value of <code><XXX ></code>.</p>	<p>According to MAX Testbench (pessimistic) estimation, the size of at least one of the resulting .dat files may exceed disk space. If this is intentional, you can ignore the warning and proceed with the simulation. Otherwise, use a value smaller or equal to the value computed by MAX Testbench.</p>	<p>Automatic split-out is enabled when the <code>-split_out</code> option is not set. So either remove this option and let MAX Testbench automatically compute and set the required split out value, or follow the warning recommendation and set a value smaller or equal to the one specified. If you know you have enough space and the logic simulator runs correctly, you can proceed with the simulation. For further details, see the description of the W-038 message.</p>
<p>W-044: Detected invalid multibit scan cell, simulation cannot be performed in parallel mode.</p>	<p>MAX Testbench detected multibit scan cells that were incorrectly described. In this case, a parallel mode simulation is not possible since the respective scan cell can't be correctly identified in the design.</p>	<p>Check the input STIL file and the TestMAX ATPG parameters for errors.</p>
<p>W-045: Unified STIL Flow disabled due to errors detected in the DFT structure processing. Mode forced to serial only simulation.</p>	<p>MAX Testbench detected unexpected errors while processing the DFT structure. As a result, the parallel simulation mode of the unified STIL flow is disabled.</p>	<p>Check other errors and warnings reported during the MAX Testbench execution.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-046: Unified STIL Flow for Sequential Plus is only supported for multi-core architectures. Mode forced to serial only simulation.</p>	<p>MAX Testbench detected only one core during the processing of the DFT structure. In this case, the parallel simulation mode of the unified STIL flow is disabled.</p>	<p>Check other errors and warnings reported during the MAX Testbench execution.</p>
<p>W-047: Specified <code>cfg_reverse_bus_order</code> cannot be found in the design, ignored .</p>	<p>MAX Testbench detected that the identified port name is incorrect.</p>	<p>Make sure you set the correct port name in the STIL file.</p>
<p>W-048: Failing scan cell name display feature may have an impact on simulator performance .</p>	<p>The failing scan cell name display functionality increases the Verilog testbench size and the memory usage. On large designs, this feature can impact the simulator performance.</p>	<p>Disable the failing scan cell name display functionality and use the <code>-failing_cell_report</code> option to post-process the simulator log file. Example: <code>stil2verilog pats.stil</code> <code>-failing_cell_report simv.log</code></p>
<p>W-049: Specified Inverted Output port cannot be found in the design or it's not a scanout port, ignored.</p>	<p>MAX Testbench is unable to localize the specified port in the DUT port list or it is not a scanout port, which is currently the only port type supported.</p>	<p>Make sure you set the correct scanout port name in the STIL file.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-050: Parallel release time (%0f) is bigger than strobe time or clock edge time; parallel simulation may fail.</p>	<p>The release time value specified by <code>cfg_parallel_release_time</code> command is not compatible with the strobe time or clock edge time.</p>	<p>If parallel simulation fails, check the release time value.</p>
<p>W-051: <code>ctlCompressor</code> <code><name></code> has <code><value></code> unconnected modes: ignored.</p>	<p>The <code>ctlCompressor</code> block has some unconnected modes and MAX Testbench ignores them.</p>	<p>If this is not the expected behavior, fix the unconnected modes in the STIL file.</p>
<p>W-052: Compressors are using only <code><value></code> out of <code><value></code> connections of <code>UnloadSerial</code> <code>izer <name></code>.</p>	<p>Some of the <code>UnloadSerializer</code> connections are unused.</p>	<p>Check the input STIL file to determine if this is the intended behavior.</p>
<p>W-053: Invalid signal <code><name></code>.</p>	<p>MAX Testbench detects an invalid signal name because of an escape character and attempts to fix the issue.</p>	<p>If the simulator fails during Verilog testbench compilation, check the specified signal name in the STIL file.</p>
<p>W-054: Serial simulation not supported for this DFT architecture . Forcing Parallel only simulation.</p>	<p>The tool detected a DFT architecture where USF STIL file can be used only for Parallel simulation.</p>	<p>Serial simulation is not supported for this USF STIL file.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-055: Detected an invalid InternalShiftStart value in the SerializerStructures block, automatically computing a new value.</p>	<p>The tool detected a DFT Serializer type configuration where the InternalShiftStart statement present in the STIL seems invalid.</p>	<p>The tool will automatically compute internally a new value but if the Parallel simulation fails please review the original InternalShiftStart value.</p>
<p>W-056: Requesting Unified STIL Flow 2 (USF2) algorithm for unsupported scan compression architecture. Switching to traditional USF mode</p>	<p>Unified STIL Flow 2 (USF2) cannot be applied for the selected scan compression architecture. Therefore only traditional USF can be applied for now.</p>	<p>Please contact Synopsys to be informed if and when the selected scan compression architecture will be supported.</p>
<p>W-057: Missing required information (internal nodes) for Unified STIL Flow 2 (USF2). Switching to traditional USF mode</p>	<p>Unified STIL Flow 2 (USF2) requires certain mandatory information that are missed. Therefore only traditional USF can be applied for now.</p>	<p>This could be a bug or because an incompatible STIL file was used. Please contact Synopsys for support.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-058: Invalid number of scan inputs/outputs of SeqCompressor, ignoring</p>	<p>STIL analysis shows that the ScanChainGroup is incorrectly described (or missing)</p>	<p>Please fix the order in the STIL file.</p>
<p>W-059: Disabling the Enhanced Debug Mode for Combined Pattern Validation (EDCPV)</p>	<p>Due to some consistency checks, EDCPV mode cannot be activated, the generated testbench will not be able to pinpoint the exact failing scan cell in parallel simulation mode.</p>	<p>Please refer to the User's Guide (or contact Synopsys) to understand the requirements for EDCPV mode.</p>
<p>W-060: Scanout is being observed in capture mode, may result in parallel simulation failure</p>	<p>STIL analysis shows that scanout are being observed in the capture mode. It can result in parallel simulation failure</p>	<p>Please fix the issue in STIL file</p>
<p>W-061: No v2 license found for this site. Turning off PI/PO Optimization .</p>	<p>To Use PI/PO optimization feature, you need to MaxTB v2 or need to provide Test-CA license with MaxTB.</p>	<p>Please use MaxTB v2 or get Test-CA license</p>
<p>W-062: No v2 license found for this site. Turning off PSD v3.0 Processing.</p>	<p>To Use PSD v3.0 feature, you need to provide Test-CA license with MaxTB. If you want to switch to legacy PSD, Please use -legacy in run_sim command while generating PSD data from TetraMAX ATPG</p>	<p>Please get Test-CA license or use -legacy in run_sim command.</p>

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
W-063: No PSD file available for the cores	There are no PSD files found for the mentioned cores. Hence the Enhanced Debug mode won't be available for these cores during the simulation.	Please Update the STIL file to point to PSD files.
W-064: MISR might not be correctly filled in the misr_load proc for parallel simulation		
W-065: Mismatch in the number of chains in misr_group and number of inputs to misr_load proc		
W-066: Partition Group selects no chains	The specified group doesn't have any chains specified in the SelectedChains block.	Please make sure this is expected.
W-067: ActiveScanChains not found in definition	It seems that the ActiveScanChains is not correctly described in the definition of the procedure.	Please make sure this is expected.
W-068: Ignoring -first option	-first option not supported with -generic_testbench option.	Generate testbench using -generic_testbench option and then generate only .dat file with -patterns_only and -first

Table 12 MAX Testbench Warning Messages (Continued)

Warning Message	Description	What Next
<p>W-069: Parallel Simulation strobe cannot be provided for Scancell</p>	<p>In parallel Simulation this Scancell cannot be strobed due to presence of SNPS_PIPELINE keywords in the ScanChain. The value will be masked.</p>	<p>Generate STIL file with Cell type information enabled.</p>

Informational Message Descriptions

Table 3 lists all MAX Testbench informational messages and their descriptions.

Table 13 Informational Message Descriptions

Info Message	Description	What Next
<p>I-001 - nshift parameter is greater or equal than the maximum scan chain length (%d in the current design)</p>	<p>This message indicates that the value specified for the nshift parameter is greater or equal than the maximum scan chain length. In this situation, as expected, the simulation becomes a serial one.</p>	<p>This is an expected behavior.</p>
<p>I-002- Time unit sets to <value></p>	<p>This is a message to inform the user that he is about to overwrite the automatic setting for this parameter with a specified value using the cfg_time_unit parameter from the configuration file.</p>	<p>This is an expected behavior</p>
<p>I-003- Time precision sets to <value></p>	<p>This is a message to inform the user that he is about to overwrite the automatic setting for this parameter with a specified value using the cfg_time_precision parameter from the configuration file.</p>	<p>This is an expected behavior</p>
<p>I-004- Multiple assignments for signal <name> in signal group <name>, using WFCMap and proceeding with <value>, last label <name></p>	<p>This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), the WFCMap resulting value, and the name of the last Label observed during processing. This message is displayed only in verbose mode.</p>	<p>This is an expected behavior</p>

Table 13 Informational Message Descriptions (Continued)

Info Message	Description	What Next
I-005- Event ForceOff (Z) interpreted as CompareUnknown (X) in the event waves of cluster "X" of Signal "%s" in WFT "%s"	This message describes how the tool interprets certain 'unusual' constructs found in the waveform table. These constructs are usually encountered when processing older versions of STIL.	This is an expected behavior
I-006- Multiple assignments for sig <name>, using WFCmap and proceeding with <value>	This message occurs when a signal is assigned multiple values inside a statement. The signal can be part of a SignalGroup or all the assignments can be SignalGroups. If possible, the tool will report the location where this happens, the parent Macro/Procedure name (if any), the WFCMap resulting value, and the name of the last Label observed during processing. This message is displayed only in verbose mode.	This is an expected behavior
I-007- Event ForceOff (Z) interpreted as CompareUnknown (X) in the event waves of WFT "%s" containing both compare and force types	This message informs the user about how the tool interprets certain 'unusual' constructs found in the waveform table. Usually encountered when processing older versions of STIL.	This is an expected behavior
I-008- Requesting <name> EVCD file generation (use "tmax_vcde" simulator compiler definition to enable file generation)	User specified a EVCD file in the configuration file. The tool will update the testbench but the simulation will not generate the EVCD file by default.	Specify the "tmax_vcde" simulator compiler definition to enable file generation
I-009- Updated Serializer Tail Pipeline internally to zero due to shorter Serializer data length	In the case of DFTMAX Serializer with slow pipelines (core pipelines), for some configurations TestMAX ATPG does not consider the Serializer Tail pipeline stages as expected by MAX Testbench. When this occurs, MAXTestbench attempts to compensate for this behavior.	This situation rarely occurs.

Table 13 Informational Message Descriptions (Continued)

Info Message	Description	What Next
I-011- The following clocks will not be pulsed during the parallel Shift: <i>list_of_clocks</i>	Lists the clocks that will not be used during the parallel shift simulation.	See the I-011 manpage for additional details.

Troubleshooting MAX Testbench

The following sections describe how to resolve MAX Testbench-generated errors:

- [Introduction](#)
- [Troubleshooting Compilation Errors](#)
- [Troubleshooting Mismatches](#)
- [Debugging Simulation Mismatches Using the write_simtrace Command](#)

Introduction

You can run a design against a set of predefined stimulus and check (validate) the design response against an expected response. This process mimics the behavior of the tester against a device under test.

Problems might occur with

- incorrect or incomplete STIL data
- incorrect connections of the device to this stimulus in the testbench
- incorrect device response due to structural errors or timing problems inside the design

Ultimately, the goal of using a testbench is to validate that the device response, often with accurate internal timing, does match the response expected in the STIL data.

There are alternative and additional troubleshooting strategies to what is presented in this section. The most important aspects when testing are knowledge of the design and remembering the fundamental characteristics of the test you're troubleshooting.

Troubleshooting Compilation Errors

This section describes some of the typical error messages you encounter during compilation when using VCS or Ncsim.

These error messages are related to the following parameters or issues:

- [FILELENGTH Parameter](#)
- [NAMELENGTH Parameter](#)
- [Memory Allocation](#)
- [MDEPTH Parameter](#)

FILELENGTH Parameter

The following error message appears if you exceed the maximum file length:

```
XTB Error: cannot open /disk/path.to.a.large.file.name.maxtb_psd.dat PSD
file. Disabling Enhanced Debug USF mode...
```

By default, the FILELENGTH parameter in MAX Testbench is set to 1024 characters, which corresponds to the 1024 character limit imposed by NCSIM. In some cases, you can set this parameter to a higher limit at the compilation stage either in the testbench file or at the simulation command line.

You can use the following MAX Testbench parameter to change the maximum file length:

```
parameter FILELENGTH = 1024; // max length for file names
```

If you are using a set of long paths, you can set the Verilog FILELENGTH parameter in the testbench, using the following syntax:

```
-pvalue+tb_name. FILELENGTH=your_value
```

You also might encounter the following error:

```
Warning-[STASKW_CO] Cannot open file
/disk/some.path.name.to.a.very.large.file.name.maxtb.Verilog.gz, 8535
The file
  /disk/some.path.name.to.a.very.large.file.name.maxtb.maxtb_psd.dat '
could not be opened. No such file or directory.
Ensure that the file exists with proper permissions.
XTB Error: cannot
open /disk/some.path.name.to.a.very.large.file.name.maxtb_psd.dat PSD
file. Disabling Enhanced Debug USF mode...
```

For exceptionally long paths, you can override the Verilog parameter in the testbench and specify an extended file length at the simulation recompile command line using the following syntax:

```
vcs -pvalue+tb_name. FILELENGTH=your_value
```

NAMELENGTH Parameter

For parallel strobe data (PSD) files, the default filename length is 800 characters. If you exceed this length, the following message appears:

```
Warning-[STASKW_CO] Cannot open file
./LongName.p.maxtb.v, 1278
The file 'ReallyLongName.p.maxtb_psd.dat' could not be opened. No such
file or directory.
Ensure that the file exists with proper permissions.
XTB Error: cannot open ReallyLongName.p.maxtb_psd.dat PSD file. Disabling
Enhanced Debug USF mode...
```

To correct this error, you can set the NAMELENGTH parameter in the testbench or at the simulation recompile line using the following syntax:

```
vcs -pvalue+tb_name.NAMELENGTH=800
```

Memory Allocation

The following error message identifies a memory allocation error:

```
XTB Error: size of test data file filename .dat exceeding testbench memory
allocation. Exiting...
(recompile using -pvalue+design1_test.tb_part.MDEPTH=<###>).
```

In this case, you need to recompile the testbench using the following Verilog parameter to adjust the memory allocation:

```
-pvalue+design1_test.tb_part.MDEPTH=depth)
```

For more information, see [MAX Testbench Runtime Programmability](#).

MDEPTH Parameter

The following warning message appears during compilation:

```
Warning-[STASKW_RMIEAFL] Illegal entry
maxtb.v, 934
Illegal entry found at file /temp/patterns/maxtb.dat line 422
while executing $readmem.
Please ensure that the file has proper entries.
```


This happens if you change the value of the default MDEPTH using the `-pvalue +top_test.MDEPTH` option and the specified value is too small for the size of the `.dat` file used with the testbench.

If this warning is ignored, this leads to an error in the testbench simulation later. This warning should always be changed to error so that if the MDEPTH is fixed. This can be done using the `-error=STASKW_RMIEAFL` simulation runtime option.

Troubleshooting Miscompares

The following sections describe the process of debugging failures (miscompares) detected when simulating a design using MAX Testbench and a set of generated STIL pattern data:

- [Handling Miscompare Messages](#)
- [Understanding MAX Testbench Parallel Miscompares](#)
- [Localizing a Failure Location](#)
- [Adding More Fingerprints](#)

These sections also present some techniques for using MAX Testbench to assist in the analysis of simulation mismatch messages when they occur during a simulation run. These techniques start with the direct approach:

- Understanding the simulation mismatch message completely
- Proceeding to some advanced options to assist in debugging the overall simulation behavior
- Miscompares are most commonly the misapplication of STIL data and caused by either incorrect design constructs for this data
- STIL constructs for the design or the context of the application

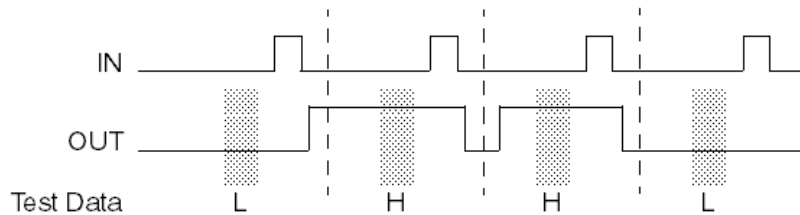
Handling Miscompare Messages

Test data is sampled at distinct points in the test pattern, which are called test strobos. Test strobos indicate whether the device is operating properly or not in response to the stimulus provided by the test data.

In general, miscompares happen only on outputs (or bidirectional signals in the output mode). This limits the visibility into both the device operation and the test data expectations, which can make analyzing these failures more complicated. Furthermore, these output measurements are placed to occur at locations of a stable device response to assure repeatable test operation. And finally, output strobe miscompares often identify an internal failure that might have happened some time in the past. All of these issues complicate the analysis process.

In the following figure, the limited visibility into the design behavior is shown by output strobe data on signal “out” that indicates this signal remains high between two test Vectors, although the actual device operation has a period of a low state between these two measurements. This is not incorrect, in fact it is probably expected design operation.

Figure 165 Measurement Points on “OUT”



This section details the four forms of miscompare messages generated by Verilog DPV and the information that each message contains.

Miscompare Message 1

```
STILDPV: Signal SOT expected to be 1 was X
```

```
At time 1240000, v# 5
```

```
With WaveformTable "_default_WFT_"
```

```
At Label: "pulse"
```

```
Current Call Stack: "capture_CLK"
```

This miscompare message is generated from a STIL Vector when an output response does not match the expected data present in the test data. The message contains a fingerprint of information to consider when analyzing this failure. It reports the nature of the error and where it happened, but does not indicate why.

- The expected state in the STIL test data, and the actual state seen in the simulation during this test strobe.
- Both the simulation time index and the STIL vector number, to cross-reference this failure in simulation time with the test data.
- The current WaveformTable name active in this vector, to help correlate this failure with the STIL data and identify what timing was active at this failure.
- The last STIL vector label seen during execution of the STIL test data. Again, this helps to correlate the failure with the STIL data. Be aware that the label might be the last one

seen if there is no label on this vector (the message reports “Last Label Seen:” if the label is not on this vector itself).

- The procedure and macro call stack, if this failure happens from inside a procedure or macro call (or series of calls).

Both the labels and the call stack information might be lists of multiple entries. Verilog DPV separates multiple entries with a backslash (\) character.

Miscompare Message 2

```
STILDPV: Signal SOT expected to be 1 was X
```

```
At time 9640000, V# 97
```

```
With WaveformTable "_default_WFT_"
```

```
Last Previous Label (6 cycles prior): pulse"
```

```
Current Call Stack: load_unload"
```

```
TestMAX ATPG pattern 7, Index 5 of chain c1 (scancell A1)
```

If the failure occurs during an identified unload of scan data during the simulation with the simulation executing serial scan simulation, then the failure message will contain an additional line of information that identifies:

- The failing pattern number from the TestMAX ATPG information.
- The index into the Shift operation that reported the failure.
- The name of the failing scan chain.
- The name of the scan cell that aligns with this index.

The index specified in this message is relative to the scan cell order identified in the ScanStructures section of the STIL data; index 1 = the first scan cell in the ScanStructures section and so on.

Miscompare Message 3

```
STILDPV: Parallel Mode Scancell A1 expected to be 1 was X
```

```
At time 9040100, V# 91
```

```
With WaveformTable "_default_WFT_"
```

```
TestMAX ATPG pattern 7, Index 5 of chain c1
```

If the failure occurs during an identified unload of scan data during the simulation with the simulation executing parallel scan simulation, then the failure message is formatted differently. It identifies:

- The failing scan cell, and the expected and actual states of that cell.
- The time that this failure was detected (beware: in parallel mode this is the time that the parallel-measure operation was performed. This is inside the Shift operation being performed, but it might not correlate with a strobe time inside a Vector, because the scan data must be sampled before input events occur).
- The WaveformTable active for this Shift.
- The failing pattern number from the TestMAX ATPG information.
- The index into the Shift operation that reported the failure.
- The name of the failing scan chain.

Like miscompare message 2, the index specified in this message is relative the scan cell order identified in the `ScanStructures` section of the STIL data; index 1 = the first scan cell in the `ScanStructures` section and so on.

Miscompare Message 4

```
STILDPV: Signal SOT changed to 1 in a windowed strobe at time 940000
```

Output strobes can be defined to be instantaneous samples in time, or “window strobes” that validate an output remains at the specified state for a region of time.

When window strobes are used, an additional error might be generated if an output transitions inside the region of that strobe. This error message identifies the signal, the state it transitioned to, and the simulation time that this occurred.

For an example of the scenario that generates this message, see [Figure 5](#).

Understanding MAX Testbench Parallel Miscompares

The following example shows the VCS script used for parallel simulation for MAX Testbench:

```
vcs -full64 -R \  
    -l parallel_stil.log \  
    +delay_mode_zero +tetramax  
  
par.v \  
-v ../lib/class.v \  
../1_dftc/result/lt_timer_flat.v \  
  
+define+tmax_rpt=1 \  
+define+tmax_msg=10
```

Localizing a Failure Location

When a failure occurs, your first debugging step is to localize the failure in the STIL data file.

The following sections describe how to localize a failure by interpreting the fingerprint information:

- [Resolving the First Failure](#)
- [Miscompare Fingerprints](#)
- [Additional Troubleshooting Help](#)

When the failure is localized, you need to determine if it's reasonable to test this output signal at this location.

With STIL constructs, an output remains in the last specified operation (the last WaveformCharacter asserted) until that operation (WaveformCharacter) is changed on that signal.

In the example that follows, a signal called “tdo” is being tested in a Vector after a Shift operation. But in the two Vectors, “tdo” is not included, because it is expected that this signal should remain in the last tested state, or should this signal have been set to an untested value (generally an “X” WaveformCharacter for TestMAX ATPG tests). Notice that the “tck=P” signal is repeated in the last two vectors, because it does not remain in the last tested state.

```
load_unload {  
W _default_WFT_  
...  
Shift { V { tdi=#; tdo=#; tck=P; } }  
V { tdi=#; tdo=#; tck=P; tms=1; }  
V { tck=P; tms=1; }  
V { tck=P; tms=0; }  
}
```

Resolving the First Failure

Subsequent failures can be caused by cascading effects; the very first error is the best error to start examining. Because basic scan patterns, starting with a scan load and ending with a scan unload, are self-contained units, failures in one scan pattern do not typically propagate—unless the failure is indicative of a design or timing fault that persists throughout the patterns (or the patterns have sequential behavior).

Don't take "first" literally as the first printed mismatch, all mismatches that happen at the same time step (or even at different times, but in the same STIL vector), are all a consequence of a problem that was functionally detected at this point. Any error generated in the first failing vector is a good starting point.

Miscompare Fingerprints

The following sections explain how to interpret the information contained in the miscompare messages and how to troubleshoot various situations:

- [Expected versus Actual States](#)
- [Current Waveform Table](#)
- [Labels and Calling Stack](#)

Expected versus Actual States

The first piece of data to analyze is the expected state (specified in the test data), and the actual state present from the simulation run.

Are all the actual states an "X" value? This can indicate initialization issues, or the loss of the internal design state during operation caused by glitches or transient events. If an "X" is found in the simulation, start tracing it backward in both the design and in simulation time—where did that X come from?

Are the mismatches hard errors? For example is a "1" expected, but it is actually a "0"? This could be caused by one of the following:

- Timing problems in the design
- Strobe positioning
- Extra or missing clocks
- Glitches, or transients

Current Waveform Table

The next piece of data in the mismatch message to analyze is the WaveformTable reference.

What are the event times specified for this strobe? What are the event times on the other inputs? Are the event relationships proper—was the test developed with the strobe events after (or before) the input events and is that timing relationship maintained in this WaveformTable?

Is there enough time between the input events and the output strobes? Does the design have time to settle before the strobe measurement?

TestMAX ATPG has distinct event ordering requirements, and the timing specified in the WaveformTable needs to be compatible with the test generation. In particular, the strobe

times must be placed before the clock pulse (pre-clock measure) or after the clock pulse (post-clock measure).

The name of the WaveformTable can sometimes help locate the failure as well. In particular for path delay environments, the name of the WaveformTable can identify the launch, capture, or combined events and isolate the failing Vector that uses that named WaveformTable.

Labels and Calling Stack

The final piece of information to analyze in the mismatch message is the referenced label and the current call stack at the failing location. This can often isolate the location of a mismatch by the presence of the label or the name of the procedure currently active when this mismatch occurred.

What activity is happening here? Is it a capture or scan operation? Is an output strobe expected here?

Additional Troubleshooting Help

Sometimes the information contained in the mismatch message is not sufficient to localize the failure in the STIL data. When this happens, the first thing to do is to activate the tracing options to get more information about what was being simulated when the failure occurred. The next section describes how to activate the MAX Testbench trace options.

Sometimes tracing might not get clearly to the failing location either. The last recourse is to edit the STIL data itself and add more information.

Adding More Fingerprints

If you cannot identify the location of a failure, you might need to edit the STIL data and add additional information. The most helpful construct to add is the Label statements to a Vector that did not have distinct labels (see following example). Because the previous label is always printed in the miscompare message, adding labels directly can eliminate ambiguity in identifying that failing location.

```
load_unload {  
  
W _default_WFT_  
  
...  
  
Shift { V { tdi=#; tdo=#; tck=P; } }  
  
V { tdi=#; tdo=#; tck=P; tms=1; }  
  
l_u_post_2: V { tck=P; tms=1; }  
  
l_u_post_3: V { tck=P; tms=0; }  
  
}
```

Labels might be added in STIL data files generated by TestMAX ATPG or might be added to the procedure definitions (if the label is added to a procedure) defined in the STL procedure file data sent to TestMAX ATPG as well, if TestMAX ATPG is used to regenerate the STIL data test.

Debugging Simulation Mismatches Using the `write_simtrace` Command

This section describes the process for using the `write_simtrace` command to assist in debugging ATPG pattern mismatches found during a Verilog simulation. You can use this command in conjunction with simulation mismatch information to create a new Verilog module to monitor additional nodes. A typical flow using TestMAX ATPG and VCS is also provided.

The following topics are covered in this section:

- [Overview](#)
- [Debugging Flow](#)
- [Input Requirements](#)
- [Using the `write_simtrace` Command](#)
- [Understanding the Simtrace File](#)
- [Error Conditions and Messages](#)
- [Example Debug Flow](#)
- [Restrictions and Limitations](#)

Overview

Analyzing simulation-identified mismatches of expected behavior during the pattern validation process is a complex task. There are many reasons for a mismatch, including:

- Response differences due to internal design delays
- Differences due to effects of the “actual” timing specified
- Formatting errors in the stimulus
- Fundamental errors in selecting options during ATPG

Each situation might contribute to the causes for a mismatch. The only evidence of a failure is a statement generated during the simulation run that indicates that the expected state of an output generated by ATPG differs from the state indicated by the simulation. Unfortunately, there is minimal feedback to help you identify the cause of the situation.

To understand the specific cause of the mismatch, you need to compare two sets of simulation data: the ATPG simulation that produced the expected state and the behavior of the Verilog simulation that produced a different state.

After you identify the first difference in behavior, there are still several more steps in the analysis process. You will need to trace back this first difference through the design elements (and often back through time) to identify the cause of the difference. The process of tracing back through time involves re-running the simulation data to produce additional data; as a result, the analysis of this issue is an iterative process,

The key to identifying the discrepancies between the environments is to correlate the information between the Verilog simulation and the TestMAX ATPG simulation. TestMAX ATPG includes a graphical schematic viewer (GSV) with simulation annotation capability. Verilog also has several mechanisms to provide access to internal simulation results that are common to all Verilog simulators.

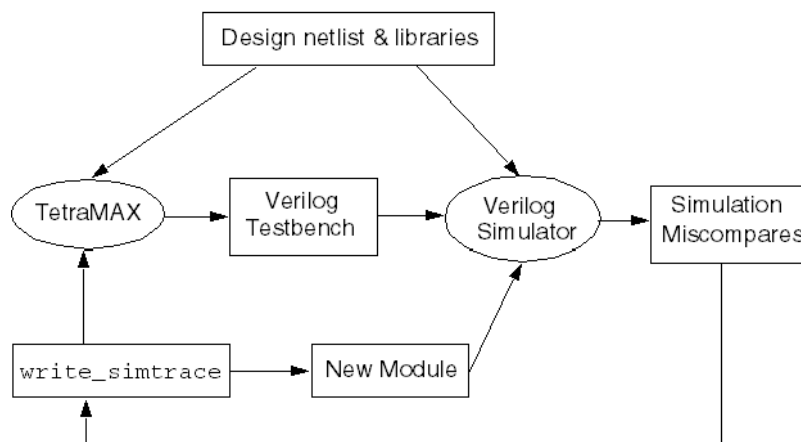
The `write_simtrace` command facilitates the creation of a Verilog module by adding simulation data to be used for comparison with TestMAX ATPG.

Debugging Flow

The following figure shows a typical flow for debugging simulation mismatches using the `write_simtrace` command.

This flow assumes that you are familiar with Verilog simulation. It also assumes that you are using a common netlist for both the Verilog and TestMAX ATPG environments, and that you have executed the `run_simulation` command after ATPG with no failures.

Figure 1 Debugging Simulation Mismatches Using `write_simtrace`



Note in the preceding figure that a Verilog testbench is written out after the TestMAX ATPG process, and is simulated. The simulation log file shows mismatches on scan cells or primary outputs. For each mismatch, you will need to analyze the relevant nodes in the TestMAX ATPG GSV to find their source. The `write_simtrace` command is used to generate a new Verilog module with these additional signals and read it into the simulator. If you monitor the changes on the nodes in the simulation environment at the time the mismatches occur, and correlate that data with the same pattern in TestMAX ATPG, you will eventually see some differences between the two environments that led to the divergent behavior.

The overall process of analyzing simulation mismatches is iterative. You can use the same TestMAX ATPG session for ATPG by analyzing with the GSV and running `write_simtrace`. On the other hand, the simulation would need to be rerun with the new module to monitor the specified signals.

If you do not want to keep the TestMAX ATPG session running (due to license or hardware demands, for example), it is recommended that you write out an image after DRC and save the patterns in binary format. This will ensure that you can quickly re-create the TestMAX ATPG state used for debugging.

Input Requirements

To leverage the functionality of this feature, you need to supply a common or compatible netlist for both TestMAX ATPG and the Verilog simulator.

You also need to provide a MAX Testbench format pattern. Additional testbench environments produced by Synopsys tools are supported but might require additions or modifications depending on the naming constructs used to identify the DUT in the testbench. Usage outside these flows is unsupported.

A TestMAX ATPG scan cell report, as produced by the following command, is useful for providing the instance path names of the scan cells:

```
report_scan_cells -all > chains.rpt
```

To avoid rerunning TestMAX ATPG from scratch, it is recommended that you create an image of the design after running DRC and then save the resulting ATPG patterns in binary format. This ensures that the TestMAX ATPG environment can be quickly recovered for debugging simulation mismatches if the original TestMAX ATPG session cannot be maintained.

Depending on the context and usage of Verilog, you might need to edit the output simtrace file to add a `timescale` statement. In addition, this file can be modified to identify an offset time to start the monitoring process.

You also need to modify the Verilog scripts or invocation environment to include the debug file as one of the Verilog source files to incorporate this functionality in the simulation.

Using the write_simtrace Command

The `write_simtrace` command generates a file in Verilog syntax that defines a standalone module that contains debug statements to invoke a set of Verilog operations. This debugging process references nodes specified by the `-scan` and `-gate` options. Because this is a standalone module, it references these nets as instantiated in the simulation through the testbench module; there are dependencies on these references based on the naming convention of the top module in the testbench module.

After running the `write_simtrace` command, if all nodes specified were found and the file was written as expected, TestMAX ATPG will print the following message:

```
End writing file 'filename' referencing integer nets, File_size = integer
```

This statement identifies how many nets were placed in the output file to be monitored. Note that the file name will not include the path information.

Understanding the Simtrace File

The format of the output simtrace file is shown below:

```
// Generated by TestMAX ATPG(TM)
// from command: < simtrace_command_line >

`define TBNAME AAA_tmax)testbench_1_16.dut
// `define TBNAME module_test.dut
module simtrace_1;
    initial begin
// #<time_to_start> // uncomment and enter a time to start
    $monitor("%t: <scan_data>; <gate_data>", $time(), <list of net
references>);
    end
endmodule // simtrace_1
```

The name of this module is the name of the file without an extension. The module consists of a Verilog initial block that contains an annotation (commented-out) that you can uncomment and edit to identify the time to start this trace operation.

The default trace operation uses the Verilog `$monitor` statement, which is defined in the Verilog standard and supported (with equivalent functionality) across all Verilog clones.

Each `-scan` and `-gate` option identifies a set of monitored nets in the display. Each of these sets is configured as identified below. A semicolon is placed between each different set of nodes in the display to emphasize separate options.

The `<scan_data>` is explained as follows:

If the scan reference contains a chain name and a cell name, for example, "c450:23", then the display will contain this reference name, followed by 3 state bits that represent the state of the scan element before this reference, the state of this reference, and the

state after this reference. The states before and after are enclosed in parentheses. If there is no element before (this is the first element of the chain) or no element after (this is the last element of the chain), then the corresponding state will not be present. Following the 3 states, each non-constant input to this cell is listed as well. This allows tracing of scan enable and scan clock behavior during the simulation. For example, for a cell in the middle of a chain:

```
C450:23 (0)1(1), D:0 SI:1 SE:0 CK:0
```

The `<gate_data>` is formatted similarly to the `<scan_data>` with a port name specified. The name of the signal or net is printed, followed by the resolved state of that net. For example: `Clk1:0`.

If the `gate_reference` is to a module, then the information printed looks very similar to the information for `scan_data`, with one output state of the module, followed by a listing of all non-constant inputs.

Names may be long and might traverse through the design hierarchy. By default, only the last twenty characters of the name are printed in the output statement. The `-length` option can be specified to make these names uniformly longer or shorter.

You need to read the `simtrace` file into a Verilog simulation by adding this filename to the list of Verilog source files on the Verilog command line or during invocation.

Error Conditions and Messages

The output file is not generated if there are errors on the `write_simtrace` command line. All errors are accompanied by error messages of several forms, which are described as follows:

- A standard TestMAX ATPG error message is issued for improper command arguments, missing arguments, or incomplete command lines (no arguments).
- In addition, M650 messages might be generated, with the following forms:

```
Cannot write to simulation debug file <name>. (M650)
```

```
No nodes to monitor in simulation debug file <name>. (M650)
```

These two messages indicate a failure to access a writable file, or that there were no nodes to monitor from the command line. Both of these situations mean that an output file will not be generated.

Example Debug Flow

The following use case is an example of how to use the debug flow.

After running ATPG and writing out patterns in legacy Verilog format, the simulation of the patterns results in the following lines in the Verilog simulation log file:

```

        37945.00 ns; chain7:43 0(0)  INs: CP:1 SI:0;
dd_decrypt.U878.ZN:0,
INs:  A1:1 A2:1 B1:1 B2:1 C:1;
        37955.00 ns; chain7:43 0(0) INs: CP:0 SI:0;
dd_decrypt.U878.ZN:0,
INs:  A1:1 A2:1 B1:1 B2:1 C:1

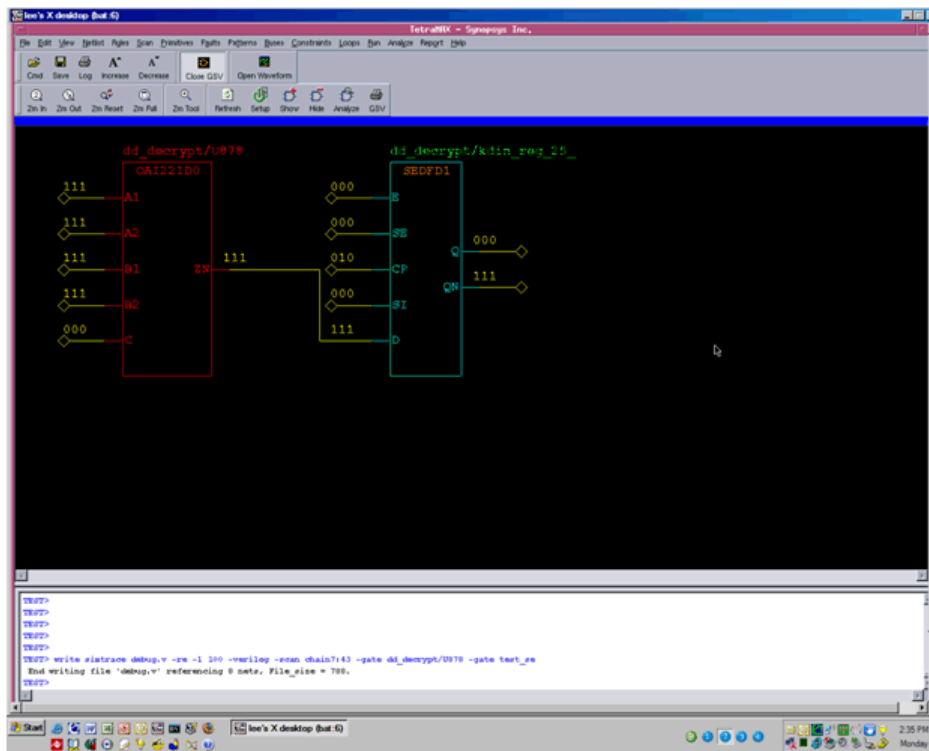
// *** ERROR during scan pattern 4 (detected during final pattern
unload)
    4 chain7 43 (exp=0, got=1) // pin SO_7, scan cell 43, T=
38040.00 ns
//      40000.00 ns : Simulation of 5 patterns completed with 1
errors
    
```

From the TestMAX ATPG scan cells report:

```
chain7 43 MASTER NN 10199 dd_decrypt/kdin_reg_25_ (SEDFD1)
```

The miscompared gates and patterns are displayed in the TestMAX ATPG GSV, as shown in the following figure.

Figure 166 Display of Miscompared Gates and Patterns



To create the debug module in TestMAX ATPG, specify the following `write_simtrace` command:

```
TEST-T> write_simtrace debug.v -re -l 100 -scan chain 7:43 \  
-gate { dd_decrypt/U878 test_se }  
End writing file 'debug.v' referencing 8 nets, File_size = 788.
```

After rerunning the simulation with the `debug.v` module, the following information is now included in the Verilog simulation log file:

```
37945.00 ns; chain7:43 1(0) INs: CP:1 SI:0; dd_decrypt.U878.ZN:1,  
INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;  
  
37955.00 ns; chain7:43 1(0) INs: CP:0 SI:0;  
dd_decrypt.U878.ZN:1,  
INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;  
  
// *** ERROR during scan pattern 4 (detected during final pattern  
unload)  
  
4 chain7 43 (exp=0, got=1) // pin SO_7, scan cell 43, T=  
38040.00 ns
```

To correlate the information that appears in the TestMAX ATPG GSV for pattern 4, look at the values in the simulation log file at the time of the capture operation. To do this, search backward from the time of the miscompare to identify when the scan enable port was enabled:

```
33255.00 ns; chain7:43 0(0) INs: CP:0 SI:0;  
dd_decrypt.U878.ZN:0,  
INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:1;  
  
33300.00 ns; chain7:43 0(0) INs: CP:0 SI:0;  
dd_decrypt.U878.ZN:0,  
INs: A1:1 A2:1 B1:1 B2:1 C:0 ; test_se:0;  
  
33545.00 ns; chain7:43 1(0) INs: CP:1 SI:0;  
dd_decrypt.U878.ZN:1,  
INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:0;  
  
33555.00 ns; chain7:43 1(0) INs: CP:0 SI:0;  
dd_decrypt.U878.ZN:1,  
INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:0;  
  
33600.00 ns; chain7:43 1(0) INs: CP:0 SI:0;  
dd_decrypt.U878.ZN:1,  
INs: A1:1 A2:1 B1:1 B2:1 C:1 ; test_se:1;
```

This example shows that the D input of the scan cell will capture the output of `dd_decrypt.U878`. Notice that there is a difference between the TestMAX ATPG value and the simulator value for `dd_decrypt.U878.C`. If you can identify the cause of this discrepancy, you will eventually find the root cause of the miscompare. By tracing the logic

cone of `dd_decrypt.U878.C` in the TestMAX ATPG GSV to primary inputs or sequential elements, the additional objects to be monitored in simulation can be easily extracted and their values compared against TestMAX ATPG.

Restrictions and Limitations

Note the following usage restrictions and limitations:

- Encrypted netlists for TestMAX ATPG or the Verilog simulator are not supported because the names provided by this flow will not match in both tools.
- Non-Verilog simulators are not supported.

Debugging Parallel Simulation Failures Using Combined Pattern Validation

This section describes how to debug parallel simulation failures using the combined pattern validation (CPV) flow. You can use this flow to precisely debug patterns by reporting the exact failing scan cell for scan compression architectures.

This debug capability is an enhancement to the existing unified STIL flow (USF) and includes interoperability between TestMAX ATPG, MAX Testbench, and VCS.

The following sections describe how to debug parallel simulation failures:

- [Overview](#)
- [Understanding the PSD File](#)
- [Creating a PSD File](#)
- [Flow Configuration Options](#)
- [Displaying the Instance Names of Failing Cells](#)
- [Debug Modes for Simulation Miscompare Messages](#)
- [Pattern Splitting](#)
- [MAX Testbench and Consistency Checking](#)
- [Using the PSD File with DFTMAX Ultra Compression](#)
- [Limitations for Debugging Simulation Failures Using CPV](#)

See Also

- [Writing STIL Patterns](#)

Overview

The combined pattern validation (CPV) parallel simulation failure debug flow is similar to the unified STIL flow (USF). However, the CPV parallel simulation flow enables you to quickly debug failures without using TestMAX ATPG to identify the chains and cell instance names with issues. It also provides support for debugging parallel simulation failures and the flexibility to use your own debug tools.

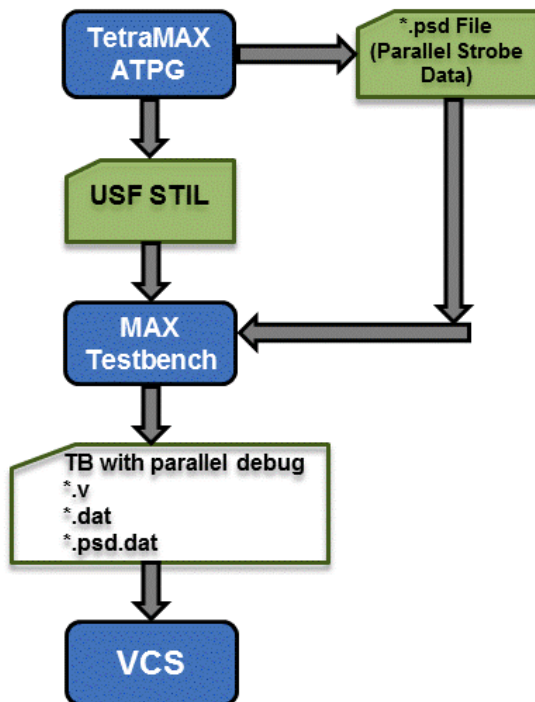
The USF has limited support for debugging parallel simulation failures. The USF debug report lists the pattern number, scan output pin, and the shift index for each failure, but it does not include the particular scan cell that failed.

For diagnosing manufacturing defects, the information provided by the USF is sufficient, since you usually only need to pinpoint the exact fault site (the location of the faulty gate or pin). However, for parallel simulation pattern debugging, you usually need to identify the exact failing scan cell and instance name.

For more information on USF, see [Writing STIL Patterns](#).

The following figure shows the basic CPV parallel simulation failure debug flow.

Figure 167 CPV Parallel Simulation Failure Debug Flow



As shown in the preceding figure, TestMAX ATPG saves the parallel test data to the parallel strobe data (PSD) file in the working directory. You then write the STIL pattern files, and MAX Testbench uses the USF file and the PSD file to generate a testbench and test data file.

MAX Testbench also generates a key test data file that holds only the parallel strobe data used during the simulation miscompare activity of the simulator. This additional MAX Testbench output file, the PSD file file (*.dat.psd), is used during the load_unload procedure as golden (expected) data, which provides comparison data at the scan chain level and failure information at the scan cell resolution level.

See Also

- [Using MAX Testbench](#)
- [Setting the Run Mode](#)

Understanding the PSD File

The PSD file is a binary format file that contains additional parallel strobe data required for debugging parallel simulation failures. You can create a separate PSD file for each pattern unload. Without compression, this file can be four to ten times larger than the original DSF parallel STIL file. You can compress the PSD file as needed using the gzip utility.

The data in the PSD file corresponds to the expected strobe (unload scan chain) data. It is coded using two bits to model states 0, 1 and X, as shown in the following example:

```
Pattern 1 (fast_sequential)
Time 0: load c1 = 0111
Time 1: force_all_pis = 0000000000 00000zzzz
Time 2: pulse_clocks ck2 (1)
Time 3: force_all_pis = 0000100100 00000zzzz
Time 4: measure_all_pos = 00zzzz
Time 5: pulse_clocks ck1 (0)
Time 6: unload c1 = 0000
```

The History section of the USF file contains attributes that link the PSD file and USF pattern file. This information uses STIL annotation, as shown in the following example:

```
Ann {* PSDF = last_100 *}
Ann {* PSDS = 1328742765 *}
Ann {* PSDA = #0#0/0 *}
```

Note the following:

- PSDF — Identifies the PSD file name and location.
- PSDS — Identifies the unique signature (composed of a date and specific ID number) of the PSD file corresponding to the USF file.
- PSDA — Identifies the number of partitions when more than one PSD file is used.

TestMAX ATPG does not use the STIL `Include` statement to establish the USF to PSD file link. This means the additional parallel strobe data does not need to use the STIL syntax, which could overload the USF file with large amounts of test information.

The following figure shows examples of the attributes in the USF file and the corresponding hex data in the PSD file.

Figure 168 USF File and PSD File Example

USF File with PSD Fields in Bold	PSD File (Hex format)
<pre> STIL 1.0 { Design 2005; } Header { ... History { Ann {*Wed May 5 03:00:07 2010 *} Ann {*PSDF = ./my_psd.bin *} Ann {*PSDS = <date><ses_id>*} ... Signals {} SignalGroups {} ... Pattern "_pattern_" { </pre>	<pre> // PSD File generated by TetraMAX V. XXXX // <signature> // <total_pat_nb> <pat_id> // PSD for pattern 0 <nb_loads> // PSD for load 1 <chain_1_id> <parallel_strobe_data> <chain_2_id > <parallel_strobe_data> ... <chain_N_id > <parallel_strobe_data> // PSD for load 2 <chain_1_id> <parallel_strobe_data> ... // PSD for load L <pat_id> // PSD for pattern 1 <nb_loads> // PSD for load 1 </pre>

Creating a PSD File

There are two ways to create a PSD file:

- Using the ATPG flow

Specify the `-parallel_strobe_data_file` option of the `set_atpg` command and the `run_atpg` command. This process is described in [Using the run_atpg Command to Create a PSD File](#).

Note:

This flow is not supported with threaded ATPG, use the Run Simulation flow instead.

- Using the Run Simulation flow

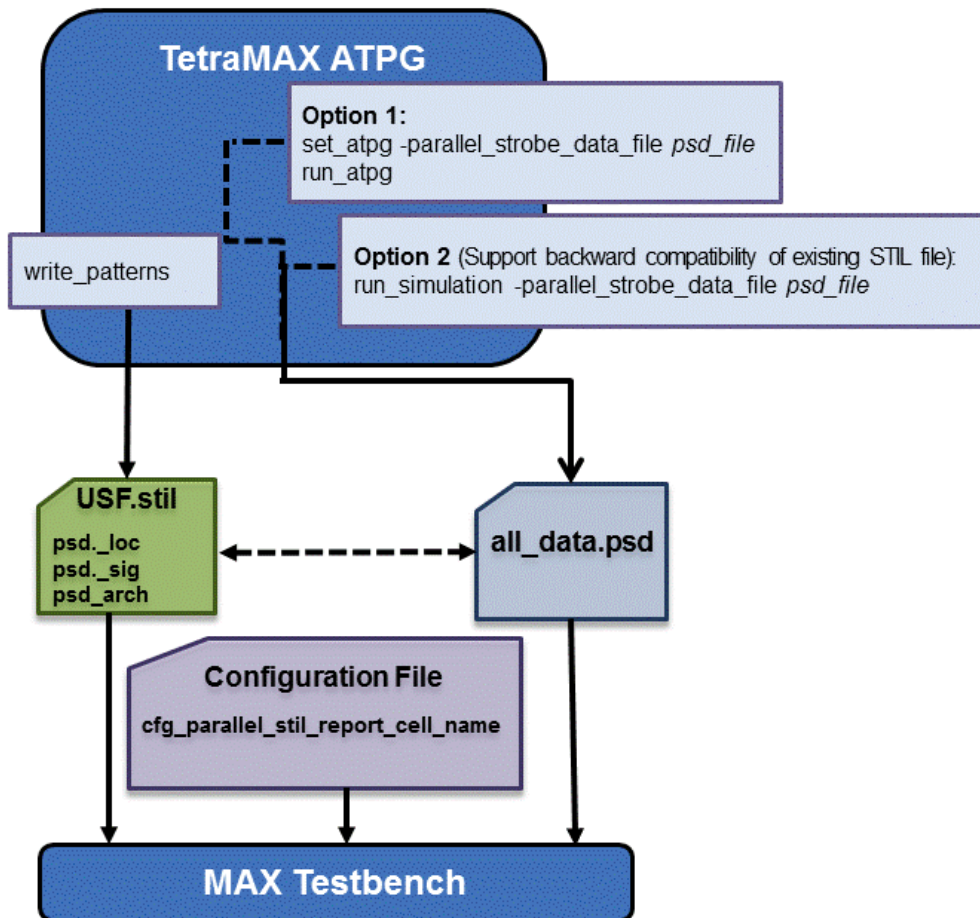
Specify the `-parallel_strobe_data_file` option of the `run_simulation` command to create a PSD file and support the backward compatibility of an existing STIL file. This process is described in [Using the run_simulation Command to Create a PSD File](#).

Note:

This is only supported flow for threaded ATPG.

The following figure shows these options in a flow.

Figure 169 Options for Creating a PSD File



Using the run_atpg Command to Create a PSD File

Note:

The following is not supported with threaded ATPG.

To generate a PSD file during the ATPG flow, you need to specify the

`-parallel_strobe_data_file` option of the `set_atpg` command and the `run_atpg` command.

You can also specify the `report_settings atpg` command to print the settings in the PSD file.

The following example shows how to generate a PSD file using the `run_atpg` command:

```
TEST-T> set_atpg -parallel_strobe_data_file psd_file \  
         -replace_parallel_strobe_data_file  
TEST-T> report_settings atpg  
atpg = parallel_strobe_data_file=psd_file,  
timing_exceptions_au_analysis=no, num_processes=0;  
TEST-T> run_atpg  
TEST-T> write_patterns out.stil -format stil  
TEST-T> write_testbench -input usf.1040.stil \  
         -output usf.1040 -replace -parameter \  
         { -first 10 -last 40 -config config.file -verbose \  
         -log mxtb.log}  
Executing 'stil2Verilog'...  
maxtb> Starting from test pattern 10  
maxtb> Last test pattern to process 40  
maxtb> Total test patterns to process 31  
maxtb> Detected a Scan Compression mode.  
maxtb> Generating Verilog testbench for both serial and parallel load  
mode...
```

Note the following:

- When you invoke MAX Testbench, the PSD file specified in the `set_atpg` command is automatically used. If you do not want to include the PSD file, specify the following option during simulation compilation:

```
tmax_usf_debug_strobe_mode=0
```

- The `write_testbench` command in the previous example references a configuration file called `my_config`. This file contains the following command:

```
set cfg_parallel_stil_report_cell_name 1
```

This command is described in detail in [Displaying Instance Names](#).

Using the `run_simulation` Command to Create a PSD File

You use the `-parallel_strobe_file` option of the `run_simulation` command to create a PSD file that supports the backward compatibility of an existing STIL file.

The following example shows how to use the `run_simulation` command to create a PSD file:

```
set_atpg -noparallel_strobe_data_file  
  
set_patterns -external usf.stil -delete  
Warning: Internal pattern set is now deleted. (M133)  
End parsing STIL file usf.stil with 0 errors.  
End reading 22 patterns, CPU_time = 23.00 sec, Memory = 0MB  
  
report_patterns -summary
```

```

                Pattern Summary Report
-----
#internal patterns                                0
#external patterns (usf.stil)                    22
#fast_sequential patterns                        22
-----

run_simulation -parallel_strobe_data_file \
  test_tr_resim.psd -replace
Created parallel strobe data file 'test_tr_resim.psd'
Begin good simulation of 22 external patterns.
Simulation completed: #patterns=22, #fail_pats=0(0), #failing_meas=0(0),
CPU time=11.00
Total parallel strobe data patterns: 22, external patterns: 22

write_patterns usf_resim.stil -format stil -replace -external
Warning: STIL patterns defaulted to parallel simulation mode. (M474)
Patterns written reference 158 V statements, generating 802 test cycles
End writing file 'usf_resim.stil' with 22 patterns, File_size = 1531782,
CPU_time = 23.0 sec.

report_patterns -summary
                Pattern Summary Report
-----
#internal patterns                                0
#external patterns (usf.stil)                    22
#fast_sequential patterns                        22
-----

write_testbench -input usf_resim.stil -output usf_resim \
  -replace -parameter { -log mxtb_resim.log -verbose \
  -config my_config }
```

Note the following:

- For TestMAX ATPG-generated ATPG patterns, you should use the `run_simulation` command without any additional options. In this case, TestMAX ATPG automatically uses the appropriate simulation algorithm based on the type of pattern input. TestMAX ATPG recognizes patterns produced using Basic Scan or Fast-Sequential mode, but Full-Sequential mode patterns are not supported in this flow.
- You can also improve the performance of non-threaded ATPG by using the `-num_processes` option of the `set_simulation` command. This option specifies the use of multiple CPU cores. For example, the `set_simulation -num_processes 4` command specifies the use of 4 cores. With threaded ATPG `run_simulation` will use number of threads specified by `set_simulation -num_threads`
- The `write_testbench` command in the previous example references a configuration file called `my_config`. This file contains the following command:

```
set cfg_parallel_stil_report_cell_name 1
```

This command is described in detail in the next section, [Displaying Instance Names](#).

- When you invoke MAX Testbench, the PSD file specified in the `run_simulation` command is automatically used. If you don't want to include the PSD file, specify the following option during simulation compilation:

```
tmax_usf_debug_strobe_mode=0
```

Flow Configuration Options

In the example flow shown in [Creating a PSD File](#), MAX Testbench uses as input a PSD file created from TestMAX ATPG and a configuration file that specifies the reporting of instance names. Depending on your debugging needs and simulation resources, you can use different combinations of this input to MAX Testbench.

For example, if you do not want to reference the instance names in the simulation mismatch messages, you can exclude this information from the configuration file as described in [Displaying Instance Names](#). Or, if you do not want to reference the strobe data provided in the PSD file (see [Understanding the PSD File](#)), you can exclude this file.

The following table shows a summary of MAX Testbench mismatch debug support.

Figure 170 MAX Testbench Simulation Mismatch Support

Mismatch Debug Instance name	MAX Testbench			
	Dual STIL Flow		Unified STIL Flow	
	Serial	Parallel	Serial	Parallel
Legacy	NA	With CFG File	NA	With CFG File
Compression & serializer	NA	With CFG File	NA	With CFG File & PSD File

The following section shows examples of these reporting options.

Example Simulation Mismatch Messages

You can use different configuration combinations of input to report various simulation mismatch messages.

Chapter 32: Using MAX Testbench Debugging Parallel Simulation Failures Using Combined Pattern Validation

The following sections show examples of the various miscompare messages:

- **Example 1** shows messages that appear when neither a PSD file or a configuration file is used as input to MAX Testbench.
- **Example 2** shows messages that appear when a PSD file is used, but not a configuration file.
- **Example 3** shows messages that appear when you use both a PSD file and a configuration file as input.
- *Verbosity Setting Examples* shows messages with the trace reporting verbosity level set to 0 (the default) using the `+tmax_msg` runtime option.

Example 1

Example 1 Messages That Appear With No PSD File and No Configuration File

```

Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1_Full164; Runtime version G-2012.09-SP1_Full164; Apr 24 22:29 2013
*****
MAX_TB Version H-2013.03-SP1
Test Protocol File generated from original file "../patterns/pats_nopsd_H-2013.03-SP1.stil"
STIL file version: 1.0
Enhanced Runtime Version: use <sis_exec> +tmax_help for available runtime options
*****

XTB: Starting parallel simulation of 43 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 0
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_so3, scan cell 0
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 1
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 0
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_so4, scan cell 2

XTB: Begin parallel scan load for pattern 10 (T=5100.00 ns, V=52)
XTB: Begin parallel scan load for pattern 15 (T=7600.00 ns, V=77)

```

Example 2

Example 2 Messages That Appear With a PSD File and No Configuration File

Chapter 32: Using MAX Testbench Debugging Parallel Simulation Failures Using Combined Pattern Validation

```

Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1_Full164; Runtime version G-2012.09-SP1_Full164; Apr 24 21:31 2013
*****
MAX TB Version H-2013.03-SP1
Test Protocol File generated from original file ".../patterns/pats_H-2013.03-SP1.stil"
STIL File version: 1.0
Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime options
*****

XTB: Enabling Enhanced Debug Mode. Using mode 0 (conditional parallel strobe).
XTB: Starting parallel simulation of 42 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 0
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s03, scan cell 0
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s03, scan cell 1
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s03, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s04, scan cell 0
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s04, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s04, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 1
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 3
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 2, scan cell 2
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 3
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 3, scan cell 1
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 3, scan cell 3
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 3, scan cell 4
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 3
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 7, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 0
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 9, scan cell 2
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 10, scan cell 0
XTB: Begin parallel scan load for pattern 10 (T=5100.00 ns, V=52)
XTB: Begin parallel scan load for pattern 15 (T=7600.00 ns, V=77)

```

Example 3

Example 3 Messages That Appear With a PSD File and a Configuration File

```

Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1_Full164; Runtime version G-2012.09-SP1_Full164; Apr 24 21:31 2013
*****
MAX TB Version H-2013.03-SP1
Test Protocol File generated from original file ".../patterns/pats_H-2013.03-SP1.stil"
STIL File version: 1.0
Enhanced Runtime Version: use <sim_exec> +tmax_help for available runtime options
*****

XTB: Enabling Enhanced Debug Mode. Using mode 0 (conditional parallel strobe).
XTB: Starting parallel simulation of 43 patterns
XTB: Using 0 serial shifts
XTB: Begin parallel scan load for pattern 0 (T=200.00 ns, V=3)
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 0
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s01, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s03, scan cell 0
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s03, scan cell 1
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s03, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s04, scan cell 0
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_s04, scan cell 1
>>> At T=2740.00 ns, V=28, exp=0, got=1, pin test_s04, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 1, cell name mic0_alu0_accu_q_reg[3]
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 1, scan cell 3, cell name mic0_alu0_accu_q_reg[1]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 2, scan cell 2, cell name mic0_alu0_accu_q_reg[7]
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 3, cell name mic0_alu0_accu_q_reg[6]
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 3, scan cell 1, cell name mic0_ctr10_s_inpc_q_reg
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 3, scan cell 3, cell name mic0_ctr10_s_dr_cen_q_reg
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 3, scan cell 4, cell name mic0_ctr10_s_alu_cntrl_q_re
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2, cell name mic0_ctr10_s_state_reg[1]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 3, cell name mic0_ctr10_s_state_reg[0]
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 7, scan cell 1, cell name mic0_ir0_ir_q_reg[11]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 0, cell name mic0_pc0_prog_counter_q_reg
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 9, scan cell 2, cell name mic0_pc0_prog_counter_q_reg
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 10, scan cell 0, cell name mic0_pc0_prog_counter_q_re
XTB: Begin parallel scan load for pattern 10 (T=5100.00 ns, V=52)
XTB: Begin parallel scan load for pattern 15 (T=7600.00 ns, V=77)

```

Verbosity Setting Examples

You can further control the reporting of simulation miscompare messages by specifying the `+tmax_msg` runtime option, or by setting the `cfg_message_verbosity_level` command in the MAX Testbench configuration file. For details on the `+tmax_msg` option, see "Setting the Verbose Level."

The following examples show how the messages appear when you set the verbosity level to 0 (the default) using the `+tmax_msg` runtime option.

Example 4 Using a PSD File and No Configuration File With Verbosity Level 0

```
#####
MAX TB
Test Protocol File generated from original file "pats.usf.stil"
STIL file version: 1.0
NO CONFIGURATION FILE
#####
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 2
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 2
```

Example 5 Using a PSD File and Configuration File with Verbosity Level 0

```
#####
MAX TB
Test Protocol File generated from original file "pats.usf.stil"
STIL file version: 1.0
USING THE CONFIGURATION FILE
#####
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
XTB: searching corresponding parallel strobe failures...
>>> At T=2740.00 ns, V=28, exp=1, got=0, chain 2, scan cell 2, cell
name mic0.alu0.accu_q_reg[7]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 4, scan cell 2, cell
name mic0.ctrl10.s_state_reg[1]
>>> At T=2740.00 ns, V=28, exp=0, got=1, chain 9, scan cell 2, cell
name mic0.pc0.prog_counter_q_reg[5]
```

Example 6 Using a Configuration File and No PSD file with Verbosity Level 0:

```
XTB: Begin parallel scan load for pattern 5 (T=2600.00 ns, V=27)
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so1, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so3, scan cell 2
>>> Error during scan pattern 5 (detected from parallel unload of
pattern 4)
>>> At T=2740.00 ns, V=28, exp=1, got=0, pin test_so4, scan cell 2
>>> Error during scan pattern 7 (detected from parallel unload of
pattern 6)
```

Displaying the Instance Names of Failing Cells

You can set MAX Testbench to report the instance names of failing cells during the simulation of a parallel-formatted STIL file. To enable this reporting, set the

`cfg_parallel_stil_report_cell_name` command to 1 in the configuration file (0 is the default), as shown in the following example:

```
set cfg_parallel_stil_report_cell_name 1
```

Important: Memory consumption is affected when you enable the reporting of instance cells. The best way to turn off parallel simulation failure debug mode without recompiling the simulation executable is to set the following runtime option:

```
./simv +tmax_usf_debug_strobe_mode=0
```

The following example shows how MAX Testbench reports instances of failing cells:

```
Error during scan pattern 28 (detected during parallel unload
of pattern 27)
At T=33940.00 ns, V=340, exp=0, got=1, chain 35, scan cell
1, cell name U_CORE.dd_d.o_tval_reg
At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell
7, cell name U_CORE.dd_d.o_data_reg_3_
At T=33940.00 ns, V=340, exp=1, got=0, chain 35, scan cell
9, cell name U_CORE.dd_d.o_data_reg_1_
```

The following table shows the configurations supported by MAX Testbench for debugging parallel mismatches with instance names. The configuration file is used when the

`cfg_parallel_stil_report_cell_name 1` option is specified. The PSD file is the parallel strobe data file used for parallel simulations.

Figure 171 Support for Debugging Parallel Mismatch With Instance Names

TetraMAX, DFTMAX, and DFTMAX Ultra	MAX Testbench	
Command: <code>write_patterns -parallel</code>	Dual STIL Flow <code>-nounified_stil_flow</code>	Unified STIL Flow: <code>-unified_stil_flow</code>
Legacy	Supported with configuration file	Supported with configuration file
DFTMAX Ultra compression	Supported with configuration file	Supported with configuration file and PSD file
Serializer compression	Supported with configuration file	Supported with configuration file and PSD file

See Also

- [Configuring MAX Testbench](#)
- [Debugging Parallel Simulation Failures for Combined Pattern Validation \(CPV\)](#)

Debug Modes for Simulation Mismatch Messages

You can specify modes for reporting various levels of details of simulation runtime mismatch messages for scan compression technology. To do this, use the `+tmax_usf_debug_strobe_mode` predefined simulation command option. The syntax for this option is as follows:

```
+tmax_usf_debug_strobe_mode=<0, 1, 2, 3>
```

Each mode is described as follows:

0 - Disables parallel simulation failure debug and generates normal error messages related only to the scan output. This mode is useful for increasing simulation performance when you only want to quickly determine the pass/fail status of very large designs.

1 - Specifies the default mode, referred to as the "Conditional parallel strobe mode." This mode generates mismatch simulation messages using parallel strobe data that is applied only to USF failures.

2 - This mode, referred to as the "Unconditional parallel strobe mode," concurrently activates the USF and the CPV parallel strobe data for generating mismatch messages for each pattern.

3 - Generates miscompare messages only for internal errors using parallel strobe data applied to each pattern. The messages generated from this mode do not indicate if a parallel strobe failure is propagated to the primary scan output (after the compressor).

You can also specify this option as a command in the Runtime field of the testbench (*.v) file produced by MAX Testbench. However, the simulation command line always overrides the default specification of the testbench file.

The following table summarizes the errors reported for each mode. An "Error" is actually a reported mismatch message generated during scan-unload processing. "Normal IO Errors" refer to error messages generated during scan that report errors relative to the scan output. "Internal Errors" refer to error messages generated during scan that report the error relative to an internal scan cell.

Table 14 *Debug Modes and Reported Errors*

Mode	Normal IO Errors	Internal Errors
Mode=0	Yes	No
Mode=1	Yes	Yes
Mode=2	Yes	Yes
Mode=3	No	Yes

Note that in serial simulation, the Internal error field is not available. Only the normal I/O errors are recorded, as if you received tester failures at the I/O of the device.

The following examples show how messages for the various modes appear in the log file:

MODE 0 Log File Example

```
jv_comp_parallel_mode0.log:XTB: Enhanced Debug Mode disabled (user
request).
jv_comp_parallel_mode0.log:XTB: Simulation of 7 patterns completed with 6
mismatches (time: 2700.00 ns, cycles: 27)
-----
```

MODE 1 Log File Example

```
jv_comp_parallel_mode1.log:XTB: Enabling Enhanced Debug Mode. Using mode
1 (conditional parallel strobe).
jv_comp_parallel_mode1.log:XTB: Simulation of 7 patterns completed with 6
mismatches (1672 internal mismatches) (time: 2700.00 ns, cycles: 27)
-----
```

MODE 2 Log File Example

```
jv_comp_parallel_mode2.log:XTB: Enabling Enhanced Debug Mode. Using mode
 2 (unconditional parallel strobe).
jv_comp_parallel_mode2.log:XTB: Simulation of 7 patterns completed with 6
 mismatches (10569 internal mismatches) (time: 2700.00 ns, cycles: 27)
-----
```

MODE 3 Log File Example

```
jv_comp_parallel_mode3.log:XTB: Enabling Enhanced Debug Mode. Using mode
 3 (only parallel strobe).
jv_comp_parallel_mode3.log:XTB: Simulation of 7 patterns completed with
 (10569 internal mismatches) (time: 2700.00 ns, cycles: 27)
```

Pattern Splitting

MAX Testbench stores key simulation miscompare activity for the parallel strobe data in a *psd.dat file. This data is used during the load_unload procedure as golden (expected) data. By default, the *psd.dat file contains a maximum of 1000 patterns. When more than 1000 patterns are used, MAX Testbench automatically splits the contents of the PSD file and generates a set of corresponding set of *_psd.dat files.

You can manually specify pattern splitting in TestMAX ATPG or MAX Testbench using any of the following flow options:

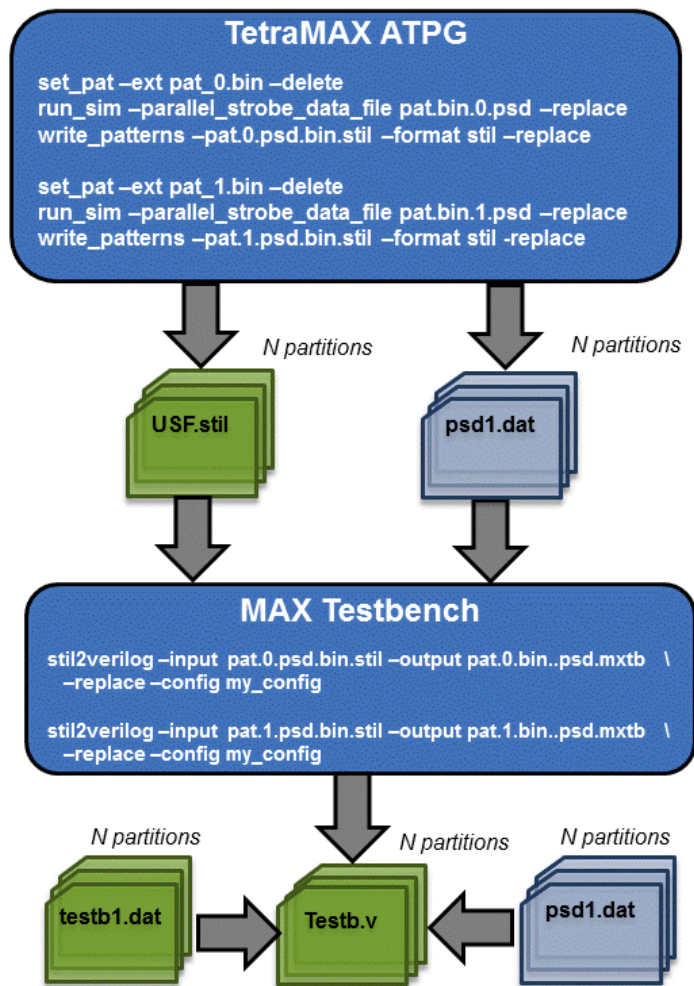
- Split the patterns using the `write_patterns` command in TestMAX ATPG before they are used by MAX Testbench. This process is described in [Splitting Patterns Using TestMAX ATPG](#).
- Use the `-split_out` option of the `write_testbench` or `stil2Verilog` commands to split the patterns in MAX Testbench. This flow is described in [Splitting Patterns Using MAX Testbench](#).
- Use the `run_simulation` command flow and the `-first` and `-last` options of the `write_testbench` or `stil2Verilog` commands to address only the failing VCS pattern sets in MAX Testbench. This flow is described in [Specifying a Range of Split Patterns Using MAX Testbench](#).

Splitting Patterns Using TestMAX ATPG

You can split patterns using the `write_patterns` command in TestMAX ATPG before using the patterns in MAX Testbench. For example, you might want TestMAX ATPG to write out 500 patterns per file. To do this, read each split STIL pattern file into TestMAX ATPG and then specify the `run_simulation -parallel_strobe_data_file` command for each pattern file.

The following figure shows the flow for using the `write_patterns` command to split patterns before using MAX Testbench. For examples of this flow, see [Examples Using TestMAX ATPG For Pattern Splitting](#).

Figure 172 Debugging Flow Using Split Patterns in Binary Format



Examples Using TestMAX ATPG For Pattern Splitting

The following examples show pattern splitting using the write_patterns command in TestMAX ATPG:

- [Set Up Example](#)
- [Example Using Pattern File From write_patterns Command](#)
- [Example Using Split USF STIL Pattern Files](#)

Set Up Example

The following example writes out split binary patterns from the same ATPG run:

```
run_atpg -auto
write_patterns pats.bin -format binary -replace -split 3
```

Example Using Pattern File From write_patterns Command

This example uses split binary pattern files from the `write_patterns` commands in the previous example, then writes out USF STIL patterns:

```
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_0.bin -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.bin.0.psd -replace
write_patterns pat.0.psd.bin.stil -format stil -replace -external
report_patterns -summary

set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_1.bin -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.bin.1.psd -replace
write_patterns pat.1.psd.bin.stil -format stil -replace -external
report_patterns -summary
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats_2.bin -delete
report_patterns -summary

run_sim -parallel_strobe_data_file pat.bin.2.psd -replace
write_patterns pat.2.psd.bin.stil -format stil -replace -external
report_patterns -summary

write_testbench -input pat.0.psd.bin.stil -output \
  pat.0.bin..psd.mxtb -replace -parameters \
  {-log mxtb_bin.0.log -verbose -config my_config}
write_testbench -input pat.1.psd.bin.stil -output pat.1.bin..psd.mxtb \
  -replace -parameters {-log mxtb_bin.1.log -verbose \
  -config my_config}
write_testbench -input pat.2.psd.bin.stil -output \
  pat.2.bin..psd.mxtb -replace -parameters \
  {-log mxtb_bin.2.log -verbose -config my_config}
```

Example Output Files:

```
pat.bin.2.psd
pat.bin.1.psd
pat.bin.0.psd
pat.2.psd.bin.stil
pat.1.psd.bin.stil
pat.0.psd.bin.stil
mxtb_bin.2.log
pat.2.bin..psd.mxtb.v
```


Chapter 32: Using MAX Testbench Debugging Parallel Simulation Failures Using Combined Pattern Validation

```
pat.2.bin..psd.mxtb.dat
pat.2.bin..psd.mxtb_psd.dat
mxtb_bin.1.log
pat.1.bin..psd.mxtb.v
pat.1.bin..psd.mxtb.dat
pat.1.bin..psd.mxtb_psd.dat
mxtb_bin.0.log
pat.0.bin..psd.mxtb.v
pat.0.bin..psd.mxtb.dat
pat.0.bin..psd.mxtb_psd.dat
```

Example Using Split USF STIL Pattern Files

The following example uses split USF STIL pattern files:

```
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_0.stil -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.usf.0.psd -replace
write_patterns pat.usf.0.psd.stil -format stil -replace -external
report_patterns -summary
set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_1.stil -delete
report_patterns -summary

run_sim -parallel_strobe_data_file pat.usf.1.psd -replace
write_patterns pat.usf.1.psd.stil -format stil -replace -external
report_patterns -summary

set_atpg -noparallel_strobe_data_file
set_patterns -ext pats.usf_2.stil -delete
report_patterns -summary
run_sim -parallel_strobe_data_file pat.usf.2.psd -replace
write_patterns pat.usf.2.psd.stil -format stil -replace -external
report_patterns -summary

write_testbench -input pat.usf.0.psd.stil -output \
  pat.usf.0.psd.mxtb -replace -parameters \
  {-log mxtb_usf.0.log -verbose -config my_config}
write_testbench -input pat.usf.1.psd.stil -output \
  pat.usf.1.psd.mxtb -replace -parameters \
  {-log mxtb_usf.1.log -verbose -config my_config}
write_testbench -input pat.usf.2.psd.stil -output \
  pat.usf.2.psd.mxtb -replace -parameters {-log \
  mxtb_usf.2.log -verbose -config my_config}
```

Example Output Files:

```
pat.usf.2.psd
pat.usf.1.psd
pat.usf.0.psd
pat.usf.2.psd.stil
pat.usf.1.psd.stil
```

Chapter 32: Using MAX Testbench Debugging Parallel Simulation Failures Using Combined Pattern Validation

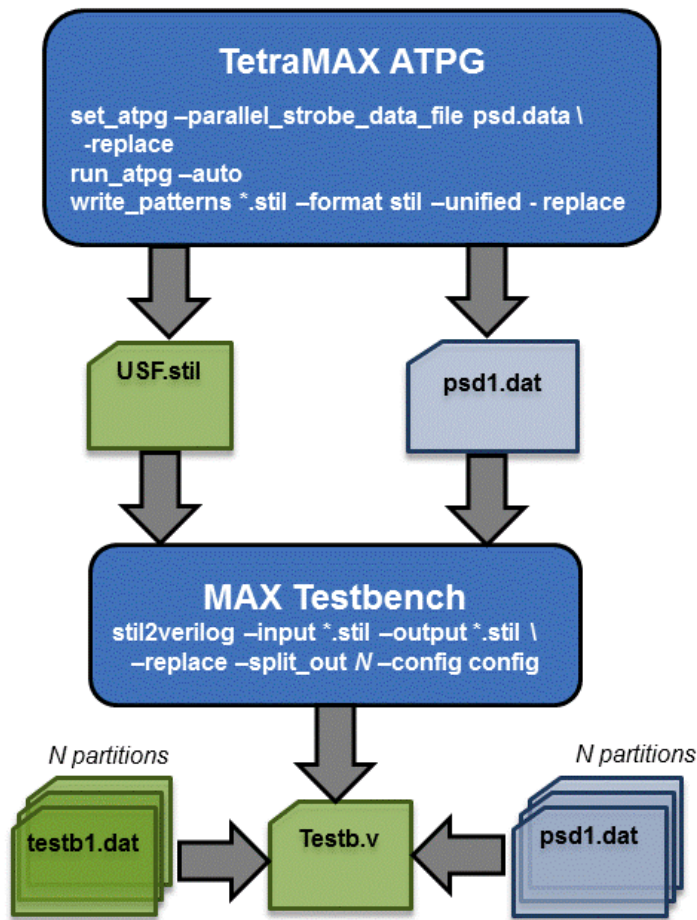
```
pat.usf.0.psd.stil  
mxtb_usf.2.log  
pat.usf.2.psd.mxtb.v  
pat.usf.2.psd.mxtb.dat  
pat.usf.2.psd.mxtb_psd.dat  
mxtb_usf.1.log  
pat.usf.1.psd.mxtb.v  
pat.usf.1.psd.mxtb.dat  
pat.usf.1.psd.mxtb_psd.dat  
mxtb_usf.0.log  
pat.usf.0.psd.mxtb.v  
pat.usf.0.psd.mxtb.dat  
pat.usf.0.psd.mxtb_psd.dat
```

Splitting Patterns Using MAX Testbench

You can manually specify pattern splitting in MAX Testbench using the `-split_out` option of the `write_testbench` or `stil2Verilog` commands.

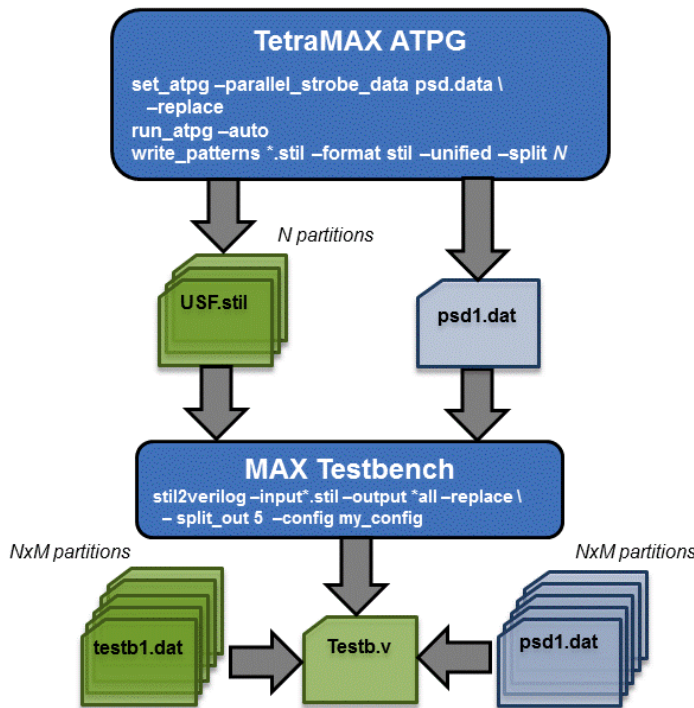
The following figure shows the flow for splitting patterns using MAX Testbench.

Figure 173 Flow for Using MAX Testbench to Split Patterns



You can also split patterns in both TestMAX ATPG and MAX Testbench. This flow is described in Figure 7.

Figure 174 Flow For Using Both TestMAX ATPG and MAX Testbench to Split Patterns



Specifying a Range of Split Patterns Using MAX Testbench

You can split a specified range of split patterns in MAX Testbench so you can better focus your debugging efforts. To do this, use the standard `run_simulation` command flow and read back only the set of binary or STIL patterns that failed in simulation, then produce the PSD file (for details, see [Using the run_simulation Command to Create a PSD File](#)).

MAX Testbench and Consistency Checking

When you run MAX Testbench, it automatically detects and processes the PSD file, and issues the following message:

```
maxtb> Detected STIL file with Enhanced Debug for CPV (EDCPV) Capability
(PSD file: psdata). Processing...
```

MAX Testbench performs a series of consistency checks between the contents of the USF file and PSD file. If any issues are detected, it generates a testbench file without the parallel strobe data, and issues the following warning message:

```
Warning: Disabling the Enhanced Debug Mode for Combined Pattern
Validation (EDCPV) corrupted PSD file due to bad file signature
(1329175245). Make sure the PSD file corresponds to the generated STIL
file (W-041)
```

The following message is specific to the debugging parallel simulation failures using the Combined Pattern Validation (CPV) flow:

```
W-041: Disabling the Enhanced Debug Mode for Unified STIL Flow (EDUSF)
```

This message is issued when the debug mode for parallel simulation failures cannot be activated because of consistency checking failures. As a result, the generated testbench is not able to pinpoint the exact failing scan cell in parallel simulation mode. MAX Testbench continues to generate the testbench files without the parallel strobe data file.

See Also

- [MAX Testbench Error Warnings and Messages](#)

Using the PSD File with DFTMAX Ultra Compression

If you are running DFTMAX Ultra compression, make sure you use the `-parallel_strobe_data_file` option of the `run_simulation` command to create a PSD file. It is important that you do not use the `set_atpg -parallel_strobe_data_file` command to create a PSD file with DFTMAX Ultra compression. In this case, the `update_streaming_patterns -remove_padding_patterns` command invalidates the PSD file.

Also, you should use only binary files for external patterns.

To create the PSD file with DFTMAX Ultra compression in a single session:

1. Run ATPG in TestMAX ATPG.

```
run_atpg -auto
```

For more information on running TestMAX ATPG, see [Running the Basic ATPG Design Flow](#) in the *TestMAX ATPG User Guide*.

2. Use the `update_streaming_patterns` command to clean up the padding patterns.

```
update_streaming_patterns -remove_padding_patterns
```

3. Write the unified STIL patterns.

```
write_patterns out.stil -format binary -replace
```

4. Delete all current internal and external patterns

```
set_patterns -delete
```

5. Specify a set of externally generated patterns.

```
set_patterns -external out.stil
```

6. Create the PSD file.

```
run_simulation -parallel_strobe_data_file comp.psd -replace
```

7. Write the final set of patterns that match the PSD file.

```
write_patterns pat_u.stil -for stil -repl -unified_stil_flow -ext
```

8. Run MAX Testbench.

```
write_testbench -input pat_u.stil -out maxtb -replace -parameters  
{-log mxtb.log -verbose -config my.cfg}
```

Script Example

The following script contains the commands for using the PSD file with DFTMAX Ultra compression.

```
run_atpg -auto  
update_streaming_patterns -remove_padding_patterns  
  
write_patterns out.stil -format binary -replace  
  
set_patterns -del  
set_patterns -ext out.stil  
run_simulation -parallel_strobe_data_file comp.psd -replace  
  
write_patterns pat_u.stil -for stil -repl -unified_stil_flow -ext  
  
write_testbench -input pat_u.stil -out maxtb -replace -parameters \  
  {-log mxtb.log -verbose -config my.cfg}
```

Limitations for Debugging Simulation Failures Using CPV

Note the following limitations related to debugging simulation failures using CPV:

- The Full-Sequential mode is not supported.
- The `set_patterns` and `run_simulation` commands are not supported for multiple contiguous runs (see [Creating a PSD File](#)). Also, update and masking flows are not supported, including pattern restore from binary and new pattern write flows, multiple pattern read back, and single merged pattern write.
- When the `set_atpg -parallel_strobe_data` command is used with DFTMAX Ultra compression, the PSD file is invalidated by a subsequent `update_streaming_patterns -remove_padding_patterns` command. Only the `run_simulation` flow should be used with DFTMAX Ultra compression.
- The `-first`, `-last`, `-sorted`, `-reorder`, and `-type` options of the `write_patterns` command are not supported.

Chapter 32: Using MAX Testbench
Debugging Parallel Simulation Failures Using Combined Pattern Validation

- The `-sorted`, `-reorder`, and `-type` options of the `write_testbench` and `stil2Verilog` commands are not supported.
- The `-last` option of the `run_simulation` command is not supported.

33

Using Loadable Nonscan Cells in TestMAX ATPG

Nonscan cells are nonscan flip-flops or latches that capture known values during the last shift cycle. You can configure TestMAX ATPG to recognize any nonscan cells that load the value of a combinational logic function of multiple scan cells during simulation or ATPG. Logic and fault simulation can simulate these load values for depths up to the length of the longest scan chain. ATPG can control these load values with a sequential depth of 1.

The following sections describe how to use loadable nonscan cells in TestMAX ATPG:

- [Simulation Support](#)
- [ATPG Support](#)
- [Multithreading ATPG](#)
- [Fault Simulation Support](#)
- [Reporting Loadable Nonscan Cells](#)
- [Analyzing](#)
- [Limitations](#)

Simulation Support

When simulating loadable nonscan cells, TestMAX ATPG supports the basic scan, two-clock, and fast-sequential simulation engines.

Simulation support for loadable nonscan cells is set by the following `set_simulation` switch:

```
-shift_cycles number
```

You can set the *number* value to specify the depth of the loading to be simulated. The maximum value of the specified *number* is 10, but also no more than the length of the longest scan chain (plus 1 for compression). The default for *number* is 0, which will not enable simulation of loadable nonscan cells.

To ensure that the `run_simulation` command reports accurate results, set the number for `set_simulation -shift_cycles` to the same value you used to generate the patterns.

Note that the `run_simulation` command will fail in subsequent runs if you set the shift cycles to a lower number than you originally set using the `set_simulation -shift_cycles` switch.

In the graphical schematic viewer (GSV), the simulated shift cycles are displayed in the “Good Sim Results” pindata, and the cycles are enclosed in curly braces; for example: “{111100}011”.

ATPG Support

Both the basic scan and two-clock ATPG engines support loadable nonscan cells (the fast-sequential engine is not supported). To enable ATPG support, first set the `set_drc -load_nonscan_cells` switch, and then set the shift cycles to 1 or higher using `set_simulation -shift_cycles`. (Note that ATPG only supports a depth of 1, although more cycles are simulated if a greater value is specified.)

Multithreading ATPG

When using [multithreading ATPG](#), if `set_simulation shift_cycles` is not set, then `run_atpg` would set it to 1 by default so that pattern generation will simulate the nonscan cells correctly. Subsequent `run_simulation` command uses `shift_cycles` of 1. If patterns are then reloaded into a new session then you must use `set_simulation -shift_cycles 1` before doing `run_sim` to avoid simulation mismatches.

Fault Simulation Support

In order to get correct fault simulation coverage, you must first specify `set_simulation -shift_cycles` to the same value you set for pattern generation, or a higher value. You must also specify the `set_drc -load_nonscan_cells` command before specifying the `run_drc` command.

Reporting Loadable Nonscan Cells

You can use the following options of the `report_nonscan_cells` command to explicitly report loadable nonscan cells:

- `load` — Reports all loadable nonscan cells when the `set_drc -load_nonscan_cells` command is specified.
- `nonx_load` — Reports loadable nonscan cells which always have a non-X value (that is, 0 or 1) during the random pattern-based analysis performed when the `set_drc -load_nonscan_cells` command is specified.

Example 1: Using the load and nonx_load Options

```
TEST-T> report_nonscan_cells load
```

```
type      behavior_data  gate_id  instance_name (type)
```

```
-----
```

```
LE        load_unstable   95  n0and (FD1)
```

```
LE        load_unstable   96  n0or  (FD1)
```

```
LE        load_unstable   97  n0aoi (FD1)
```

```
LE        load_unstable   98  n0xor (FD1)
```

```
LE        load_unstable   99  l2andf (FD1)
```

```
LE        load_unstable  100  l2orf  (FD1)
```

```
TEST-T> report_nonscan_cells nonx_load
```

```
type      behavior_data  gate_id  instance_name (type)
```

```
-----
```

```
LE        load_unstable   95  n0and (FD1)
```

```
LE        load_unstable   96  n0or  (FD1)
```

```
LE        load_unstable   97  n0aoi (FD1)
```

```
LE        load_unstable   98  n0xor (FD1)
```

Note that the report from the `load` option contains all nonscan cells that can be loaded when the `set_drc -load_nonscan_cells` command is specified. The report from the `nonx_load` option is a subset of those cells that are guaranteed to always have known values at the end the load operation.

Alternatively, you can use the `run_simulation` and `run_atpg` commands to print comments that refer to the number of nonscan cells that were loaded, as shown in the following example:

```
Shift simulation completed: #shifts=3, #nonscancells_loaded=48
```

The statement "`#shifts=3`" is the value specified by the `set_simulation -shift_cycles` command. The statement "`#nonscancells_loaded=48`" is the total number of cells that were loaded in all patterns (that is, the number of cells multiplied by the number of patterns).

After you run the `run_drc` command, a new section is printed in the `run_drc` log, as shown in the following example:

```
Nonscan rules checking completed, CPU time=0.00 sec.
-----
-----
Shift_pattern simulation setup completed: #shift_cycles=1,
#shift_patterns=5
Shift simulation completed: #shifts=1, #nonscancells_loaded=128
Shift simulation completed: #shifts=1, #nonscancells_loaded=128
Shift simulation completed: #shifts=1, #nonscancells_loaded=128
Nonscan load cell identification completed: #load_nonscan_cells=4
Begin DRC dependent learning...
```

The previous identifies four loadable nonscan cells.

Analyzing

After setting the shift cycles to the desired value using the `set_simulation -shift_cycles` command, you can analyze the loading of nonscan cells using the `analyze_nonscan_loading` command. The syntax for this command is as follows :

```
analyze_nonscan_loading
[-atleast <percentage <0 | 1 | B>>]
[-atmost <percentage <0 | 1 | B>>]
[-max number]
[-patterns number]
```

(See the `analyze_nonscan_loading` command online help topic for complete details on the syntax items.)

The `analyze_nonscan_loading` command simulates the specified number of random patterns and reports the number of nonscan cells that are loadable. By default, 1024 patterns are simulated and all nonscan cells are reported in a histogram of the percentage of patterns in which a non-X value was loaded.

Note that no instances are reported by default. To report instances, use either the `-atleast` or `-atmost` switches, taking into account the minimum or maximum percentage of the time that was loaded with the specified value (with `B` meaning either 0 or 1). To report the most possible instances, specify `"-atleast {1 B}"`. For example:

```
TEST-T> analyze_nonscan_loading -patterns 100 -atleast {10 0}
Begin nonscan loading analysis: #patterns=100, #nonscan_cells=7
Shift_pattern simulation setup completed: #shift_cycles=1,
#shift_patterns=3
Shift simulation completed: #shifts=1, #nonscancells_loaded=160
Shift simulation completed: #shifts=1, #nonscancells_loaded=160
Shift simulation completed: #shifts=1, #nonscancells_loaded=160
Shift simulation completed: #shifts=1, #nonscancells_loaded=20
```

```
Nonscan_loading analysis summary: #patterns=100, #nonscan_cells=7,
CPU_time=0.00 sec
```

```
-----
Range   nonscan cell load count
-----
0       1
1-9     1
10-19   0
20-29   0
30-39   0
40-49   0
50-59   0
60-69   0
70-79   0
80-89   0
90-99   1
100     4
-----
```

```
-----
Nonscan_loading report: threshold=10, type=atleast, value=0, max=0
-----
```

```
-----
gate_id   type   count   instance name
-----
          91   DFF      93   n0and
          95   DFF      48   l2andf
          94   DFF      47   n0xor
          93   DFF      30   n0aoi
-----
```

In the previous example, only four instances are reported even though five instances were loaded with non-X values in more than 10% of the simulations. This indicates that the listed instance that was not loaded failed to meet the `-atleast` condition that a 0 should be loaded in 10% of the simulations.

Limitations

Full-sequential patterns for loadable nonscan cells are not supported for ATPG or simulation.

It's important to note that the loadable nonscan cells feature is limited in its effectiveness. It will not be of much benefit if the load_unstable cells form a deep sequential circuit, have sequential loops, or are fed by X-generators. Since this feature has runtime penalties in DRC, ATPG, and simulation, you should make sure you are getting a net benefit when using it. You can check its effectiveness by comparing the number of S19

Chapter 33: Using Loadable Nonscan Cells in TestMAX ATPG Limitations

violations to the following line of `run_drc` output (which is only printed when the `set_drc -load_nonscan_cells` command is specified):

```
Nonscan load cell identification completed: #load_nonscan_cells=6  
(nonx=4)
```

Note that the loadable nonscan cells feature is beneficial if the `#load_nonscan_cells` value is large and/or this value comprises a large percentage of S19s violations.

34

PowerFault

The following sections describes PowerFault and its simulation:

- [PowerFault Simulation](#)
- [Verilog Simulation with PowerFault](#)
- [Faults and Fault Seeding](#)
- [PowerFault Strobe Selection](#)
- [Using PowerFault Technology](#)

PowerFault Simulation

PowerFault simulation technology enables you to validate the IDDQ test patterns generated by TestMAX ATPG.

The following sections describe PowerFault simulation:

- [PowerFault Simulation Technology](#)
- [IDDQ Testing Flows](#)
- [Licensing](#)

PowerFault-IDDQ might not work on unsupported operating platforms or simulators. See the *TestMAX ATPG Release Notes* for a list of supported platforms and simulators.

PowerFault Simulation Technology

PowerFault simulation technology verifies quiescence at strobe points, analyzes and debugs nonquiescent states, selects the best IDDQ strobe points for maximum fault coverage, and generates IDDQ fault coverage reports.

Instead of using the IDDQ fault model, you can use the standard stuck-at-0/stuck-at-1 fault model to generate ordinary stuck-at ATPG patterns, and then allow PowerFault to select the best patterns from the resulting test set for IDDQ measurement. The PowerFault simulation chooses the strobe times that provide the highest fault coverage.

PowerFault technology uses the same Verilog simulator, netlist, libraries, and testbench used for product sign-off, helping to ensure accurate results. The netlist and testbench do not need to be modified in any way, and no additional libraries need to be generated.

You run PowerFault after generating IDDQ test patterns with TestMAX ATPG, as described in [Quiescence Test Pattern Generation](#) in the *TestMAX ATPG User Guide*.

You perform IDDQ fault detection and strobe selection in two stages:

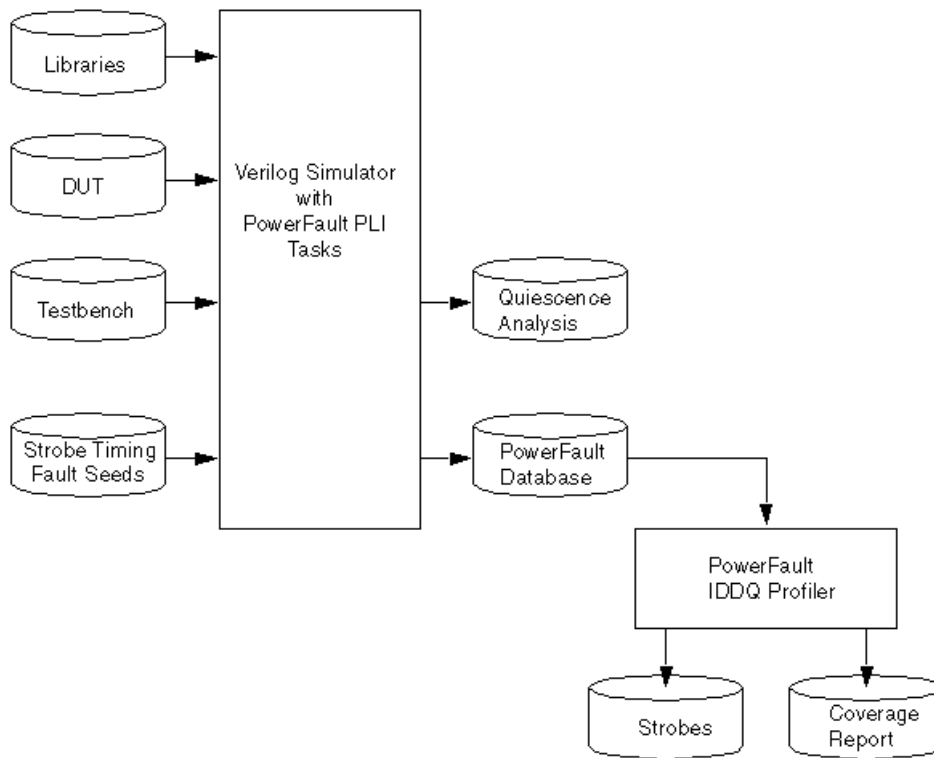
1. Run a Verilog simulation with the normal testbench, using the PowerFault tasks to specify the IDDQ configuration and fault selection, and to evaluate the potential IDDQ strobcs for quiescence.

The inputs to the simulation are the model libraries, the description of the device under test (DUT), the testbench, and the IDDQ parameters (fault locations and strobe timing information).

The simulator produces a quiescence analysis report, which you can use to debug any leaky nodes found in the design, and an IDDQ database, which contains information on the potential strobe times and corresponding faults that can be detected.

2. Run the IDDQ Profiler, *IDDQPro*, to analyze the IDDQ database produced by the PowerFault tasks. The IDDQ Profiler selects the best IDDQ strobe times and generates a fault coverage report, either in batch mode or interactively.

Figure 175 Data Flow for PowerFault Strobe Selection



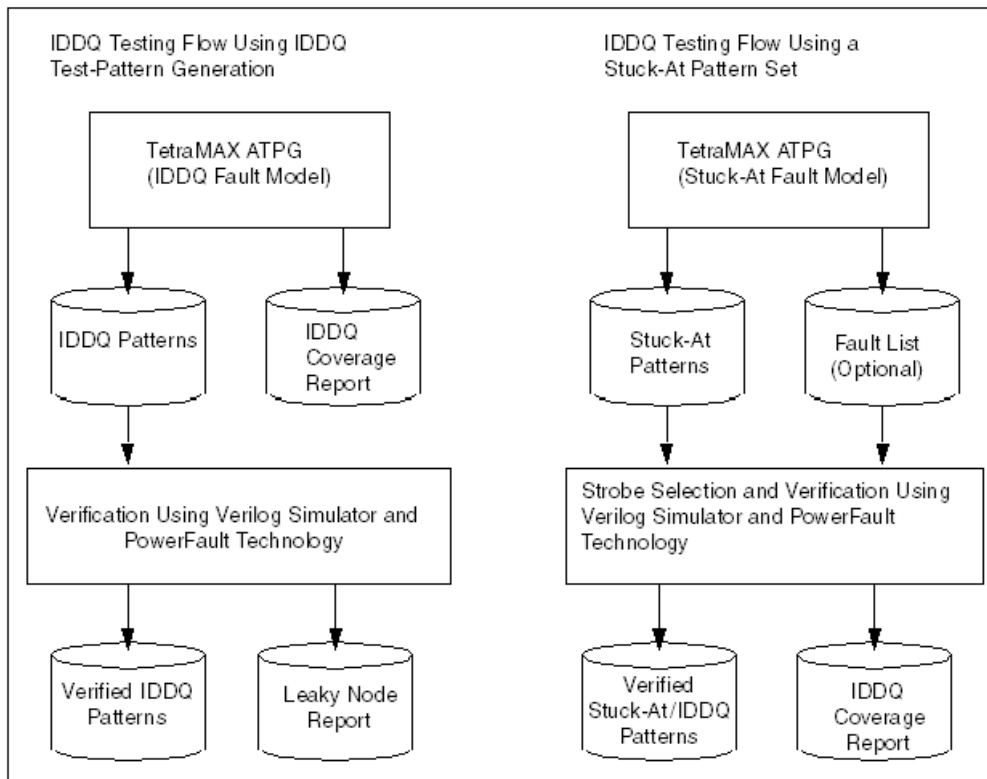
IDDQ Testing Flows

There are two recommended IDDQ testing flows:

- [IDDQ Test Pattern Generation](#) — TestMAX ATPG generates an IDDQ test pattern set targeting the IDDQ fault model rather than the usual stuck-at fault model.
- [IDDQ Strobe Selection From an Existing Pattern Set](#) — Use an existing stuck-at ATPG pattern set and let the PowerFault simulation select the best IDDQ strobe times in that pattern set.

The following figure shows the types of data produced by these two IDDQ test flows.

Figure 176 IDDQ Testing Flows



IDDQ Test Pattern Generation

In the IDDQ testing flow shown in [IDDQ Testing Flows](#), TestMAX ATPG generates a test pattern that directly targets IDDQ faults. Instead of attempting to propagate the effects of stuck-at faults to the device outputs, the ATPG algorithm attempts to sensitize all IDDQ faults and apply IDDQ strobes to test all such faults. TestMAX ATPG compresses and merges the IDDQ test patterns, just like ordinary stuck-at patterns.

While generating IDDQ test patterns, TestMAX ATPG avoids any condition that could cause excessive current drain, such as strong or weak bus contention or floating buses. You can override the default behavior and specify whether to allow such conditions by using the `set_iddq` command.

In this IDDQ flow, TestMAX ATPG 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 vector 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 a Verilog/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 TestMAX ATPG.

Having TestMAX ATPG generate the IDDQ test patterns is a very efficient method. It works best when the design uses models that are entirely structural. When the design models transistors, capacitive discharge, or behavioral elements in either the netlist or library, the ATPG might be either optimistic or pessimistic because it does not simulate the mixed-level data and signal information in the same way as the Verilog simulation module. Consider this behavior when you select your IDDQ test flow.

IDDQ Strobe Selection From an Existing Pattern Set

In the IDDQ testing flow shown in [IDDQ Testing Flows](#), PowerFault selects a near-optimum set of strobe points from an existing pattern set. The existing pattern can be a conventional stuck-at ATPG pattern or a functional test pattern. The output of this flow is the original test pattern with IDDQ strobe points identified at the appropriate times for maximum fault coverage.

In order for valid IDDQ strobe times to exist, the design must be quiescent enough of the time so that an adequate number of potential strobe points exist. You need to avoid conditions that could cause current to flow, such as floating buses or bus contention.

The specification of faults targeted for IDDQ testing is called fault seeding. There are a variety of ways to perform fault seeding, depending on your IDDQ testing goals. For example, to complement stuck-at ATPG testing done by TestMAX ATPG, you can initially target faults that could not be tested by TestMAX ATPG, such as those found to be undetectable, ATPG untestable, not detected, or possibly detected. For general IDDQ fault testing, you can seed faults automatically from the design description, or you can provide a fault list generated by TestMAX ATPG or another tool.

The Verilog/PowerFault simulator determines the quiescent strobe times in the test pattern (called the qualified strobe times) and determines which faults are detected by each strobe. Then the IDDQ Profiler finds a set of strobe points to provide maximum fault coverage for a given number of strobos.

You can optionally run the IDDQ Profiler in interactive mode, which lets you select different combinations of IDDQ strobos and examine the resulting fault coverage for each combination. This mode lets you navigate through the hierarchy of the design and examine the fault coverage at different levels and in different sections of the design.

Licensing

The Test-IDDQ license is required to perform Verilog/PowerFault simulation. This license is automatically checked out when needed, and is checked back in when the tool stops running.

By default, the lack of an available Test-IDDQ license causes an error when the tool attempts to check out a license. You can have PowerFault wait until a license becomes available instead, which lets you queue up multiple PowerFault simulation processes and have each process automatically wait until a license becomes available.

PowerFault supports license queuing, which allows the tool to wait for licenses that are temporarily unavailable. To enable this feature, you must set the `SNPSLMD_QUEUE` environment variable to a non-empty arbitrary value (“1”, “TRUE”, “ON”, “SET”, and so forth.) before invoking PowerFault:

```
unix> setenv SNPSLMD_QUEUE 1
```

Existing Powerfault behavior with `SSI_WAIT_LICENSE` will continue to be supported for backward functionality of existing customer scripts.

```
% setenv SSI_WAIT_LICENSE
```

If the license does not exist or was not installed properly, then the Verilog/PowerFault simulation will hang indefinitely without any warning or error message.

Verilog Simulation with PowerFault

PowerFault simulation technology operates as a standard programmable language interface (PLI) task that you add to your Verilog simulator. You can use PowerFault to find the best IDDQ strobe points for maximum fault coverage, to generate IDDQ fault coverage reports, to verify quiescence at strobe points, and to analyze and debug nonquiescent states.

The following sections describe Verilog simulation with PowerFault:

- [Preparing Simulators for PowerFault IDDQ](#)
- [PowerFault PLI Tasks](#)

Preparing Simulators for PowerFault IDDQ

PowerFault includes two primary parts:

- A set of PLI tasks you add to the Verilog simulator
- The IDDQ Profiler, a program that reads the IDDQ database generated by the PowerFault’s IDDQ-specific PLI tasks

Before you can use PowerFault, you need to link the PLI tasks into your Verilog simulator. The procedure for doing this depends on the type of Verilog simulator you are using and

the platform you are running. The following sections provide instructions to support the following Verilog simulators (platform differences are identified with the simulator):

- [Synopsys VCS](#)
- [Cadence NC-Verilog®](#)
- [Cadence Verilog-XL®](#)
- [Model Technology ModelSim®](#)

These instructions assume basic installation contexts for the simulators. If your installation differs, you will need to make changes to the commands presented here. For troubleshooting problems, refer to the vendor-specific documentation on integrating PLI tasks. Information about integrating additional PLI tasks is not presented here.

```
setenv SYNOPSIS root_directory  
set path=($SYNOPSIS/bin $path)
```

For a discussion about the use of the `SYNOPSIS_TMAX` environment variable, see [Specifying the Location for TestMAX ATPG Installation](#).

Then, to simplify the procedures, set the environment variable `$IDDQ_HOME` to point to where you installed Powerfault IDDQ. For example, in a typical Synopsys installation using `csh`, the command is:

```
setenv IDDQ_HOME $SYNOPSIS/iddq/
```

Note the following:

- `sparc64` and `hp64` should be used only in specific 64-bit contexts.
- PowerFault features dynamic resolution of its PLI tasks. This means that after a simulation executable has been built with the PowerFault constructs present (following the guidelines here), this executable does *not* need to be rebuilt if you change to a different version of PowerFault. Changing the environment variable `$IDDQ_HOME` to the desired version will load the runtime behavior of that version of PowerFault dynamically into this simulation run.

Using PowerFault IDDQ With Synopsys VCS

To generate the VCS simulation executable with PowerFault IDDQ, invoke VCS with the following arguments:

- Command-line argument `+acc+2`
- When running zero-delay simulations, you must use the `+delay_mode_zero` and `+tetramax` arguments.

- Command-line argument `-P $IDDQ_HOME/lib/iddq_vcs.tab` to specify the PLI table (or merge this PLI table with other tables you might already be using), a reference to `$IDDQ_HOME/lib/libiddq_vcs.a`. do not used the `-P` argument with any non-PLI Verilog testbenches.

In addition, you must specify:

- Your design modules
- Any other command-line options necessary to execute your simulation

If your simulation environment uses PLIs from multiple sources, you might need to combine the tab files from each PLI, along with the file `$IDDQ_HOME/lib/iddq_vcs.tab` for PowerFault operation, into a single tab file. See the VCS documentation for information about tab files.

The following command will enable the `ssi_iddq` task to be invoked from the Verilog source information in `model.v`:

```
vcs model.v +acc+2 -P $IDDQ_HOME/lib/iddq_vcs.tab \  
$IDDQ_HOME/lib/libiddq_vcs.a
```

For VCS 64-bit operation, if you specify the `-full64` option for VCS 64-bit contexts, you must also set `$IDDQ_HOME` to the appropriate 64-bit build for that platform: either `sparc64` for Solaris environments or `hp64` for HP-UX environments. If you do not specify the `-full64` option, then `sparcOS5` or `hp32` should be specified. Since the `-comp64` option affects compilation only, `$IDDQ_HOME` should reference `sparcOS5` or `hp32` software versions as well.

For difficulties that you or your CAD group cannot handle, contact Synopsys at:

Web: <https://solvnet.synopsys.com> Email: support_center@synopsys.com Phone: 1-800-245-8005 (inside the continental United States)

Additional phone numbers and contacts can be found at: <http://www.synopsys.com/Support/GlobalSupportCenters/Pages>

For additional VCS support, email vcs_support@synopsys.com.

Using PowerFault IDDQ With Cadence NC-Verilog

The following sections describe how to prepare for and run a PowerFault NC-Verilog simulation:

- [Setup](#)
- [Creating the Static Executable](#)
- [Running Simulation](#)

- [Creating a Dynamic Library](#)
- [Running Simulation](#)

Setup

You can use the Powerfault library with NC-Verilog in many ways. The following example describes two such flows. For both flows, set these NC-Verilog-specific environment variables:

```
setenv CDS_INST_DIR <path_to_Cadence_install_directory>
setenv INSTALL_DIR $CDS_INST_DIR
setenv ARCH <platform> //sun4v for solaris, lnx86 for linux.
setenv SNPSLMD_LICENSE_FILE <>
```

32-bit Setup

```
setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools:${LD_LIBRARY_PATH} // 32-bit
set path=($CDS_INST_DIR/tools/bin $path) // 32-bit
```

64-bit Setup

Use `+nc64` option when invoking

```
setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools/64bit:${LD_LIBRARY_PATH} //
64-bit
set path=($CDS_INST_DIR/tools/bin/64bit $path) // 64-bit
```

For the 64-bit environments use the `*cads_pic.a` libraries

Creating the Static Executable

The following steps describe how to create the static executable:

1. Create a directory to build the ncelab and ncsim variables and navigate to this directory. Create an environment variable to this path to access it quickly.

```
mkdir nc cd nc setenv LOCAL_NC "/<this_directory_path>"
```

If PowerFault is the only PLI being linked into the Verilog run, then go to step 2. If additional PLIs are being added to your Verilog environment, then go to step 3.

2. Run two build operations using your Makefile.nc

```
make ncelab $IDDQ_HOME/lib/sample_Makefile.nc make ncsim
  $IDDQ_HOME/lib/sample_Makefile.nc
```

Go to step 6.

3. Copy the PLI task and the sample makefile into the nc directory. The makefile contains the pathname of the PowerFault object file `$IDDQ_HOME/lib/libiddq_cds.a`.

```
cp $IDDQ_HOME/lib/veriusersample_forNC.c . cp
  $IDDQ_HOME/lib/sample_Makefile.nc .
```

4. Edit the example files to define additional PLI tasks.
5. Run two build operations using your `Makefile.nc`

```
make ncelab -f sample_Makefile.nc make ncsim -f sample_Makefile.nc
```
6. Ensure the directory you created is located in your path variable before the instances of these tools under the directory: `$CDS_INST_DIR`.

```
set path=($LOCAL_NC $path)
```

Running Simulation

```
ncvlog <design data and related switches>
```

```
ncelab -access +rwc <related switches>
```

```
ncsim <testbench name and related switches>
```

Make sure that the executables `ncelab` and `ncsim` picked up in the previous steps are the ones created in `$LOCAL_NC` directory, not the ones in the cadence installation path.

You can also use the single-step `ncVerilog` command as follows:

```
ncVerilog +access+rwc +ncelabexe+$LOCAL_NC/ncelab +ncsimexe+$LOCAL_NC/
ncsim <design data and other switches>
```

If using 64-bit binaries, use the “+nc64” option with the `ncVerilog` script

Creating a Dynamic Library

This section describes a flow to create a dynamic library `libpli.so` and update the path, `LD_LIBRARY_PATH` to include the path to this library. In this flow, TestMAX ATPG resolves PLI functional calls during simulation. There are two ways to build the dynamic library: either use `vconfig`, as in the first flow below, or use the `sample_Makefile.nc`, with the target being `libpli.so`.

1. Create a directory in which to build the `libpli.so` library and navigate to this directory. Set an environment variable to this location to access it quickly.

```
mkdir nc cd nc setenv LIB_DIR "<this_directory_path>"
```

2. Copy the PLI task file into the directory.

```
cp $IDDQ_HOME/lib/veriusersample_forNC.c .
```

3. Edit the sample files to define additional PLI tasks.

4. Use the `vconfig` utility and generate the script to create the `libpli.so` library. You can also use the `cr_vlog` template file shown at the end of this step.

- Name the output script `cr_vlog`.
- Select `Dynamic PLI libraries only`
- Select `build libpli`
- Ensure that you include the user template file `veriusers.c` in the link statement.

This is the `$(IDDDQ_HOME)/lib/veriusers_sample_forNC.c` file that you copied to the `$(LIB_DIR)` directory.

- Link the Powerfault object file from the pathname, `$(IDDDQ_HOME)/lib/libiddq_cds.a`

The `vconfig` command displays the following message after it completes:

```
*** SUCCESSFUL COMPLETION OF VCONFIG
***** EXECUTE THE SCRIPT: cr_vlog TO BUILD: Dynamic PLI library.
```

- Add another linking path: `$(IDDDQ_HOME)/lib` to the first compile command in the `cr_vlog` script.

The `cr_vlog` script is as follows:

```
cc -KPIC -c ./veriusers_sample_forNC.c -I$(CDS_INST_DIR)/tools/Verilog/
include -I$(IDDDQ_HOME)/lib

ld -G veriusers_sample_forNC.o $(IDDDQ_HOME)/lib/libiddq_cds.a -o
libpli.so
```

- Change the `cr_vlog` script to correspond the architecture of the machine on which it runs.
- To compile on a 64-bit machine, use the `-xarch=v9` value with the `cc` command.
- For Linux, use `-fPIC` instead of `-KPIC`. Also, you might need to replace `ld` with `gcc` or use `-lc` with `ld` on Linux.

5. Run the `cr_vlog` script to create `libpli.so` library. Ensure the directory `$(LIB_DIR)` you create is in the path, `LD_LIBRARY_PATH`.

```
setenv LD_LIBRARY_PATH ${LIB_DIR}:${LD_LIBRARY_PATH}
```

You must edit the generated `cr_vlog` script to add a reference to 64-bit environment on the `veriusers.c` compile (add `-xarch=v9`), and the `-64` option to the `ld` command.

Running Simulation

```
ncvlog <design data and related switches>
```



```
ncelab -access +rwc <related switches>
```

```
ncsim <testbench name and related switches>
```

Equivalently, single-step ncVerilog command can also be used

as

follows.

```
ncVerilog +access+rwc <design data and other switches>
```

Using PowerFault IDDQ With Cadence Verilog-XL

The following sections describe how to setup and run a PowerFault Cadence Verilog-XL simulation:

- [Setup](#)
- [Running Simulation](#)
- [Running Verilogxl](#)

Setup

To access user-defined PLI tasks at runtime, create a link between the tasks and a Verilog-XL executable using the vconfig command. The vconfig command displays a series of prompts and creates another script called cr_vlog, which builds and links the ssi_iddq task into the Verilog executable.

This is a standard procedure for many Verilog-XL users. You only need to do it only one time for a version of Verilog-XL, and it should take about 10 minutes. Cadence uses this method to support users that need PLI functionality.

After you create a link between the PowerFault IDDQ constructs and the Verilog-XL executable, you can use them each time you run the executable. The PowerFault IDDQ functions do not add overhead to a simulation run if the run does not use these functions. The functions are not loaded unless you use PowerFault IDDQ PLIs in the Verilog source files.

You do not need any additional runtime options for a Verilog-XL simulation to use PowerFault IDDQ after you create a link to it.

To create a link between the tasks and a Verilog-XL executable, do the following:

1. Set the Verilog-XL specific environment variables:

```
setenv CDS_INST_DIR <path_to_Cadence_install_directory> setenv  
INSTALL_DIR $CDS_INST_DIR setenv TARGETLIB . setenv  
ARCH <platform> setenv SNPSLMD_LICENSE_FILE <> setenv  
LD_LIBRARY_PATH $CDS_INST_DIR/tools:${LD_LIBRARY_PATH} set  
path=($CDS_INST_DIR/tools/bin $CDS_INST_DIR/tools/bin $path)
```

2. Create a directory to hold the Verilog executable and navigate into it. Set an environment variable to this location to access it quickly.

```
mkdir vlog cd vlog setenv LOCAL_XL "/<this_directory_path>"
```

3. Copy the sample veriuser.c file into this directory:

```
cp $IDDQ_HOME/lib/veriuser_sample_forNC.c .
```

4. Edit the veriuser_sample_forNC.c file to define additional PLI tasks.
5. Run the vconfig command and create the cr_vlog script to link the new Verilog executable. The vconfig command displays the following prompts. Respond to each prompt as appropriate; for example,

Name the output script `cr_vlog`. Choose a `Stand Alone` target. Choose a `Static with User PLI Application` link. Name the Verilog-XL target `Verilog`.

You can answer `no` to other options.

Create a link between your user routines and Verilog-XL. The `cr_vlog` script includes a section to compile your routines and include them in the link statement.

Ensure that you include the user template file `veriuser.c` in the link statement. This is the `$IDDQ_HOME/lib/veriuser_sample_forNC.c` file that you copied to the `vlog` directory.

Ensure that you include the user template file `vpi_user.c` in the link statement. The pathname of this file is `$CDS_INST_DIR/Verilog/src/vpi_user.c`. The `vconfig` command prompts you to accept the correct path.

Create a link to Powerfault object file as well. The pathname of this file is `$IDDQ_HOME/lib/libiddq_cds.a`

After it completes, the `vconfig` command completes:

```
*** SUCCESSFUL COMPLETION OF VCONFIG ***  
  
*** EXECUTE THE SCRIPT: cr_vlog TO BUILD: Stand Alone  
  
Verilog-XL
```

6. Add to the option `-I/$IDDQ_HOME/lib` to the first compile command in the `cr_vlog` file, which compiles the sample `veriuser.c` file.
7. Do the following before running the generated `cr_vlog` script:

Note for HP-UX 9.0 and 10.2 users:

The `cr_vlog` script must use the `-Wl` and `-E` compile options. Change the `cc` command from `cc -o Verilog` to `cc -Wl,-E -o Verilog`.

If you are using either HPUX 9.0 or Verilog-XL version 2.X, you must also create a link to the -ldld library. The last lines of the cr_vlog script must be similar to:

```
+O3 -lm -lBSD -lc1 -N -ldld
```

If you use a link editor (such as ld) instead of the cc command to create the final link, make sure you pass the -W1 and -E options as shown previously.

Note for Solaris users:

You must create a link between the cr_vlog script and the -lsocket, -lnsl, and -lintl libraries.

Check the last few lines of script and ensure these libraries are included.

8. Run the cr_vlog script. The script creates a link between the ssi_iddq task and the new Verilog executable (Verilog) in the current directory.
9. Verify that the Verilog directory appears in your path variable before other references to an executable with the same name, or reference this executable directly when running Verilog. For example,

```
set path=(./vlog $path)
```

Running Simulation

Before running simulation, ensure that the executable Verilog used to run simulation is the executable that you created in the \$LOCAL_XL directory and not the executable in the Cadence installation path.

To run simulation, use the following command:

```
Verilog +access+raw <design data and related switches>
```

Running Verilogxl

There is no command line example due to the interpreted nature of this simulation. You do not need any runtime options to enable the PLI tasks after you create a link between them and the Verilog-XL executable.

Using PowerFault IDDQ With Model Technology ModelSim

User-defined PLI tasks must be compiled and linked in ModelSim to create a shared library that is dynamically loaded by its Verilog simulator, vsim.

The following steps show you how to compile and link a ModelSim shared library:

1. Create a directory where you want to build a shared library and navigate to it; for example,

```
mkdir MTI  
cd MTI
```

2. Copy the PLI task into this directory as "veriuser.c"; for example,

```
cp $IDDQ_HOME/lib/veriuser_sample.c veriuser.c
```

3. Edit veriuser.c to define any additional PLI tasks.

4. Compile and link veriuser.c to create a shared library named "libpli.so"; for example,

```
cc -O -KPIC -c -o ./veriuser.o \  
-I<install_dir_path>/modeltech/include \  
-I$IDDQ_HOME/lib -DaccVersionLatest ./veriuser.c  
ld -G -o libpli.so veriuser.o \  
$IDDQ_HOME/lib/libiddq_cds.a -lc
```

For compiling on a 64-bit machine, use `-xarch=v9` with `cc`. For Linux, use `-fPIC` instead of `-KPIC`.

5. Identify the shared library to be loaded by `vsim` during simulation. You can do this in one of three ways:

- Set the environment variable `PLI OBJS` to the path of the shared library; for example,

```
setenv PLI OBJS  
<path_to_the_MTI_directory>/libpli.so vlog ... vsim ...
```

- Pass the shared library to `vsim` in its `-pli` argument; for example,

```
vlog ...  
vsim -pli <path_to_the_MTI_directory>/libpli.so ...
```

- Assign the path to the shared library to the `Veriuser` variable in the "modelsim.ini" file, and set the environment variable `MODELSIM` to the path of the `modelsim.ini` file; for example,

In the `modelsim.ini` file:

```
Veriuser = <path_to_the_MTI_directory>/libpli.so
```

On the command line:

```
setenv MODELSIM <path_to_modelsim.ini_file/modelsim.ini  
  
vlog ...  
  
vsim ...
```

PowerFault PLI Tasks

The following sections describe the various PowerFault PLI tasks:

- [Getting Started](#)
- [PLI Task Command Summary Table](#)
- [PLI Task Command Reference](#)

Getting Started

The first step in using PowerFault technology is to run a Verilog simulation using your normal testbench, combined with the PowerFault tasks to seed faults and evaluate potential IDDQ strobcs.

A task called `ssi_iddq` executes PowerFault commands in the Verilog file that configures the Verilog simulation for IDDQ analysis. Some of the commands are mandatory and some are optional. The commands must at least specify the device under test, seed the faults, and apply IDDQ strobcs.

For example, preparation for IDDQ testing can be as simple as adding a module similar to the following to your Verilog simulation:

```
module IDDQTEST();  
  
parameter CLOCK_PERIOD = 10000;  
  
initial begin  
  
  $ssi_iddq( "dut tbench.M88" );  
  
  $ssi_iddq( "seed SA tbench.M88" );  
  
end  
  
always begin  
  
  fork  
  
  # CLOCK_PERIOD;  
  
  # (CLOCK_PERIOD -1) $ssi_iddq( "strobe_try" );  
  
end
```

```
join
end
endmodule
```

This example contains three PowerFault commands. The first one specifies the device under test (DUT) to be `tbench.M88`. The second one seeds the entire device with stuck-at (SA) faults. Inside the `always` block, the third one invokes the `strobe_try` command to evaluate the device for IDDQ strobing at one time unit before the end of each cycle.

The order of commands in the Verilog file is important because the PLI tasks must be performed in the following order:

1. Specify the DUT module or modules (mandatory).
2. Specify other simulation setup parameters (optional).
3. Specify disallowed leaky states (optional).
4. Specify allowed leaky states (optional).
5. Specify fault seed exclusions (optional).
6. Specify fault models (optional).
7. Specify fault seeds (mandatory).
8. Run testbench and specify strobe timing (mandatory).

PLI Task Command Summary Table

The following table provides a quick summary of the PowerFault commands that you can use in Verilog files to perform PLI tasks. For detailed information on each command, see the next section, [PLI Task Command Reference](#). If you are viewing this document in online form, you can click the page number reference in the table to jump to the detailed description of the command.

Table 15 *PLI Task Command Summary*

<i>Simulation Setup Commands</i>	
<code>dut</code>	Specifies the DUT modules
<code>output</code>	Names the IDDQ database
<code>ignore</code>	Specifies black box nets and modules
<code>statedep_float</code>	Specifies the primitives that can block floating nodes
<code>io</code>	Specifies DUT ports

Table 15 *PLI Task Command Summary (Continued)*

<code>measure</code>	Specifies the rail for IDDQ measurement
<code>verb</code>	Turns verbose mode on or off (off by default)
<i>Leaky State Commands</i>	
<code>allow</code>	Allows user-specified leaky states
<code>disable SepRail</code>	Forces all top-level pullups and pulldowns in contention to be identified as leaky, see
<code>disallow</code>	Disallows user-specified leaky states
<i>Fault Seeding Commands</i>	
<code>seed SA</code>	Seeds stuck-at faults automatically
<code>seed B</code>	Seeds bridging faults automatically
<code>scope</code>	Sets the scope for faults seeded by read commands
<code>read_bridges</code>	Seeds bridging faults from a file
<code>read_tmax</code>	Seeds faults from a TestMAX ATPG fault list
<code>read_verifault</code>	Seeds faults from a Verifault fault list
<i>Fault Seed Exclusion Command</i>	
<code>exclude</code>	Excludes module instances from fault seeding
<i>Fault Model Commands</i>	
<code>model SA</code>	Configures operation of the seed SA command
<code>model B</code>	Configures operation of the seed B command
<i>Strobe Commands</i>	
<code>strobe_try</code>	Performs an IDDQ strobe evaluation if the chip is quiet; see
<code>strobe_force</code>	Forces an IDDQ strobe evaluation
<code>strobe_limit</code>	Limits the number of IDDQ strobe evaluations
<code>cycle</code>	Sets the internal cycle count
<i>Circuit Examination Commands</i>	
<code>status</code>	Prints a report on leaky nets

Table 15 *PLI Task Command Summary (Continued)*

<code>summary</code>	Prints a nodal analysis summary
----------------------	---------------------------------

PLI Task Command Reference

The following sections describe the syntax and functions of the PowerFault commands:

- [Conventions](#)
- [Simulation Setup Commands](#)
- [Leaky State Commands](#)
- [Fault Seeding Commands](#)
- [Fault Model Commands](#)
- [Strobe Commands](#)
- [Circuit Examination Commands](#)
- [Disallowed/Disallow Value Property](#)
- [Can Float Property](#)

Each command description includes the Backus-Naur form (BNF) syntax and a description of the command behavior.

Conventions

The following conventions apply to the PLI task command descriptions:

- [Special-Purpose Characters](#)
- [Module Instances and Entity Models](#)
- [Cell Instances](#)
- [Port and Terminal References](#)

Special-Purpose Characters

Several special-purpose characters are used in the command syntax descriptions, as described in the following table.

Table 16 *Special Characters in Command Syntax*

<i>Character</i>	<i>Purpose</i>
------------------	----------------

Table 16 *Special Characters in Command Syntax*
(Continued)

+	Plus-sign suffix indicates repetition of one or more
*	Asterisk suffix indicates repetition of zero or more
[]	Square brackets enclose an optional element
()	Parentheses indicate grouping
	Vertical bar separates alternative choices

When you use Verilog escaped identifiers in a command, each escape character must itself be escaped. For example, to use the name `tbench.dut.\IO(23)` with the `allow` command, use the following syntax:

```
$ssi_iddq( "allow float tbench.dut.\\IO(23)" );
```

Module Instances and Entity Models

A number of commands accept either *module-instance* or *entity-model* as a parameter. A *module-instance* is a full path name of an instantiated module, such as the module name `tbench.au.ctrl`. An *entity-model* is the definition name (not instance name) of a module. For example, `tbench.au.ctrl` might be one instance of the `IOCTRL` entity model. When you specify an entity model in a command, it applies to all instances of that model.

Cell Instances

The commands for fault seeding refer to Verilog cells. A cell instance is a module instance that has either of these characteristics:

- The module definition appears between the compiler directives ``celldefine` and ``endcelldefine`.
- The module definition is in a model library, and the `+nolibcell` option has not been used. A library is a collection of module definitions contained in a file or directory that are read by library invocation options (such as the `-y` option provided by most Verilog simulators).

If you use the `+nolibcell` option when you invoke the Verilog simulator, only modules meeting the first condition above are considered cells.

By default, PowerFault treats cells as fault primitives. It seeds faults only at cell boundaries, not inside of cells. However, some design environments generate netlists that mark very large blocks as cells. To make PowerFault seed inside those cells, use

the model SA `seed_inside_cells` command or the model B `seed_inside_cells` command.

Port and Terminal References

The commands for allowing and disallowing leaky states refer to *connection references*. A connection reference describes a port of a module or a terminal of a primitive. You can refer to a port by its name. You can also refer to ports and terminals by their index numbers, with 0 indicating the first port or terminal. For example, `port.0` refers to the first port of a module; `term.0` refers to the first terminal (the output terminal) of a primitive.

Simulation Setup Commands

The following simulation setup commands set up the general operating parameters for the PowerFault simulation, such as the name of the device under test (DUT), the name of the generated IDDQ database, and the names of the DUT ports:

- `dut`
- `output`
- `ignore`
- `io`
- `statedep_float`
- `measure`
- `verb`

dut

```
dut module-instance+
```

This command is required and must be the first `ssi_iddq-task` command executed. It specifies which instances represent the device under test. The arguments are the full path names of one or more module instances.

Here are some examples of valid `dut` commands:

```
$ssi_iddq( "dut tbench.core" );
```

```
$ssi_iddq( "dut tbench.slave tbench.master" );
```

output

```
output [mode] [label] database-name
```

```
mode ::= (create|append|replace=testbench-number)
```

```
label ::= label=string
```

This command specifies the name of the generated IDDQ database. The database is a directory that PowerFault uses to store simulation results. During the Verilog simulation, the `ssi_iddq`-task commands fill the database with information for the IDDQ Profiler. You run the IDDQ Profiler after the Verilog simulation to select stobes and generate IDDQ reports.

The following command makes the `ssi_iddq` task create a database named `/cad/sim/M88/iddq.db1`:

```
$ssi_iddq( "output /cad/sim/M88/iddq.db1" );
```

The default *mode* is `create?`, which creates the database if it does not already exist. If the database already exists, its entire contents are cleared before the new simulation results are stored.

When you use the `append` mode, the simulation results are appended to the specified database. The `append` mode allows the simulation results from multiple testbenches for a circuit to be saved into one database, as described in the “Combining Multiple Verilog Simulations” section.

The `replace` mode replaces one specified testbench result in a multiple set of results saved using the `append` mode. For the testbench number, specify 1 to overwrite the first results saved, 2 to overwrite the second results saved, and so on.

The `label` option assigns a string label to represent the current testbench. This is useful when the database is used to store results from multiple testbenches. When the IDDQ Profiler selects stobes, it uses the label to identify the testbench from which the strobe was selected.

The `append` mode is useful for a circuit that has multiple testbenches. It is much more efficient to append the results from multiple testbenches to one database, rather than create a separate database for each testbench. For details, see “Combining Multiple Verilog Simulations”.

Do not use the `append` mode with multiple concurrent simulations. For example, you cannot start four Verilog simulations at the same time and try to have each one append to the same database. If you have multiple testbenches for a circuit, you need to run them serially.

ignore

```
ignore net module-or-prim-instanceconn-ref
```

```
ignore net entity-modelconn-ref
```

```
ignore (all|core|ports) module-or-prim-instance
```

```
ignore (all|core|ports) entity-model
```

```
conn-ref ::= port-name | port.port-index |  
term.term-index  
port-name ::= scalar-port-name | vector-port-name  
[port-index]
```

The `ignore` command describes which nodes in your circuit should be ignored for IDDQ testing. Ignored nodes are excluded from analysis, fault seeding, and status reports. The same effect can be produced by using the `exclude?`, `allow fight?`, and `allow float` commands together, but using the `ignore` command is more efficient. This command overrides all built-in checkers and all custom checkers defined with the `disallow` command.

In the first two forms of the command, `conn-ref` describes which node to ignore. For example, the following command causes the node connected to the port named `INTR` in the module `tbench.core.busarb` to be ignored:

```
$ssi_iddq( "ignore net tbench.core.busarb INTR" );
```

The following command causes the node connected to the fifth port of `tbench.core.busarb` to be ignored:

```
$ssi_iddq( "ignore net tbench.core.busarb port.5" );
```

The following command causes the nodes connected to the `INTR` port of all instances of the `ARB` module to be ignored:

```
$ssi_iddq( "ignore net ARB INTR" );
```

In the last two forms of the command, the `(all|core|ports)` option describes how the command is applied to nodes of a particular module or primitive. For example, the following command causes all nodes connected to ports of the `tbench.core.pll` module to be ignored:

```
$ssi_iddq( "ignore ports tbench.core.pll" );
```

The following command causes all nodes inside `tbench.core.pll` to be ignored:

```
$ssi_iddq( "ignore core tbench.core.pll" );
```

The following command causes all nodes connected to ports and all nodes inside `tbench.core.pll` to be ignored:

```
$ssi_iddq( "ignore all tbench.core.pll" );
```

io

```
io net-instance+
```

This command lists any primary inputs and outputs (I/O pads) that are not connected to ports of the DUT modules. PowerFault assumes that each port of a DUT module is connected to an I/O pad. If your chip has I/O pads that are not connected to a port of a DUT module, you can optionally specify them with this command. Doing so might allow PowerFault to find better strobe points.

statedep_float

```
statedep_float #-and-ins#-nand-ins#-nor-ins#-or-ins
```

This command specifies the types of primitives that can block floating nodes. The default setting is:

```
$ssi_iddq( "statedep_float 3 3 2 0" );
```

By default, AND and NAND gates with up to three inputs and NOR gates with up to two inputs can block floating nodes. These primitives are commonly used to “gate” a three-state bus so that it does not cause a leakage current. For more information on this topic, see “State-Dependent Floating Nodes”.

If your foundry implements two-input OR gates so that they can block floating nodes, use this command:

```
$ssi_iddq( "statedep_float 3 3 2 2" );measure (0|1)
```

measure

```
measure (0|1)
```

This command specifies which power rail to use for IDDQ measurement. By default, PowerFault assumes that IDDQ measurements are made on the VDD (power) rail; this is the most common test method. If your automated test equipment (ATE) is configured to measure ISSQ, the current flowing through the VSS (ground) rail, use the following command:

```
$ssi_iddq( "measure 0" );
```

verb

```
verb (on|off)
```

This command turns verbose mode on and off. In verbose mode, the `ssi_iddq` task echoes every command before execution, and it also prints the result (qualified or unqualified) of every `strobe_try` command. By default, verbose mode is initially off. To turn on verbose mode, use this command:

```
$ssi_iddq( "verb on" );
```

Leaky State Commands

PowerFault has powerful algorithms for determining quiescence. By default, it recognizes two types of leaky states: floating inputs (“float”) and drive contention (“fight”). It is also

configurable; the `allow`, `disable SepRail`, and `disallow` commands let you modify the algorithms for determining quiescence.

The following sections describe the leaky state commands:

- [allow](#)
- [disable SepRail](#)
- [disallow](#)

allow

The `allow` command specifies the types of leaky states that are to be ignored. The `disallow` command defines new leaky states that would normally be unrecognized, such as leaky behavioral and external models (for more information, see “Behavioral and External Models”). The `allow` command tells PowerFault how to ignore leaky states it normally recognizes; the `disallow` command tells PowerFault how to identify leaky states it does not normally recognize.

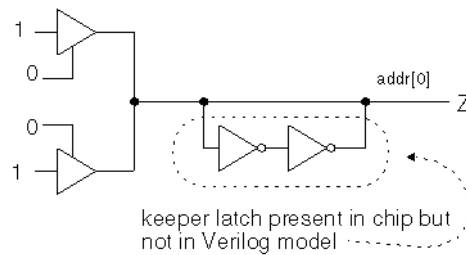
There are several different forms of this command. These are the forms that apply to specified nets, instances, or entity models:

```
allow (float|fight) net-instance  
  
allow (float|fight) module-or-prim-instance [conn-ref]  
  
allow (float|fight) entity-model [conn-ref]  
  
conn-ref ::= port-name | port.port-index | term.term-index  
  
port-name ::= scalar-port-name |  
vector-port-name[port-index]
```

These commands specify which leaky states in the design to allow (ignored by PowerFault). You can use them to have PowerFault ignore leaky states that are not present in the real chip.

Incomplete Verilog models can cause misleading leaky states, which PowerFault should ignore. For example, consider a chip that has an internal three-state bus with a keeper latch like the one shown in the following figure.

Figure 177 Three-State Bus With Keeper Latch



When the bus is fabricated on the chip, the keeper latch prevents the bus from floating. However, the Verilog model for the bus does not include the keeper latch. As a result, when the bus floats (has a Z value) during the Verilog simulation, PowerFault considers it a possible cause of high current and disqualifies any strobe try at that time.

To tell PowerFault that the bus `addr[0]` does not cause high current when it floats during the simulation, use a command like the following:

```
$ssi_iddq( "allow float tbench.iob.addr[0]" );
```

When you use a module (primitive) instance name, the `allow` command applies to all nets declared inside the instance, including those inside of submodules, and to all nets attached to the instance's ports (terminals). For example, to allow nets to float inside of and connected to `tbench.au.ctrlr?`, use this command:

```
$ssi_iddq( "allow float tbench.au.ctrlr" );
```

If you use an entity-model name, the command applies to every instance of that entity model. For example, to allow all nets to float inside of and connected to the instances of the `IOCTL` module, use the following command:

```
$ssi_iddq( "allow float IOCTL" );
```

By using the optional connection reference, you can make the command apply to a specific port or terminal. For example, if `IOCTL` has a port named `out2?`, then the following command allows the nets attached to the `out2` port of all `IOCTL` instances to float:

```
$ssi_iddq( "allow float IOCTL out2" );
```

The following command allows the nets attached to the output terminal of all `bufif0` instances to float:

```
$ssi_iddq( "allow float bufif0 term.0" );
```

To globally allow leaky states, use this command:

```
allow (all|poss) (fight|float)
```

This form of the `allow` command turns on global options that apply to every net. The `all` option makes PowerFault ignore all true and all possibly leaky states. The `poss` option makes PowerFault ignore just the possibly leaky states; true leaky states are still disallowed. For a description of true and possibly floating nodes, see “Floating Nodes and Drive Contention”.

This form of the `allow` command is most useful for verifying strobe timing and debugging test vectors. For example, if you want to find vectors that definitely have drive contention (so that you can measure it on your ATE), use these commands:

```
$ssi_iddq( "allow poss fight" );  
$ssi_iddq( "allow all float" );
```

In this case, only vectors with true drive contention are disqualified because all floating nodes and all nodes with possible drive contention are ignored.

Here is the form of the command for allowing leaky states inside cells:

```
allow cell (fight|float)
```

This form of the `allow` command applies to every net that is internal to a cell. Nets connected to cell ports and nets outside of cells are not affected. The `fight` option makes PowerFault ignore all true and possible drive contention on nets inside of cells. The `float` option makes PowerFault ignore all true and possibly floating nets inside of cells. For a description of true and possibly floating nodes, see “Floating Nodes and Drive Contention”.

This form of the `allow` command is most useful when your cell libraries have many internal nets that are erroneously flagged as floating or contending. This most commonly happens when cells use dummy local nets (nets not present in the real chip) for the purpose of timing checks. If you know that all the nets internal to your cells are always quiescent, you can use these commands:

```
$ssi_iddq( "allow cell fight" );  
$ssi_iddq( "allow cell float" );
```

disable SepRail

Current measurements, performed at test, are subject to the configuration of the test equipment when considering current contributions. Typically, many test environments use separate power supplies for the device signals (often referred to as "pin electronics") from the primary power supply for the device itself.

Because of these separate supplies, some leaky conditions might not contribute current that is measured from the device rails or primary power supply. In particular, out-of-state pullups or pulldowns on the IO of the device might not contribute to measured IDDQ current. Eliminating test vectors that do not contribute leaky current can reduce the overall effectiveness of a set of IDDQ tests. Remember, only pullups and pulldowns that are

associated with the top-level signals of the design are considered here. Internally, all current-generating situations are considered.

By default, IddQTest will not identify all out-of-state pull conditions on top-level IO signals as leaky. Certain situations are allowable. In particular, internal pulls (pullups or pulldowns that are part of the internal device definition) that are pulling to the opposite state of the measured rail (for example, internal pulldowns for IddQ measurements) will not be identified as leaky. External pulls (pullups or pulldowns that are external to the device referenced with the `dut` command) that are pulling to the same state as the measured rail (for example, external pullups for IddQ measurements) will also not be identified as leaky.

To override this default behavior, and force *all* out-of-state conditions with pullups and pulldowns at the top level of the design to be identified as leaky, the option `disable SepRail` must be specified. This can be specified as:

```
$ssi_iddq( "disable SepRail" );
```

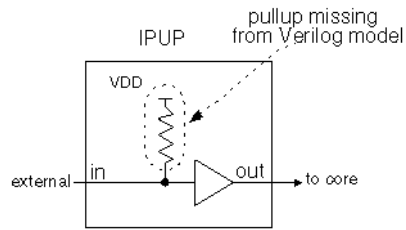
disallow

```
disallow module-or-prim-instance leaky-condition  
disallow entity-model leaky-condition  
leaky-condition ::= expr  
expr ::= ( expr ) | expr && expr | expr || expr  
| conn-ref == value | conn-ref != value  
conn-ref ::= port-name | port.port-index | term.term-index  
port-name ::= scalar-port-name |  
vector-port-name[bit-index]  
value ::= 0|1|Z|X
```

This command describes specific leaky states that would not otherwise be recognized. At every strobe try, PowerFault examines your entire netlist for leaky states. If your chip has leaky states that cannot be detected by analyzing the Verilog netlist, you might need to use the `disallow` command.

For example, consider the case where the input pads on your chip have pullups as shown in the following figure, but these pullups are missing from your Verilog models.

Figure 178 Input Macro With Pullup



If `IPUP` is the entity model for your input pad and its input port is named `in`, use the following command to tell PowerFault that the DUT is leaky when the input is 0:

```
$ssi_iddq( "disallow IPUP in == 0" );
```

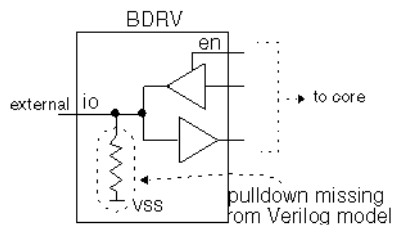
You can also refer to a port or terminal by its index number. Index numbers start at zero. For example, if port `in` is the second port in the `IPUP` port list, then the preceding command example is equivalent to the following command:

```
$ssi_iddq( "disallow IPUP port.1 == 0" );
```

The `leaky-condition` argument specifies an entity model or a particular instance that is nonquiescent. This condition is a Boolean expression describing the combination of port values or terminal values that make the chip leaky. If you specify an entity model, the condition applies to all instances of the entity model.

For example, assume the bidirectional pads on your chip have pulldowns as shown in the following figure, but those pulldowns are missing from your Verilog model.

Figure 179 Bidirectional Macro With Pulldown



To tell PowerFault that `BDRV` is an entity model that is leaky when port `io` is high and port `en` is high, use this command:

```
$ssi_iddq( "disallow BDRV ( io == 1 ) && ( en == 1 )" );
```

The `==` and `!=` operators differ from their Verilog counterparts. The expression `(conn-ref == value)` is true only if the values match exactly. For example, if `io` is X, then the expression `(io == 1)` is *not* true.

The following form of the `disallow` command turns on global options, which apply to every net:

```
disallow (Xs|Zs|Caps)
```

Turning on these options makes PowerFault follow pessimistic rules for determining quiescence. By default, nets at three-state (Z), unknown (?X?), and capacitive (Caps) values are allowed as long as they do not cause leakage. In other words, a net can be at Z if it does not have any loads.

To make PowerFault compatible with less-sophisticated IDDQ tools that disallow every X or Z, use these commands:

```
$ssi_iddq( "disallow Xs" );  
$ssi_iddq( "disallow Zs" );
```

Using these `disallow` commands, no Xs or Zs are allowed because a single X or Z implies nonquiescence and disqualifies an IDDQ strobe try. Because PowerFault analyzes the netlist in detail, if your chip is modeled structurally (the logic is implemented with Verilog user-defined primitives and ordinary primitives), you probably do not need to use this form of the `disallow` command. It is better to describe only the specific leaky states, so that more strobe times are allowed.

Fault Seeding Commands

At the beginning of the simulation, before using the `strobe_try` command to evaluate strobos for IDDQ testing, you need to tell PowerFault where to seed faults. For this purpose, you can use `seed` commands to seed faults automatically, or `read` commands to seed faults from an existing fault list.

The `seed` and `read` commands are cumulative. If you want to seed some faults automatically and seed some faults from a fault list, use both the `seed` and `read` commands.

The following sections describe the various seeding commands:

- [seed SA](#)
- [seed B](#)
- [scope](#)
- [read_bridges](#)
- [read_tmax](#)
- [read_verifault](#)

- [read_zycad](#)
- [exclude](#)

seed SA

```
seed SA module-instance+
seed SA net-instance+
```

This command seeds both stuck-at-0 and stuck-at-1 faults in each of the specified instances or nets. For module instances, PowerFault performs automatic hierarchical seeding of each module and all its lower-level modules. The placement of fault seeds (ports, terminals, and so on) is determined by the current fault model. For more information, see [Fault Model Commands](#).

Here are some examples of valid `seed SA` commands:

```
$ssi_iddq( "seed SA tbench.M88.IO tbench.M88.CORE" );
$ssi_iddq( "seed SA tbench.M88.IO.CO tbench.M88.IO.IRDY" );
```

seed B

```
seed B module-instance+
seed B net-instancenet-instance
```

This command automatically seeds bridging faults throughout the specified instances or between two specified nets. For module instances, PowerFault performs automatic hierarchical seeding of each module and all its lower-level modules. The placement of fault seeds (between ports, terminals, and so on) is determined by the current fault model. For more information, see [Fault Model Command](#).

Here are some examples of valid `seed B` commands:

```
$ssi_iddq( "seed B tbench.M88.IO" );
$ssi_iddq( "seed B tbench.M88.IO.SHF0 tbench.M88.IO.SHF1" );
```

scope

```
scope module-instance
```

This command sets the scope for the faults seeded by subsequent `read_type` commands. By default, PowerFault expects full path names for all fault entries. Some ATPG environments generate fault entries that have incomplete path names (for example, without the testbench module name). For those environments, use the `scope` command to specify a prefix for all path names.

For example, the following four commands tell PowerFault to do the following: seed faults from files `tbench.core` and `tbench.io?`, consider all names in `U55.flist` to be relative to `tbench.core?`, and consider all names in `U24.flist` to be relative to `tbench.io?`:

```
$ssi_iddq( "scope tbench.core" );  
$ssi_iddq( "read_tmax U55.flist" );  
$ssi_iddq( "scope tbench.io" );  
$ssi_iddq( "read_tmax U24.flist" );
```

read_bridges

```
read_bridges file-name
```

This command reads the names of net pairs from a file (one pair per line) and seeds a bridging fault between each listed pair. For example, a file containing the following two lines would seed bridging faults in the `tbench.M88` module between `TA` and `TB?`, and between `PA` and `PB?`:

```
tbench.M88.TA tbench.M88.TB  
tbench.M88.PA tbench.M88.PB
```

read_tmax

```
read_tmax [strip] fault-classes* file-name
```

```
fault-classes ::= (DS|DI|AP|NP|UU|UO|UT|UB|UR|AN|NC|NO|-- )
```

This command reads fault entries from a TestMAX ATPG fault list. By default, only fault entries in the AP, NP, NC, and NO classes are seeded. If you want to seed faults in other classes, use the `fault-classes` argument to specify the fault classes. For definitions of these fault classes, refer to the *TestMAX ATPG User Guide*.

For example, the following command seeds faults in the `fa1` file that are in the following classes: possibly detected (AP, NP), undetectable (UU, UT, UB, UR), ATPG untestable (AN), and not detected (NC, NO):

```
$ssi_iddq( "read_tmax AP NP UU UT UB UR AN NC NO fa1" );
```

By default, PowerFault remembers all the comment lines and unseeded faults in the fault list, so that when it produces the final fault report, you can easily compare the report to the original fault list. If you do not want to save this information (it requires extra disk space), use the `strip` option:

```
$ssi_iddq( "read_tmax strip AP NP UU UT UB UR AN NC NO fa1" );
```

read_verifault

```
read_verifault [strip] status-types* file-name
```

```
status-types ::= (detected|potential|undetected|
drop_task|drop_active|drop_looping|drop_detected |
drop_potential|drop_pli|drop_hyper_active|
drop_hyper_mem|untestable )
```

This command reads fault seeds from a Verifault-XL fault list. By default, only fault descriptors without status or with the status `undetected` or `potential` are seeded. If you want to seed faults with other status types, use the `status-types` argument to specify the status types.

For example, the following command seeds all faults with status `potential`, `undetected`, or `untestable` from the file `M88.flist?`:

```
$ssi_iddq( "read_verifault potential undetected untestable
M88.flist" );
```

By default, PowerFault remembers all the comment lines and unseeded faults in the fault list, so that when it produces the final fault report, you can easily compare the report to the original fault list. If you do not want to save this information (it requires extra disk space), use the `strip` option:

```
$ssi_iddq( "read_verifault strip potential undetected
untestable M88.flist" );
```

read_zycad

```
read_zycad [strip] fault-types* result-types* file-name
fault-types ::= (i|o|n)
result-types ::= (C|D|H|I|M|N|O|P|U)
```

This command reads fault seeds from a Zycad fault origin file. By default, only fault origins with the node type (`n`) and the undetected (`u`) or not run yet (`N`) or possibly (`P`) result are seeded. If you want to seed other fault types or results, use the `fault-types` and `result-types` arguments to specify them.

For example, the following command seeds all input and output faults with the impossible (`I`) and possibly (`P`) result from the file `M88.fog?`:

```
$ssi_iddq( "read_zycad i o I P M88.fog" );
```

exclude

```
exclude module-instance+
exclude primitive-instance+
```

```
exclude entity-model+
```

The `exclude` command excludes specified parts of the design from fault seeding. This command specifies instances and entities that are to be excluded from the fault seeding performed by the `seed?`, `read_tmax?`, `read_verifault?`, and `read_zycad` commands.

For example, to exclude all instances of the `BRAM16` entity from fault seeding, use the following command:

```
$ssi_iddq( "exclude BRAM16" );
```

To exclude individual instances, specify the full path name of each instance:

```
$ssi_iddq( "exclude tbench.M88.io tbench.M88.mem" );
```

The `exclude` command excludes only instances from seeding. It does not exclude them from being checked for leaky states. If you need to ignore a leaky state, use the `allow` command, described in [Leaky State Commands](#).

Fault Model Commands

The `model` commands determine where the `seed` commands will place faults. Therefore, if you use a `model` command, you must execute it before the `seed` command. When you specify a module instance name in the `seed` command, the seeding algorithm performs a hierarchical traversal of the instance, seeding faults on the ports and terminals specified by the current fault model. By default, this traversal stops at cell boundaries.

The settings made with a `model` command are not cumulative. The current model is based only on the most recent `model` command. In other words, each `model` command overwrites the settings made by the previous `model` command.

The following sections describe the fault model commands:

- [model SA](#)
- [model B](#)

model SA

```
model SA directionsa-placement [seed_inside_cells]  
direction ::= (port_IN|port_OUT|term_IN|term_OUT)+  
sa-placement ::= (all_mods|leaf_mods|cell_mods|prims)+
```

This command specifies where the `seed SA` command seeds stuck-at faults. The following table summarizes the command options.

Table 17 Options for Stuck-At Fault Models

Direction Options

Table 17 Options for Stuck-At Fault Models (Continued)

<code>port_IN</code>	Enables stuck-at faults on input ports of chosen modules
<code>port_OUT</code>	Enables stuck-at faults on output ports of chosen modules
<code>term_IN</code>	Enables stuck-at faults on input terminals of primitives
<code>term_OUT</code>	Enables stuck-at faults on output terminals of primitives
Stuck-At Placement Options	
<code>all_mods</code>	Chooses all modules for port stuck-at faults
<code>leaf_mods</code>	Chooses leaf modules for port stuck-at faults
<code>cell_mods</code>	Chooses cell modules for port stuck-at faults
<code>prims</code>	Chooses primitives for terminal stuck-at faults
Seed Inside Cells Option	
<code>seed_inside_cells</code>	Enables fault seeding inside cells

The default stuck-at fault seeding behavior is equivalent to the following `model SA` command:

```
model SA port_IN port_OUT term_IN term_OUT
leaf_mods cell_mods prims
```

With the default stuck-at fault model, faults are seeded on input and output ports of cell and leaf modules, and on input and output terminals of every primitive, but not inside cells. Primitives and modules found inside of cells are ignored. A leaf module is a module that does not contain any instances of submodules.

If you want to seed inside cells, include the `seed_inside_cells` option. For example, these two lines seed stuck-at faults on output terminals of every primitive, including those inside cells:

```
$ssi_iddq( "model SA term_OUT prims seed_inside_cells" );
$ssi_iddq( "seed SA tbench.M88" );
```

For detailed examples showing how the `model SA` command options affect the placement of fault seeds, see [Options for PowerFault-Generated Seeding](#).

model B

```
model B bridge-placement [seed_inside_cells]

bridge-placement ::= (cell_ports|fet_terms|
gate_IN2IN|gateIN2OUT|vector)+
```

This command specifies where the `seed B` command seeds bridging faults. A bridging fault is a short circuit between two different functional nodes in the design. A fault of this type is considered detected by an IDDQ strobe when one node is at logic 1 and the other is at logic 0.

The following table summarizes the `model B` command options.

Table 18 Options for Bridging Fault Models

<i>Bridge Placement Options</i>	
<code>cell_ports</code>	Enables bridging faults between adjacent ports of cells and between each input and output port of cells (if the cell has two or fewer output ports)
<code>fet_terms</code>	Enables bridging faults between all pairs of terminals of field effect transistor (FET) switches
<code>gate_IN2IN</code>	Enables bridging faults between adjacent input terminals of non-FET primitives (including UDPs)
<code>gate_IN2OUT</code>	Enables bridging faults between all pairs of input and output terminals of non-FET primitives (including UDPs)
<code>vector</code>	Enables bridging faults between adjacent bits of expanded vectors
<i>Seed Inside Cells Option</i>	
<code>seed_inside_cells</code>	Enables fault seeding inside cells

The default bridging fault seeding behavior is equivalent to the following `model B` command:

```
model B cell_ports fet_terms gate_IN2IN gate_IN2OUT vector
```

With the default bridging fault model, bridging faults are seeded between the ports of cells, the terminals of primitives, and the bits of expanded vectors. No seeding is performed inside cells.

To seed other types of bridging faults, specify them with the `model B` command. For example, these two lines seed bridging faults between the ports of all cells inside `tbench.M88`:

```
$ssi_iddq( "model B cell_ports" );  
$ssi_iddq( "seed B tbench.M88" );
```

For detailed examples showing how the `model B` command options affect the placement of fault seeds, see [Options for PowerFault-Generated Seeding](#).

Strobe Commands

After you specify the DUT modules and seed the faults, you need to describe the IDDQ strobe timing. When the testbench is running, it must use either the `strobe_try` or `strobe_force` command to indicate when it is appropriate to apply an IDDQ strobe.

The following sections describe the various strobe commands:

- [strobe_try](#)
- [strobe_force](#)
- [strobe_limit](#)
- [cycle](#)

strobe_try

```
strobe_try
```

You should have the testbench invoke the `strobe_try` command at as many potential strobe times as possible. The `strobe_try` command tells PowerFault that the circuit is stable and can be tested for quiescence.

For example, you can use the following line just before the end of each cycle:

```
$ssi_iddq("strobe_try")
```

At each occurrence of this line, PowerFault determines whether the circuit is quiescent, allowing an IDDQ strobe to be applied. If the `verb on` command has been executed, the simulator reports the result of each `strobe_try?`, allowing you to identify nonquiescent strobe times.

You should use the `strobe_try` command one time per tester cycle, and it should be the last event of the cycle. For example, if you have delay paths that take multiple clock cycles, do not use the command when those paths are active.

strobe_force

```
strobe_force
```

This command turns off quiescence checking and allows PowerFault to consider all strobe times. Use this command only if you are sure the chip is quiescent. For example, you can use it if your technology provides an `IDDQ_OK` signal that forces the chip into quiescence.

If you know the quiescent points in your simulation, you can use the `strobe_force` command rather than the `strobe_try` command to reduce the simulation runtime. With the `strobe_force` command, PowerFault does not need to check the entire chip for quiescence at each strobe try.

strobe_limit

```
strobe_limit max-strobes
```

This command terminates the Verilog simulation when `max-strobes` qualified strobe points have been found.

For example, the following command stops the simulation after 100 qualified strobe points have been found:

```
$ssi_iddq( "strobe_limit 100" );
```

cycle

```
cycle cycle-number
```

This command sets the initial PowerFault cycle number, an internal counter maintained by PowerFault. The cycle number has no affect on finding or selecting IDDQ strobcs. It is used during Verilog simulations and during strobe selection to report a cycle number along with the simulation time of each strobe.

By default, the cycle number begins at 1 and is incremented after every strobe try. If your test program does not strobe on every cycle, you can use the `cycle` command to synchronize PowerFault with the cycle count of your test program. For example, if your cycle count begins at 0 instead of 1, use this command:

```
$ssi_iddq( "cycle 0" );
```

The `cycle` command can also accept a nonstring argument, allowing you to set the cycle number to the value of a simulation variable. For example:

```
always @testbench.CYCLE  
  
$ssi_iddq( "cycle", testbench.CYCLE );
```

Circuit Examination Commands

The circuit examination commands, `status` and `summary?`, provide information on the location and cause of IDDQ testing problems found in the design.

The following sections describe the circuit examination commands:

- [status](#)
- [summary](#)

status

```
status [drivers]
(leaky|nonleaky|both|all_leaky) [file-name]
```

This command determines why your circuit is quiescent or nonquiescent at a particular simulation time. It is most useful when you are having difficulty producing qualified strobe points.

If there is a persistent leaky node in your circuit (for example, caused by an always-active pulldown), PowerFault will not be able to find quiescent strobe points. Fortunately, the `status leaky` command can quickly identify any leaky nodes, allowing you to improve your test program so that it produces more quiescent strobe points.

Use the following command to print out all the net conditions that imply that the circuit is not quiescent:

```
$ssi_iddq( "status leaky" );
```

The command prints out the name of each leaky net and the reason that the net's value implies that the circuit is not quiescent. There are two possible causes for a leaky node: a floating input or drive contention.

Here is an example of a report generated by the `status` command:

```
Time 35799
top.dut.ioctl.stba is leaky. Re: float
top.dut.ioctl.addr[0] is leaky. Re: fight
top.dut.ioctl.addr[1] is leaky. Re: possible fight
```

If you use the `status` command and the `strobe_try` command in the same simulation run, and you want the status report to include the first strobe, you must execute the first `status` command before the first `strobe_try` command.

Use the following command to print out all the net conditions that imply that the circuit is quiescent:

```
$ssi_iddq( "status nonleaky" );
```

Use the following command to print out all the net conditions that imply that the circuit is or is not quiescent:

```
$ssi_iddq( "status both" );
```

The output of the `status` command can be quite long because it can contain up to one line for every net in the chip. You can direct the output to a file instead of to the screen. For example, to write the leaky states into a file named `bad_nets?`, use the following command:

```
$ssi_iddq( "status leaky bad_nets" );
```

The simulator creates the `bad_nets` file the first time it executes the `status` command. When it executes the `status` command again in the same simulation run, it appends the output to the `bad_nets` file, together with the current simulation time. This creates a report of the leaky states at every disqualified strobe time.

By default, the `leaky` option reports only the first occurrence of a leaky node. If the same leaky condition occurs at different strobe times, the report says “All reported” at each such strobe time after the report of the first occurrence. To get a full report on all leaky nodes, including those already reported, use the `all_leaky` option instead of the `leaky` option, as in the following example:

```
$ssi_iddq( "status all_leaky bad_nodes" );
```

This can produce a very long report.

The `drivers` option makes the `status` command print the contribution of each driver. However, it reports only gate-level driver information. For example, consider the following command:

```
$ssi_iddq( "status drivers leaky bad_nodes" );
```

The command produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
St0<- top.dut.mmu.UT344
St1<- top.dut.mmu.UT366
StX<- resolved value
top.dut.mmu.TDATA is leaky: Re: float
HiZ<- top.dut.mmu.UT455
HiZ<- top.dut.mmu.UT456
```

In this example, `top.dut.mmu.DIO` has a drive fight. One driver is at strong 0 (`st0`) and the other is at strong 1 (`st1`). The contributing value of each driver is printed in Verilog strength/value format (described in section 7.10 of the IEEE 1364 Verilog LRM).

The same `status` command without the `drivers` option produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
```

```
top.dut.mmu.TDATA is leaky: Re: float
```

summary

```
summary file-name
```

When you use the `summary` command, PowerFault prints a summary at the end of the simulation that describes problem nodes. It lists the nodes reported by the `status` command and also lists the nodes that were not reported but might cause problems.

The summary for each node is reported in this format:

```
net-instance-name: property+
```

The `summary` command merges simulation information reported by the `status` command with static information from the formal analyzer. For example, consider the case where the `status` command produces the following output:

```
Time 3999
tbench.M88.SELM.RESET is leaky: Re: float
tbench.M88.VEE[0] is leaky: Re: float
HiZ <- tbench.M88.CB.vee0.out
HiZ <- tbench.M88.LB.vee0.out
Time 12999
tbench.M88.DIO[1] is leaky: Re: possible fight
St0 <- tbench.M88.dpad1_clr
StX <- tbench.M88.dpad1_snd
StX <- resolved value
tbench.M88.BIO is leaky: Re: disallowed X
tbench.M88.U244 is leaky: Re: ARAM (WR_EN == 1 && DATA[0]
== Z)
```

The corresponding summary might look like this:

```
Summary of problem nodes:
tbench.M88.SELM.RESET: did float : unconnected
tbench.M88.VEE[0]: did float : not muxed
tbench.M88.DIO[1]: did fight : can float : not muxed
```

```
tbench.M88.BIO: disallowed value
tbench.M88.U244: disallow ARAM (WR_EN == 1 && DATA[0] == Z)
tbench.M88.APP.POW: constant fight
```

The summary lists nodes that can cause problems for IDDQ testing. It might also identify node properties that are considered design problems. For example, if floating nodes are illegal in your design environment, you should check to see whether any nodes have the “did float” or “can float” property.

The more your circuit is modeled at the gate level, the more accurate the summary is.

The following table lists and describes the node properties reported by the `summary` command.

Table 5 Node Properties Reported by `summary` Command

Node Property	Description
did float	The node was reported as floating (or possibly floating) during simulation.
did fight	The node was reported as having (or possibly having) drive contention during the simulation.
did pull	The node was reported as having (or possibly having) an active pullup/pulldown during simulation.
disallowed value	The node was reported as violating a simple <code>disallow</code> command during the simulation.
disallow <i>expr</i>	The node was reported as violating a compound <code>disallow</code> command during the simulation. <i>expr</i> contains the text of the <code>disallow</code> command.
can float	The node can float, but was not reported as floating during the simulation.
can fight	The node can have drive contention, but was not reported as having this condition during the simulation.
can pull	The node has pullups/pulldowns, but they were not active during the simulation.
not muxed	The node has multiple drivers that are not multiplexed. In other words, the control logic for the drivers does not always enable one and only one driver at a time.

Node Property	Description
unconnected	The node is an unconnected input.
constant	The node has a constant current. In other words, it has both a pullup and a pulldown.

Disallowed/Disallow Value Property

A node with the “disallowed value” property violated a simple `disallow` command at some time during the simulation. Here are some examples of simple `disallow` commands:

```
$ssi_iddq( "disallow tbench.M88 (BIO == X)" );
```

```
$ssi_iddq( "disallow BUF3I (out == 0)" );
```

A node with the “disallow *expr*” property violated a compound `disallow` command at some time during the simulation. Here are some examples of compound `disallow` commands:

```
$ssi_iddq( "disallow ARAM (WR_EN == 1 && DATA[0] == Z)" );
```

```
$ssi_iddq( "disallow PHMX (in == 1 && en != 0)" );
```

Can Float Property

Each node with the “can float” property requires special consideration because it can cause high current. Each such node was never reported as floating during the simulation because of one or more of these conditions:

- The node never floated.
- The node floated but was blocked.
- The node floated but did not have a load (it was not connected to a gate-level input).

For more information, see [Floating Nodes and Drive Contention](#).

Faults and Fault Seeding

The process of specifying fault locations for IDDQ testing is called *fault seeding*. You can have PowerFault seed faults automatically from the design description, or you can use a fault list generated by TestMAX ATPG or another tool.

The following sections describe faults and fault seeding:

- [Fault Models](#)
- [Fault Seeding](#)
- [Options for PowerFault-Generated Seeding](#)

Fault Models

The TestMAX ATPG and Verilog/PowerFault environments support several different types of fault models which are described in the following sections:

- [Fault Models in TestMAX ATPG](#)
- [Fault Models in PowerFault](#)

Fault Models in TestMAX ATPG

In TestMAX ATPG, the term “fault model” refers to the type of fault used for test pattern generation.

- For IDDQ testing, there are two choices: stuck-at and IDDQ. The stuck-at fault model is the standard, default model most often used to generate test patterns.
- The IDDQ fault model is used to generate test patterns specifically for IDDQ testing.

There are two types of IDDQ fault models, the pseudo-stuck-at model and the toggle model.

The fault model choice in TestMAX ATPG determines how the ATPG algorithm operates. For the stuck-at model, TestMAX ATPG attempts to propagate the effects of faults to the scan elements and device outputs. For the IDDQ model, TestMAX ATPG attempts to control all nodes to 0 and 1 while avoiding conditions that violate quiescence.

For more information on TestMAX ATPG fault models, see the *TestMAX ATPG User Guide* or consult the TestMAX ATPG online help.

Fault Models in PowerFault

In the PowerFault environment, the term “fault model”? refers to the algorithm used to seed faults in the design when you use the `seed SA` command to seed stuck-at faults or the `seed B` command to seed bridging faults.

Stuck-At Faults

A stuck-at-0 fault is considered detected when the node in question is placed in the 1 state, the circuit is quiescent, and an IDDQ strobe occurs. Similarly, a stuck-at-1 fault is considered detected when the node is placed in the 0 state, the circuit is quiescent, and an IDDQ strobe occurs.

To seed stuck-at faults from a TestMAX ATPG fault file, use the `read_tmax` command. Similar commands are available to seed faults from a Verifault fault list. To seed stuck-at faults automatically throughout the design based on the locations of the modules, cells, primitives, ports, and terminals in the design, use the `model SA` and `seed SA` commands.

Untestable faults are ignored during fault detection and strobe selection, but they are still listed in fault reports for reference. Faults untestable by PowerFault include stuck-at-0 faults on supply0 wires and stuck-at-1 faults on supply1 wires.

Bridging Faults

A bridging fault involves two nodes. The fault is considered detected when one node is placed in the 1 state, the other is placed in the 0 state, the circuit is quiescent, and an IDDQ strobe occurs. For an accurate fault model, the two nodes in question must be physically adjacent in the fabricated device, so that actual bridging between the nodes is possible in a defective device.

You can seed bridging faults by reading them from a list (which could be generated by an external tool) by using the `read_bridges` command. You can also seed bridging faults automatically between adjacent cell ports, between terminals of field effect transistor (FET) switches, between the terminals of gate primitives, and between adjacent vector bits. In this case, *adjacent* means “right next to each other in the Verilog description.” To seed bridging faults in this manner, use the `model B` and `seed B` commands.

Fault Seeding

At the beginning of the Verilog/PowerFault simulation, before using the `strobe_try` command to evaluate strobos for IDDQ testing, you need to tell PowerFault where to seed faults. To do this, you can use `seed` commands to seed faults automatically or the `read_tmax` command to seed faults from an existing fault list.

The `seed` and `read_tmax` commands are cumulative. If you want to seed some faults automatically and seed some faults from a fault list, you can use both the `seed` and `read_tmax` commands, and all of the faults seeded by the two commands are used.

The following sections describe fault seeding:

- [Seeding From a TestMAX ATPG Fault List](#)
- [Seeding From an External Fault List](#)
- [PowerFault-Generated Seeding](#)

Seeding From a TestMAX ATPG Fault List

To seed the design with stuck-at faults from a TestMAX ATPG fault list, use the `read_tmax` command. In this command, you specify the TestMAX ATPG fault file name, and optionally, the detectability classes of faults to be seeded.

In TestMAX ATPG, you create a fault file upon completion of test pattern generation by using the `write_faults` command. Typically, you write a complete fault list using a command similar to the following:

```
write_faults mylist.faults -replace -all
```

Before you generate the fault list, you need to set the hierarchical delimiter character in TestMAX ATPG. PowerFault expects the delimiter character to be a period. By default, TestMAX ATPG uses the forward slash (/) character. To generate the fault list in a compatible format, use the following `set_build` command before you build the model:

```
set_build -hierarchical_delimiter .
```

The generated fault file describes each fault in terms of type (stuck-at-0 or stuck-at-1), detectability class, and location in the design. For example:

```
sa0 DS .testbench.fadder.co
sa1 DS .testbench.fadder.co
sa0 DS .testbench.fadder.sum
sa1 DS .testbench.fadder.sum
...
```

Each fault class and each hierarchical group of fault classes has a two-character abbreviation. For example, DS stands for “detected by simulation.”

The TestMAX ATPG fault classes are defined in the following list:

```
DT - detected
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
UO - undetectable, unobservable
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

TestMAX ATPG places each fault into one of the bottom-level fault classes. For more information about fault classes, refer to the *TestMAX ATPG User Guide*.

By default, the PowerFault command `read_tmax` seeds faults in the AP, NP, NC, and NO classes. If you want to seed faults belonging to classes other than the default set, you need to specify the classes in the `read_tmax` command. For example, the following command seeds faults in the `fa1` file that belong to the following classes: possibly detected (AP, NP), undetectable (UU, UT, UB, UR), ATPG untestable (AN), and not detected (NC, NO):

```
$ssi_iddq( "read_tmax AP NP UU UT UB UR AN NC NO fa1" );
```

One way to use this command is to target undetectable and possibly detected faults in TestMAX ATPG. In this way, PowerFault complements TestMAX ATPG to obtain the best possible overall test coverage. If adequate coverage of these faults is obtained with just a few IDDQ strobcs and if your tester time budget allows it, you can then seed faults throughout the design with the `seed SA` command and generate additional IDDQ strobcs to obtain even better IDDQ test coverage.

Seeding From an External Fault List

If you use the Verifault-XL fault simulator, you can seed the design with faults from a Verifault fault list or fault dictionary. Similarly, if you use the Zycad fault simulator, you can seed the design with faults from the Zycad `f`

To seed faults from these types of files, use the `read_verifault` command, described in “`read_verifault`”.

To seed the design with bridging faults from a file-based list, use the `read_bridges` command. For details, see “`read_bridges`” in the [PowerFault PLI Tasks](#) section.

PowerFault-Generated Seeding

To have PowerFault automatically seed the design, use the `seed SA` command to seed stuck-at faults or the `seed B` command to seed bridging faults. To specify how these seeding algorithms operate, use the `model SA` and `model B` commands. For details, see “Fault Model Commands” in the [PowerFault PLI Tasks](#) section.

Options for PowerFault-Generated Seeding

For PowerFault-generated seeding, use the `seed SA` and `seed B` commands. The `model SA` and `model B` commands specify the behavior of the seeding algorithms.

The following sections provide some specific examples showing how you can use the `model SA` and `model B` command options to control the seeding of faults in the design:

- [Stuck-At Fault Model Options](#)
- [Bridging Faults](#)

For basic information on using the `model SA` or `model B` command, see “model SA” or “model B” in [PowerFault PLI Tasks](#) section.

Stuck-At Fault Model Options

The `model SA` command determines where the `seed SA` command seeds stuck-at faults. The following table lists and describes the fault model options available in the `model SA` command.

Table 19 Options for Stuck-At Fault Models

<i>Direction Options</i>	
<code>port_IN</code>	Enables stuck-at faults on input ports of chosen modules
<code>port_OUT</code>	Enables stuck-at faults on output ports of chosen modules
<code>term_IN</code>	Enables stuck-at faults on input terminals of primitives
<code>term_OUT</code>	Enables stuck-at faults on output terminals of primitives
<i>Stuck-At Placement Options</i>	
<code>all_mods</code>	Chooses all modules for port stuck-at faults
<code>leaf_mods</code>	Chooses leaf modules for port stuck-at faults
<code>cell_mods</code>	Chooses cell modules for port stuck-at faults
<code>prims</code>	Chooses primitives for terminal stuck-at faults
<i>Seed Inside Cells Option</i>	
<code>seed_inside_cells</code>	Enables fault seeding inside cells

The `all_mods?`, `leaf_mods?`, and `cell_mods` options specify which types of modules will have port faults. The `port_IN` and `port_OUT` options specify which types of ports from those modules are seeded with stuck-at faults.

The `prims` option specifies that any primitive instance found within a seeded module will have terminal faults. The `term_IN` and `term_OUT` options specify which types of terminals from those primitives are seeded with stuck-at faults.

Here is a specific example to help demonstrate how these options work. Assume that you have the following Verilog description of a testbench module called `tbench.M88`:

```
module M88();

  hier hmod( hout, hin );
  leaf lmod( lout, lin );
  cell cmod( cout, cin );
  nand( nout, nin1, nin2 );
endmodule

module hier( out, in );
  output out;
  input in;
  leaf lmod( lout, lin );
endmodule

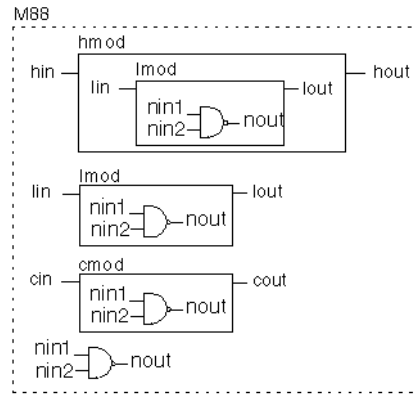
module leaf( out, in );
  output out;
  input in;
  nand( nout, nin1, nin2 );
endmodule

`celldefine
module cell( out, in );
  output out;
  input in;
  nand( nout, nin1, nin2 );
```

```
endmodule
`endcelldefine
```

At the top level of hierarchy, this testbench module contains a hierarchical module (?hmod?), a leaf-level module (?lmod?), a module that has been defined as a cell (?cmod?), and a primitive gate (?nand?). The following figure shows a circuit diagram corresponding to this Verilog description.

Figure 180 Circuit Example for Stuck-At Fault Seeding



Default Stuck-At Fault Seeding

By default, the `seed SA` command seeds port faults on leaf and cell modules and seeds terminal faults on primitives. The default behavior is equivalent to using the following `model SA` command:

```
model SA port_IN port_OUT term_IN term_OUT
leaf_mods cell_mods prims
```

Suppose that you start stuck-at seeding using the default model:

```
$ssi_iddq( "seed SA tbench.M88" );
```

This command seeds stuck-at faults on the following nets:

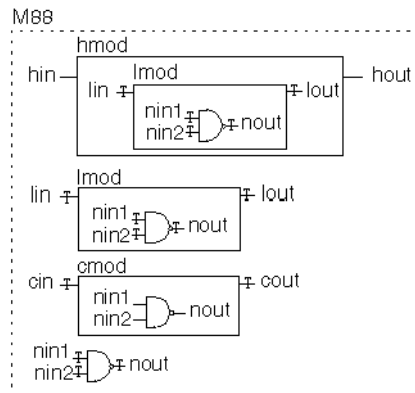
```
tbench.M88.lmod.lin
tbench.M88.lmod.lout
tbench.M88.hmod.lmod.nin1
tbench.M88.hmod.lmod.nin2
tbench.M88.hmod.lmod.nout
tbench.M88.lin
```

Chapter 34: PowerFault
Faults and Fault Seeding

```
tbench.M88.lout
tbench.M88.lmod.nin1
tbench.M88.lmod.nin2
tbench.M88.lmod.nout
tbench.M88.cin
tbench.M88.cout
tbench.M88.nin1
tbench.M88.nin2
tbench.M88.nout
```

The following figure shows the circuit diagram with each seeded fault marked with an asterisk (*).

Figure 181 Seed Locations: Default Stuck-At Fault Model



all_mods

The `all_mods` option chooses all modules for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of all modules inside `tbench.M88`:

```
$ssi_iddq( "model SA port_IN port_OUT all_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

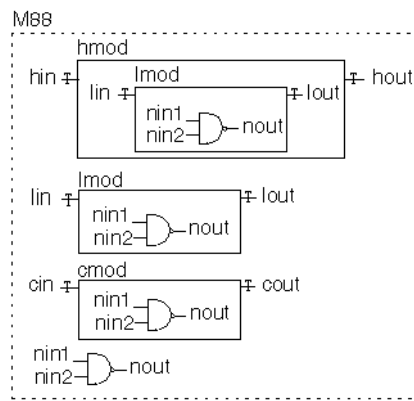
```
tbench.M88.hin
tbench.M88.hout
tbench.M88.hmod.lin
```


Chapter 34: PowerFault
Faults and Fault Seeding

```
tbench.M88.hmod.lout
tbench.M88.lin
tbench.M88.lout
tbench.M88.cin
tbench.M88.cout
```

The following figure shows the resulting locations of seeds using this fault model.

Figure 182 Seed Locations: all_mods Stuck-At Fault Model



cell_mods

The `cell_mods` option chooses cells for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of every cell module inside `tbench.M88`?:

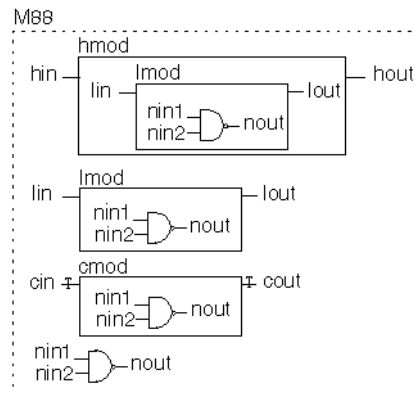
```
$ssi_iddq( "model SA port_IN port_OUT cell_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.cin
tbench.M88.cout
```

The following figure shows the resulting locations of seeds using this fault model.

Figure 183 Seed Locations: cell_mods Stuck-At Fault Model



leaf_mods

The `leaf_mods` option chooses leaf-level modules for port stuck-at faults. Thus, the following two lines seed faults on the input and output ports of every leaf module inside `tbench.M88`:

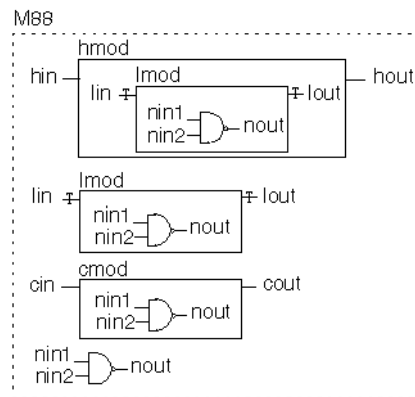
```
$ssi_iddq( "model SA port_IN port_OUT leaf_mods" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lin
tbench.M88.hmod.lout
tbench.M88.lin
tbench.M88.lout
```

The following figure shows the resulting locations of seeds using this fault model.

Figure 184 Seed Locations: leaf_mods Stuck-At Fault Model



prims

The `prims` option chooses primitives for terminal stuck-at faults. Thus, the following two lines seed faults on the input terminal of every primitive inside `tbench.M88`:

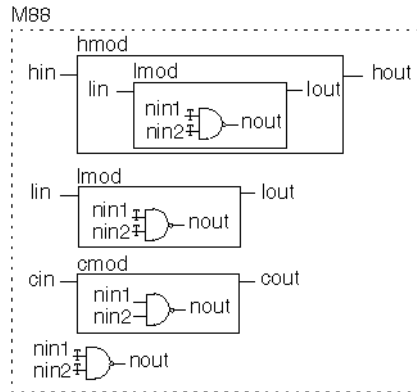
```
$ssi_iddq( "model SA term_IN prims" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

- tbench.M88.hmod.lmod.nin1
- tbench.M88.hmod.lmod.nin2
- tbench.M88.lmod.nin1
- tbench.M88.lmod.nin2
- tbench.M88.nin1
- tbench.M88.nin2

The following figure shows the resulting locations of seeds using this fault model.

Figure 185 Primitive Input Stuck-At Fault Model



The following two lines seed faults on the output terminal of every primitive inside tbench.M88:

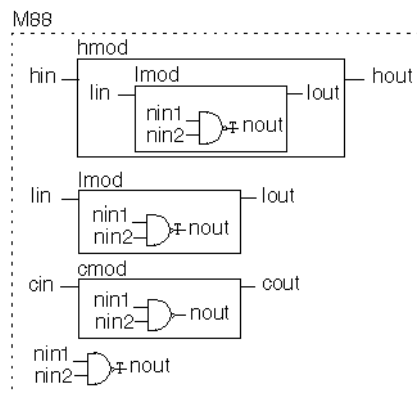
```
$ssi_iddq( "model SA term_OUT prims" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lmod.nout
tbench.M88.lmod.nout
tbench.M88.nout
```

The following figure shows the resulting locations of seeds using this fault model.

Figure 186 Seed Locations: Primitive Output Stuck-At Fault Model



seed_inside_cells

The `seed_inside_cells` option enables seeding of faults inside cells. Thus, the following two lines seed faults on the output terminal of every primitive inside `tbench.M88?`, including those inside cells:

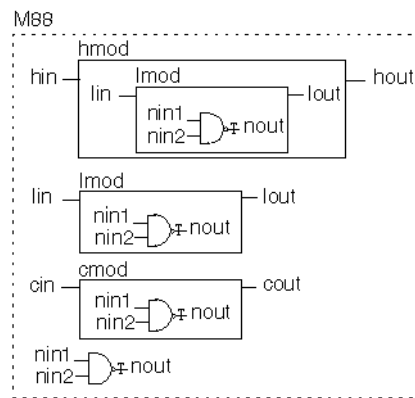
```
$ssi_iddq( "model SA term_OUT prims seed_inside_cells" );
$ssi_iddq( "seed SA tbench.M88" );
```

As a result, stuck-at faults are seeded on the following nets:

```
tbench.M88.hmod.lmod.nout
tbench.M88.lmod.nout
tbench.M88.cmod.nout
tbench.M88.nout
```

The following figure shows the resulting locations of seeds using this fault model.

Figure 187 Primitive Output Seeding for `seed_inside_cells`



Bridging Faults

The `model B` command determines where the `seed B` command seeds bridge faults. The following table lists and describes the bridge placement options available for the `model B` command.

Table 20 Options for Bridging Fault Models

Bridge Placement Options	
<code>cell_ports</code>	Enables bridging faults between adjacent ports of cells and between each input and output port of cells (if the cell has no more than two output ports)

Table 20 Options for Bridging Fault Models (Continued)

<code>fet_terms</code>	Enables bridging faults between all pairs of terminals of FET switches
<code>gate_IN2I N</code>	Enables bridging faults between adjacent input terminals of non-FET primitives (including UDPs)
<code>gate_IN2O UT</code>	Enables bridging faults between all pairs of input and output terminals of non-FET primitives (including UDPs).
<code>vector</code>	Enables bridging faults between adjacent bits of expanded vectors

Seed Inside Cells Option

<code>seed_insi de_cells</code>	Enables fault seeding inside cells
-------------------------------------	------------------------------------

cell_ports

The `cell_ports` option seeds bridging faults between adjacent ports of each cell, and also between the cell inputs and outputs if the cell has no more than two output ports. Ports are considered adjacent when they appear next to each other in the module's port list definition. For example, consider the following module definition:

```
\celldefine
module bsel( out, in1, in2, in3 );
output out;
input in1, in2, in3;
endmodule
\endcelldefine
```

The following port pairs are considered adjacent:

```
out, in1
in1, in2
in2, in3
```

As a result, the `cell_ports` option seeds five bridging faults: three between pairs of adjacent ports and two more between the inputs and outputs. This is the bridging fault list:

```
out, in1
in1, in2
```

```
in2, in3
```

```
out, in2
```

```
out, in3
```

fet_terms

The `fet_terms` option seeds bridging faults between all pairs of terminals of each FET switch. This results in four bridging faults for a CMOS switch or three bridging faults for any other type of switch.

For example, consider this primitive:

```
nmos UF44( out, data, ctl );
```

The `term_fets` option seeds these three bridging faults:

```
out, data
```

```
out, ctl
```

```
data, ctl
```

gate_IN2IN

The `gate_IN2IN` option seeds bridging faults between adjacent input terminals of gates. Terminals are considered adjacent when they appear next to each other in the primitive's terminal list.

For example, consider the following primitive:

```
and U2033( out, in1, in2, in3 );
```

The `gate_IN2IN` option seeds the following two bridging faults:

```
in1, in2
```

```
in2, in3
```

gate_IN2OUT

The `gate_IN2OUT` option is like the `gate_IN2IN` option, except that it seeds bridging faults between inputs and outputs. For the previous example, the `gate_IN2OUT` option seeds the following three bridging faults:

```
out, in1
```

```
out, in2
```

```
out, in3
```

vector

The `vector` option seeds bridging faults between adjacent bits of a vector. Two bits are considered adjacent when they have an index within one unit of each other.

For example, consider the following vector:

```
wire [3:0] dvec;
```

The `vector` option seeds the following three bridging faults:

```
dvec[3], dvec[2]
dvec[2], dvec[1]
dvec[1], dvec[0]
```

seed_inside_cells

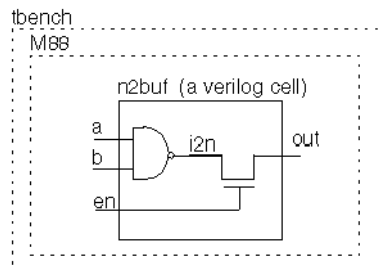
The `seed_inside_cells` option enables seeding of faults inside cells.

Assume that you have a circuit with a module `tbench.M88` that contains an instance of the following cell:

```
`celldefine
module n2buf( a, b, en, out);
input a, b, en;
output out;
nmos( out, n2out, en );
nand( a2out, a, b );
endmodule
`endcelldefine
```

The following figure shows a circuit diagram for this cell.

Figure 188 Example Circuit for Bridging Faults



The following two lines seed bridging faults between cell ports and between FET-switch terminal pairs inside `tbench.M88`?:

```
$ssi_iddq( "model B cell_ports fet_terms" );  
$ssi_iddq( "seed B tbench.M88" );
```

These commands seed five bridging faults between the ports of `n2buf`?:

```
a, b  
b, en  
a, out  
b, out  
en, out
```

By default, no faults are seeded inside of cells. Therefore, the internal net `i2n` is not considered for fault seeding. To include this internal node, use the `seed_inside_cells` option:

```
$ssi_iddq( "model B cell_ports fet_terms  
seed_inside_cells" );  
$ssi_iddq( "seed B tbench.M88" );
```

In this case, the following additional bridging faults are seeded:

```
i2n, en  
i2n, out
```

PowerFault Strobe Selection

After you run a Verilog/PowerFault simulation, you can use the PowerFault strobe selection tool, `IDDQPro`, to select a set of strobe times to obtain maximum fault coverage. `IDDQPro` uses the information in the `IDDQ` database produced by the Verilog/PowerFault simulation.

The following sections describe PowerFault strobe selection:

- [Overview of IDDQPro](#)
- [Invoking IDDQPro](#)
- [Interactive Strobe Selection](#)
- [Strobe Selection Tutorial](#)

- [Understanding the Strobe Report](#)
- [Fault Report Formats](#)
- [Verifault Interface](#)
- [Iterative Simulation](#)

Overview of IDDQPro

IDDQPro is a strobe selection tool that operates on the IDDQ database produced by a Verilog/PowerFault simulation. IDDQPro selects a set of strobe times to maximize fault coverage for a given number of strobuses.

When you run a Verilog/PowerFault simulation, the `output` command in the PowerFault Verilog module specifies the name of the IDDQ database. The database contains information on seeded faults and the faults detected at each qualified strobe time.

When you invoke IDDQPro, you specify the database name and the number of strobuses you want to use. IDDQPro analyzes the database and finds a set of strobuses that maximizes the number of faults detected.

You can run IDDQPro in batch mode or interactive mode.

- In batch mode, IDDQPro selects a set of strobuses and reports the results.
- In interactive mode, IDDQPro displays a command prompt.

You can interactively enter commands to select strobuses, display reports, and traverse the hierarchy of the design.

IDDQPro produces two report files: a strobe report (`iddq.srpt`) and a fault report (`iddq.frpt`).

- The strobe report shows the time value and cumulative fault coverage of each selected strobe point.
- The fault report lists the status of each seeded fault, either detected or undetected, for the complete set of selected strobuses.

Each report file starts with a header that summarizes the report contents and tells you how to interpret the information provided.

After you use IDDQPro to select a set of strobuses, it is a good idea to copy and save the strobe report file so that you will not need to generate it again. The strobe report can take a long time to generate. It is not as important to save the fault report file because you can quickly regenerate it, as long as you have the strobe report file.

Invoking IDDQPro

You invoke IDDQPro at an operating system prompt. The following sections describe the process for invoke IDDQPro:

- [ipro Command Syntax](#)
- [Strobe Selection Options](#)
- [Report Configuration Options](#)
- [Log File and Interactive Options](#)

ipro Command Syntax

The full Backus-Naur form (BNF) description of the command syntax for IDDQPro is as follows:

```
ipro options* iddq-database-name+
options ::=
-strb_lim max-strobes |
-cov_lim percent-cov |
-ign_ucov |
-strb_set file-name |
-strb_unset file-name |
-strb_all |
-prnt_fmt (tmax|verifault|zycad) |
-prnt_nofrpt |
-prnt_full |
-prnt_times |
-path_sep (./) |
-log file-name |
-inter
```

The command consists of the keyword `ipro?`, followed by zero or more options, followed by one or more IDDQ database names. A typical command specifies a limit on the number of strobes with the `-strb_lim` option and specifies a single IDDQ database. For example:

```
ipro -strb_lim 5 iddq
```

This command invokes IDDQPro, specifies a maximum limit of five strobes, and specifies `iddq` as the name of the IDDQ database.

Here are some more examples of IDDQPro invocation commands:

```
ipro -strb_lim 5 iddqdb1 iddqdb2
```

```
ipro -strb_lim 8 /net/simserver/CCD/iddq
```

```
ipro -strb_lim 10 iddq
```

```
ipro -strb_lim 10 -cov_lim 0.95 iddq
```

```
ipro -strb_lim 10 -cov_lim 0.95 -prnt_fmt verifault iddq
```

Strobe Selection Options

You can control strobe selection by using the following `ipro` command options:

```
-strb_lim max-strobes
```

```
-cov_lim percent-cov
```

```
-strb_set file-name
```

```
-strb_unset file-name
```

```
-strb_all
```

If you do not use any options, IDDQPro selects strobes until it either uses up all the possible strobe points or reaches the absolute maximum coverage possible.

-strb_lim

The `-strb_lim` option specifies the maximum number of strobe points to select. The practical maximum number depends on the test equipment being used. Typically, only five to ten IDDQ strobes are allowed per test. IDDQPro attempts to obtain the best possible coverage, given the specified maximum number of strobes.

For example, to limit the number of selected strobes to ten, you would use a command such as the following:

```
ipro -strb_lim 10 iddq
```

-cov_lim

The `-cov_lim` option specifies the target fault coverage percentage. Strobe selection stops when fault coverage reaches or exceeds this limit. Coverage is expressed as a decimal fraction between 0.00 and 1.00. For example, to choose as many strobes as necessary to reach 80 percent fault coverage, you would use a command such as the following:

```
ipro -cov_lim 0.80 iddq
```

-strb_set

The `-strb_set` option causes IDDQPro to select the strobe times listed in a file. IDDQPro evaluates the effectiveness of the strobes listed in the file. If you have a set of strobe times you think are good for IDDQ testing, put them into a file, with one time value per line.

For example, to force the selection of strobes at times 29900 and 39900, put those two times into a file named `stimes` like this,

```
29900
```

```
39900
```

and then use a command such as the following:

```
ipro -strb_set stimes -strb_lim 8 /cad/sim/M88/iddq
```

As a result of this command, IDDQPro selects the two specified strobe times, plus six other strobe times that it selects with its regular coverage-maximizing algorithm. The usual strobe report, `iddq.srpt?`, includes all eight strobes. In addition, IDDQPro generates a separate strobe evaluation report called `iddq.seval?`, which shows the coverage obtained by just the two file-specified strobe times.

If you are using multiple testbenches, specify the testbench number before each strobe time. Testbench numbering starts at 1. For example, to select the strobes at times 299 and 1899 in the first testbench and time 399 in the second testbench, enter the following lines in the strobe time file:

```
tb=1 299
```

```
tb=1 1899
```

```
tb=2 399
```

To regenerate a fault report from a saved strobe report, use the `-strb_set` option and specify the name of the strobe report file. For example:

```
ipro -strb_set iddq.srpt -strb_lim 5 iddq
```

-strb_unset

The `-strb_unset` option prevents IDDQPro from selecting the strobe times listed in a file. If you have a set of strobe times that you do not want IDDQPro to use, put them into a file, with one value per line. For example, if you want to prevent the strobes at times 59900 and 89900 from being selected, put those two times into a file named `bad_stimes` and then use a command such as the following:

```
ipro -strb_unset bad_stimes -strb_lim 8 /cad/sim/M88/iddq
```

As a result of this command, IDDQPro selects eight strobe times using its regular coverage-maximizing algorithm, but excluding the strobes at times 59900 and 89900. If you are using multiple testbenches, specify the testbench number before each strobe time as explained previously for the `-strb_set` option.

-strb_all

The `-strb_all` option causes IDDQPro to select all qualified strobe points, starting with the first strobe time, instead of using the coverage-maximizing algorithm. The strobe report and fault report show the coverage obtained by making an IDDQ measurement at every qualified strobe point.

Although it is usually impractical to make so many measurements, the `-strb_all` option is useful because it determines the maximum possible coverage that can be obtained from your testbench or testbenches. In addition, the resulting fault report identifies nets that never get toggled; they are reported as undetected.

Report Configuration Options

You can control the generation of the fault report by IDDQPro by using the following `ipro` command options:

```
-prnt_fmt (tmax|verifault|zycad)
-prnt_nofrpt
-prnt_full
-prnt_times
-path_sep
-ign_ucov
```

-prnt_fmt

The `-prnt_fmt` option specifies the format of the fault report produced by IDDQPro. The format choices are `tmax?`, `verifault?`, and `zycad?`. The default format is `tmax?`.

In the default format, the faults are reported as shown in the following example:

```
sa0 NO .testbench.fadder.co
```

Chapter 34: PowerFault

PowerFault Strobe Selection

```

sal DS .testbench.fadder.co
sa0 DS .testbench.fadder.sum
sal NO .testbench.fadder.sum
...

```

To generate a fault report in Zycad `.fog` format, use a command similar to the following:

```
ipro -prnt_fmt zycad -strb_lim 5 iddq
```

In the Zycad configuration, faults are reported as shown in the following example:

```

@testbench.fadder
co 0 n U
co 1 n D
sum 0 n D
sum 1 n U
...

```

To generate a fault report in Verifault format, use a command similar to the following:

```
ipro -prnt_fmt verifault -strb_lim 5 iddq
```

In the Verifault configuration, faults are reported as shown in the following example:

```

fault net sa0 testbench.fadder.co 'status=undetected';
fault net sal testbench.fadder.co 'status=detected';
fault net sa0 testbench.fadder.sum 'status=detected';
fault net sal testbench.fadder.sum 'status=undetected';
...

```

-prnt_nofrpt

Use the `-prnt_nofrpt` option to suppress generation of the fault report. Otherwise, by default, IDDQPro generates the `iddq.frpt` fault report every time the program is run in batch mode.

-prnt_full, -prnt_times, and -path_sep

The `-prnt_full?`, `-prnt_times?`, and `-path_sep` options control the generation of Zycad-format fault reports. These options do not affect on Verifault-format fault reports.

The `-prnt_full` option controls the reporting of hierarchical paths. By default, faults are divided into groups, with the cell name shown at the beginning of each group. Only the leaf-level net name is shown in each line.

Here is an example taken from a report in the default Zycad reporting format:

```
@tbench.M88  
  
sio24 0 n D  
  
sio24 1 n D  
  
sio25 0 n D  
  
sio25 1 n U
```

If you use the `-prnt_full` option, the full hierarchical paths are reported in each line, as shown in the following example:

```
tbench.M88.sio24 0 n D  
  
tbench.M88.sio24 1 n D  
  
tbench.M88.sio25 0 n D  
  
tbench.M88.sio25 1 n U
```

The `-prnt_times` option causes the fault report to include the simulation time at which each fault was first detected. For example, with the `-prnt_times` options, the same faults as described in the preceding example are reported as follows:

```
tbench.M88.sio24 0 n 129900 D  
  
tbench.M88.sio24 1 n 39900 D  
  
tbench.M88.sio25 0 n 455990 D  
  
tbench.M88.sio25 1 n U
```

The `-path_sep` option specifies the character for separating the components of a hierarchical path. The default character is a period (.) so that path names are compatible with Verilog. If you want Zycad-style path names, select the forward slash character (/) instead, as in the following example:

```
ipro -prnt_fmt zycad -prnt_full -path_sep / -strb_lim 5 iddq
```

Then the same faults described previously are reported as follows:

```
/tbench/M88/sio24 0 n D  
  
/tbench/M88/sio24 1 n D  
  
/tbench/M88/sio25 0 n D
```



```
/tbench/M88/sio25 1 n U
```

-ign_uncov

The `-ign_uncov` option prevents IDDQPro from using the “potential” status in the fault report. All faults are still listed, but faults that would normally be reported as potential are instead reported as undetected. This option also prevents IDDQPro from generating coverage statistics for uninitialized nodes in the strobe report. For information on uninitialized nodes, see [Faults Detected at Uninitialized Nodes](#).

Log File and Interactive Options

The `-log` option lets you specify the name of the IDDQPro log file. The log file contains a copy of all messages displayed during the IDDQPro session. By default, the log file name is `iddq.log`.

By default, IDDQPro runs in batch mode. This means that IDDQPro reads the IDDQ database, selects the strobe times, produces the strobe report and fault report files, and returns you to the operating system prompt.

The `-inter` option lets you run IDDQPro in interactive mode. In this mode, IDDQPro displays a prompt. You interactively select strobcs manually or automatically, request the reports that you want to see, and optionally browse through the hierarchy of the design.

The IDDQPro interactive commands are described in the next section, [Interactive Strobe Selection](#).

Interactive Strobe Selection

To use IDDQPro in interactive mode, invoke it with the `-inter` option, as in the following example:

```
% ipro -inter iddq
```

When IDDQPro is started in interactive mode, it loads the results from the Verilog simulation and waits for you to enter a command. No strobcs are selected and no reports are generated until you enter the commands to request these actions.

At the interactive command prompt, you can enter commands to select strobcs, display reports, and traverse the hierarchy of the design. When you change to a lower-level module in the design hierarchy, the reports that you generate apply only to the current scope of the design.

The following table lists and briefly describes the interactive commands. The following sections provide detailed descriptions of these commands.

Table 21 *IDDQPro Interactive Commands*

Command	Description
<code>cd</code>	Changes the interactive scope to lower-level instance
<code>desel</code>	Prevents selection of specified strobe times
<code>exec</code>	Executes a list of interactive commands in a file
<code>help</code>	Displays a summary description of all commands or one command
<code>ls</code>	Displays a list of lower-level instances at the current level
<code>prc</code>	Prints a fault coverage report
<code>prf</code>	Prints a list of all seeded faults and their detection status
<code>prs</code>	Prints a list of all qualified strobcs and their status
<code>quit</code>	Terminates IDDQPro
<code>reset</code>	Cancelcs all strobc selections and detected faults
<code>sela</code>	Selects strobcs automatically using the coverage-maximizing algorithm
<code>selall</code>	Selects all strobcs
<code>selm</code>	Selects one or more strobcs manually, specified by time value

To run an interactive IDDQPro session, you typically use the following steps:

1. Select the strobcs automatically or manually, or select all strobcs (`?sela?`, `selm?`, or `selall?`).
2. If you want to analyze just a submodule of the design, change to hierarchical scope for that module (`?ls?`, `cd?`).
3. Print a strobc report, coverage report, and fault report (`?prs?`, `prc?`, `prf?`).
4. Repeat steps 1 through 3 to examine different sets of strobcs or different parts of the design. Use the `reset` command to select an entirely new set of strobcs.
5. Exit from IDDQPro (`?quit?`).

By default, the output of all interactive commands is sent to the terminal (stdout). The printing commands, especially `prf` and `prs?`, can produce very long reports. If you want to redirect the output of one of these commands to a file, use the `-out` option.

cd

```
cd module-instance
```

The `cd` command changes the current scope of the analysis to a specified module instance. You can use this command to produce different reports for different parts of the design. For example, to print separate fault reports for modules `top.M88.alu` and `top.M88.io?`, enter the following commands:

```
cd top.M88.alu  
prf -out alu.frpt  
cd top.M88.io  
prf -out io.frpt
```

To get a listing of modules in the current hierarchical scope, use the `ls` command. To move up to the next higher level of hierarchy, use the following command:

```
cd ..
```

desel

```
desel strobe-times* selm-options*  
strobe-times ::= [tb=testbench-number] simulation-time  
selm-options ::= -in file-name | -out file-name
```

The `desel` (deselect) command prevents IDDQPro from selecting one or more specified strobe times when you later use the `sela` or `selall` command. The strobe times can be explicitly listed in the command line or read from an input file.

If the `desel` command specifies strobos that are currently selected, they are first deselected. The specified strobos are all made unselectable by subsequent invocations of the `sela` or `selall` command. However, they can still be selected manually with the `selm` command.

For example, the following command deselects the two strobos at 59900 and 89900 and prevents them from being selected automatically by a subsequent `sela` or `selall` command:

```
desel 59900 89900
```

If you are using multiple testbenches, you can deselect strobos from different testbenches. For example, the following command manually deselects strobos at time 799 and 1299 from testbench 1 and a strobe at time 399 from testbench 2:

```
desel tb=1 799 tb=1 1299 tb=2 399
```

exec

```
exec file-name
```

The `exec` command executes a list of interactive commands stored in a file.

help

```
help [command-name]
```

The `help` command displays help on a specified interactive command. If you do not specify a command name, the `help` command provides help on all interactive commands.

ls

```
ls
```

The `ls` command lists the lower-level instances contained in the current scope of the design. To change the hierarchical scope, use the `cd` command.

prc

```
prc [-out file-name]
```

The `prc` (print coverage) command displays a report on the fault coverage of instances in the current hierarchical scope. This report shows which blocks in your design have high coverage and which have low coverage.

This command reports statistics on seeded faults. Faults that were not seeded during the Verilog/PowerFault simulation (such as faults detected by a previous run) are not included in the fault coverage statistics.

prf

```
prf [-fmt (tmax|verifault)] [-full] [-times]  
[-out file-name]
```

The `prf` (print faults) command displays a report on the faults in the instances contained in the current hierarchical scope.

The output of this command is just like the default fault report file produced in batch mode, `iddq.fprt`, except that the `prf` command lists the status of faults beneath the current hierarchical scope, rather than all faults in the whole design.

The `prf` command complements the `prc` command. The `prc` command shows which blocks have low coverage, and the `prf` command shows which faults are causing the low coverage.

prs

```
prs [-out file-name]
```

The `prs` (print strobe) command displays the time value for every qualified IDDQ strobe. For each selected strobe, the number of incremental (additional new) faults detected by the strobe is also reported.

quit

```
quit
```

The `quit` command terminates IDDQPro.

reset

```
reset
```

The `reset` command clears the set of selected strobess and detected faults, allowing you to start over.

sela

```
sela sela-options*
```

```
sela-options ::=
```

```
-cov_lim percent_cov |
```

```
-strb_lim max_strobes |
```

```
-out file-name
```

The `sela` (select automatic) command automatically selects strobess using a coverage-maximizing algorithm. This is the same selection algorithm IDDQPro uses in batch mode.

The `-cov_lim` and `-strb_lim` options work exactly like the command-line options described in [Strobe Selection Options](#).

The `-out` option redirects the output of the command to a specified file.

selm

```
selm strobe-times* selm-options*
```

```
strobe-times ::= [tb=testbench-number] simulation-time
```

```
selm-options ::= -in file-name | -out file-name
```

The `selm` (select manual) command lets you manually select strobess by specifying the strobe times. You can explicitly list the strobe times in the command line or read them from an input file using the `-in` option.

After you run this command, IDDQPro analyzes the strobe set and reports the results. To redirect the output to a file, use the `-out` option.

The `selm` and `sela` commands work together in an incremental fashion. Each time you use one of these commands, it adds the newly selected strobes to the list of previously selected strobes. This continues until the maximum possible coverage is achieved, after which no more strobes can be selected. If the IDDQPro analysis determines that a manually selected strobe fails to detect any additional faults, the selection is automatically canceled.

For example, consider the following two commands:

```
selm 29900 39900  
sela -strb_lim 6
```

The first command manually selects the two strobes at 29900 and 39900. The second command automatically selects six more strobes that complement the first two strobes and maximize the fault coverage.

To clear all strobe selections and start over, use the `reset` command.

If you are using multiple testbenches, you can select strobes from different testbenches. For example, the following command manually selects strobes at times 799 and 1299 in testbench 1 and the strobe at time 399 in testbench 2:

```
selm tb=1 799 tb=1 1299 tb=2 399
```

selall

```
selall [-out file-name]
```

The `selall` (select all) command automatically selects every qualified strobe, starting with the first strobe time and continuing until the maximum possible coverage is achieved or all qualified strobes are selected.

Although it is usually impractical to make so many measurements, the `-selall` command is useful because it determines the maximum possible coverage that can be obtained from your testbench or testbenches. If you use the `prf` command after the `selall` command, the resulting fault report identifies nets that never get toggled; they are reported as undetected.

Strobe Selection Tutorial

After you install the Synopsys IDDQ option to TestMAX ATPG, you can do the Strobe Selection Tutorial to test the installation and to get an introduction to PowerFault strobe selection.

This tutorial is intended to be a brief demonstration, not a comprehensive training session.

The following sections guide you through the Strobe Selection Tutorial:

- [Simulation and Strobe Selection](#)
- [Interactive Strobe Selection](#)

Simulation and Strobe Selection

The `$IDDQ_HOME/samples` directory contains some examples of designs and scripts to demonstrate PowerFault capabilities. One example is a simple one-bit full adder. In the following set of tutorial procedures, you will run a script that simulates the testbench and selects a set of IDDQ strobe times in the testbench:

- [Examine the Verilog File](#)
- [Run the doit Script](#)
- [Examine the Output Files](#)

Examine the Verilog File

The following steps show you how to examine the Verilog design file.

1. Change to the directory `$IDDQ_HOME/samples/fadder?`.
2. Look for two files in the directory: the `doit` script and the `fadder.v` Verilog file.
3. Using any text editor, view the contents of the `fadder.v` file.

The `fadder.v` Verilog file contains three modules: `testbench?`, `iddqtest?`, and `fadder?`.

The `testbench` module is the testbench for the full adder. It tests every possible input pattern, from `b000` through `b111`, and prints out the port values at one time unit before the end of each cycle.

The `iddqtest` module invokes the PLI tasks for IDDQ analysis. It contains the following `$ssi_iddq` commands:

```
$ssi_iddq( "dut testbench.fadder" );  
  
$ssi_iddq( "seed SA testbench.fadder" );  
  
// strobe 1 time unit before end of cycle  
  
forever begin  
  
# (testbench.CYCLE - 1)  
  
$ssi_iddq( "strobe_try" );  
  
# 1;  
  
end
```

The first command defines the device under test to be `testbench.fadder?`. The second one seeds stuck-at faults throughout the entire device. The third one performs IDDQ strobe evaluation one time unit before the end of each cycle.

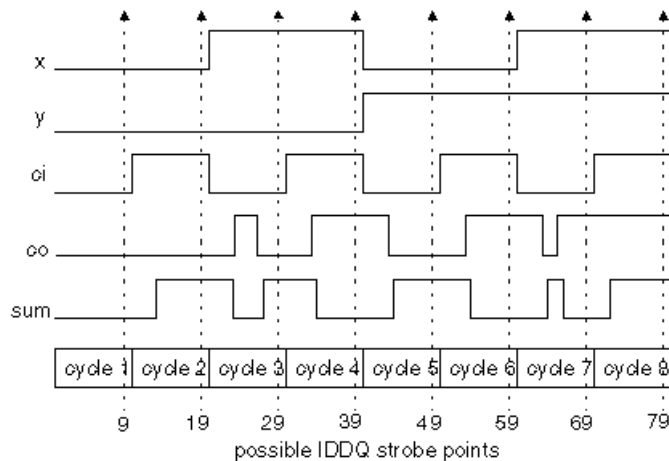
The `fadder` module is a gate-level description of the device under test, a single-bit full adder implemented with NOR gates. Each gate has a unit delay. Given two input bits (`x` and `y`) and a carry-in bit (`ci`), the full adder computes the sum bit and the carry-out (`co`) bit. The model implements the following Boolean equations:

$$co = (x \& y) \mid (x \& ci) \mid (y \& ci)$$

$$sum = x \wedge y \wedge ci$$

The following figure shows the stimulus, response, and IDDQ strobe points for the full adder simulation.

Figure 189 Full Adder Simulation Strobe Points



Run the doit Script

The following steps show you how to run the `doit` script, which runs the Verilog/Powerfault simulation and IDDQ Profiler.

1. Using any text editor, view the contents of the `doit` (do it) file. This is a script that creates a directory for the simulator output, invokes the Verilog simulator (with IDDQ PLI tasks), and runs the IDDQ Profiler to select the strobe times.
2. If necessary, edit the file to work with your system configuration. For example, if your simulator is invoked by a command other than `vcs` or `Verilog?`, modify the line that invokes the simulator.
3. Run the script.

The script runs the Verilog simulation, which produces the following results:

```
time co sum {x,y,ci}
9 0 0 000
19 0 1 001
29 0 1 010
39 1 0 011
49 0 1 100
59 1 0 101
69 1 0 110
79 1 1 111
```

The `$ssi_iddq` tasks produce the following summary report:

```
IDDQ-Test
Strobes (qualified/tested) = 8/8
Faults seeded (stuck-ats/bridges) = 32/0
Created IDDQ database: iddq
```

This report tells you that eight strobes were tested, and all eight were found to be quiescent.

The script then invokes the IDDQ Profiler, which selects some of the eight quiescent strobes. It generates two files: a strobe report named `iddq.srpt` and a fault report named `iddq.frpt`. The script then tells you the path to the output files.

```
Loading seeds
Beginning strobe selection
...
Strobe selection complete
Strobe report is printed to iddq.srpt
Fault report is printed to iddq.frpt
```

Examine the Output Files

The following steps show you how to examine the report files.

1. Go to the directory containing the `fadder` output files. Find the subdirectory called `iddq?`, which contains the IDDQ database generated by the `$ssi_iddq` PLI tasks, and the two IDDQ Profiler output files, `iddq.srpt` and `iddq.frpt?`.
2. Examine the contents of the strobe report file, `iddq.srpt?`. You should see the following report:

```
# Date: day/date/time # Reached requested fault coverage. # Selected
3 strobos out of 8 qualified. # Fault Coverage (detected/seeded)
= 100.0% (32/32) # Timeunits: 1.0ns # # Strobe: Time Cycle Cum-Cov
Cum-Detects Inc-Detects 9 1 50.0% 16 16 39 4 84.4% 27 11 49 5 100.0%
32 5
```

The report shows the requested level of fault coverage, 100 percent, was achieved by three strobos. A table shows the time values and cycle numbers of the selected strobos, the cumulative fault coverage achieved by each successive strobe, the cumulative number of faults detected with each successive strobe, and the incremental (additional) faults detected with each successive strobe.

3. Examine the contents of the fault report file, `iddq.frpt?`. The report shows the list of faults and the test result for each fault:

```
sa0 DS .testbench.fadder.co sa1 DS .testbench.fadder.co sa0
DS .testbench.fadder.sum sa1 DS .testbench.fadder.sum sa0
DS .testbench.fadder.x sa1 DS .testbench.fadder.x sa0
DS .testbench.fadder.y sa1 DS .testbench.fadder.y sa0
DS .testbench.fadder.ci sa1 DS .testbench.fadder.ci sa0
DS .testbench.fadder.u12_out sa1 DS .testbench.fadder.u12_out sa0
DS .testbench.fadder.u10_out sa1
DS .testbench.fadder.u10_out ... sa0 DS .testbench.fadder._x sa1
DS .testbench.fadder._x sa0 DS .testbench.fadder._y sa1
DS .testbench.fadder._y
```

The test result for each fault is either DS (detected by simulation) or NO (not observed). In this case, all faults were detected. Each fault is identified by fault type (sa0 = stuck-at-0, sa1 = stuck-at-1) and the hierarchical net name.

Interactive Strobe Selection

In the previous steps of this tutorial, you used the IDDQ Profiler in batch mode, which is the default operating mode. In this mode, the IDDQ Profiler selects a set of strobos and attempts to obtain the requested fault coverage with the fewest possible strobos.

You can also use the IDDQ Profiler in interactive mode to perform strobe and fault coverage analysis. In a typical interactive session, you select a set of strobos, print a strobe report and a fault coverage report for that set of strobos, and then repeat this process for different sets of strobos. You can examine the status of all faults or just the faults within a specified hierarchical scope.

The following sections guide you through the interactive strobe selection portion of this tutorial:

- [Select Strobes Automatically](#)
- [Select All Strobes](#)
- [Select Strobes Manually](#)
- [Cumulative Fault Selection](#)

Select Strobes Automatically

The following steps show you how to use the IDDQ Profiler to automatically select the strobes in a single step:

1. In the directory containing the fadder output files, execute the following command:

```
% ipro -inter iddq
```

The `ipro -inter` command invokes the IDDQ Profiler in interactive mode, and the `iddq` argument specifies the name of the IDDQ database to use for the interactive session.

2. At the IDDQ Profiler prompt (`>`), enter the “select automatic” command:

```
> sela
```

This command invokes the same strobe selection algorithm used in batch mode. The IDDQ Profiler responds as follows:

```
... # Reached requested fault coverage. # Selected 3 strobes out of  
8 qualified. # Fault Coverage (detected/seeded) = 100.0% (32/32) #  
Timeunits: 1.0ns # # Strobe: Time Cycle Cum-Cov Cum-Detects  
Inc-Detects 9 1 50.0% 16 16 39 4 84.4% 27 11 49 5 100.0% 32 5
```

The list of selected strobes is the same as in batch mode.

3. Enter the “print coverage” command:

```
> prc
```

The IDDQ Profiler responds as follows:

```
Fault coverage for top modules
```

```
Instance NumDet NumFaults %Coverage (stuck-at bridge)  
testbench 32 32 100.0% (32/32 0/0)
```

For the current set of selected strobes, 32 out of 32 faults are detected, and coverage is 100 percent.

4. Enter the “print faults” command:

```
> prf
```

The IDDQ Profiler produces the same fault report that you saw earlier in the `iddq.frpt` file:

```
sa0 DS .testbench.fadder.co sa1 DS .testbench.fadder.co ... sa0  
DS .testbench.fadder._y sa1 DS .testbench.fadder._y
```

5. Enter the “reset” command:

```
> reset
```

This command clears the set of selected strobes and detected faults.

Select All Strobes

The following steps show you how to manually select all possible strobes.

1. Enter the “select all” command:

```
> selall
```

The IDDQ Profiler responds as follows:

```
# Selected all qualified strobes. # Selected 5 strobes out of 8  
qualified. # Fault Coverage (detected/seeded) = 100.0% (32/32) #  
Timeunits: 1.0ns # # Strobe: Time Cycle Cum-Cov Cum-Detects  
Inc-Detects 9 1 50.0% 16 16 19 2 62.5% 20 4 29 3 84.4% 27 7 39 4  
90.6% 29 2 49 5 100.0% 32 3
```

All qualified strobes were selected in sequence, starting with the first strobe at `time=9`, until the target coverage of 100 percent was achieved. Five strobes were required, rather than the three selected by the `sel_a` (select automatic) command.

2. Reset the strobe selection and detected faults:

```
> reset
```

Select Strobes Manually

The following steps show you how to select strobes manually.

1. Enter the following “select manual” command to manually select a single strobe at `time=39`:

```
> selm 39
```

The IDDQ Profiler responds as follows:

```
# Selected 1 strobes out of 8 qualified. # Fault Coverage  
(detected/seeded) = 50.0% (16/32) # Timeunits: 1.0ns # # Strobe: Time  
Cycle Cum-Cov Cum-Detects Inc-Detects 39 4 50.0% 16 16
```

This single strobe detected 16 faults, providing coverage of 50 percent.

2. To find out which faults have not yet been detected, enter the “print faults” command:

```
> prf
```

You should see the following response:

```
sa0 DS .testbench.fadder.co sa1 NO .testbench.fadder.co sa0
NO .testbench.fadder.sum sa1 DS .testbench.fadder.sum ... sa0
DS .testbench.fadder._x sa1 NO .testbench.fadder._x sa0
NO .testbench.fadder._y vsa1 DS .testbench.fadder._y
```

The second column shows `DS` for “detected by simulation” or `NO` for “not observed.”

3. Enter the following command to see a list of modules:

```
> ls
```

The IDDQ Profiler responds as follows:

```
ls testbench
```

This simple model has only one level of hierarchy. In a multilevel hierarchical model, you can change the scope of the design view by using the `ls?`, `cd module_name?`, and `cd ..` commands. When you use the `prf` command, only the faults residing within the current scope (in the current module and below) are reported. Similarly, a coverage report generated by the `prc` command applies only to the current scope.

4. Enter the following command to manually select another strobe at time=49:

```
> selm 49
```

The IDDQ Profiler responds as follows:

```
# Selected 2 strobcs out of 8 qualified. # Fault Coverage
(detected/seeded) = 87.5% (28/32) # Timeunits: 1.0ns # # Strobe: Time
Cycle Cum-Cov Cum-Detects Inc-Detects 39 4 50.0% 16 16 49 5 87.5% 28
12
```

The IDDQ Profiler adds each successive strobe selection to the previous selection set. The report shows the cumulative coverage and cumulative defects detected by each successive strobe.

5. Look at the fault list:

```
> prf
```

6. Reset the strobe selection and detected faults:

```
> reset
```

Cumulative Fault Selection

The following steps show you how to combine manual and automatic selection techniques:

1. Manually select the two strobes at time=19 and time=29:

```
> selm 19 29
```

The IDDQ Profiler responds as follows:

```
# Selected 2 strobes out of 8 qualified. # Fault Coverage  
(detected/seeded) = 78.1% (25/32) # Timeunits: 1.0ns # # Strobe: Time  
Cycle Cum-Cov Cum-Detects Inc-Detects 19 2 50.0% 16 16 29 3 78.1% 25  
9
```

2. Enter the “select automatic” command:

```
> sela
```

The IDDQ Profiler responds as follows:

```
# Reached requested fault coverage. # Selected 4 strobes out of 8  
qualified. # Fault Coverage (detected/seeded) = 100.0% (32/32) #  
Timeunits: 1.0ns # v# Strobe: Time Cycle Cum-Cov Cum-Detects  
Inc-Detects v 19 2 50.0% 16 16 29 3 78.1% 25 9 59 6 96.9% 31 6 69 7  
100.0% 32 1
```

The `sela` command keeps the existing selected strobes and applies the automatic selection algorithm to the remaining undetected faults. In this case, four strobes were required to achieve 100 percent coverage.

3. Reset the strobe selection and detected faults:

```
> reset
```

4. Continue to experiment with the commands you have learned. For help on command syntax, use the `help` command:

```
> help
```

or

```
> help command_name
```

5. When you are done, exit with the `quit` command:

```
> quit
```

Understanding the Strobe Report

A strobe report (iddq.srpt file) is generated when you run IDDQPro in batch mode and each time you select strobcs in interactive mode. The following sections describe a strobe report:

- [Example Strobe Report](#)
- [Fault Coverage Calculation](#)
- [Adding More Strobcs](#)
- [Deleting Low-Coverage Strobcs](#)

Example Strobe Report

A strobe report lists the selected strobcs in time order and shows the following information for each strobe:

- The simulation time
- The simulation cycle number
- The cumulative coverage achieved
- The cumulative number of faults detected
- The incremental (additional new) faults detected

The report gives you an idea of the effectiveness of each strobe. A large jump in coverage indicates a valuable strobe. A very small increase in coverage indicates a strobe with little value.

Here is an example of a strobe report:

```
# IDDQ-Test strobe report
# Date: day date time
# Reached requested fault coverage.
# Selected 6 strobcs out of 988 qualified.
# Fault Coverage (detected/seeded) = 90.3% (23082/25561)
# Timeunits 1.0ns
# Strobe: Time Cycle Cum-Cov Cum-Detects Inc-Detects
19990 2 48.3% 12346 12346
329990 33 69.0% 17637 5291
```

```
2109990 211 74.2% 18966 1329
2129990 213 77.9% 19912 946
2759990 276 85.7% 21906 1994
2809990 281 90.3% 23082 1176
```

Fault Coverage Calculation

The fault coverage statistics in a strobe report include the following types of faults:

- [Faults Detected by Previous Runs](#)
- [Undetected Faults Excluded From Simulation](#)
- [Faults Detected at Uninitialized Nodes](#)

Faults Detected by Previous Runs

For example, the following report indicates that faults were detected by previous runs:

```
# Reached requested fault coverage.
# Selected 8 strobos out of 755 qualified.
# Fault Coverage (detected/seeded) = 90.0% (90/100)
# Faults detected by previous runs = 60
```

In this example, an existing fault list was read into the Verilog simulation with `read_tmax` or a similar command. That fault list had 60 faults that were already detected, either by an external tool such as Verifault or by a previous IDDQPro run. Therefore, the eight selected strobos only detected 30 more faults than the 60 that were already detected.

Undetected Faults Excluded From Simulation

The following report indicates that undetected faults were excluded from simulation:

```
# Reached requested fault coverage.
# Selected 4 strobos out of 2223 qualified.
# Fault Coverage (detected/seeded) = 85.0% (170/200)
# Undetected faults excluded from simulation = 20
```

The fault list read in by `read_tmax` or a similar command had 20 faults that were undetected but excluded. Perhaps the fault list covered the entire chip, but 20 faults were excluded from seeding at the I/O pads. The four selected strobos detected 170 faults and did not detect 30 faults. However, of the 30 undetected faults, only 10 were simulated by IDDQPro.

Faults Detected at Uninitialized Nodes

The following report indicates that faults were detected at uninitialized nodes:

```
# Reached requested fault coverage.  
# Selected 5 strobes out of 2223 qualified.  
# Fault Coverage (detected/seeded) = 92.5% (370/400)  
# Faults detected at un-initialized nodes = 10
```

If an uninitialized node is driven to X (unknown rather than floating) during every selected vector, a strobe detects one stuck-at fault, either stuck-at-0 or stuck-at-1, because the node is driven to either 1 or 0. However, it is not known which type of fault is detected. The report indicates that 370 out of 400 faults were detected. Of the 370 detected faults, 10 have an unknown type, corresponding to the 10 nodes that were never initialized.

Adding More Strobes

After a Verilog/PowerFault simulation, you can use IDDQPro repeatedly to evaluate the effectiveness of different strobe combinations. It is not necessary to rerun the Verilog/PowerFault simulation each time.

You can use the strobes selected from an IDDQPro run as the initial strobe set for subsequent runs. For example, consider the following sequence of commands:

```
ipro -strb_lim 6 /cad/sim/M88/iddq  
mv iddq.srpt stimes  
ipro -strb_set stimes -strb_lim 8 /cad/sim/M88/iddq
```

The first command runs IDDQPro and selects six strobe points. The second command copies the strobe report file to a new file. The third command invokes IDDQPro again, using the strobe report from the first run as the initial strobe set, and selects two additional strobe points. After the second run, the strobe report file (iddq.srpt) contains eight strobe points, consisting of the six original strobes plus two new ones.

Deleting Low-Coverage Strobes

If you identify a strobe that provides very little additional coverage, you can delete it from the strobe report and run IDDQPro again to recalculate the coverage:

1. Run IDDQPro to select an initial set of strobes:

```
ipro -strb_limit 8 iddq
```

2. Save the strobe report to a separate file:

```
mv iddq.srpt stimes
```

3. Edit the new file and delete the strobe that provides the fewest incremental fault detections.
4. Run IDDQPro again, using the edited file for initial strobe selection:

```
ipro -strb_limit 8 -strb_set stimes iddq
```

For best results, delete only one strobe at a time and run IDDQPro each time to recalculate the coverage. Coverage lost by deleting multiple strobes cannot be calculated by simple addition of the incremental coverage because of overlapping coverage.

Fault Report Formats

A fault report (`iddq.frpt` file) is generated when you run IDDQPro in batch mode and each time you use the `prf` command in interactive mode. The fault report lists all the seeded faults and their detection status.

You can choose the fault report format by using the `-prnt_fmt` option when you invoke IDDQPro. The format choices are the TestMAX ATPG, Verifault, and Zycad formats. The default is TestMAX ATPG.

The following sections describe the various fault report formats:

- [TestMAX ATPG Fault Report Format](#)
- [Verifault Fault Report Format](#)
- [Listing Seeded Faults](#)

TestMAX ATPG Fault Report Format

A fault report in TestMAX ATPG format lists one fault descriptor per simulated fault. Each fault descriptor shows the type of fault, the fault status (DS=detected by simulation, NO=not observed), and the full net name (or two net names for a bridging fault).

Here is a section of a fault report in TestMAX ATPG format:

```
sa0 DS tb.fadder.co
sa1 DS tb.fadder.co
sa0 DS tb.fadder.sum
sa1 DS tb.fadder.sum
```

The fault report shows five faults, all of which are detected by the selected strobes. All five faults involve nets in the `tb.fadder` module instance. The first four faults are stuck-at-0 and stuck-at-1 faults for the `co` and `sum` nets. The last fault is a bridge fault between the `x` and `ci` nets.

Verifault Fault Report Format

A fault report in Verifault format lists one fault descriptor per simulated fault. Each fault descriptor begins with the keyword `fault?`, followed by type of the fault, the full name of the net, and the fault status.

Here is a section of a fault report in Verifault format:

```
fault net sa0 tb.fadder.co 'status=detected';  
fault net sa1 tb.fadder.co 'status=detected';  
fault net sa0 tb.fadder.sum 'status=detected';  
fault net sa1 tb.fadder.sum 'status=detected';  
fault bridge wire tb.fadder.x tb.fadder.ci  
'status=detected';
```

The fault report shows five faults, all of which are detected by the selected strobcs. All five faults involve nets in the `tb.fadder` module instance. The first four faults are stuck-at-0 and stuck-at-1 faults for the `co` and `sum` nets. The last fault is a bridge fault between the `x` and `ci` nets.

Listing Seeded Faults

The IDDQ database stores the faults seeded by the Verilog/PowerFault simulation in a compact binary format. Usually, you use IDDQPro to select strobcs, calculate the fault coverage, and print a fault report that lists all the seeded faults along with their detection status. However, there might be times you want a list of the seeded faults without selecting strobcs. For example, if there are no quiet strobe points to select, IDDQPro cannot generate the fault report.

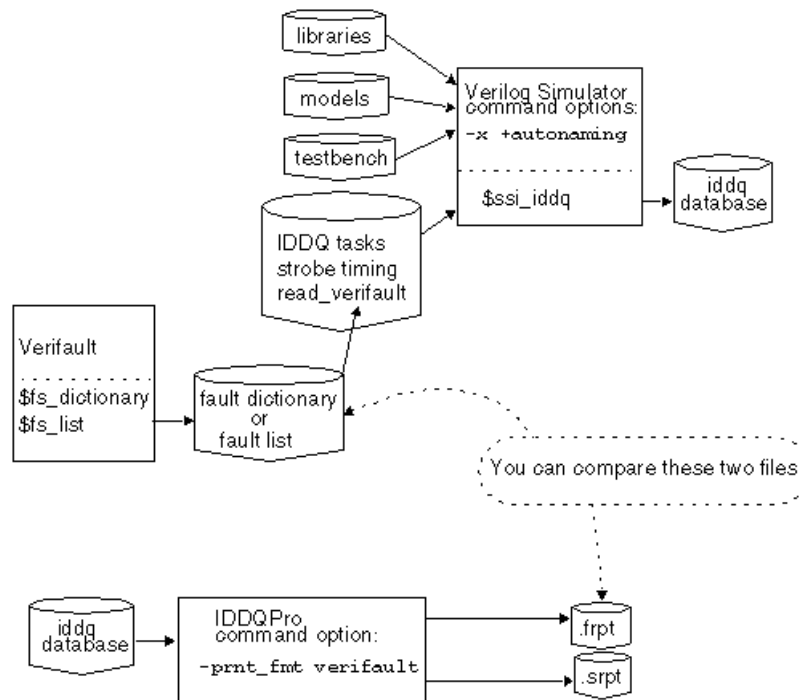
To generate a list of seeded faults under these circumstances, start IDDQPro in interactive mode, and then use the `prf` command to generate a fault report, and redirect the output to a file:

```
ipro -inter iddq-database-name  
prf -out iddq.frpt  
quit
```

Verifault Interface

You can seed a design with faults taken from a Verifault fault list. The following figure shows the data flow for this type of fault seeding.

Figure 190 Data Flow for Verifault Interface



To seed faults from Verifault fault dictionaries and fault lists, use the `read_verifault` command in the Verilog/PowerFault simulation, as described in “[read_verifault](#)” in the [PowerFault PLI Tasks](#) section. By default, PowerFault remembers all the comment lines and unseeded faults in the Verifault file, so that when it produces the final fault report, you can easily compare the report to the original file.

When you use the `read_verifault` command to seed fault descriptors generated by Verifault, and your simulator is Verilog-XL, use the `-x` and `+autonaming` options when you start the simulation:

```
Verilog -x +autonaming iddq.v ...
```

Otherwise, the `read_verifault` command might not be able to find the nets and terminals referenced by your fault descriptors.

By default, the `read_verifault` command seeds both prime and nonprime faults. When you run IDDQPro after the Verilog simulation to select strobos and print fault reports, all fault coverage statistics produced by IDDQPro include nonprime faults. If you want to see statistics for only prime faults, seed only those faults. For example, you can create a fault list with just prime faults and use that list with the `read_verifault` command.

By default, IDDQPro generates fault reports in TestMAX ATPG format. To print a fault report in Verifault format, use the `-prnt_fmt verifault` option:

```
ipro -prnt_fmt verifault -strb_lim 5 iddq-database-name
```

When you use multiple testbenches, the fault report files show only the comment lines from the first testbench. PowerFault does not try to merge the comment lines from the fault list in the second and subsequent testbenches with those in the first testbench.

If you mix fault seeds from other formats, like using the `read_zycad` command to seed faults from a Zycad .fog file, the Zycad faults detected in previous iterations are counted in the coverage statistics but are not printed in the fault report.

Iterative Simulation

You can run PowerFault iteratively, using each successive testbench to reduce the number of undetected faults. This feature is supplied only for backward compatibility with earlier versions of PowerFault. In general, you get better results by using the multiple testbench methodology explained in [Combining Multiple Verilog Simulations](#).

In the following example, you have two testbenches and you want to choose five strobes from each testbench. All of the PowerFault tasks have been put into one file named `ssi.v?`.

This is the procedure to perform simulations iteratively:

1. In `ssi.v?`, seed the entire set of faults, using either the `seed` command or the `read` commands.

2. Run the Verilog simulation with the first testbench:

```
vcs +acc+2 -R -P $IDDQ_HOME/lib/iddq_vcs.tab
testbench1.v ssi.v ... $IDDQ_HOME/lib/libiddq_vcs.a
or
Verilog testbench1.v ssi.v ...
```

3. Run IDDQPro to select five strobe points:

```
ipro -strb_lim 5 ...
```

4. Save the fault report and strobe report:

```
mv iddq.srpt run1.srpt
mv iddq.frpt run1.frpt
```

5. Edit and change `ssi.v` so that it seeds only the undetected faults in `run1.frpt?`:

```
...
```

```
$ssi_iddq( "read_tmax run1.frpt" );
```

```
...
```

6. Run the Verilog simulation again, using the second testbench:

```
vcs +acc+2 -R -P $IDDQ_HOME/lib/iddq_vcs.tab
```

```
testbench2.v ssi.v ... $IDDQ_HOME/lib/libiddq_vcs.a
```

```
or
```

```
Verilog testbench2.v ssi.v ...
```

7. Run IDDQPro again to select five strobe points from the second testbench:

```
ipro -strb_lim 5 ...
```

8. Save the fault report and strobe report:

```
mv iddq.srpt run2.srpt
```

```
mv iddq.frpt run2.frpt
```

After completion of these steps, run1.srpt contains five strobe points for the first testbench and run2.srpt contains five strobe points for the second testbench.

If you have more than two testbenches, repeat steps 5 through 8 for each testbench, substituting the appropriate file names each time.

Using PowerFault Technology

The following sections provide information on using PowerFault simulation technology:

- [PowerFault Verification and Strobe Selection](#)
- [Testbenches for IDDQ Testability](#)
- [Combining Multiple Verilog Simulations](#)
- [Improving Fault Coverage](#)
- [Floating Nodes and Drive Contention](#)
- [Status Command Output](#)
- [Behavioral and External Models](#)
- [Multiple Power Rails](#)
- [Testing I/O and Core Logic Separately](#)

PowerFault Verification and Strobe Selection

You can use PowerFault simulation technology to perform the following IDDQ tasks:

- [Verifying TestMAX ATPG IDDQ Patterns for Quiescence](#)
- [Selecting Strobes in TestMAX ATPG Stuck-At Patterns](#)
- [Selecting Strobe Points in Externally Generated Patterns](#)

Verifying TestMAX ATPG IDDQ Patterns for Quiescence

When you use the TestMAX ATPG IDDQ fault model, TestMAX ATPG generates test patterns that have an IDDQ strobe in every pattern. When you write the patterns to a Verilog-format file, TestMAX ATPG automatically includes the PowerFault tasks necessary for verifying quiescence at every strobe.

To verify TestMAX ATPG IDDQ test patterns for quiescence, use the following procedure:

1. In TestMAX ATPG, use the `write_patterns` command to write the generated test patterns in STIL format. For example, to write a pattern file called `test.stil`, you could use the following command:

```
write_patterns test.stil -internal -format stil
```

2. Using MAX Testbench, create a Verilog testbench (for details, see [Using the stil2Verilog Command](#)). For example, to write a Verilog testbench called `test.v` you could use the following command:

```
stil2Verilog test.stil test
```

3. If you want to specify the name of the leaky node report file, open the test pattern file in a text editor and search for all occurrences of the `status drivers leaky` command, and change the default file name to the name you want to use. This is the default command:

```
// NOTE: Uncomment the following line to activate
// processing of IDDQ events
// define tmax_iddq
`$ssi_iddq("status drivers leaky top_level_name.leaky");
```

Substitute your own file name as in the following example:

```
`$ssi_iddq("status drivers leaky my_report.leaky");
```

Save the edited test pattern file.

4. Run a Verilog/PowerFault simulation using the test pattern file.

The simulator produces a quiescence analysis report, which you can use to debug any leaky nodes found in the design.

Selecting Strobes in TestMAX ATPG Stuck-At Patterns

Instead of generating test patterns specifically for IDDQ testing, you can use TestMAX ATPG to generate ordinary stuck-at ATPG patterns and then use PowerFault simulation technology to choose the best strobe times from those patterns. To do this, you need to modify the Verilog testbench file to enable the simulator's IDDQ tasks.

This is the general procedure:

1. In TestMAX ATPG, use the `write_patterns` command to write the generated test patterns in STIL format. For example, to write a pattern file called `test.stil`, you could use the following command:

```
write_patterns test.stil -internal -format stil
```

2. Using MAX Testbench, create a Verilog testbench (for details, see [Using the stil2Verilog Command](#)). For example, to write a Verilog testbench called `test.v` you could use the following command:

```
stil2Verilog test.stil test
```

3. Open the test pattern file in a text editor.
4. At the beginning of the file, find the following comment line:

```
// `define tmax_iddq
```

Remove the two forward slash characters to change the comment into a ``define tmax_iddq` statement. This enables the PowerFault tasks that TestMAX ATPG has embedded in the testbench.

Instead of activating the ``define tmax_iddq` statement in the file, you can define `tmax_iddq` when you invoke the Verilog simulator. For example, when you invoke VCS, use the `+define+tmax_iddq=0+` option.

5. If you want to specify the name of the leaky node report file, search for all occurrences of the `status drivers leaky` command and change the default file name to the name you want to use. This is the default command:

```
`$ssi_iddq("status drivers leaky top_level_name.leaky");
```

6. Save the edited test pattern file.
7. Run a Verilog simulation using the edited test pattern file.
8. Run the IDDQ Profiler.

When you run the Verilog/PowerFault simulation, the IDDQ system tasks evaluate each strobe time for fault coverage. When you run the IDDQ Profiler, it selects the best strobe times.

Selecting Strobe Points in Externally Generated Patterns

You can use PowerFault simulation technology to select strobes from testbenches generated by sources other than TestMAX ATPG. The procedure depends on the testbench source:

- For test vectors generated by other ATPG tools, edit the testbench to add the PowerFault tasks.
- For functional (design verification) test vectors, edit the testbench to add the PowerFault tasks and determine timing for the tester vector. Use t-1, the last increment of time within a test cycle, for IDDQ strobes.
- For BIST (built-in self-test), control the clock with tester and determine timing for the tester vector. Use t-1 for IDDQ strobes.

To see how to edit the testbench to add PowerFault tasks, you can look at some Verilog testbenches generated by TestMAX ATPG. For example, after the `initial begin` statement, you need to insert `$ssi_iddq` tasks to invoke the PowerFault commands:

```
initial begin
//Begin IddQTest initial block
$ssi_iddq("dut adder_test.dut");
$ssi_iddq("verb on");
$ssi_iddq("seed SA adder_test.dut");
$display("NOTE: Testbench is calling IDDQ PLIs.");
$ssi_iddq("status drivers leaky LEAKY_FILE");
//End of IddQTest initial block
...
end
```

You also need to find the capture event and insert the PowerFault commands to evaluate a strobe at that point. For example:

```
event capture_CLK;
always @ capture_CLK begin
->forcePI_default_WFT;
```

```
#140; ->measurePO_default_WFT;

#110 PI[4]=1;

#130 PI[4]=0;

//IddQTest strobe try

begin

$ssi_iddq("strobe_try");

$ssi_iddq("status drivers leaky LEAKY_FILE");

end

//IddQTest strobe try

end
```

Testbenches for IDDQ Testability

When you create a testbench outside of the TestMAX ATPG environment, the following design principles can significantly improve IDDQ testability:

- [Separate the Testbench From the Device Under Test](#)
- [Drive All Input Pins to 0 or 1](#)
- [Try Strokes After Scan Chain Loading](#)
- [Include a CMOS Gate in the Testbench for Bidirectional Pins](#)
- [Model the Load Board](#)
- [Mark the I/O Pins](#)
- [Minimize High-Current States](#)
- [Maximize Circuit Activity](#)

Separate the Testbench From the Device Under Test

For better IDDQ testability, maintain a clean separation of the testbench from the device under test (DUT). The Verilog DUT module should model only the structure and behavior of the chip. Put the chip-external drivers and pullups in the testbench. The testbench should also generate stimulus for the chip and verify the correctness of the chip's outputs.

Drive All Input Pins to 0 or 1

The mapping of testbench Xs to automated test equipment (ATE) drive signals is not well defined. The results depend on how the active load on the ATE is programmed. Because

Xs can be mapped to VDD, VSS, or some intermediate voltage, such as $(VDD-VSS)/2$, avoid having your testbench drive Xs into the chip. PowerFault reports input pins driven to X as “possible float.”

Try Strobes After Scan Chain Loading

To minimize simulation time and database size when you run a Verilog/PowerFault simulation, do not perform a `strobe_try` on every serialized scan load step. Instead, use `strobe_try` only after the entire scan chain is loaded.

If your simulation does a parallel scan load or you are using functional vectors, use `strobe_try` before the end of each cycle.

Include a CMOS Gate in the Testbench for Bidirectional Pins

If your chip has bidirectional I/O pins, place a CMOS gate inside the testbench to transmit the signal between the testbench driver and the I/O pad. For details, see [Use Pass Gates](#).

Model the Load Board

Take into account external connections to the DUT. When a chip is tested by ATE, it resides on a load board. The load board is a printed circuit board that provides the encapsulating environment in which the chip is tested. It can contain pullups/pulldowns, latches for three-state I/O pins, power/ground connections, and so on.

In general, your Verilog testbench should model the load board as accurately as possible. Any pullups/pulldowns/latches that would exist on the load board should be modeled in the testbench. In general, if a chip requires pullups to operate correctly in a real system, you can assume they are needed on the load board also.

Mark the I/O Pins

The top-level ports of each DUT module are assumed to be primary I/O ports and are given special treatment by PowerFault. If the testbench drives the DUT through other ports, use the `io` command to tell PowerFault about these ports. For information on the `io` command, see “io” in the [PowerFault PLI Tasks](#) section.

Minimize High-Current States

Try to minimize times when analog, RAM, and I/O cells are in current-draining states. Put them into standby mode when possible and write a complete set of test vectors for analog/RAM/I/O standby mode.

Because IDDQ testing can be performed when the circuit is in a low-current state, try to minimize the number of vectors that put the circuit into high-current states. For maximum coverage, you might need to repeat the vectors that are normally applied during high-current states. For example, if your I/O pads have active pullups during some vectors, you

can apply those same vectors again when the pullups are disabled, so that IDDQ testing can be performed on those vectors.

Maximize Circuit Activity

Try to toggle each node during low-current states. Some easy methods for achieving high circuit activity include:

- Shift alternating 0/1 patterns into scan registers.
- Apply alternating 0/1 patterns to data and address lines.

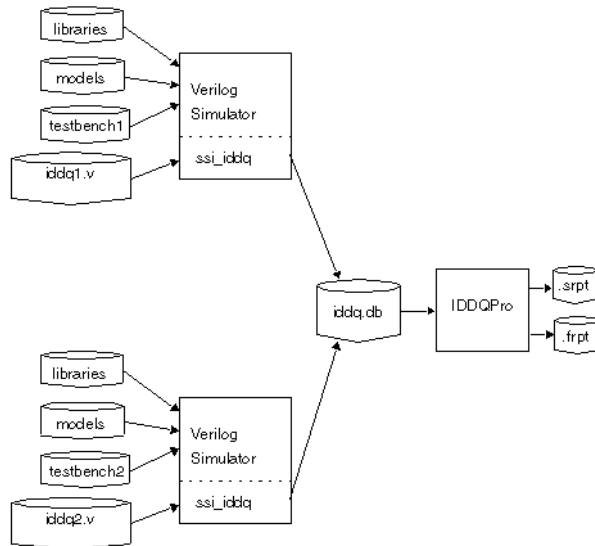
Combining Multiple Verilog Simulations

If you use different Verilog simulation runs to test different portions of a device or to drive a device into different states, you can use PowerFault technology to choose a set of strobe times for maximum fault coverage over all the resulting testbenches. For example, if there are 30 testbenches and your tester time budget allows only five IDDQ strobings, the five selected strobings ought to provide the best coverage out of all possible strobings in all 30 testbenches.

If you want to improve coverage efficiency within a single testbench, see [Deleting Low-Coverage Strobings](#).

To combine multiple simulation results, you can merge the IDDQ information from each successive Verilog/PowerFault simulation into a single database. Then you can apply the IDDQ Profiler to that single database. This process is illustrated in the following figure.

Figure 191 Using Multiple Testbenches



The following procedure is an example of a strobe selection session using two testbenches and a budget of five IDDQ strobes. The PowerFault PLI tasks for `testbench1` and `testbench2` are in files named `iddq1.v` and `iddq2.v`, respectively.

1. In `iddq1.v` and `iddq2.v`, seed the entire set of faults, using either the `seed` command or `read` commands. For example:

```
$ssi_iddq( "seed SA iddq1.v" );
```

```
$ssi_iddq( "seed SA iddq2.v" );
```

2. In `iddq1.v`, use the `output create` command to save the simulation results to an IDDQ database named `iddq.db`?:

```
$ssi_iddq( "output create label=run1 iddq.db" );
```

3. In `iddq2.v`, use the `output append` command to append the simulation results to the database you created in Step 2:

```
$ssi_iddq( "output append label=run2 iddq.db" );
```

4. Run a Verilog/PowerFault simulation using `testbench1.v` and `iddq1.v`?.
5. Run a Verilog/PowerFault simulation using `testbench2` and `iddq2.v`?.

6. Run the IDDQ Profiler to select five good strobe points from the `iddq.db` database:

```
ipro -strb_lim 5 iddq
```

A strobe report for multiple testbenches shows both the testbench number and simulation time within the respective testbench for each selected strobe. Testbench names and labels are listed in the header of the strobe report. Testbenches are numbered in sequence, starting with 1.

When you use multiple testbenches, the fault report files show only the comment lines from the first testbench. PowerFault does not try to merge the comment lines from the fault list in the second and subsequent testbenches with those in the first testbench.

Improving Fault Coverage

PowerFault does not require additional design-for-test (DFT) circuitry or modifications to your testbench, models, or libraries. It does not require configuration files, and it runs on any Verilog chip design.

If PowerFault is unable to find enough qualified strobos to provide satisfactory fault coverage, you might be able to find more qualified strobos by using the techniques described in the following sections:

- [Determine Why the Chip Is Leaky](#)
- [Evaluate Solutions](#)

Determine Why the Chip Is Leaky

The first step is to run the Verilog/PowerFault simulation to determine why the chip is leaky at strobe times. At each strobe try, PowerFault examines your chip for leaky states. If it finds any leaky states, it disqualifies the strobe point.

To check the leaky states for each strobe point, use the `status` command after the `strobe_try?`, as in the following example:

```
always begin
    fork
        # CLOCK_PERIOD;
        # (CLOCK_PERIOD -1)
    begin
        $ssi_iddq( "strobe_try" );
        $ssi_iddq( "status drivers leaky bad_nodes" );
    end
    join
```

```
end
```

This example creates a file called `bad_nodes` that describes each leaky state at each strobe point. For example:

```
Time 3999
top.dut.vee[0] is leaky: Re: float
HiZ <- top.dut.veePad0.out
top.dut.DIO[1] is leaky: Re: fight
St0 <- top.dut.dpad1_cld
St1 <- top.dut.dpad1_snd
StX <- resolved value
```

For each `status` command, the simulator reports the simulation time and a list of leaky nodes. In the report, the full path name of each net is followed by a reason (such as `Re: float?`) and a list of drivers and their contribution to the net value. For example, in the preceding example, `top.dut.vee[0]` is floating because its lone driver (`?top.dut.veePad0?`) is in the high-impedance state.

For a complete description of the output of the `status` command, see [Status Command Output](#). For more information on leaky states, see [Leaky State Commands](#).

Evaluate Solutions

After you identify and understand the leaky states, you need to decide how to eliminate or ignore them so that you can change unqualified strobes into qualified ones. Use any of the following methods:

- [Use the allow Command](#)
- [Configure the Verilog Testbench](#)
- [Configure the Verilog Models](#)

Use the allow Command

The `allow` command can make PowerFault ignore leaky states that you know are not present in the real chip. For example, incomplete Verilog models can cause misleading leaky states that prevent PowerFault from qualifying strobe points. For more information, see [Leaky State Commands](#).

Configure the Verilog Testbench

In some cases, you can fix leaky states by modifying the Verilog testbench, as described in the following sections:

- [Drive All Input Pins to 0 or 1](#)
- [Use Pass Gates](#)
- [Model the Load Board](#)
- [Mark the I/O Pins](#)

Drive All Input Pins to 0 or 1

Make sure the testbench initializes all primary inputs. If your testbench drives Xs into the primary input pins of the device under test (DUT), PowerFault disqualifies the vector and flags those pins as “possible float.” PowerFault takes the conservative position that Xs driven by the testbench might translate to the automated test equipment (ATE) turning off the drive signal and allowing the input pin to float.

If your ATE replaces Xs with a default drive value (either VDD or VSS), then driving Xs should be allowed. In that case, use the `allow float` command on all your input pins, as in the following example:

```
$ssi_iddq( "allow float testbench.chip.RE" );  
$ssi_iddq( "allow float testbench.chip.ABUS[0]" );  
$ssi_iddq( "allow float testbench.chip.ABUS[1]" );
```

Use Pass Gates

If your chip has bidirectional I/O pins, place a CMOS gate inside the testbench to transmit the signal between the testbench driver and the I/O pad.

The following code shows how two registers in the testbench are connected to signals that feed the DUT pins:

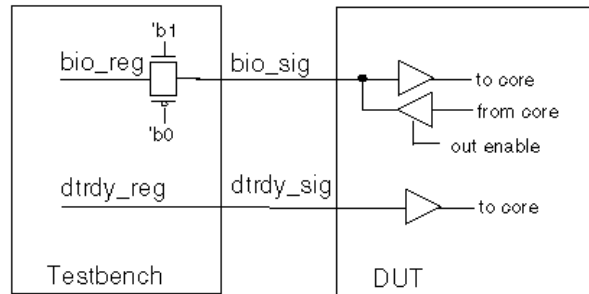
```
reg bio_reg, dtrdy_reg; // registers to hold stimulus  
  
// drive bidirectional "bio" signal through pass gate  
wire bio_tmp = bio_reg;  
  
cmos( bio_sig, bio_tmp, `b1, `b0);  
  
// drive input signal directly  
wire dtrdy_sig = dtrdy_reg;  
  
// hookup signals to dut
```



```
dut dut( bio_sig, dtrdy_sig, ... );
```

Notice how the input signal `dtrdy_sig` is driven directly by the `dtrdy_reg` register, but the bidirectional signal `bio_sig` is driven through the `cmos` primitive, as shown in the following figure.

Figure 192 Pass Transistor Between the Testbench and DUT



Model the Load Board

When a chip is tested by ATE, it resides on a load board. The load board is a printed circuit board that provides the encapsulating environment in which the chip is tested. It can contain pullups/pulldowns, latches for three-state I/O pins, power and ground connections, and so on.

In general, your Verilog testbench should model the load board as accurately as possible. Any pullups, pulldowns, and latches that would exist on the load board should be modeled in the testbench. In general, if a chip needs pullups to operate correctly in a real system, you can assume they are needed on the load board also.

Mark the I/O Pins

The top-level ports of a DUT module are assumed to be primary I/O ports and are given special treatment by PowerFault. If the testbench drives the DUT through other ports, use the `io` command to tell PowerFault about these ports. For information on the `io` command, see [PowerFault PLI Tasks](#).

Configure the Verilog Models

In general, the more your chip is modeled at a structural level (using gates, switches, and wires), the better for IDDQ testing. If your cells model logic behaviorally rather than with built-in Verilog primitives and user-defined primitives (UDPs), PowerFault might find fewer qualified strobe points. For details, see the following sections:

- [Drive All Buses Possible](#)
- [Gate Buses That Cannot Be Driven](#)

- [Use Keeper Latches](#)
- [Enable Only One Driver](#)
- [Avoid Active Pullups and Pulldowns](#)
- [Avoid Bidirectional Switch Primitives](#)

Drive All Buses Possible

Because floating buses can disqualify strobe points, try to always drive internal buses. Either configure the control logic to always enable one driver for the bus, or use keeper latches (holders).

For example, here is a bus that has two drivers that are fully multiplexed:

```
bufif1 ( addr0, X[0], sel ); // driver 1
bufif1 ( addr0, Y[0], sel_bar ); // driver 2
not ( sel_bar, sel ); // inverter
```

Gate Buses That Cannot Be Driven

If driving the bus is not always possible or desirable, gate the bus so that when it does float, the effect is blocked. For example, here is a bus that has two drivers and one load:

```
bufif1 ( addr0, X[0], x_en ); // driver 1
bufif1 ( addr0, Y[0], y_en ); // driver 2
or ( x_or_y_en, x_en, y_en ); // qualifier
and ( addr0_qualified, addr0, x_or_y_en ); // load
```

The bus value is blocked at the load (AND gate) when neither driver is active. If you want to use OR gates to block floating buses, use the `statedep_float` command. For more information on this command, see “[statedep_float](#)” in the [PowerFault PLI Tasks](#) section. For more information on blocking floating buses, see [State-Dependent Floating Nodes](#).

Use Keeper Latches

If a bus cannot always be driven or gated, consider using keeper latches (also called “keepers”). A keeper retains the last value driven onto the bus. It has a weaker drive strength than normal bus drivers so that it can be overdriven.

Keepers should be modeled structurally. For example, here is a bus that has two drivers and one keeper:

```
bufif1 ( addr0, X[0], x_en ); // driver 1
bufif1 ( addr0, Y[0], y_en ); // driver 2
```

```
buf (pull0,pull1) ( addr0, addr0 ); // keeper
```

Avoid modeling keepers behaviorally or with continuous assignments:

```
wire (pull0,pull1) addr0 = addr0; // AVOID THIS
```

Use only strength-restoring gates such as `buf` for modeling keepers. Avoid using switch primitives (`?nmos?`, `pmos?`, `cmos?`) for modeling keepers:

```
rnmos( addr0, addr0, 'b1 ); // AVOID THIS
```

Enable Only One Driver

Because bus contention disqualifies strobe points, initialize all control logic (enabling lines) for bus drivers. Furthermore, if possible, configure the control logic to enable only one driver for the bus at a time.

Avoid Active Pullups and Pulldowns

Active pullups and pulldowns can also disqualify strobe points, so use keeper latches on three-state buses rather than pullups or pulldowns. PowerFault treats each of the following elements as a pullup or pulldown:

- `pullup` and `pulldown` primitives
- `tri1` and `tri0` nets
- `wand` and `wor` nets

Conflicting values on “wired AND” nets are reported as active pullups, and conflicting values on “wired OR” nets are reported as active pulldowns.

When you must use pullups or pulldowns, model them structurally like this:

```
wire n26;  
  
pullup( n26 );  
  
OR  
  
tri1 n26;
```

Avoid modeling pullups and pulldowns behaviorally or with continuous assignments, as in the following example:

```
wire (highz0,pull1) n26 = n26; // AVOID THIS
```

Avoid Bidirectional Switch Primitives

Avoid using the `rtran?`, `rtranif1?`, and `rtranif0` primitives. If possible, replace them with `nmos?`, `pmos?`, or `cmos` primitives.

Floating Nodes and Drive Contention

PowerFault recognizes certain types of floating nodes and drive contention, and reports them according to their classification. The following sections describe floating nodes and drive contention:

- [Floating Node Recognition](#)
- [Drive Contention Recognition](#)

Floating Node Recognition

The following sections describe floating node recognition:

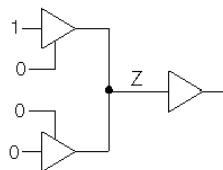
- [Leaky Floating Nodes](#)
- [Floating Nodes Ignored by PowerFault](#)
- [State-Dependent Floating Nodes](#)
- [Configuring Floating Node Checks](#)
- [Floating Node Reports](#)
- [Nonfloating Nodes](#)

Leaky Floating Nodes

PowerFault identifies the following types of floating nodes as leaky:

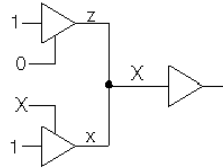
- *True floating node* — This is a node at Z, which does not have any active drivers, as shown in the following figure.

Figure 193 True Floating Node Example



- *Possibly floating node* — This is a node at X that might not have an active driver, as shown in the following figure, or an undriven capacitive node. A capacitive node is a Verilog net with small, medium, or large strength.

Figure 194 Possibly Floating Node Example

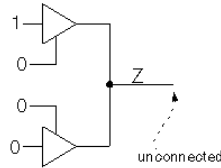


Floating Nodes Ignored by PowerFault

PowerFault ignores (does not report) these types of floating nodes:

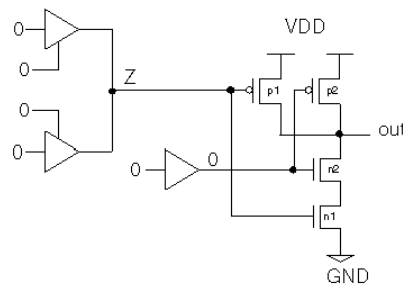
- *Floating node without a load* — This is a node that does not drive anything, as shown in the following figure.

Figure 195 Floating Node Without Load Example



- *State-dependent floating node* — This is a node that can be allowed to float because its effects are blocked by the states of other inputs, as shown in the following figure.

Figure 196 Blocked Floating Node Example



State-Dependent Floating Nodes

For AND, NAND, and NOR gates, the IDDQ effect of a floating input can be blocked by the other inputs. For example, if one input to a two-input NAND gate is floating but the other input is 0, the floating input is blocked so that it cannot cause a leakage current.

In the preceding figure, the 0 input turns off transistor n2, so there is no conducting path from VDD to VSS through transistors p1 and n1. If the 0 input was 1 instead, PowerFault would identify the floating input as leaky.

By default, all inputs of 2-input and 3-input AND/NAND gates and 2-input NOR gates are treated as state-dependent floating nodes. By default, gates with more inputs and other types of gates are not allowed to have floating inputs. You can change the input limit for the AND, NAND, and NOR gates by using the `statedep_float` command. For more information, see “`statedep_float`” in “[PowerFault PLI Tasks](#) .”

Configuring Floating Node Checks

Using the `allow` and `disallow` commands, you can configure how floating nodes are recognized. The `allow` command lets you do the following:

- Allow a particular node to float
- Allow all nodes to float
- Allow possible floating nodes (true floating nodes are still disallowed)

The `disallow` command lets you do the following:

- Disallow a Z on a particular node
- Disallow Zs on all nodes

For a complete description of the `allow` and `disallow` commands, see [PowerFault PLI Tasks](#).

Floating Node Reports

The `status leaky` command reports a list of floating nodes and nodes with drive contention. In order to save space, it reports only the floating node at the first strobe where the node is leaky. To get a report on all floating nodes (including those previously reported), use the `all_leaky` option with the `status` command. For example:

```
$ssi_iddq( "status drivers all_leaky bad_nodes" );
```

Nonfloating Nodes

To get a list of leaky nodes, use the following command:

```
$ssi_iddq( "status leaky" );
```

To get a list of nonleaky nodes, use the following command:

```
$ssi_iddq( "status nonleaky" );
```

This command reports a list of nodes that are not floating and do not have drive contention, together with the reason that each node was found to be nonleaky. This

information can be useful when you think a node should be reported as floating, but it is not.

Drive Contention Recognition

PowerFault identifies the following types of drive contention:

- *Pullups and pulldowns* — For example, see the active pullup in [Figure 197](#).
- *Contention between multiple bus drivers* — For example, see the true drive fight in [Figure 198](#).

Figure 197 Active Pullup

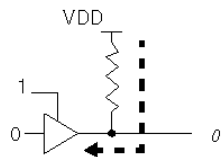
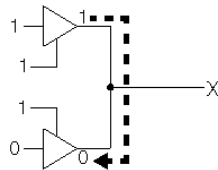
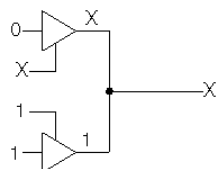


Figure 198 True Drive Fight



PowerFault makes a distinction between true and possible drive contention. A true fight occurs when a net has both a 0 (VSS) driver and a 1 (VDD) driver. A possible fight occurs when one or more drivers are at X on a bus with multiple drivers, as shown in the following figure.

Figure 7 Possible Drive Fight



PowerFault also warns about unusual connections that indicate static leakage. The first time you execute the `status` command, it writes warning messages to the simulation log file about the following conditions:

- A node connected to both VSS (supply0) and VDD (supply1)
- A node connected to both a pullup and a pulldown

Status Command Output

The output of the `status` command can help you determine the cause of floating nodes and drive contention. Eliminating or reducing these types of leaky states not only makes your design more IDDQ-testable, it can also reduce the device power consumption.

The following sections describe the `status` command output:

- [Status Command Overview](#)
- [Leaky Reasons](#)
- [Nonleaky Reasons](#)
- [Driver Information](#)

Status Command Overview

The `status` command is executed during the Verilog/PowerFault simulation. It reports the nodes found to be leaky or nonleaky. For information on the command syntax, see “status” in [PowerFault PLI Tasks](#).

The status of each node is reported in this format:

```
net-instance-name is (leaky|non-leaky). Re: reason
```

The instance name of each net is followed by a reason that explains why the node was found to be leaky or nonleaky. For example:

```
top.dut.TBIN is leaky: Re: float
```

```
top.dut.DIO is leaky: Re: possible float
```

The `status` command distinguishes between true and possible leaks. Possible leaks arise when nodes and drivers have unknown values (X). In the preceding example, `top.dut.TBIN` is truly floating (Z), whereas `top.dut.DIO` is possibly floating.

By default, the `status leaky` command reports only the first occurrence of a leaky node. When there are leaky nodes at a strobe, and all these leaky nodes have been reported at previous strobe times, the command prints the message “All reported.”

Leaky Reasons

The status command determines that a node is leaky for either a standard or user-defined reason. A standard reason is reported when the node is leaky due to a built-in quiescence check, such as fight, float, pullup, or pulldown. A user-defined reason is reported when the node violates a condition specified by the `disallow` command.

[Table 22](#) lists the standard leaky reasons and [Table 23](#) lists the user-defined leaky reasons.

Table 22 *Standard Leaky Reasons*

Reason	Description
Fight	A drive fight between two or more drivers of equal strength. One driver is at 0 and another is at 1.
Pullup	An active pullup. A net with a pullup is being driven to 0. Any time a stronger driver at 0 is overriding a weaker driver at 1, the net is flagged as having an active pullup.
Pulldown	An active pulldown. A net with a pulldown is being driven to 1. Any time a stronger driver at 1 is overriding a weaker driver at 0, the net is flagged as having an active pulldown.
Float	A floating input node; an input node that is undriven (Z).
Possible Fight	A possible drive fight. One driver at X might be fighting with another driver (see Floating Nodes and Drive Contention).
Possible Pullup	A possible active pullup. A net with a pullup is being driven by an X. Any time a stronger driver at X is overriding a weaker driver at 1, the net is flagged as having a possible pullup.
Possible Pulldown	A possible active pulldown. A net with a pulldown is being driven by an X. Any time a stronger driver at X is overriding a weaker driver at 0, the net is flagged as having a possible pulldown.
Possible Float	A possible floating input node. The node is at X, but might have no active drivers (see Floating Nodes and Drive Contention).

Table 23 *User-Defined Leaky Reasons*

Reason	Description
Disallowed 0	A <code>disallow</code> command flags the net's present state (0) as leaky.

Table 23 User-Defined Leaky Reasons (Continued)

Reason	Description
Disallowed 1	A disallow command flags the net's present state (1) as leaky.
Disallowed X	A disallow command flags the net's present state (X) as leaky.
Disallowed Z	A disallow command flags the net's present state (Z) as leaky.
Disallow all Xs	A disallow X command flags the net's state (X) as leaky.
Disallow all Zs	A disallow Z command flags the net's present state (Z) as leaky.
Disallow all Caps A	A disallow Caps command flags the net's present capacitive state as leaky.
Disallowed 0	A disallow command flags the net's present state (0) as leaky.
Disallowed 1	A disallow command flags the net's present state (1) as leaky.

A user-defined leaky reason appears when a node has a state specifically disallowed by a `disallow` command. For example:

```
$ssi_iddq( "disallow top.dut.SDD == 0" );  
$ssi_iddq( "disallow Z" );
```

These two `disallow` commands produce a report like the following:

```
top.dut.SDD is leaky: Re: disallowed 0  
top.dut.BIO is leaky: Re: disallow all Zs
```

In this example, `top.dut.SDD` is 0, which is disallowed by the first `disallow` command; and `top.dut.BIO` is Z, which is disallowed by the second `disallow` command.

Nonleaky Reasons

Table 24 lists the standard nonleaky reasons and Table 25 lists the user-defined nonleaky reasons.

Table 24 Standard Nonleaky Reasons

Reason	Description
0 or 1	The node is a quiet 0 or 1.
Z no loads	The node is floating, but not connected to any inputs.
Z blocked	The node is floating, but is blocked (see Floating Nodes and Drive Contention).
X no contention	The node is driven to X (it is not floating) and has no contention; it is probably uninitialized.
Possible float no loads	The node is X and might be floating, but is not connected to any inputs.
Possible float blocked	The node is X and might be floating, but is blocked.

Table 25 User-Defined Nonleaky Reasons

Reason	Description
Allowable float	The node is (or possibly is) floating, but an <code>allow</code> command permits it.
Allowable fight	The node has (or possibly has) drive contention, but an <code>allow</code> command allows it.
Allow all fights	The node has (or possibly has) drive contention, but an <code>allow</code> command allows all contention.
Allow poss fights	The node possibly has drive contention, but an <code>allow</code> command allows possible contention.
Allow all floats	The node is (or possibly is) floating, but an <code>allow</code> command allows all floats.

Table 25 User-Defined Nonleaky Reasons (Continued)

Reason	Description
Allow poss floats	The node is possibly floating, but an <code>allow</code> command allows all possible floats.

A user-defined nonleaky reason appears when a node has a state specifically allowed by an `allow` command. For example:

```
$ssi_iddq( "allow fight top.dut.PL" );
$ssi_iddq( "allow all float" );
```

These two `allow` commands can produce a report like the following:

```
top.dut.PL is non-leaky: Re: allowable fight
top.dut.BIO is non-leaky: Re: allow all floats
```

Driver Information

To determine why a net is floating or has drive contention, its drivers must be examined. Simulation debuggers and even some system tasks (such as the `$showvar` task in the Verilog simulator) can perform this examination. You can also use the `drivers` option of the `status` command, but this option generates only gate-level driver information.

The `drivers` option causes the `status` command to print the contribution of each driver. For example:

```
$ssi_iddq( "status drivers leaky bad_nodes" );
can produce output like:
top.dut.mmu.DIO is leaky: Re: fight
St0 <- top.dut.mmu.UT344
St1 <- top.dut.mmu.UT366
StX <- resolved value
top.dut.mmu.TDATA is leaky: Re: float
HiZ <- top.dut.mmu.UT455
HiZ <- top.dut.mmu.UT456
```

In this example, `top.dut.mmu.DIO` has a drive fight. One driver is at strong 0 (`?st0?`) and the other at strong 1 (`?st1?`). The contributing value of each driver is printed in Verilog strength/value format, as described in section 7.10 of the IEEE 1364 Verilog LRM.

The same status command without the `drivers` option produces a report like this:

```
top.dut.mmu.DIO is leaky: Re: fight
top.dut.mmu.TDATA is leaky: Re: float
```

Behavioral and External Models

PowerFault examines the structure of your Verilog HDL model to determine whether the chip is quiescent. PowerFault looks for bus contention, floating inputs, active pullups, and other current-drawing states.

If you use behavioral models or external models (like LMC, LAI, or VHDL cosimulated models) to simulate subblocks of the chip, PowerFault cannot to determine when those subblocks are quiescent. As a result, it might select strobe points that are inappropriate for IDDQ testing. To prevent this from happening, use the `disallow` command.

The following sections describe the `disallow` command in more detail:

- [Disallowing Specific States](#)
- [Disallowing Global States](#)

Disallowing Specific States

The `disallow` command is a flexible command that lets you describe the leaky states for all instances of a behavioral or external model. One or more commands can describe which input, output, or internal states correspond to nonquiescence.

For example, the three following `disallow` commands describe when instances of the BRAM and DAC entities are leaky:

```
$ssi_iddq( "disallow BRAM (REFRESH == 1 && ENABLE == 0)" );
$ssi_iddq( "disallow BRAM (WRITE_EN == 1 || READ_EN == 1)" );
$ssi_iddq( "disallow DAC (port.0 != 0 && port.1 != 0)" );
```

Disallowing Global States

You can use the `disallow` command to disallow all nets in the Verilog simulation from having a particular value. This is useful if the libraries contain behavioral gate models. For example, if the three-state buffers are not modeled with Verilog primitives or UDPs, then PowerFault might not be able to detect bus contention.

Here is an example of a three-state buffer modeled behaviorally:

```
module BUF0 (out, data, control);  
  
output out;  
  
input data, control;  
  
wire out = ( control == 0 ) ? data : `bZ;  
  
endmodule
```

To prevent bus contention during an IDDQ strobe, you can disallow all Xs with this command:

```
$ssi_iddq( "disallow X" );
```

If disallowing all Xs is too pessimistic, you can use a specific `disallow` command for each three-state buffer entity. For example, if you have two types of three-state buffers, BUF0 and BUF1, use the following commands:

```
$ssi_iddq( "disallow BUF0 ( out == X )" );  
$ssi_iddq( "disallow BUF1 ( out == X )" );
```

If the libraries contain behavioral gate models, PowerFault might not be able to detect floating buses (buses with all drivers turned off). To prevent floating buses during an IDDQ strobe, you can disallow all Zs with this command:

```
$ssi_iddq( "disallow Z" );
```

If disallowing all Zs is too pessimistic, you can use a `disallow` command for each three-state buffer entity. For example, you could use the following commands:

```
$ssi_iddq( "disallow BUF0 ( out == Z )" );  
$ssi_iddq( "disallow BUF1 ( out == Z )" );
```

For more information on the `disallow` command, see [Leaky State Commands](#).

Multiple Power Rails

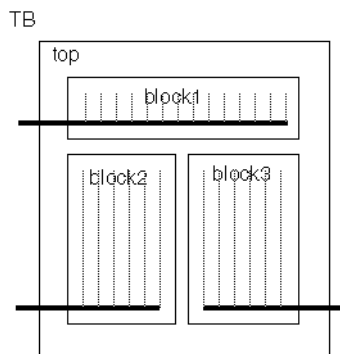
This section describes how to apply PowerFault to a chip with multiple power rails, where each power rail feeds a separate logic block on the chip. The overall strategy is as follows:

1. Determine the number of IDDQ test points for each block.
2. For one block, run a Verilog/PowerFault simulation, seeding only the faults in that block; and use IDDQPro to select strobes for the block.

3. Repeat step 2 for each block in the design. Exclude any strobes that have already been selected for previous blocks.
4. To determine the fault coverage for each block using the full set of strobes, run IDDQPro separately on each database, manually selecting all strobes selected in steps 2 and 3.

Here is an example. Suppose you have a chip with three power rails, as shown in the following figure.

Figure 199 Chip With Three Power Rails



Step 1

Select two IDDQ strobes for each block.

Step 2

Run a Verilog simulation, seeding faults only in block1. The Verilog simulation produces a database named db1 (see [Figure 200](#)). You then use IDDQPro to automatically select two strobes from the database and save the strobe report in accum.strobes (see [Figure 201](#)):

```
ipro -strb_lim 2 -prnt_nofrpt db1
```

```
mv iddq.srpt accum.strobes
```

Figure 200 Create a Database for Block 1

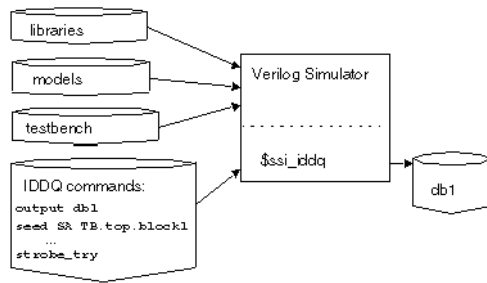
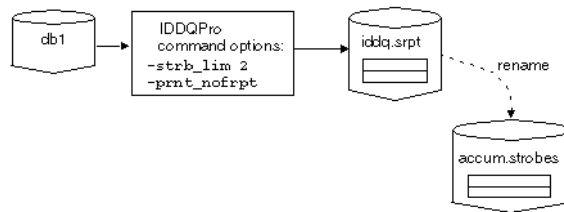


Figure 201 Select Two Strobes for Block 1



Step 3

Run the next Verilog simulation, this one seeding faults only in block2. The Verilog simulation produces a database named db2. You then use IDDQPro to automatically select two strobes from db2 and append the two strobes to accum.strobes (see the following figures):

```
ipro -strb_lim 2 -strb_unset accum.strobes -prnt_nofrpt db2
```

```
cat iddq.srpt >> accum.strobes
```


Figure 202 Create a Database for Block 2

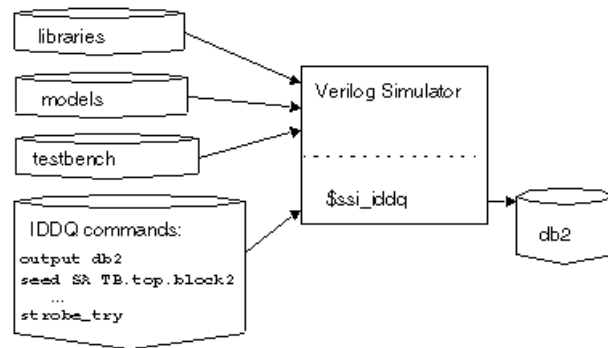
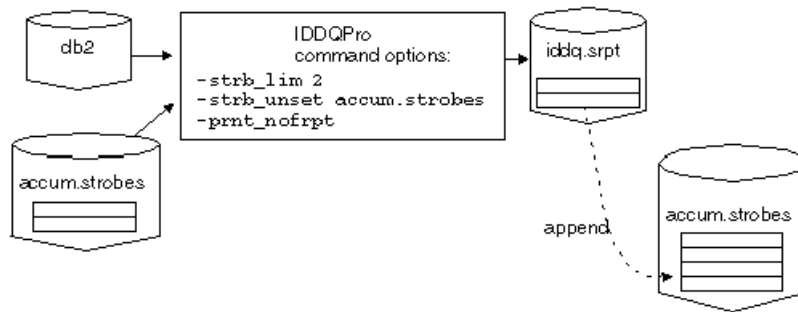


Figure 203 Select Two Strobes for Block 2



To complete step 3, you run the last Verilog simulation, this one seeding faults only in block3. The Verilog simulation produces a database named db3. You then use IDDQPro to automatically select two strobes from db3 and append the two strobes to accum.strobes?:

```
ipro -strb_lim 2 -strb_unset accum.strobes -prnt_nofrpt db3
cat iddq.srpt >> accum.strobes
```

The accum.strobes file now has six strobes (two for each block). The strobes you selected for any one block might be qualified for the other two blocks, so in step 4 you will try to select all six strobes.

Step 4

To begin step 4, you run IDDQPro to manually select six strobes from db1?. You select the strobes stored in accum.strobes and save the resulting strobe and fault reports:

```
ipro -strb_lim 6 -strb_set accum.strobes db1
mv iddq.srpt iddq.srpt1
mv iddq.frpt iddq.frpt1
```

Continuing step 4, you run `IDDQPro` to manually select six strobos from `db2`. You select the strobos stored in `accum.strobos` and save the resulting strobe and fault reports:

```
ipro -strb_lim 6 -strb_set accum.strobos db2
mv iddq.srpt iddq.srpt2
mv iddq.frpt iddq.frpt2
```

To finish step 4, you repeat the same procedure using `db3`:

```
ipro -strb_lim 6 -strb_set accum.strobos db3
mv iddq.srpt iddq.srpt3
mv iddq.frpt iddq.frpt3
```

Conclusion

After step 4 is complete, you have selected a total of six strobos (two for each block). The three individual strobe reports describe the fault coverage of the six strobos for each of the three blocks. The three individual fault reports describe the detected faults for each of the three blocks.

Testing I/O and Core Logic Separately

PowerFault looks at the chip as a whole. By default, everything in the DUT module, including I/O pads, must be quiescent to qualify a strobe point for IDDQ testing.

If the I/O pads and core logic have separate power rails, you can probably increase fault coverage by testing the core logic separately. This is because you can test the core at times when the I/O pads are leaky, assuming that you are able to measure IDDQ just for the core logic (ignoring the current drawn by the I/O pads).

To qualify strobos just for the core logic, use the `allow` command to ignore floating I/O pins and drive contention at I/O pins. This command makes PowerFault ignore all leaky states at the I/O pads. Also use the `exclude` command to prevent faults from being seeded inside the I/O pads.

Here is an example:

```
$ssi_iddq( "allow float top.dut.clk33_pad" );
$ssi_iddq( "allow fight top.dut.clk33_pad" );
$ssi_iddq( "exclude top.dut.clk33_pad" );
$ssi_iddq( "allow float top.dut.dto_pad" );
$ssi_iddq( "allow fight top.dut.dto_pad" );
```

```
$ssi_iddq( "exclude top.dut.dto_pad" );
```

35

Types of Reports

You can use the following output reports for various types of analysis in TestMAX ATPG:

[Analyze Buses](#)

[Analyze Faults](#)

[ATPG Constraints](#)

[ATPG Primitives](#)

[Report Buses](#)

[report_capture_masks](#)

[Cell Constraints](#)

[Clocks](#)

[Commands](#)

[report_delay_paths](#)

[Diagnosis](#)

[Faults](#)

[Memory](#)

[Modules](#)

[Net Connections](#)

[Nets](#)

[Nofaults](#)

[Nonscan Cells](#)

[Patterns](#)

[PI Constraints](#)

[PI Equivalences](#)

[Pin Data](#)

[PO Masks](#)
[Primitives](#)
[Rules](#)
[run_build_model](#)
[run_drc](#)
[run_fault_sim](#)
[run_justification](#)
[run_simulation](#)
[Scan Ability](#)
[Scan Cells](#)
[Scan Chains](#)
[Scan Path](#)
[Settings](#)
[Summaries](#)
[Version](#)
[Violations](#)
[Wires](#)

Output From the report_scan_ability Command

You can use output from the `report_scan_ability` command to view data on nonscan cells that have been selected to behave as scan cells using the `set_scan_ability` command. The syntax for this command is as follows:

```
report_scan_ability [-max d]
```

Standard Format

```
gate_id type instance_name
-----
 97299 DFF /core/host/d_l2odd_reg
 97302 DFF /core/host/rd_stat_reg_1
```

gate_id

Indicates the [primitive ID](#) number of the nonscan cell which has been selected to be treated as a virtual scan cell.

type

Indicates the primitive type on the reported virtual scan cell. Possible primitive types are DLAT and DFF.

instance_name

Indicates the [instance name](#) of the virtual scan cell.

See Also

- [report_scan_ability](#)

Output From the report_scan_cells Command

You can use output from the [report_scan_cells](#) command to view scan cell information for selected scan cells. The syntax for this command is as follows:

```
report_scan_cells < <chain_name [position]> | -shadows | -all >  
[-pins] [-max d] [-reverse] [-verbose]
```

Standard Format

chain	cell	type	inv	gate#	instance_name (type)
c1	0	MASTER	IN	147	/reg4/r (N_LAT)
c1	1	MASTER	IN	145	/reg3/r (N_LAT)
		DSLAVE	IN	146	/reg4/lat1 (P_LAT)
c1	2	MASTER	NI	143	/reg2/r (N_LAT)
		SCANTLA	IN	144	/reg3/lat1 (P_LAT)

chain

Indicates the chain name in which the reported scan cell resides.

cell

Indicates the cell position in the scan chain of the reported scan cell. Position 0 indicates the scan cell closest to the scanout pi, position 1 is adjacent to it, and so forth.

type

Indicates the state element type of a primitive associated with the reported scan cell. The possible choices are MASTER, SLAVE, SHADOW, OBS_SHADOW (observable shadow), DSLAVE (dependent slave or lockup latch), and SCANTLA (scannable transparent latch).

inv

Indicates the inversion relationships of the primitive associated with the reported scan cell. The inversion data consists of 2 characters each of which can be N (not inverted) or I (inverted). The first character indicates the inversion relationship of the primitive to the chain input port. The second character indicates the inversion relationship of the primitive to chain output port. Additional explanation of this inversion reference can be found in the topic: [Troubleshooting Pattern Simulation Failures](#)

gate#

Indicates the [primitive ID](#) number of the gate.

instance_name

Indicates the [instance name](#) of the gate.

(type)

Indicates the name of library module from which the primitive was derived.

-Pin Format

chain	cell	type	inv	gate#	instance_name (type)
c1	0	MASTER	IN	147	/reg4/r (N_LAT)
		input	I	147	/reg4/r/D (N_LAT)
		output	N	147	/reg4/r/Q (N_LAT)
c1	1	MASTER	IN	145	/reg3/r (N_LAT)
		DSLAVE	IN	146	/reg4/lat1 (P_LAT)
		input	I	145	/reg3/r/D (N_LAT)
c1	2	output	N	146	/reg4/lat1/Q (P_LAT)
		MASTER	NI	143	/reg2/r (N_LAT)
		SCANTLA	IN	144	/reg3/lat1 (P_LAT)
		input	N	143	/reg2/r/D (N_LAT)
		output	N	144	/reg3/lat1/Q (P_LAT)

Description

The -pin format includes all data reported in the standard format plus two additional entries for each scan cell that identify its input and output pins. The type field is used to indicate either input or output. The inversion field consists of only one character. For the input pin, this indicates the inversion of the pin relative to the chain input. For the output pin, this

indicates the inversion of the pin relative to the chain output. The full pin pathname of the pin is given in the instance_name field.

Verbose Format

```

chain cell type   edge inv gate# clocks                               instance_name
(type)
-----
C1      4096 MASTER TE    NN  1647 - c1mhz - tck - rst      /B/id_0 (SF1)
C1      4095 MASTER TE    NN  1646 - c1mhz - tck - rst      /B/id_4 (SF1)
C1      4094 MASTER TE    NN  1648 - c1mhz - tck - rst      /B/swait (SF2)
C1      4093 MASTER TE    NN  1678 - c1mhz - tck - rst      /B/th_iq_0 (SF1)
:      :      :      :      :      :
C1      4078 MASTER TE    NN  1663 - c1mhz - tck - rst      /B/tal_iq_7
(SF1)
C1      4077 MASTER LE    NN  5810 + tck                               /A/tdo_lock
(SF4)
        DSLAVE TE    NN  4275 - c1mhz - tck - rst
        /A/C/mblk/peak/dd15 (SF1)
C1      4076 MASTER TE    NN  4295 - c1mhz - tck - rst
        /A/C/mblk/peak/dd14 (SF1)
C1      4075 MASTER LS    NN  1309 + c1mhz + tck + rst
        /A/C/mblk/peak/LOCKUP2 (LAT1)
        DSLAVE TE    NN  2224 - tck - c1mhz - rst      /A/C/sblk/aga0
(SF2)
C1      4074 MASTER TE    NN  2223 - tck - c1mhz - rst      /A/C/sblk/aga1
(SF1)
:      :      :      :      :      :
C1      3950 MASTER TE    NN  4291 - c1mhz - tck - rst
        /C/peak/current_19 (SF1)
C1      3949 MASTER LS    NN  1308 + c1mhz + tck + rst      /C/peak/LOCKUP3
(LAT1)
        DSLAVE TE    NN  4048 - tst32k - c1mhz
        /A/C/micro_blk/egdma/save0 (SF1)
C1      3948 MASTER TE    NN  4047 - tst32k - c1mhz
        /A/C/micro_blk/egdma/save
:      :      :      :      :      :
C1      3771 MASTER LE    NN  1861 + c1mhz + tck + rst      /B/rgf_8 (SF3)
C1      3770 MASTER LE    NN  1858 + c1mhz + tck + rst      /B/rgf_7 (SF3)
        SCANTLA UP  NN  0832 ? c1mhz ? tck ? rst      /B/rgf_LOCKUP
(LAT2)
C1      3769 MASTER LE    NN  1811 + c1mhz + tck + rst      /B/ci8_6 (SF3)
:      :      :      :      :      :
C1      3557 MASTER LE    NN  1945 + c1mhz + tck + rst      /B/rgd_4 (SF3)
        SHADOW LE    IN  1751 + c1mhz + tck + rst      /B/rgd_3 (DF2)
C1      3556 MASTER LE    NN  1948 + c1mhz + tck + rst      /B/rgd_2 (SF3)
        SHADOW LE    IN  1753 + c1mhz + tck + rst      /B/rgd_1 (DF2)
:      :      :      :      :      :
C1      160  MASTER LE    NN  5812 + c1mhz + tck + rst
        /dbsn/cb_d_2/scan_out (SF4)
    
```


Chapter 35: Types of Reports

Output From the report_scan_cells Command

```

                SHADOW LS    NN  1482  + tck
/dbsn/cb_d_2/data_in (LAT1)
C1      159 MASTER LE    NN  5916  + clmhz + tck + rst
/dbsn/cb_h/scan_out (SF4)
                SHADOW LS    NN  1506  + tck
/dbsn/cb_b_6/data_in (LAT1)
                SHADOW LS    NN  1582  + tck
/dbsn/cb_h/data_in (LAT1)
                SHADOW LS    NN  1596  + tck
/dbsn/cb_b_3/data_in (LAT1)
                SHADOW LS    NN  1597  + tck
/dbsn/cb_b_4/data_in (LAT1)
                SHADOW LS    NN  1598  + tck
/dbsn/cb_b_5/data_in (LAT1)
                SHADOW LE    IN  2106  + tck + clmhz + rst
/C/clockgen/kprev (DF1)
:      :      :      :

```

The `-verbose` format includes all data reported in the standard format plus the following additional columns:

edge

Indicates the clock edge on which the device captures during a scan shift. **LE** indicates leading edge (a 0->1 transition for a clock with off state 0, or 1->0 transition for a clock with off state 1). **TE** indicates trailing edge. **LS** indicates the device is level sensitive and not edge sensitive. **UP** indicates unstable pin, when all clocks are off, a clock pin of the state element is unstable.

clocks

Indicates all clock names and associated polarity which affect the contents of the state element. The identification of clocks considers only true clock inputs of DFFs/DLATs and ignores the set/reset inputs. A polarity of "+" indicates a 0->1 transition of this clock pin at the top level of the chip causes a capture. A polarity of "-" indicates a 1->0 transition of the clock at the top level causes a capture. A polarity of "?" indicates the polarity is not given, as for a scan **TLA** device.

See Also

- [report_scan_cells](#)
- [Scan Cells](#)

Output From the report_scan_chains Command

You can use output from the `report_scan_chains` command to view scan chain information. The syntax for this command is as follows:

```
report_scan_chains [-verbose]
```

Standard Format

chain	group	length	input_pin	output_pin
c1	sg0	230	/scanin1	/scanout1
c2	sg0	225	/scanin2	/scanout2
c3	sg0	230	/scanin3	/scanout3

chain

Indicates the name of the chain to be reported.

group

Indicates the group name associated with the report chain. The group name is normally internally generated and is of the form "sgN" where N is the unique group ID number. Scan chains in the same group are assumed to be loaded and unloaded in parallel. When multiple scan groups are defined in the DRC file, the group name will appear here.

length

Indicates the number of scan cells in the reported chain.

input_pin

Indicates the pin name of the chain input pin.

output_pin

Indicates the pin name of the chain output pin.

Verbose Format

chain	group	#LE/#TE/#LS	input_pin	output_pin	clocks
c1	sg0	230/0/1	/a_scanin1	/a_scanout1	clk1 rst clk2 #dslaves=2
c2	sg0	200/25/0	/a_scanin2	/a_scanout2	clk2 #shadows=14
c3	sg0	115/115/0	/a_scanin3	/a_scanout3	phi_1 phi_2
c4	sg0	224/0/0	/a_scanin4	/a_scanout4	clk1 set rst #scantlas=3
c5	sg0	0/175/0	/a_scanin5	/a_scanout5	nclk

chain

Indicates the name of the chain to be reported.

group

Indicates the group name associated with the report chain.

#LE/#TE/#LS

Provides a breakdown of the master cells in the scan chain into how many are leading edge (LE), trailing edge (TE), or level sensitive (LS). The sum of these will equal the total scan length reported in the standard report.

input_pin

Indicates the pin name of the chain input pin.

output_pin

Indicates the pin name of the chain output pin.

clocks

Provides a list of all clocks associated with master and other cells in the scan chain. This list includes any clock physically attached to the cells, including asynchronous set, reset, shift, and capture operations. In addition, if the scan chain contains any DSLAVE, SHADOWS, OBSERVABLE SHADOWS, or scan transparent latch devices, the count of these types of cells will appear in this area.

See Also

- [Understanding Compressor Connections in the Output from report_scan_chains](#)
- [report_scan_chains](#)
- [Scan Cell Types](#)

Output From the report_scan_path Command

You can use output from the report_scan_path command to view the gates in a segment of a scan chain path. The syntax for this command is as follows:

```
report_scan_path chain_name <SCO | cell_position> <SCI | cell_position>
```

Standard Format

```
Scan path for chain=c1: begin_position=SCO, end_position=SCI  
/SDO2 (148) PO (_PO)
```

Chapter 35: Types of Reports

Output From the report_scan_path Command

```

    --- I 137-/reg4/Q
    SDO2 O
PO usage: scanout (c1)
/reg4 (137) BUF (DFFP)
    --- I 147-/reg4/r/Q
    Q O 148-SDO2
/reg4/r (147) DLAT (N_LAT)
    !SB I (/TIE_1)
    !RB I (/TIE_1)
    CK I 14-
    D I 146-/reg4/lat1/Q
    Q O 137-
scan_behavior: MASTER(LS/-) chain=c1 cell_id=0 invert_data=IN obs=noproc
/reg4/lat1 (146) DLAT (P_LAT)
    !SB I (/TIE_1)
    !RB I (/TIE_1)
    CK I 19-
    D I 116-/reg3/Q
    Q O 147-/reg4/r/D
scan_behavior: DSLAVE(LS/-) chain=c1 cell_id=1 invert_data=IN

```

Verbose Format

```

TEST> report_scan_path big_chain 1 2 -verbose
    Scan path for chain=big_chain: begin_position=1,
end_position=2
    dff1/base_dff1 (15) DFF (base_dff)
set      P I (TIE_0)
reset    P I (TIE_0)
clk      I 6-
data     I 13-dff1/base_mux0/out
q        O 12-/dff0/base_mux0/d1
8-/gate5/in ...
scan_behavior: MASTER(LE/-) chain=big_chain cell_id=1 invert_data=NN
obs=noproc
set_reset_usage: set=no, reset=no, unstable_flag=no,
set_dominance=clocks_only, reset_dominance=clocks_only dff1/base_mux0
(13) MUX (base_mux)
sel      I 2-se
d0       I 8-gate5/out
d1       I 16-dff2/base_dff1/q
out      O 15-/dff1/base_dff1/data
dff2/base_dff1 (16) DFF (base_dff)
set      P I (TIE_0)
reset    P I (TIE_0)
clk      I 6-
data     I 5-dff2/base_mux0/out
q        O 13-/dff1/base_mux0/d1
scan_behavior: MASTER(LE/-) chain=big_chain cell_id=2 invert_data=NN
obs=noproc

```

```
set_reset_usage: set=no, reset=no, unstable_flag=no,  
set_dominance=clocks_only, reset_dominance=clocks_only
```

Description

The first line of the scan path report identifies the beginning and ending positions of the scan chain of the reported scan path. The scan chain path is displayed backward beginning at the selected starting scan cell (SCO indicates begin point is chain output) and ending at the selected ending scan cell (SCI indicates end point is chain input). The gates in the scan path are displayed in the standard primitive report format. The scan path report does not report gates which do not have a single sensitized path to its preceding scan path gate.

Note that the `scan_behavior` line contains an `invert_data` statement, as shown in bold in the following example:

```
scan_behavior: MASTER(LE/-) chain=big_chain cell_id=2 invert_data=NI  
obs=noproc
```

The `invert_data` statement in the example displays net inversion data in a two-character format: the "N" character refers to normal (non-inverting) behavior, and the "I" character refers to inverting behavior. The first character of the sequence corresponds to the area of the scan cell between the scan-in port and the `_DFF` primitive. The second character corresponds to the area of the scan cell between the primitive to the scan-out port.

In the following example, net inversion occurs from the scan-in port to the `_DFF` primitive of the scan cell, and the scan cell behavior is normal (non-inverting) from the primitive to the scan-out port:

```
invert_data=IN
```

The behavior can be the same in both locations of the scan cell, as shown in the following examples:

```
invert_data=NN  
invert_data=II
```

See Also

- [report_scan_path](#)
- [Primitives Report](#)

Output From the report_settings Command

You can use output from the `report_settings` command to view the current settings defined by any of the set commands. The syntax for this command is as follows:

```
report_settings
```

Standard Format

```
atpg =
allow_clockon_measures=no, abort_limit=10,
capture_cycles=0, coverage=100.00, decision=norandom,
di_analysis=yes, full_seq_abort_limit=10, full_seq_atpg=no,
full_seq_time=(10.0,0.0), full_seq_merge=off, merge=off,
basic_min_detects_per_pattern=(0,0),
fast_min_detects_per_pattern=(0,0),
full_min_detects_per_pattern=(0,0),
new_capture=no, patterns=0, prevention=norandom,
random_fill=yes, store=yes, summary=yes, time=(0.0,0.0),
verbose=no, post_capture_contention_prevention=no,
chain_test=0011;

debug=off, hex=yes, max_intervals=1000, max_pattern_cells=250,
max_seed_patterns=16, max_total_cells=250,
min_lfsr_length=20,
multi_seeds_per_interval=no, num_,
num_patterns_per_interval=256, randomize_pis=no,
chain_test=no,
sim_misr=yes, verbose=no, dump=none,
pi_assumed_scan=no, po_assumed_scan=no,
use_cell_constraints=no, use_constant_value_cells=no;
build = add_buffer=yes, delete_unused_gates=yes, fault_boundary=lowest,
hierarchical_delimiter='/', limit_fanout=256,
undriven_bidi=PIO,
net_connections_change_netlist=yes,
merge: bus_keepers=yes
cascaded_gates_with_pin_loss=no
equivalent_dlat_dff=on
wire_to_buffer=yes
feedback_paths=yes
flipflop_from_dlat=yes
mux_from_gates=pin-preserve
tied_gates_with_pin_loss=no
wire_to_buffer=yes
xor_from_gates=pin-preserve
buses = external_Z=Z, fault_contention=TIEX
commands = abort=yes, history=yes
contention = allow_multiple_drivers_on=yes, atpg=yes, bidi=yes, bus=yes,
dff_dlat=no, float=no, pre_capture_clock_check=yes,
post_capture_clock_check=yes, ram=no,
retain_bidi_direction=no,
severity=warning, verbose=no, wire=no;
delay = diagnostic_propagation=no, launch_cycle=any_launch,
mask_nontarget_paths=no,
pi_changes=yes, po_measures=yes, relative_edge=no,
robust_fill=yes,
simulate_hazards=yes, allow_reconverging_paths=no;
drc = test_proc_file=none,
```

Chapter 35: Types of Reports

Output From the report_settings Command

```

allow_unstable_set_resets=no, bidi_control_pin=no,
clock=any,
controller_clock=no, disturb_clock_grouping=yes,
initialize_dff_dlat=X, multi_captures_per_load=yes,
oscillation=500,
remove_false_clocks=no, shadows=on,
skew=1, store_setup=no, store_stability_patterns=no,
TLAs=yes, trace=off,
unstable_lsrams=no, z_check_with_constraints=no
observe_procedure=any, pipeline=no;
faults = atpg_effectiveness=no, au_credit=0, equiv_code=(--),
fault_coverage=no, model=stuck, report=uncollapsed,
summary=noverbose;
iddq = atpg=no, float=no, strong=yes, toggle=no, weak=yes, write=yes
exclude_ports=all;
learning = atpg_equivalence=on(sim_passes=32, test_passes=5000),
common_input=yes, equivalent_latches=yes,
implication=medium,
max_feedback_sources=100, verbose=no;
match_names = match_names options are not set
messages = display=yes, double_slash=no, level=standard,
transcript_comments=yes,
logfile=off
netlist = celldefine=yes, check_only_used_udps=yes, dominance=on,
conservative_mux=combination
enable_portfaults=yes, escape=cond, max_errors=10,
pin_assign=no, redefined_module=last, scalar_net=no,
sequential_modeling=no, suppress_faults=yes,
xmodeling=yes;
patterns = source=internal,
histogram_summary=no, load_summary=no,
verilog_lastscan=yes;
physical = verbosity_level=1 (default)
pindata = none;
-shift_character=X, -constrain_character=X;
primitive_report= interval=0, max_fanout=2, time=clock, verbose=no;
random_patterns = clock=none, length=1024, observe_type=master;
simulation = basic_scan=yes, bidi_fill=off, data=(0:-1), measure=pat,
oscillation=(10,2), store_memory_contents=no,
words_per_pass=32, xclock_give_xout=no, verbose=no;
wgl = chain_list=all, forces_during_load=previous, group_bidis=no,
inversion_reference=master, last_scan=yes,
macro_usage=no, pad=no,
scan_map=dash, scan_data_format=pre_measured,

bidi_map=(Z0,-0) (Z1,-1) (0X,0-) (1X,1-) (XX,X-) (ZX,-X) (-X,--) (ZZ,-Z) (Z-,--)
workspace sizes = connectors=20000, decisions=5000, line=50000,
string=2048,

ydf = YDF schema is not set

```

Description

This report gives the current setting for most options selected with a "set" command.

See Also

- [report_settings](#)

Output From the report_summaries Command

You can use output from the [report_summaries](#) command to view primitive, fault, pattern, library cell, memory, optimization, sequential depth, or CPU usage summaries.

Standard Format

```
TEST> set_faults -fault_coverage
TEST> report_summaries faults
      Uncollapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   144501
Possibly detected    PT    4224
Undetectable         UD    1634
ATPG untestable     AU    9335
Not detected         ND    6811
-----
total faults 166505
test coverage 91.53%
fault coverage 86.78%
-----
```

This report gives the current fault summary report followed by the current pattern summary report.

Verbose Format

```
TEST> set_faults -summary verbose
TEST> report_summaries
      Collapsed Fault Summary Report
-----
fault class          code #faults
-----
Detected             DT   28689
  detected_by_simulation DS  (1822)
  detected_by_implication DI (26867)
Possibly detected    PT    617
  not_analyzed-pos_detected NP  (617)
Undetectable         UD    976
```


Chapter 35: Types of Reports

Output From the report_summaries Command

```

undetectable-unused          UU      (8)
undetectable-tied            UT     (505)
undetectable-blocked         UB     (463)
ATPG untestable              AU     6795
  atpg_untestable-not_detected AN     (6795)
Not detected                  ND    102341
  not-controlled             NC     (6171)
  not-observed               NO    (96170)
-----
total faults 139418
test coverage 20.95%
-----
                          Pattern Summary Report
-----
#internal patterns          25
  #basic_scan patterns      25
-----

```

The verbose report displays the 11 classes of faults that make up the five major fault categories. These numbers are enclosed in parenthesis "(" and sum to produce the number of the major category. Fault classes with zero counts are not listed.

Primitives Report

```

TEST> report summaries primitives
                          Gate Summary Report
-----

```

```

#primitives                101071
#primary_inputs             251
#primary_outputs           148
#primary_bidis              128
#DLATs                      227
  #TLAs                      195
  #nonscan                   32
#DFFs                       11369
  #nonscan                   416
  #scan                      10953
#BUSs                        577
  #contention_fails          289
#BUFs                        4108
#INVs                        6425
#ANDs                        9740
#NANDs                       21086
#ORs                         5689
#NORs                        2981
#XORs                        2207
#XNORs                       944
#TIEOs                      11357
#MUXs                       20017
#WIRES                       23

```

```
#BUFZs          138
#TSDs           3656
-----
```

The primitives report list the breakdown of ATPG primitives in the flattened model by types. Indented values such as #TLAs and #nonscan sum to form the entry just above them #DLATs. The total count of all ATPG primitives is given by #primitives.

Library Cells Report

```
TEST> report_summaries library
      Library Cells Report
-----
```

```
Cell name used
-----
```

```
an2p1      2589
an2p2       111
an8p1        58
aoi211p1    376
fdrtp1     4416
na4p1         8
no2p1     4593
no7p1         4
no8p1        13
oai31p1     82
or2p8        16
or7p2         1
or8p1        53
sfdrstp1    134
-----
```

The library cells report summaries the number of library cells used by the flattened design list by cell name. The determination of a library cell vs. a hierarchical design module is not an foolproof process and so this report could be slightly off by missing some cells. Your ASIC vendor generally provides similar functionality in their tools which are more accurate.

Optimizations Report

```
TEST> report_summaries optimizations
      Optimizations Report
-----
```

```
optimization #occurrences #primitives #pins #modules
type          eliminated   lost   optimized
-----
unused gates 105129         105129    759     74
tied gates   0                24      0      0
buffers     220542          220542     0    560
```

Chapter 35: Types of Reports

Output From the report_summaries Command

inverters	64131	64131	0	209
cascaded gates	808	808	0	6
SWs as BUFs	24	0	0	1
DLATs as BUFs	0	0	0	0
MUXs	22061	109195	45	33
XORs	0	0	0	0
equiv. DLAT/DFE	43	0	0	10
DLATs as DFFs	0	0	0	0
DFFs as DLATs	0	0	0	0
BUS keepers	0	0	0	0
feedback paths	0	0	0	0

total	412738	499829	804	590

The optimizations summary report provides details of the number of gates removed during optimization process as well as whether there were any fault sites dropped (#pins lost) during this effort.

Sequential Depths Report

```
TEST> report_summaries sequential_depths
type      depth gate_id
-----
Control   7      91788
Observe   5      1496
Detect    10     1496
```

The sequential depths report provides the maximum sequential depth found for control, observe, and detection of a fault location. A representative [gate ID](#) corresponding to this maximum is provided. There can be additional locations with equal depth but only one is given.

See Also

- [Coverage Calculations](#)
- [Fault Classes](#)
- [report_faults](#)
- [report_primitives](#)
- [report_patterns](#)

- [set_faults](#)
- [Understanding Flattening Optimization](#)

Output From the report_version Command

You can use output from the report_version command to identify the current version of TestMAX ATPG or the virtual address space in which TestMAX ATPG is executing. The syntax for this command is as follows:

```
report_version
  [-full | -short | -address]
  [-banner]
  [-verbose]
```

Standard Format

Indicates the program major version.

```
BUILD-T> report_version
H-2013.03-SP5-CS1-tcl
```

Full Format

Indicates the major version, the compile date, and the hour, minute, and second of the last compile.

```
BUILD-T> report_version -full
H-2013.03-SP5-CS1-i131004_180753-tcl
```

Short Format

Indicates the program internal version and represents the date the program was last compiled.

```
BUILD-T> report_version -short
i131004-tcl
```

Address Format

Reports whether the executable is using 32-bit or 64-bit virtual address space. A 32-bit executable can use at most 2.1 to 3.8 GB of virtual memory, depending on the platform. So, your design plus libraries plus patterns must all fit in this range of memory. A 64-bit executable does not have this limit and is typically limited by the size of physical RAM plus swap space of the workstation on which it is running.

```
BUILD-T> report_version -address  
32bit executable
```

Banner Format

Generates the product name and the full version report. This consists of the product name (TestMAX ATPG), short version string, the date, and the time of day stamp of the currently running version.

```
BUILD-T> report_version -banner  
TestMAX ATPG (TM) H-2013.03-SP5-CS1-i131004_180753 -tcl
```

Verbose Format

Generates the product name and the full version report (same as the `-banner` option), followed by the date version for subsystem tasks.

```
BUILD-T> report_version -verbose  
TestMAX ATPG (TM) H-2013.03-SP5-CS1-i131004_180753  
util020306 edif102400 udp030603 otddl11798 veri091708 net031604  
flat121611  
stil032009 syn121400 vhd1120203 wgl100308 api101507 vcde_102602  
tpapi040700 lbist111207-tcl
```

See Also

- [report_version](#)

Output From the report_violations Command

You can use output from the `report_violations` command to view rule violation data. The syntax for this command is as follows:

```
report_violations <violation_id | rule_id | rule_type  
| -all> [-max d]
```

Standard Format

```
Warning: Unconnected module input pin ( N_LAT/NR ).  
(B8-1)  
Warning: Undriven module internal net ( DFFP/notify_reg ).  
(B9-1)  
Warning: Undriven module internal net ( DFFRLP/notify_reg ).  
(B9-2)  
Warning: Nonscan DLAT /reg0/lat1 (138) disturbed during  
time 0 of load_unload procedure. (S19-1)
```

```
Warning: Nonscan DLAT /reg2/lat1 (142) disturbed during  
time 0 of load_unload procedure. (S19-2)
```

This report displays the rule violation occurrence message for selected violations. The message is preceded by text that indicates the severity of the violation (either Warning: or Error:).

See Also

- [report_violations](#)

Output From the report_wires Command

You can use output from the report_wires command to view data associated with wire gates.

```
report_wires [id | -summary | -all ]  
[-contention <fail | pass | abort>] [-max d] [-verbose]
```

Summary Format

```
Wire summary: #wire_gates=2  
Contention status: #pass=2, #fail=0, #abort=0,  
#not_analyzed=0
```

#wire_gates

Indicates the total number of WIRE gates in the simulation model.

Contention status

Indicates the number of WIRE gates that are in each WIRE contention category.

Standard Format

```
TEST> report_wires -all  
      contention      zstate #drivers  
gate_id status/capture status strong/weak behavior_data  
-----  
      79 pass ...          ...      4 0  clock  
     206 pass ...          ...      5 0
```

gate_id

Indicates the [primitive ID](#) number of the WIRE gate.

contention status

Indicates the contention status category of the WIRE gate.

contention capture

This field is not used for WIRE gates.

zstate status

This field is not used for WIRE gates.

#drivers strong/weak

Indicates the number of strong drivers followed by the number of weak drivers that are connected to the WIRE gate.

behavior_data

This lists the learned behavior information for the WIRE gate. This includes logical behavior and connectivity to clocks.

Verbose Format

```
TEST> report_wires -all -verbose
      contention      zstate #drivers
gate_id status/capture status strong/weak behavior_data
-----
   79 pass ...          ...      4 0   clock
      inputs: 75 (BUF) 76 (BUF) 77 (BUF) 78 (BUF)
  206 pass ...          ...      5 0
      inputs: 203 (BUF) 204 (BUF) 205 (BUF) 201 (BUF) 202 (BUF)
```

gate_id...

All data contained in the standard report is also included in the verbose report.

inputs:

This identifies the gates connected to all inputs of the WIRE gate. Each entry gives the [primitive ID](#) number followed by the primitive type in parentheses.

See Also

- [report_wires](#)
- [Wire Gate](#)

Output From the analyze_buses Command

You can use output from the analyze_buses command to analyze potential problems associated with buses. The syntax for this command is as follows:

```
analyze_buses <gate_id | -all>
[-exclusive [first | all] | -prevention | -zstate ] [-update]
```

Standard Format - defaults

```
analyze_buses 28510
Bus 28510 (28406,28111) failed contention check
(values available in pattern 0).
```

text

The text indicates the results of performing contention ability checking on the bus gate with gate_ID number 28510. The bus gate failed this check which indicates a pattern could be created that resulted in a contention condition on the analyzed bus gate. In this case, the contention occurred on the two bus drivers with gate_ID numbers 28406 and 28111. The internal gate values for this pattern are displayable by selecting the pin_data to be "pattern 0".

Standard Format - zstate

```
analyze_buses 3640 -zstate
BUS analysis required 0 remade decisions.
Bus 3640 failed Z-state ability check (values available in pattern 0).
```

text

The text indicates the results of performing Z-state ability checking on the bus gate with gate_ID number 3640. The bus gate failed this check which indicates a pattern could be created that resulted in a Z state condition on the analyzed bus gate. In this case, test generation successfully created this pattern with no remade decisions. The internal gate values for this pattern are displayable by selecting the pin_data to be "pattern 0".

Standard Format - exclusive

```
analyze_buses -all -exclusive first
Bus Contention results: #pass=68, #bidi=34, #fail=0, #abort=0, CPU
time=129.45
```


text

The text summarizes the contention ability analysis for all bus gates. The fields are:

- #pass - The number of bus gates which are proven incapable of contention.
- #bidi - The number of bidirectional bus gates which are otherwise incapable of contention.
- #fail - The number of bus gates which are proven capable of contention.
- #abort - The number of bus gates whose contention ability analysis was aborted.
- CPU time - The total CPU time in seconds required to perform the contention ability analysis.

Standard Format - prevention

```
analyze_buses -all -prevention
BUS analysis required 0 remade decisions.
Busses passed contention prevention (values available in pattern 0).
```

text

The text indicates the results of performing test generation to satisfy the conditions required to prevent contention on bus gates which are contention sensitive. In this case, the test generation successfully created a pattern that satisfied these conditions with no remade decisions. The internal gate values for this pattern are displayable by selecting the pin_data to be "pattern 0".

See Also

- [report_buses](#)
- [set_contention](#)

Output From the analyze_faults Command

You can use output from the analyze_faults command to determine why certain faults are not detected. The syntax for this command is as follows:

```
analyze_faults < < pin_pathname -stuck <0|1> [-observe gate_id]
[-display] > | <-class fault_class>... > > [-max max_faults]
[-source_points min_percent max_points] [-verbose]
```

Standard Format for Blocked pin_pathname

```
analyze_faults /i0/MUX_RST_DPE/A -stuck 0
-----
Fault analysis performed for /i0/MUX_RST_DPE/A stuck at 0
(input 1 of MUX gate 4133).
Current fault classification = AN (atpg_untestable-
not_detected).
-----
Connection data: to=MASTER,RESET from=CLOCK
Fault is blocked from detection due to constrained values.
Blockage point is gate /i0/MUX_RST_DPE (4133).
Source of blockage is gate /TEST (11).
```

Connection data:

The connection data indicates connectivity between the fault site and potential observe points and trouble points. In this case, the fault site can only be observed at a master element of a scan cell. It also connects forward to the reset input of a state element and connects backward to a defined clock pin.

Blockage point

The analysis determined that the fault was untestable due to a blockage that was the result of some constrained value. The gate at which the blockage occurred is displayed and indicates the point at which it becomes impossible to further propagate a fault effect.

Source of blockage

The analysis determined that the blockage was due to value constraints. The gate that was the origin of the constraint is given. Generally, these are constrained pins, tie gates, or state elements which have restricted behavior for the current ATPG process.

Standard Format for Successful pin_pathname

```
analyze_faults /U1/Q -stuck 0
-----
Fault analysis performed for /U1/Q stuck at 0 (output of BUF gate 63).
Current fault classification = DS (detected_by_simulation).
-----
Connection data: to=REGPO,MASTER,TS_ENABLE
Fault site control to 1 was successful (data placed in parallel pattern
0).
Observe_pt=7277(DFF) test generation was successful (data placed in
parallel
pattern 1).
```

The gate_report data is now set to "pattern:1".

Connection data:

The connection data indicates connectivity between the fault site and potential observe points and trouble points. In this case, the fault site can be observed at a regular primary output (not connected to a clock) or a master element of a scan cell. It also connects forward to enable input of a tristate gate (TSD or SW). The fault site had no backward connectivity to a potential trouble point.

Fault site control

The fault analysis successfully created a pattern that set the fault site to the required value which for this fault was a "1" state. The simulated values for this pattern at any selected gate can be displayed by setting pindata to "pattern 0".

Observe Point

The fault analysis successfully created a pattern that detected the fault at the indicated observe point (a DFF gate with gate_id number 7277). The simulated values for this pattern at any selected gate can be displayed by setting pindata to "pattern 1".

gate_report data

The pindata setting used when displaying gates is set to the pattern which detected the fault (pattern 1). The automated display will select all gates in a path between the fault site and the observe point with their simulated values that result from that pattern.

Standard Format for Class

```
TEST> analyze_faults -class an
Fault analysis summary: #analyzed=2375, #unexplained=17.
1660 faults are untestable due to constrain values.
674 faults are untestable due to constrain value blockage.
 25 faults are connected from CAPTURE_CHANGE.
194 faults are connected to CLKPO.
106 faults are connected to CLOCK.
116 faults are connected from CLOCK.
141 faults are connected to DSLAVE.
  6 faults are connected to FAIL_WIRE.
  2 faults are connected from RAM.
  9 faults are connected to RAM_ADR.
22 faults are connected to RAM_DATA.
  4 faults are connected to RAM_WRITE.
84 faults are connected to RESET.
48 faults are connected to SET.
11 faults are connected to TLA.
```

Chapter 35: Types of Reports

Output From the analyze_faults Command

```
22 faults are connected from TLA.  
189 faults are connected to TS_ENABLE.  
58 faults are connected to FAIL_TE.
```

#analyzed

This is the total number of collapsed faults that were analyzed.

#unexplained

This is the total number of analyzed collapsed faults that could not be clearly associated with with any distinct type of problem.

due to constrain values

This is the total number of collapsed faults that were identified as untestable due to constrained values. These constraints are normally due to PI constraints, tie gates, or restricted behavior of state elements.

due to constrain value blockage

This is the total number of collapsed faults that were identified as untestable due to observe blockages that resulted from constrained values.

connected to/from CAPTURE_CHANGE

This is the total number of collapsed faults that had connectivity to a **TE**(state element that captures on a clocks trailing edge) which is driven by a **LE** (state element that captures on the same clocks leading edge).

connected to CLKPO

This is the total number of collapsed faults that had connectivity to primary outputs which have connection to clock input pins.

connected to/from CLOCK

This is the total number of collapsed faults that had connectivity from clock pins.

connected to DSLAVE

This is the total number of collapsed faults that had connectivity to dependent slave scan cell state elements.

connected to FAIL_WIRE

This is the total number of collapsed faults that had connectivity to WIRE gates which failed the contention ability checking (capable of contention).

connected to/from RAM

This is the total number of collapsed faults that had connectivity to or from RAM MEMORY gates.

connected to RAM_ADR

This is the total number of collapsed faults that had connectivity to RAM_ADR gates.

connected to RAM_DATA

This is the total number of collapsed faults that had connectivity to RAM_DATA gates.

connected to RAM_WRITE

This is the total number of collapsed faults that had connectivity to a write input of a RAM MEMORY gate.

connected to/from RESET

This is the total number of collapsed faults that had connectivity to a reset input of a DLAT or DFF gate.

connected to/from SET

This is the total number of collapsed faults that had connectivity to a set input of a DLAT or DFF gate.

connected to/from TLA

This is the total number of collapsed faults that had connectivity to or from a [transparent latch](#).

connected to TS_ENABLE

This is the total number of collapsed faults that had connectivity to an enable input of a TSD or SW gate.

connected to FAIL_TE

This is the total number of collapsed faults that had connectivity between a LE (leading edge) sensitive element and a TE (trailing edge) sensitive element.

Blockage and Constraint Value Source Points Format

```
TEST-T> analyze_faults -class AN -source_points 5 0
Fault analysis summary: #analyzed=802, #unexplained=0.
420 faults are untestable due to constrain values.
382 faults are untestable due to constrain value blockage.
22 faults are connected to DSLAVE.
2 faults are connected to TIEX.
62 faults are connected to TLA.
28 faults are connected to CLOCK.
6 faults are connected to RESET.
34 faults are connected to NONSCAN_LOAD.
```

Chapter 35: Types of Reports

Output From the analyze_faults Command

```

6 faults are connected from CLOCK.
22 faults are connected from TLA.
126 faults are connected from TIEX.

Blockage Points - Total 145:
-----

Blockage Value Source Points - Total 3:
-----
i_test_se 37.16%

Constraint Value Source Points - Total 27:
-----
test_model 17.08%
i_test_se 5.74%

```

Note in the previous example that the blockage point data does not include pin pathnames. In this case, all 145 blockage points did not exceed the specified threshold of 5% of the explained faults for class AN. The following example shows the number of constrained sources, but not the percentage:

```

TEST-T> analyze_faults -class AN -source_points 0 5
... (same fault counts as before) ...

Blockage Points - Total 145:
-----

duto1/U_clk_control_i_0/pipeline_or_tree_l_reg 0.87%
duto2/U_clk_control_i_0/pipeline_or_tree_l_reg 0.75%
duto1/U_clk_control_i_0/load_n_meta_0_l_reg 0.62%
SNPS_PipeTail_test_so5_1 0.62%
SNPS_PipeTail_test_so6_1 0.62%

Blockage Value Source Points - Total 3:
-----
i_test_se 37.16%
i_pll_bypass 1.00%
i_pll_mode 0.25%

Constraint Value Source Points - Total 27:
-----
test_model 17.08%
i_test_se 5.74%
duto1/U_clk_control_i_0/U_cycle_ctr_i/count_int_reg_0_ 3.99%
duto2/U_clk_control_i_0/U_cycle_ctr_i/count_int_reg_0_ 3.99%
duto1/U_clk_control_i_0/U_cycle_ctr_i/count_int_reg_1_ 3.74%

```

Verbose Format

```

analyze_faults -class au -verbose -max 4
Fault analysis summary: #analyzed=802, #unexplained=1.
349 faults are untestable due to constrain values.

```

Chapter 35: Types of Reports

Output From the report_atpg_constraints Command

```

sa1 AN /i2/i1/U395/B
sa1 AN /i2/i1/DIA_reg_9/D
sa1 AN /i2/i1/DIC_reg_9/D
sa1 AN /i1/i3/U395/B
409 faults are untestable due to constrain value blockage.
sa0 AN /i1/i1/U395/C
sa1 AN /i1/i1/U406/B
sa1 AN /i1/i1/U414/B
sa1 AN /i1/i1/U416/B
141 faults are connected to DSLAVE.
sa1 AN /i3/U30xNDMARQ_CK_MUX/S
sa1 AN /i3/U63/Z
sa0 AN /i3/U63/Z
sa1 AN /i3/U30xNDMARQ_CK_MUX_25/S
11 faults are connected to TLA.
: : : :
: : : :
2 faults are connected from RAM.
sa1 AN /myram/dout[9]
sa0 AN /myram/dout[8]

```

fault paths

When the verbose option is selected, the standard fault data is given for each of the faults that were placed in the associated fault analysis category (up to the user-selectable maximum allowed limit).

See Also

- [set_contention](#)
- [set_faults](#)
- [report_faults](#)

Output From the report_atpg_constraints Command

You can use output from the report_atpg_constraints command to analyze data from the current ATPG constraints list. The syntax for this command is as follows:

```
report_atpg_constraints [-summary | -all] [-max d]
```

Summary Format

```
ATPG constraint summary: #constraints=3, #DRC_constraints=0
```

#constraints

Indicates the total number of ATPG constraints in the current list.

#DRC_constraints

Indicates the number of DRC ATPG constraints in the current list. DRC constraints can only be added/removed in DRC command mode and are considered during the DRC process.

Standard Format

```
name                val DRC site
-----
my_constraint1      1   no  /my_atpg_primitive1 (20217)
my_constraint2      0   no  /core/txrx/gook/U1842 (1911)
my_constraint3      1   yes /core/alu/PIE/U44 (12683)
```

name

Indicates the user-selected name assigned to the ATPG constraint.

val

Indicates the value of the ATPG constraint. It is "0", "1", or "Z".

DRC

Indicates whether the ATPG constraint is to be checked during DRC.

site

Indicates the site of the ATPG constraint. It gives the [pin pathname](#) with the [primitive ID](#) number in parentheses.

See Also

- [report_atpg_constraints](#)

Output From the report_atpg_primitives Command

You can use output from the report_atpg_primitives command to analyze data from the current ATPG primitives list. The syntax for this command is as follows:

```
report_atpg_primitives <atpg_primitive_name | -summary | -all> [-max d]
[-verbose]
```

Summary Format

```
ATPG gate summary: EQUIV=1
```

<gate_type>= *d*

Indicates the total number of ATPG primitives for each used ATPG primitive type.

Standard Format

```
name           gate_id type  inputs
-----
my_atpg_prim1 20217  EQUIV 861 ~990
```

name

Indicates the user-selected name assigned to the [ATPG primitive](#).

gate_id

Indicates the [primitive ID](#) number of the ATPG primitive.

type

Indicates the primitive type of the ATPG primitive. It can be AND, OR, SEL1, SEL01, or EQUIV.

inputs

Indicates the inputs of the ATPG primitive. For each input, the primitive ID number of the connecting primitive is given. A "~" character that precedes a primitive ID number indicates the input is inverted before use.

Verbose Format

```
/my_atpg_prim1 (20217) EQUIV (_EQUIV)
--- I () 861-/pix/fifo/late/reg_stack/U42/Y
--- I () 990-/pix/fifo/late/counter/U232/Y
my_atpg_prim1 0 ()
```

Description

The verbose format gives the standard primitive report for reported ATPG primitives.

See Also

- [report_primitives](#)
- [report_atpg_primitives](#)

Output From the report_buses Command

You can use output from the report_buses command to analyze data associated with BUS primitives.

```
report_buses  
  
< gate_id | -behavior <buf | inv | and | or | xor | xux  
| tie0 | tie1 | tiez> | -bidis | -contention  
  
<fail | pass | abort | bidi> | -keepers | -pull | -weak  
| -zstate <fail | pass | abort | bidi> | -summary | -all >  
  
[-max d] [-verbose]
```

Summary Format

```
Bus summary: #bus_gates=102, #bidi=33, #weak=0,  
#pull=25, #keepers=0  
Contention status: #pass=69, #bidi=33,  
#fail=0, #abort=0, #not_analyzed=0  
Z-state status : #pass=26, #bidi=33,  
#fail=43, #abort=0, #not_analyzed=0  
Learned behavior : none
```

#bus_gates

Indicates the total number of BUS gates in the simulation model.

#bidi

Indicates the number of BUS primitives with an external bidirectional connection.

#weak

Indicates the number of BUS primitives with only weak connections.

#pull

Indicates the number of BUS primitives with both strong and weak connections.

#keepers

Indicates the number of BUS primitives connected to a bus keeper.

Contention status

Indicates the number of BUS primitives that are in each BUS contention category.

Z-state status

Indicates the number of BUS primitives that are in each Z-state ability category.

Learned behavior

Indicates the number of BUS primitives that are in each learned behavior category.

Standard Format

```
          contention zstate #drivers
gate_id status/capture status strong/weak behavior_data
-----
154      pass          stable pass          1 1
206      pass          stable pass          1 1
```

gate_id

Indicates the [primitive ID](#) number of the BUS primitive.

contention status

Indicates the contention status category of the BUS primitive.

contention capture

Indicates whether the BUS primitive can change contention condition after a capture clock due to driver enable line connectivity to scan cells.

zstate status

Indicates the Z-state status category of the BUS primitive.

#drivers strong/weak

Indicates the number of strong drivers followed by the number of weak drivers that are connected to the BUS primitive.

behavior_data

This lists the learned behavior information for the BUS primitive. This includes logical behavior, connectivity to clocks, and bus keeper ability.

Verbose Format

```
          contention zstate #drivers
gate_id status/capture status strong/weak behavior_data
-----
154      pass          stable pass          1 1
inputs: 15 (PI) 153 (wBUF)
206      pass          stable pass          1 1
inputs: 22 (PI) 205 (wBUF)
```

gate_id...

All data contained in the standard report is also included in the verbose report.

inputs:

This identifies the primitives connected to all inputs/bidis of the BUS primitive. Each entry gives the [primitive ID](#) number followed by the primitive type in parentheses. A "w" preceding a primitive type indicates a weak connection.

See Also

- [report_buses](#)
- [Bus Gate](#)

Output From the report_cell_constraints Command

You can use output from the report_cell_constraints command to analyze cell constraints added with the add_cell_constraints command. The syntax for this command is as follows:

```
report_cell_constraints
```

Standard Format

```
type chain pos. site name
-----
1  c10    6  /MAIN/ALU/TP/FI/FIFO/\stat[3]
OX c34    17 /MAIN/BOZ/RT/RTI_1/\reg3[3]
```

type

Indicates the type of the scan cell constraint.

chain

Indicates the chain that contains the constrained scan cell.

pos.

Indicates the position of the constrained scan cell in its chain.

site name

Indicates the name that was used to identify the scan cell. This can be a pathname of a pin that connects to the constrained scan cell or the name of an instance that contains the scan cell. If the cell constraint was not identified by a name, this field is blank.

See Also

- [report_cell_constraints](#)

Output From the report_clocks Command

You can use output from the report_clocks command to reference all pins defined to be clocks. The syntax for this command is as follows:

```
report_clocks [-matrix] [-intclocks] [-verbose]
```

Standard Format

clock_name	off	usage
/CLK	0	master shift
/RSTB	1	master reset

clock_name

Indicates the [port name](#) of the [clock](#).

off

Indicates the [off state](#) of the [clock](#). The off state can be either 0 or 1.

usage

This lists usage information associated with the clock. This includes the names of scan cell memory types that it connects to, whether it is used to perform shifting, and connectivity to set/reset lines.

Matrix Format

The dynamic clock pair group ability can be displayed using the `-matrix` switch. The clock report matrix will give a table where each row indicates the potential grouping relationship of a candidate clock with each of the other candidate clocks.

```

id# clock_name      type  0  1  2  3  4  5  6  7
-----
0   clk             C   --- --A --A --A --A --A --A .
1   iopclk11        C   B-- --- --A BPA BPA BPA BPA .
2   iopclk12        C   B-- B-- --- --A BPA BPA BPA .
3   iopclk21        C   B-- BPA B-- --- --A BPA BPA .
4   iopclk22        C   B-- BPA BPA B-- --- BPA BPA .
5   iopclk31        C   B-- BPA BPA BPA BPA --- --A .
6   iopclk32        C   B-- BPA BPA BPA BPA B-- --- .
7   iopclk41        C   B-- BPA BPA BPA BPA BPA BPA .
8   iopclk42        C   B-- BPA BPA BPA BPA BPA BPA .
9   tx_intf1_clk    C   --- --A --A --A --A --A --A .
10  tx_intf2_clk    C   --- --A BPA --A --- --A BPA .
11  tx_intf3_clk    C   --- --A BPA --A BPA --A --- .
12  tx_intf4_clk    C   -D- BPA BPA --A BPA --A BPA .
13  por             R   --- --A --A --A --A --A --A .
14  rst            SR  --- --- --- --- --- --- --- .
-----
clock 1          clock 2
id# #cells #masks id# #cells #masks masked gates
-----
0    244    8    12   347    0 13628 13629 13630
                                13639 13645 13647
                                13649 14700

```

id#

Indicates the identification number of the clock.

clock_name

Indicates the clock name.

type

Indicates the current [clock](#) type. The clock type can be any combination of C (flip-flops), CW (RAM write control), S (set) or R (reset).

0 1 2...

One column for each clock.

BPA, --A, B--, ...

Indicates the possible relationship between a row clock and a column clock. These relationships are described using three characters --

The first character can be:

'B' indicating a valid before relationship
(row clock can be applied before column clock).

'-' indicating no before relationship.

The second character can be:

'P' indicating a valid parallel relationship.

'D' indicating an allowed parallel relationship that
has an acceptable level of disturbances.

'-' indicating no parallel relationship.

The third character can be:

'A' indicating a valid after relationship
(row clock can be applied after column clock).

'-' indicating no after relationship.

More information is available at the end of the matrix table report if the dynamic clock grouping feature has been enabled with disturbed clock grouping.

id1 id2

Indicates the clock pairs using the clock id#.

#cells

Indicates the number of scan cells.

C1 #masks

Indicates the number of disturbed scan cells clocked by id1, as well as the total number of scan cells clocked by id1.

C2 #masks

Indicates the number of disturbed scan cells clocked by id2, as well as the total number of scan cells clocked by id2.

masked gates

Indicates the gate number of all cells disturbed when id1 and id2 are pulsed simultaneously. If a basic scan pattern pulses both clocks at the same time, then all those cells will capture an 'X'.

Internal Clocks Format

Using the `-INTClocks` option, you can report all internal clocks and all information associated with each internal clock.

```
report_clocks -intclocks

int_clock_instance_name      gate_id off source cycle conditions
-----
ioclk_pll_cntr/U2           654    0    29  0    960=1 (0,4)
                             1    961=1 (0,5)
                             2    962=1 (0,6)
intclk_pll_cntr/U2          700    0    30  0    982=1 (0,4)
                             1    983=1 (0,5)
                             2    984=1 (0,6)
-----
PLL clock_off pattern: 960=0 961=0 972=0 982=0 983=0 984=0
-----
```

int_clock_instance_name and gate_id

Both of these columns correspond. They are the pin declared as an internal clock.

off state

The clock off state (in practice, it is always 0).

source

The gate id declared as the PLL source for the internal clock.

cycle

The clock cycle numbers starting from 0. There are as many as the number defined for the internal clock.

conditions

This column starts with the gate id of the controlling element and the value that it must be set to to enable that pulse. When a clock chain is used, the gate id is for the clock chain flip-flop. The two numbers in parentheses are the `pll_cycle` and the `pll_pulse` for the corresponding internal clock pulse. For each external event (cycle) a number of PLL pulses is simulated and a check is performed to determine which one turns on the internal clock.

PLL clock_off pattern

The pattern that prevents any internal clock pulses. This uses the same gate ids as in the conditions field, in the opposite states.

Verbose Format

A verbose format of report clock can be displayed using the `-Verbose` argument. This report gives detailed information for how a clock is used in the design.

```
report_clocks -verbose
              scan connections      nonscan connections
clock_name   off clock set reset   clock set reset usage
-----
Ram_Clk      0    0    0    0      256    0    0  RAM nonscan_DFF
BPCICLK      0   56    0    0         0    0    0  master shift
Pixel_Clk    0  525    0    0         0    0    0  master shift
nReset       1    0   440    7         0    0    0  PO master set
reset
```

clock_name

Indicates the clock name.

off

Indicates the off state of the corresponding clock.

scan connections

Indicates the number of scan cell ports connected to this clock. The ports taken in account are the clock, set, and reset ports. The number of scan connections will match the number of scan registers, except for those scan registers that cannot capture (those with DRC Rule C15 violations)

nonscan connections

Indicates the number of nonscan cell ports connected to this clock. The ports taken in account are the clock, set, and reset ports. Cells with constraints, such as "constant value cells" might not be counted here. The number of nonscan connections will exclude those nonscan cells that cannot capture (those with DRC Rule C16 violations) and some classes of transparent latches (TLAs).

usage

Indicates the functional usage of the clock.

See Also

- [add_clocks](#)
- [remove_clocks](#)
- [report_clocks](#)

Output From the report_commands Command

You can use output from the Report Commands command to display command history or to display a list of commands or specific command syntax similar to the help command. The syntax for this command is as follows:

```
report_commands [command_name | -all] [-usage] [-history [-depth d]]
```

Summary Format

```
add_atpg_constraints      add_atpg_primitives
add_cell_constraints      add_clocks
add_equivalent_nofaults  add_faults
add_net_connections      add_nofaults
add_pi_constraints        add_pi_equivalences
```

#constraints

Indicates the total number of ATPG constraints in the current list.

Description

The summary report gives a list of all supported commands. The capitalized characters indicate the minimum characters necessary to enter the command.

Standard Format

```
add_clocks <off_state> port_name...
```

Description

The standard report displays the full usage of the reported command. The capitalized characters indicate the minimum characters necessary to enter the command or its arguments. Arguments enclosed in angled brackets < > are required. Arguments enclosed in square brackets "[]" are optional.

Usage Format

```
add_clocks <off_state> port_name...
    Allowed usage = DRC
```

Description

The usage report displays the standard command report and also gives restricted usage information. The restrictions include the allowed command modes and effect on deletion of internal patterns.

See Also

- [Report Commands](#)

Output From the report_memory Command

You can use output from the report_memory command to analyze memory gate data for selected memory gates. The syntax for this command is as follows:

```
report_memory <gate_id | instance_name | -all | -summary>  
[-max d] [-contents <address | all>] [-verbose]
```

SUMMARY FORMAT

```
RAM summary: #RAMS=1, #clock_unstable=1, #load_unstable=0,  
#read_only=0.  
ROM summary: #ROMS=0, #hot_read=0, #readoff_X=0, #readoff_0=0,  
#readoff_1=0.
```

#RAMS

Indicates the total number of memory primitives that behave as RAMs.

#clock_unstable

Indicates the total number of RAM memory primitives that are unstable when clocks are off.

#load_unstable

Indicates the total number of RAM memory primitives that do not hold state during a load operation.

#read_only

Indicates the total number of RAM memory primitives that qualify for read_only treatment.

#ROMS

Indicates the total number of memory primitives that behave as ROMs.

#hot_read

Indicates the total number of ROM memory primitives that have their read line always active.

#readoff_X

Indicates the total number of ROM memory primitives that have their read_off state set to X.

#readoff_0

Indicates the total number of ROM memory primitives that have their read_off state set to 0.

#readoff_1

Indicates the total number of ROM memory primitives that have their read_off state set to 1.

Standard Format

```
type gate_ID instance_path memory_file
```

```
-----  
RAM 76 /withram ram2file.i
```

type

This indicate the type of memory primitive. The type can be either RAM or ROM.

gate_id

Indicates the [primitive ID](#) number of the memory primitive.

instance_path

Indicates the [instance name](#) of memory primitive.

memory_file

Indicates the name of the memory_file that defined the memory contents.

Verbose Format

```
                                #ports width address stable off  
type gate_ID wrt/rd addr/data min/max clk/load value instance_type
```

```
-----  
RAM 76 2 3 4 8 0 15 no yes 0 _MEMORY
```

type

Indicates the type of memory primitive. The type is either RAM or ROM.

gate_id

Indicates the **primitive ID** number of the memory primitive.

#ports wrt/rd

Indicates the number of ports associated with the memory primitive. The first field is the number of write ports contained in the memory primitive. The second field is the number of read ports it is connected to.

width addr/data

Indicates the number of address lines of the memory primitive followed by its number of data lines.

address min/max

Indicates the valid range of addresses for the memory primitive. The first field is the minimum address value and the second field is the maximum address value.

stable clk/load

Indicates the write stability behavior of the memory gate. The first field indicates if it is stable when all clocks are placed at their **off state**. The second field indicates if it holds state during the scan load operation.

off value

Indicates the value to be placed on the read port outputs when the read line is inactive. Possible choices are X, 0, or 1.

instance_type

The indicates the module type from which the memory primitive was derived.

Standard Format With Constants

```
type gate_ID instance_path memory_file
```

```
-----  
Description  
RAM 76 /withram ram2file.i  
0 : 0000XXXX  
1 : 11101111  
2 : 11001101  
3 : 10101011  
4 : 0000XXXX  
5 : 11101111  
6 : 11001101
```

Chapter 35: Types of Reports

Output From the report_modules Command

```

7 : 10101011
8 : 0010XXXX
9 : 11101111
a : 11001101
b : 10101011
c : 0000XXXX
d : 11101111
e : 11001101
f : 10101011

```

Description

The contents data can be selected to be added to either the standard format or the verbose format and is placed after the data for each reported memory primitive. One line per reported address is displayed which contains two fields. The first field is the address value in hex and the second field is a string of values for each data line of that address.

See Also

- [report_memory](#)
- [Memory Gate](#)

Output From the report_modules Command

You can use output from the report_modules command to analyze data associated with selected netlist modules.

```
report_modules [ name | -unreferenced | -undefined | -errors | -summary |
  -all ] [-verbose]
```

Summary Format

```
Modules: #UNKNOWN_FORMAT=19 #STRUCT_VERILOG=474
#BEH_VERILOG=24 (#unsupported_beh=24)
```

Description

The summary report indicates the number of modules in each used category. Possible categories include:

unknown format (UNKNOWN_FORMAT)

structural Verilog (STRUCT_VERILOG)

behavioral Verilog (BEH_VERILOG)

Verilog UDPs (UDP_VERILOG)

Standard Format

```
module name tot( i/ o/ io) inst refs(def'd) used
-----
FJK3 7( 5/ 2/ 0) 6 0 (Y) 0
```

module name

Indicates the name of the reported module.

pins tot(i/ o/ io)

This gives the counts for the external pin usage of the module. The first field reports the total number of pins in the module. Inside the parentheses are given the number of module input pins, number of module output pins, and number of module bidi pins.

inst

Indicates the number of instances inside the module.

refs(def'd)

Indicates the number of times this module was referenced from other modules. The field inside the parentheses indicates if the module was defined (y indicates module was defined, N indicates module was not defined).

used

Indicates the total number of instantiations of the module that was used in the simulation model.

Verbose Format

```
module name tot( i/ o/ io) inst refs(def'd) used
-----
FJK3 7( 5/ 2/ 0) 6 0 (Y) 0
  Inputs : J ( ) K ( ) CP ( ) CD ( ) SD ( )
  Outputs : Q ( ) QN ( )
  U1 : _INV conn=( I:SD O:n2 )
  U2 : _INV conn=( I:Q O:QN )
  U3 : _INV conn=( I:CD O:n3 )
  U4 : _MUX conn=( I:Q I:J I:n1 O:n4 )
  U5 : _INV conn=( I:K O:n1 )
  FJK3 : _DFE conn=( I:n2 I:n3 I:CP I:n4 O:Q )
```


Description

The verbose format for a module includes the all standard format data plus reports all of its input, output, inout, and instance data.

Inputs

The input line reports all the input pins of the module. Each input entry gives its pin name and any attributes of the pin placed in parentheses.

Outputs

The output line reports all the output pins of the module. Each output entry gives its pin name and any attributes of the pin placed in parentheses.

Inouts

The inout line reports all the bidi pins of the module. Each inout entry gives its pin name and any attributes of the pin placed in parentheses.

Instance lines

The data for each module instance is given on separate lines. The first field of an instance entry gives the [instance name](#). The second field is the instance type. The next field identifies all the connections of the instance pins. For each pin connection, the pin type is given first ("I" indicates input pin and "O" indicates output pin), followed by a colon, and ending with the name of the net (or pin) it is connected to.

See Also

- [report_modules](#)

Output From the report_net_connections Command

You can use output from the report_net_connections command to view data associated with all net connections added with the add_net_connections command. The syntax for this command is as follows:

```
report_net_connections
```

Standard Format

```
Connection PI, net gwx12z/amd/dp/uto_reg_0/D, -disconnect  
Connection PO, net gwx12z/amd/dp/U943/Z
```



```
Connection TIEX, net dout[4], module RGB_ctrl  
Connection TIE0, net GND
```

The report indicates the type of connection PI, PO, TIEX, and so forth., the name of the net to which the connection is made, and the keyword `-disconnect` if the original driver is to be disconnected, or the [module name](#), if the net reference is within a specific module only.

See Also

- [report_net_connections](#)

Output From the report_nets Command

You can use output from the `report_nets` command to analyze data associated with specific nets.

```
report_nets net_name [-module name]
```

Standard Format

```
TEST> report_nets /core/rgb/pwm/n6  
/core/rgb/pwm/n6 _INTERN ( )  
  I PM3/test_sei  
  I PR1/test_sei  
  I PR2/test_sei  
  I PR3/test_sei  
  I PREGS/test_sei  
  O U5/N01  
  
TEST> report_nets n56 -module hash  
hash/n56 _INTERN ( )  
  I U16/H02  
  I U17/H02  
  I U22/H01  
  O U23/N01  
  
TEST> report_nets PIN_ENA -module hash  
hash/PIN_ENA _PI ( )  
  I U17/H01  
  
TEST> report_nets pwm_dout[0] -module hash  
hash/pwm_dout[0] _PO ( )  
  O U16/N01
```

The standard net report lists the net pathname; the type of connection of _PI, _PO, _PIO, or _INTERN; and each connection for which the net is an input "I" or an output "O".

See Also

- [report_nets](#)

Output From the report_nofaults Command

You can use output from the report_nofaults command to analyze all faults with the nofault attribute. The syntax for this command is as follows:

```
report_nofaults <instance_name | pin_pathname [-stuck<0|1|01>] | -all>
```

Standard Format

```
sa1 ** /mux0/SL
sa0 ** /buf0/Y
sa1 ** /mux1/SL
```

Description

Each line gives information for a single nofaulted fault and contains the three standard fault fields. This first field indicates the stuck value for the fault. The second field is an ignored fault class field and is always displayed as two asterisks "**". The third field indicates the pin pathname associated with the fault site.

See Also

- [report_nofaults](#)

Output From the report_nonscan_cells Command

You can use the output from the report_nonscan_cells command to analyze behavioral data on nonscan flip-flops and latches. The syntax for this command is as follows:

```
report_nonscan_cells
<-summary | -all | c0 | c1 | cu | l0 | l1
| tla | le | te | ls | ram_out | unstable_set_resets | load | nonx_load>
```

```
[-max d] [-tlas [no_clock | hot_clock | x_clock]]  
[-unique] [-verbose]
```

Summary Format

```
Nonscan cell summary: #DFF=416 #DLAT=227  
tla_usage_type=no_clock_tla  
Nonscan behavior: #CU=64 #TLA=131 #LE=407 #TE=9 #LS=32  
Load disturbs : #CU=64 #TLA=131 #LE=407 #TE=9 #LS=32  
TLA behavior: #no_clock=131, #hot_clock=0, #X_clock=0
```

Nonscan cell summary:

This line indicates the total number of nonscan flip-flops (#DFF) and nonscan latches (#DLAT) in the simulation model. The line also indicates the global transparent latch (TLA) behavior. Possible TLA behaviors include "none" (no TLAs), "no_clock_tla" (no TLAs are connected to clocks), "hot_clock_tla" (TLAs are connected to clocks but all clock inputs of TLAs are active when clocks are off), and "X_clock_tla" (at least one TLA is connected to a clock and is not a hot_clock_tla).

Nonscan behavior:

This line lists the number of nonscan cells in each used nonscan cell category. Possible categories are **C0** (constant 0), **C1** (constant 1), **CU** (clock unstable), **L0** (load 0), **L1** (Load 1), **TLA** (transparent latch), **LE** (clock-off stable leading edge DFF), **RAM_out** (Macro Out bit of a memory cell), **TE** (clock-off stable trailing edge DFF), and **LS** (clock-off stable DLAT).

Load disturbs:

This line lists the number of nonscan cells in each used nonscan cell category that could not hold state during the scan load process.

TLA behavior:

This line lists the number of transparent latches in each **TLA** category. Possible TLA categories include TLAs not connected to clocks (**#no_clock**), TLAs whose clock port is always active when clocks are off (**#hot_clock**), and TLAs which are connected to clocks with the clock input being at an indeterminate state when all clocks are off (**#X_clock**).

Standard Format

```
type behavior_data gate_id instance_name (type)  
-----  
CU load_unstable 89286 /core/gum/boo/reg42 (LD1)  
L0 load_unstable 2315 /core/NFF1 (FD2)
```

```
L1 load_unstable 2416 /core/NFF2 (FD4)
TE load_stable 43220 /core/gum/reg2 (FD1)
```

type

Indicates the nonscan behavior that is used for scan based simulation and test generation. This type is one of: [C0](#), [C1](#), [CU](#), [L0](#), [L1](#), [TLA](#), [LE](#), [RAM_out](#), [TE](#), or [LS](#).

behavior_data

This lists behavior data associated with the nonscan cell. If the type is TLA, this is the TLA category as described above. Otherwise, it will indicate whether it is stable ([load_stable](#)) or unstable ([load_unstable](#)) during the scan load process.

gate_id

Indicates the [primitive ID](#) number of the nonscan cell.

instance_name

Indicates the [instance name](#) for the nonscan cell.

(type)

Indicates the module type from which the nonscan cell was derived.

See Also

- [report_nonscan_cells](#)

Output From the report_patterns Command

The output report produced by the [report_patterns](#) command helps you analyze pattern data from the internal or external pattern buffer. This report can be displayed in the following formats:

- [Summary Format](#)
- [Standard Format](#)
- [Pattern Type Format](#)

Summary Format

The `-summary` option produces a summary report that displays the number of patterns in the active internal pattern set. If there are active external patterns, the name of the

external pattern file and the number of external patterns are also included, as shown in the following example:

```
TEST-T> report_patterns -summary
Pattern Summary Report
-----
#internal patterns 448
#basic_scan patterns 448
#external patterns (chk2_fifo.vcde) 218
-----
```

Standard Format

If the starting pattern number is the only parameter specified with the `report_patterns` command, the standard format report is printed, as shown in the following example:

```
TEST-T> report_patterns 0
Pattern 0 (basic_scan)
Time 0: load chain1 =
    0100000000 1101000000 0101111101 1000000000 1010100011 0001010100
    1011000010 0100110111 0010010100 0001000010 1011001000 1001010001

Time 0: load chain2 =
    1110001011 0110001110 1011101011 0100110111 111100
Time 1: force_all_pis =
    0101110100 0000100011 1110100010 0001110010 0ZZZZZZZZZ ZZZZZZZZZZ

Time 2: measure_all_pos =
    1011XXXXX1 1000110100 0111001111 1111111011 0101000001
Time 3: pulse clocks /iigclk2x (69)
Time 4: unload chain1 =
    1111111110 1001111111 1111111101 1000000000 1010100011 0001010101
    1011000000 1111010110 0100100100 1001000010 10001001XX XXXX100011

Time 4: unload chain2 =
    1110001011 0110001110 1011101011 0100110111 111100
```

The following parameters are included in this report:

Time

Indicates the time of the applied pattern event. Possible pattern events include: `load`, `force_all_pis`, `measure_all_pos`, `pulse clocks`, `applied procedures` (master observe or shadow observe), and `unload`.

load chain_name

Displays the load values of a scan chain. The values are placed on the chain input pin in the same order as they appear for each shift cycle.

force_all_pis

Displays the respective values for placing the primary inputs and bidirectional ports. The inputs and bidirectional ports are sequentially ordered based upon their **primitive ID** numbers. Use the `report_primitives -pis -pio` command to determine the order of the characters.

measure_all_pos

Displays the respective values for measuring the primary outputs and bidirectional ports. The inputs and bidirectional ports are sequentially ordered based upon their **primitive ID** numbers. Use the `report_primitives -pos -pios` command to determine the order of the characters.

pulse clocks

Identifies the pulse clocks, including their associated **port name** and **gate ID** number.

apply procedure <procedure_name> (ID=%d) number times

Displays the number of times a master observe or shadow observe procedure is applied.

unload chain_name

Identifies an unloaded scan chain. The chain output port values are measured in the same order in which they appear for each shift cycle.

Pattern Type Format

Pattern type-based data is printed for all patterns when you specify both the `-type` and `-all` options of the `report_patterns` command.

TEST-T> `report_patterns -type -all`

Patn num	Pattern type	Cycle count	Load count	Observe proc	Clocks used
0	basic_scan	1/0	1	-	-
1	basic_scan	1/1	1	-	sdr_clk,sys_clk,pclk
2	basic_scan	1/1	1	-	sdr_clk,sys_clk,pclk
3	basic_scan	1/1	1	-	sdr_clk,prst_n
4	basic_scan	1/1	1	-	sdr_clk,sys_clk,pclk
5	basic_scan	1/1	1	-	sdr_clk,sys_clk,pclk
6	basic_scan	1/1	1	-	sdr_clk,sys_clk,pclk

Chapter 35: Types of Reports

Output From the report_patterns Command

```

7      basic_scan 1/1      1      -      sdr_clk,prst_n
8      basic_scan 1/1      1      -      sys_clk,pclk
9      basic_scan 1/1      1      -      prst_n
10     basic_scan 1/1      1      -      prst_n
11     basic_scan 1/1      1      -      prst_n
12     basic_scan 1/1      1      -      prst_n
13     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
14     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
15     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
16     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
17     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
18     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
19     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
20     basic_scan 1/1      1      -      prst_n
21     basic_scan 1/1      1      -      sdr_clk,sys_clk,pclk
22     basic_scan 1/1      1      -      sdr_clk,sys_cl
23     basic_scan 1/1      1      -      sdr_clk
24     basic_scan 1/1      1      -      sdr_clk,pclk
25     basic_scan 1/1      1      -      sys_clk

```

The following parameters are included in this report:

Patn num

Identifies the pattern number.

Pattern type

Identifies the type of pattern (basic scan, fast-sequential, or full-sequential).

A basic scan pattern is a combinational pattern that uses one scan load, one capture clock (optional), and one scan unload. A fast-sequential pattern can use up to 10 scan loads and 10 captures.

Cycle count

The cycle count is comprised of two values: N1/N2. The N1 value is the number of force-measure pairs in a pattern. This value corresponds to the number of tester cycles used by the pattern. The N2 value is the number of capture procedures with a clock pulse. Since force-measure pairs do not always include a clock pulse, the N2 value is usually less than or equal to the N1 value.

However, for designs with on-chip clocks (OCCs), the N2 value can be larger than the N1 value (for details, see the "[Understanding the Cycle Count for Designs with OCCs](#)" section).

Load count

Indicates the number of scan loads used by the pattern.

Observe proc

Identifies a master observe or shadow observe procedure.

Clocks used

Identifies the active clock used by the pattern. The order of the clocks is the same as the order used by the pattern.

Understanding the Cycle Count for Designs with OCCs

In the clock format report created by the `report_patterns` command, the cycle count (N1/N2) is comprised of force-measure pairs (N1) and capture procedures with a clock pulse (N2). In most cases, the N2 value is less than or equal to the N1 value. However, for designs with OCCs, the N2 value can be greater than the N1 value, as shown in the following example:

```
Patn Pattern Cycle Load Observe Clocks num type count count proc
used -----
fast_seq 2/8 2 - CLK1,CLK2,RefClk CLK1,CLK2,RefClk CLK1,CLK2,RefClk
CLK1,CLK2,RefClk CLK1,CLK2,RefClk CLK1,CLK2,RefClk CLK1,CLK2,RefClk
CLK1,CLK2,RefClk1 fast_seq 2/8 2 - CLK1,CLK2,RefClk CLK1,CLK2,RefClk
CLK1,CLK2,RefClk CLK1,CLK2,RefClk CLK1,CLK2,RefClk CLK1,CLK2,RefClk
CLK1,CLK2,RefClk CLK1,CLK2,RefClk
```

Designs with OCCs typically have a different cycle count than standard designs because the cycle count is computed in a different manner:

- In the example, the N1 value is reported as 2 because both patterns are multiple load patterns (the load count is 2).
- The N2 value is equal to the N1 value multiplied by the greater of the value specified by either the `set_atpg-min_ateclock_cycles` command or the `set_drc-num_pll_cycles` command.

The `set_atpg -min_ateclock_cycles` command specifies the minimum number of system cycles for each pattern. By default, the number of ATE Clock cycles is the same as the value of the `set_drc -num_pll_cycles` command. In this case, the `set_drc -num_pll_cycles` command is set to 4. As a result, N2 is equal to 8: the N1 value (2) multiplied by the PLL cycles value (4)

See Also

- [Reporting Patterns](#)
- [Specifying the Pattern Source](#)
- [Summaries Report](#)

Output From the report_pi_constraints Command

You can use output from the report_pi_constraints command to identify all primary input and bidirectional ports that have been constrained. The syntax for this command is as follows:

```
report_pi_constraints
```

Standard Format

```
pin_name constrain_value  
-----  
/TEST_MODE 1
```

pin_name

Indicates the [port name](#) of the primary input or bidi pin on which the constraint is placed.

constrain_value

Indicates the value of the PI constraint. It is "0", "1", "X", or "Z".

See Also

- [report_pi_constraints](#)

Output From the report_pi_equivalences Command

You can use output from the report_pi_equivalences command to view all primary input and bidirectional ports that have been constrained to be equivalent with the add_pi_equivalences command. The syntax for this command is as follows:

```
report_pi _equivalences
```

Standard Format

```
pin_name equivalent pins  
-----  
/NL0 /NL1 ~/NL2  
/ENA_P ~/ENA_N
```

pin_name

Indicates the [port name](#) of the primary input or BIDI pin on which the equivalence relationships have been defined.

equivalent pins

This lists all ports which have been specified to be equivalent (or inverted) with the original port. A "~" character that precedes a port name indicates that the port is inverted relative to the original port.

See Also

- [report_pi_equivalences](#)

Output From the report_po_masks Command

You can use output from the report_po_masks command to view all primary outputs that have been constrained to be masked with the add_po_masks command.

```
report_po_masks
```

Standard Format

```
output pin_name  
-----  
/D0  
/D1
```

output pin_name

Indicates the [port name](#) of the primary output or bidi port whose measured value is to be masked.

See Also

- [report_po_masks](#)

Output From the report_primitives Command

You can use output from the report_primitives command to analyze data on ATPG primitives in the model created by the run_build_model command. The syntax for this command is as follows:

```
report_primitives
```

Chapter 35: Types of Reports

Output From the report_primitives Command

```
< id | instance_name | net_name | pin_pathname |
-ports | -pis | -pos | -pios | -type gate_type | -summary >
[-max d]
```

Summary Format

```
Gate Summary Report
-----
#primitives 101071
#primary_inputs 251
#primary_outputs 148
#primary_bidis 128
#DLATs 227
  #TLAs 195
  #nonscan 32
#DFFs 11369
  #nonscan 416
  #scan 10953
#BUSs 577
  #contention_fails 289
#BUFs 4108
#INVs 6425
#ANDs 9740
#NANDs 21086
#ORs 5689
#NORs 2981
#XORs 2207
#XNORs 944
#TIEOs 11357
#MUXs 20017
#WIRES 23
#BUFZs 138
#TSDs 3656
-----
```

Description

The summary reports the total number of gates in the simulation model and the number of gates for each primitive type. The data for a primitive type is on a single line and includes two fields. The first field identifies the primitive type and the second field gives the number of gates of that primitive type in the simulation model. For the DLAT primitive type, a lower level primitive type [TLA](#) (transparent latch) is also give.

Standard Format

```
/gate2/U4 (24) DFF (FD1S)
  --- I () (/TIE_0)
  --- I () (/TIE_0)
  CP I (010) 13-7gate2/A1/Z
  --- I (XXX) 17-
  Q O (XXX) 32-q4
scan_behavior: MASTER(LE/-) chain=c1 cell_id=0 invert_data=NN obs=noproc
```

First Line

The first line of the standard primitive report contains 4 fields. The first field is the [instance name](#) of the gate, the second field is the [primitive ID](#) number enclosed in parentheses, the third field is the primitive type, and the fourth field is the module type from which the primitive was derived in parentheses.

Pin Lines

The next lines of the primitive report are the pin lines. These lines are indented and each pin contains 3 required fields with an optional pin data field. The first field is the name of the pin ("---" indicates no netlist pin name exists). A "!" character preceding an input pin name indicates the pin is inverted. The second field indicates the pin type. It might be I (input pin), O (output pin), or B (bidirectional pin). A "W" character preceding an input/bidi pin type indicates a weak pin. If primitive report is not set off, the next field is the pin data enclosed in parentheses. The pin data is selected using the set_pindata command. The final field indicates the connections of the pin. The pin connection gives the ID number of the connecting primitive and its pin pathname if it exists. An output pin can have multiple pin connections and the additional connections is placed on separate lines. The number of reported output pin connections might be selected with the default set to 2. Connections to global tie gates are indicated by a tied primitive type placed in parentheses.

Usage Lines

The final lines of a primitive report are unindented usage lines. These include scan behavior, nonscan behavior, external pin behavior (clock, scanin, scanout, constrained, masked, and equivalence), BUS behavior, BUS keeper behavior, MEMORY primitive behavior, READP primitive behavior, and MOUT primitive behavior.

Note that the `scan_behavior` line contains an `invert_data` statement, as shown in bold in the following example:

```
scan_behavior: MASTER(LE/-) chain=big_chain cell_id=2
invert_data=NI obs=noproc
```

The `invert_data` statement in the example displays net inversion data in a two-character format: the "N" character refers to normal (non-inverting) behavior, and the "I" character refers to inverting behavior. The first character of the sequence corresponds to the area of the scan cell between the scan-in port and the `_DFF` primitive. The second character corresponds to the area of the scan cell between the primitive to the scan-out port.

In the following example, net inversion occurs from the scan-in port to the `_DFF` primitive of the scan cell, and the scan cell behavior is normal (non-inverting) from the primitive to the scan-out port:

```
invert_data=IN
```

The behavior can be the same in both locations of the scan cell, as shown in the following examples:

```
invert_data=NN
```

```
invert_data=II
```

Verbose Format

```
/gate2/U4 (24) DFF (FD1S)
  --- I () (/TIE_0)
  --- I () (/TIE_0)
  CP I (010) 13-/gate2/A1/Z
  --- I (XXX) 17-
  Q 0 (XXX) 32-q4
scan_behavior: MASTER(LE/-) chain=c1 cell_id=0 invert_data=NN obs=noproc
Connection data: to=REGPO from=TLA
```

Description

The verbose primitive report (selected by using the `-verbose` option of the `set_primitive_report` command) displays the standard primitive report plus all learned data associated with the gate. This includes connectivity to/from special points, learned implications, learned equivalences, wire primitive blockage, common input behavior, and learned functional behavior.

Pin/PI/PO/PIO Format

```
gate_id type pin_name (behavior_data)
-----
  0 PI /clk clock(off=0,master,shift)
  1 PI /a
```

gate_id

Indicates the [primitive ID](#) number for the pin gate.

type

Indicates the primitive type of the pin gate. It might be "PI", "PO", or "PIO".

pin_name

Indicates the pin name of the pin gate.

(behavior_data)

This lists any behavior data associated with the pin gate. This will include clock data, scanin/scanout data, constrain pin data, and equivalence data.

See Also

- [report_primitives](#)

Output From the report_rules Command

You can use output from the report_rules command to view data associated with rule violations. The syntax for this command is as follows:

```
report_rules [rule_id | rule_type | -all] [fail]
```

Standard Format

```
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
```

rule

Indicates the name of the rule to be reported.

severity

Indicates the current severity of the reported rule. Possible severity choices include ignore, warning, error, and fatal. Fatal is a special kind of error severity whose severity cannot be changed.

#fails

Indicates the current number of failures for the reported rule.

description

This gives a brief textual description of the reported rule.

See Also

- [report_rules](#)

Output From the run_build_model Command

The `run_build_model` command builds an in-memory simulation model from the design modules that have been read in. You can view the results of this process in the output. The syntax for the `run_build_model` command is as follows:

```
run_build_model [top_module] [-weakgates]
```

Standard Format

```
run_build_model RGB2
-----
Begin build model for topcut = RGB2 ...
-----
There were 38494 primitives and 6007 faultable pins removed during model
optimizations
Warning: Rule B6 (undriven module inout pin) was violated 2 times.
Warning: Rule B7 (undriven module output pin) was violated 178 times.
Warning: Rule B8 (unconnected module input pin) was violated 923 times.
Warning: Rule B10 (unconnected module internal net) was violated 32
times.
Warning: Rule B12 (undriven instance input pin) was violated 2 times.
End build model: #primitives=93831, CPU_time=5.03 sec, Memory=27MB
-----
Begin learning analyses...
End learning analyses, total learning CPU time=11.00
-----
```

#primitives=

This indicates the total number of primitives in the simulation model that was built. Each primitive in this model is assigned a unique ID number between 0 and one less than #primitives.

CPU_time=

This indicates the cumulative number of CPU seconds that was used for the model build process.

Memory=

This indicates the amount of memory that is used to store the simulation model.

Verbase Format

```
run_build_model RGB2
-----
Begin build model for topcut = RGB2 ...
-----
Begin flattening of 211306 instantiated modules, Memory=24MB...
Flattened 100000 instantiated modules, Memory=47MB.
Flattened 200000 instantiated modules, Memory=77MB.
Unused primitives eliminated: 19154, pins lost: 13565
Buffers eliminated: 160443, pins lost: 0
Inverters eliminated: 25444, pins lost: 0
Primitives eliminated due to cascaded gate structures: 1255, pins lost:
0
There were 206296 primitives and 13565 faultable pins removed during
model optimizations
Number common input primitives=37990
Ranking completed: #ranked_gates=226851, #feedback_paths=0
Build memory summary: peak=103MB (10.0% final freed), final=90MB,
bytes/primitive=161
Build memory usage: primitives=65.7%, instances=30.9%, others=3.4%
End build model: #primitives=244759, Memory=90MB
-----
Begin learning analyses...
Fanout-free region identification completed: #regions=94366
Begin common input learning...
Learned tied gates: #XOR=11981
Learned gate types: #BUF=401 #INV=318 #XOR=12048 #XNOR=42
Learned relations: #equivalence=10893 (classes=4360)
Common input analysis completed:
Tied analysis completed: #tied_gates=14739, #blocked_gates=88/0,
#blocked_inputs=12772
End learning analyses, total learning CPU time=23.2
-----
```


Flattening of Instantiated Models results:

These are status messages that indicate the progress of flattening the total number of instantiated modules.

Primitive elimination results:

This is an expert level message that occurs during the building of the simulation model. It indicates the number of gates that were removed from the simulation model due to the following reasons:

Unused primitives eliminated: All primitives whose outputs have no path to either a top level output or an internal state element are removed. They are unused and do not affect circuit operation.

Buffers eliminated: Inverters eliminated:

All buffers and inverters are removed unless they are required as placeholders for fault sites. In addition, an inverter can only be removed if the inverters output connects to an ATPG primitive's input which has a selectable inversion control for it's inputs.

Primitives eliminated due to cascaded gate structures:

Cascaded gate optimization is performed by identifying two gates in series that can be merged into a single gate. An example of this would be two 2-input AND gates in series which could be replaced by a single 3-input AND gate.

Number common input primitives:

Primitives with inputs in common, such as a 2-input NAND gate driven from the same net are replaced with their equivalent INV function.

Ranking completed:

This is an expert level message that occurs during the building of the simulation model. It indicates the number of ranked gates, the number of feedback path networks, and the CPU time in seconds that was used to perform the ranking and feedback path analysis.

Fanout-free region identification

This is an expert level message that occurs during the building of the simulation model. It indicates the number of fanout free regions which were identified and the CPU time in seconds that was used to perform the analysis.

Begin common input learning...

This is an expert level message that occurs during the building of the simulation model. It indicates the beginning of the common input gate analysis.

Learned tied gates:

This is an expert level message that occurs during the building of the simulation model. It indicates the number of learned tied gates for various gate types that were learned during the common input gate analysis.

Learned gate types:

This is an expert level message that occurs during the building of the simulation model. It indicates the number of learned learned gate functions for various gate types that were learned during the common input gate analysis.

Learned relations:

This is an expert level message that occurs during the building of the simulation model. It indicates the number of gates with equivalence relationships and the total number of equivalence relationships that were learned during the common input gate analysis.

Common input analysis completed

This is an expert level message that occurs during the building of the simulation model. It indicates the end of the common input gate analysis and the CPU time in seconds that was used to perform the analysis.

Begin ATPG equivalence analysis...

These are expert level messages that occur during the building of the simulation model.

The first message indicates the beginning of the ATPG analysis and the settings that determine how much effort will be attempted. The number of simulation passes indicate the number of random patterns that is simulated to create a list of tie and equivalence candidates. The maximum number of test passes indicated the maximum allowed number of test generation attempts before the process is terminated.

The second message summarizes the results of the random pattern simulation. These results include the number of potential tie gates (first number is for tie0 and the second number is tie1), the number of potential equivalence gates (and number of associated equivalence groups), and the CPU time in seconds that was used for simulation. If the ATPG analysis is aborted due to hitting the maximum allowed test generation attempts, a message is given indicating the process was prematurely aborted.

The last message indicates the end of the ATPG analysis and summarizes the results. The results include the number of learned tie gates, the number of learned equivalence gates, the number of aborted ATPG attempts (the first number is during the tie learning and the second number is the equivalence learning), and the CPU time in seconds that has been used for ATPG analyses.

Tied analysis completed:

This is an expert level message that occurs during the building of the simulation model. It indicates the end of the tied gate analysis and summarizes the results. The results include the number tied gates, the number of gates whose fault effects are blocked due to tied gates, and the number of gate inputs whose fault effect is blocked due to tied values on other inputs.

See Also

- [set_atpg](#)
- [set_learning](#)
- [set_messages](#)
- [Understanding Flattening Optimizations](#)
- [Building the ATPG Model](#)

Output From the run_fault_sim Command

The run_fault_sim command performs fault simulation using the current fault list and the current selection of the pattern source. The syntax for this command is as follows:

```
run_fault_sim [-fast_pattern d] [-store] [-sequential]
```

Standard Format

```
run_fault_sim -sequential
```


Begin sequential fault simulation of 17447 faults on 500 external patterns.

#faults simulated	pass detect	#faults total	cum. detect	#faults active	test coverage	process CPU time
1023	89	1023	89	17358	0.53%	501.00
2047	12	1024	101	17346	0.60%	1002.00
3070	49	1023	150	17297	0.90%	1503.00
4093	44	1023	194	17253	1.16%	2004.00
5116	29	1023	223	17224	1.34%	2505.00
6141	26	1025	249	17198	1.49%	3006.00
7164	1	1023	250	17197	1.49%	3507.00
8187	19	1023	269	17178	1.60%	4008.00
9211	5	1024	274	17173	1.63%	4509.00
10234	1	1023	275	17172	1.64%	5010.00
11257	2	1023	277	17170	1.65%	5511.00

Chapter 35: Types of Reports

Output From the run_fault_sim Command

```

12280      11 1023      288   17159      1.72% 6012.00
13303       0 1023      288   17159      1.72% 6513.00
14326       9 1023      297   17150      1.77% 7014.00
15349       0 1023      297   17150      1.77% 7515.00
16372       4 1023      301   17146      1.79% 8016.00
17395      13 1023      314   17133      1.89% 8517.00
17447       3  52      317   17130      1.91% 9018.00
Fault simulation completed: #faults_simulated=17447,
test_coverage=1.91%, CPU time=9019.00

```

#faults simulated

This indicates the cumulative number of faults which are simulated in this run. This count can be either uncollapsed (default) or collapsed depending on the current fault report selection.

pass #faults detect/total

This indicates the number of faults which are detected in the current pass followed by the number of faults simulated in this pass. These counts can be either uncollapsed (default) or collapsed depending on the current fault report selection.

cum #faults detect/active

This indicates the cumulative number of faults which are detected in the current run followed by the number of faults which are still active. These counts can be either uncollapsed (default) or collapsed depending on the current fault report selection.

test coverage

This indicates the test coverage after the current pass was completed. The test coverage is calculated using either uncollapsed (default) fault counts or collapsed fault counts depending on the current fault report selection.

process CPU time

The indicates the cumulative CPU time in seconds that has been used for the current run.

See Also

- [run_simulation](#)
- [set_simulation](#)

Output From the run_justification Command

The run_justification command creates a pattern that satisfies user-specified conditions placed on any number of internal (or external) circuit nodes. The syntax for this command is as follows:

```
run_justification [-set <<id | pin_pathname> <0|1|z>>...]
                 [-verbose] [-store] [-previous] [-noconstraints] [-noprevention]
```

Standard Format (Examples)

```
TEST> run_justification -set 1000 0 -store
Justification required 0 remade decisions.
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 1 constraint violations (ID=1,
pat1=0). (M179)
Justification pattern is stored in internal pattern set.

TEST> run_justification -set /dac/ramdp/U46/B 1
Successful justification: pattern values available in pattern 0.

TEST> run_justification -set /dac/ramdp/U98/A 0 -store
Successful justification: pattern values available in pattern 0.
Justification pattern is stored in internal pattern set.

TEST> run_justification -set /dac/ramdp/U46/B 1 -store -previous
Unsuccessful justification: test status was atpg_untestable.

TEST> run_justification -set BIDI_EN 1 host/dram/U67/E 0 dram/U166/E 0
Successful justification: pattern values available in pattern 0.
Warning: 1 patterns rejected due to 8 bus contentions (ID=83341,
pat1=0). (M181)
```

Justification Status

The first line of the report indicates the status of the test generation effort that attempted to simultaneously satisfy all the specified conditions. If the justification was successful, then the message will also indicate a pattern number. When the pindata is set to that pattern number, the logic simulation values for the justification pattern can be displayed in the schematic viewer or by use of the report_primitives command. If the justification is not successful, the message will indicate why test generation failed. Possible explanations are abort, redundant, and ATPG untestable. These terms have the same meaning as the standard fault classes.

Store Message

If the justification was successful and the `-store` option was used, the created pattern is stored in the current internal pattern set and a message displayed indicating that the pattern was stored. The pattern is typically stored as internal pattern 0, but can also be the special "Fast-Sequential" pattern. More than one pattern can be created.

Warning Messages

If the justification was successful, the resulting pattern is fully simulated. If this pattern has a bus contention condition or it fails to satisfy a constraint, then a warning message (M181) is given. Normally, the justification process will also satisfy ATPG constraints and contention prevention. However, bidirectional pins are purposely left at X if they were not necessary for satisfying the justification conditions. This can result in bus contention conditions and the warning messages should be ignored. The contention checking or constraints checking can be adjusted or disabled using the `set_contention` command. Disabling the contention checks will avoid the extra justification effort to satisfy those conditions and any corresponding warning messages.

See Also

- [Fault Classes](#)
- [report_primitives](#)
- [set_contention](#)
- [set_pindata](#)

Output From the run_simulation Command

The `run_simulation` command performs simulation of the current pattern source determined by the `set_patterns` command and reports any differences between simulated and expected values. The syntax for this command is as follows:

```
run_simulation [-sequential] [-store] [-max_fails d]
[-nocompare] [-nox_difference]
[[pin_pathname <0|1>] | [chain_name <position> <0|1>]][-failure_file
file_name] [-replace] [-last_pattern d]
```

Standard Format

```
run_simulation -sequential
Begin sequential simulation of 16 external patterns.
```

```
Simulation completed: #patterns=16/56, #fail_pats=0(0),  
#failing_meas=0(0), CPU time=34.27
```

#patterns=N1/N2

This indicates the number of patterns in the entire pattern set followed by the total number of time periods in the pattern set. A pattern can have multiple times at which measures are performed.

#fail_pats=N1(N2)

This indicates the number of patterns which experiences simulation mismatches followed by the number of patterns which only had possible mismatches (simulated value at X for an expected measure).

#failing_meas=N1(N2)

This indicates the number of measures at POs or scan cells which experienced simulation mismatches followed by the number of measures which were possible mismatches (simulated value at X for an expected measure).

CPU time

This indicates the total amount of CPU time in seconds required to perform the simulation.

With Simulation Mismatches

```
set_patterns -external pat.bin  
End parsing binary file pat.bin with 0 errors.  
End reading 3 patterns, CPU_time = 0.00 sec, Memory = 0MB  
run_simulation flop5/TE 0  
Warning: The added fault(s) affect scan chain load/unload operation.  
(M453)  
Begin simulation of 3 external patterns with cell chain1-1 stuck at 0.  
  0 test_so[1] 2 (exp=1, got=0)  
  0 test_so[1] 3 (exp=1, got=0)  
  1 test_so[1] 1 (exp=1, got=0)  
  1 test_so[1] 2 (exp=1, got=0)  
  2 test_so[1] (exp=1, got=0)  
Simulation completed: #patterns=3, #fail_pats=3(0), #failing_meas=5(0),  
CPU time=0.00
```

The first column identifies the pattern number. The second column identifies the failing output pin. The third column identifies the scan cell position in the chain. This column exists only if the failure occurred during scan chain unload. The fourth column is made up of two fields. The first field indicates the expected value to be measured from the pattern source, and the second field indicates the simulated value.

If the patterns are read in the external pattern buffer using the `-split_patterns` option, you should note the following: When there are simulation mismatches (for example, due to using the `-stuck` or `-slow` options), if the output of the `run_simulation` command is stored in a file using the `-failure_file` option, the failure file contains all the necessary directives to run the split diagnosis.

For example:

```
run_simulation -stuck {0 CHIP_TOP/FIFO/U45/Z}
.pattern_file_name chain_test_pat.stil
.pattern_file_name test_0.stil
.first_pattern continue
3 test_so1 6 (exp=1, got=0) // Compressor failure
3 test_so2 6 (exp=1, got=0) // Compressor failure
3 test_so3 6 (exp=0, got=1) // Compressor failure
4 test_so1 6 (exp=0, got=1) // Compressor failure
4 test_so2 6 (exp=0, got=1) // Compressor failure
4 test_so3 6 (exp=1, got=0) // Compressor failure
.pattern_file_name test_1.stil
.first_pattern continue
.pattern_file_name test_2.stil
.first_pattern continue
15 test_so1 6 (exp=0, got=1) // Compressor failure
15 test_so2 6 (exp=0, got=1) // Compressor failure
15 test_so3 6 (exp=1, got=0) // Compressor failure
.pattern_file_name test_3.stil
.first_pattern continue
22 test_so1 6 (exp=1, got=0) // Compressor failure
22 test_so2 6 (exp=1, got=0) // Compressor failure
22 test_so3 6 (exp=0, got=1) // Compressor failure
Simulation completed: #patterns=30, #fail_pats=4(0),
#failing_meas=12(0), CPU time=0.02
```

Using the `-max_fails` Option

This section describes the output of the `run_simulation` command using the `-max_fails` option.

For example if the first seven failures in the file are:

```
1 so 1 (exp=1, got=0)
1 so 3 (exp=0, got=1)
2 so 1 (exp=1, got=0)
2 so 3 (exp=0, got=1)
5 so 0 (exp=1, got=0)
```

If you use `-max_fails 3`, TestMAX ATPG does not merely generate:

```
1 so 1 (exp=1, got=0)
1 so 3 (exp=0, got=1)
2 so 1 (exp=1, got=0)
```


This would only be a partial set of failures for pattern 2, which contains the third failure.

If this data was given to diagnosis, it might not be able to diagnose it with the partial failures. Therefore, TestMAX ATPG does not stop at the third failure. Instead of generating incomplete failures for pattern 2, it generates the complete failures for the last pattern (pattern 2) as follows:

```
1 so 1 (exp=1, got=0)
1 so 3 (exp=0, got=1)
2 so 1 (exp=1, got=0)
2 so 3 (exp=0, got=1)
```

Therefore, `-max_fails` values of 2, 3, 4, and 5 all generate the same result.

If you use the `-max_fails 6` option, TestMAX ATPG generates:

```
0 po0 (exp=0, got=1)
1 po0 (exp=0, got=1)
1 so 0 (exp=1, got=0)
1 so 2 (exp=0, got=1)
1 so 3 (exp=0, got=1)
2 so 2 (exp=1, got=0)
2 so 3 (exp=1, got=0)
```

Using the `-progress_message` Option of the `set_simulation` Command

You can use the `-progress_message` option of the `set_simulation` command to print a progress message for every specified number of simulation passes. A simulation pass is 32 basic scan patterns, or from 1 to 10 (but usually 3) fast-sequential patterns.

The following example shows a simulation of 107 patterns, of which 69 are basic scan patterns and the remainder are fast-sequential patterns. When the simulation of basic scan patterns completes, the count of simulation passes has to restart, which causes a larger gap at that point.

```
TEST-T> set_simulation -progress_message 2
TEST-T> run_simulation
Begin good simulation of 107 internal patterns.
Simulated 64 patterns, CPU time=0.00 sec.
Simulated 75 patterns, CPU time=0.01 sec.
Simulated 81 patterns, CPU time=0.02 sec.
Simulated 87 patterns, CPU time=0.02 sec.
Simulated 93 patterns, CPU time=0.03 sec.
Simulated 99 patterns, CPU time=0.03 sec.
Simulated 105 patterns, CPU time=0.04 sec.
Simulation completed: #patterns=107, #fail_pats=0 (0), #failing_meas=0 (0),
CPU time=0.04
TEST-T> set_simulation -progress_message 4
TEST-T> run_simulation
```

Chapter 35: Types of Reports

Output From the run_simulation Command

```
Begin good simulation of 107 internal patterns.  
Simulated 81 patterns, CPU time=0.02 sec.  
Simulated 93 patterns, CPU time=0.03 sec.  
Simulated 105 patterns, CPU time=0.04 sec.  
Simulation completed: #patterns=107, #fail_pats=0(0), #failing_meas=0(0),  
CPU time=0.04
```

See Also

- [set_simulation](#)
- [run_fault_sim](#)

36

Glossary

[At-speed Clock](#)
[ATPG Primitive ID](#)
[Primitive Name](#)
[Black Box](#)
[Bus Keeper](#)
[Capture Clock](#)
[Capture Clock Edge](#)
[Capture Vector](#)
[Circuit Path](#)
[Clock](#)
[Clock Cone](#)
[Clock Unstable - CU](#)
[Command Abbreviation](#)
[Command Comments](#)
[Command Repeat](#)
[Constant 0 - C0](#)
[Constant 1 - C1](#)
[Continuation Character](#)
[Copy Paste](#)
[Delay Path](#)
[Effect Cone](#)
[Empty Box](#)
[False Path](#)

Fanin Number
Fanout Number
Head of Path
Instance Name
Launch Clock
Launch Clock Edge
Leading Edge - LE
Level Sensitive - LS
Load 0 - L0
Load 1 - L1
Loop ID
Majority Gate
Module Name
Module Pinname
Net Name
Non-robust Detection of a Path Delay Fault
Non-robust Test (For a Path Delay Fault)
Null Module
Off-path Input
Off State
Output Redirection
Path Delay Fault
Pin Pathname
Port Name
Primitive ID
RAM_out
Robust Detection of a Path Delay Fault
Robust Test (For a Path Delay Fault)

[Scan Clock](#)
[SCOAP](#)
[Sequential Model Port Priorities](#)
[Setup Vector](#)
[Shift Position](#)
[Simulation Events](#)
[Tail of the Path](#)
[Test For a Path Delay Fault](#)
[Trailing Edge - TE](#)
[Transparent Latch - TLA](#)
[Set / Resets](#)

At-speed Clock

An *at-speed clock* is the pair of clock edges applied at the same effective cycle time as the specified operating frequency for the design.

ATPG Primitive ID

The *ATPG primitive ID* is a numeric value that TestMAX ATPG assigns to a primitive that has been added to the simulation model by the `add_atpg_primitives` command. This ATPG primitive ID is used wherever a regular primitive ID can be used. It is most often used in the `add_atpg_constraints` command.

In the following example, an ATPG primitive is added, which is the "Equiv" function with four input pins. The new primitive is assigned the ID of 20201.

```
DRC> add_atpg_primitives my_atpg_gate equiv \  
    /BLASTER/MAIN/CPU/TP/CYCL/CDEC/U1936/in1 \  
    /BLASTER/MAIN/ALU/TP/CYCL/CDEC/U6/in1 \  
    /BLASTER/MAIN/ALU/TP/CYCL/CDEC/U16/in2 \  
    /BLASTER/MAIN/ALU/TP/CYCL/CDEC/U13/in0  
  
Gate with ID#=20201 has been added to the ATPG  
primitive list.
```

```
DRC> report_atpg_primitives -all
name          id#      type  inputs
-----
my_atpg_gate  20201   EQUIV 861 990 1431 902
```

See Also

- [add_atpg_primitives](#)
- [ATPG Primitive Name](#)
- [Primitive ID](#)
- [report_atpg_primitives](#)

ATPG Primitive Name

The ATPG primitive name is a symbolic label you assign to a primitive that you are adding to the simulation model with the `add_atpg_primitives` command. This ATPG primitive name is used in the `add_cell_constraints` or `remove_atpg_constraints` command.

In the following example, an ATPG primitive is added, which is the "Equiv" function with four input pins. The new primitive is assigned the name "my_atpg_prim".

```
DRC> add atpg primitives my_atpg_prim 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
Gate with ID#=20201 has been added to the ATPG primitive
list.
DRC> report atpg primitives -all
name id# type inputs
-----
my_atpg_prim 20201 EQUIV 861 990 1431 902
```

See Also

- [add_atpg_constraints](#)
- [add_atpg_primitives](#)
- [ATPG Primitive ID](#)

- [EQUIV Primitive](#)
- [remove_atpg_constraints](#)
- [report_atpg_primitives](#)

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 TIE primitives.

When you declare a module that does not have a module definition as a black box, TestMAX ATPG 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.

See Also

- [Null Module](#)

Bus Keeper

A *bus keeper* is a cell, usually with a single pin attached to a bus net and designed to hold the last driven state on that net. A bus keeper always has a weak drive strength so that it does not compete with a strong driver. The bus keeper is initialized to a 1 or 0 by a strong driver, and when all drivers on the net are off, the bus keeper continues to drive the last value (with weak drive). Bus keepers are often used to avoid floating net conditions on internal buses.

Capture Clock

The *capture clock* is the clock used to capture the final value resulting from V2 at the tail of the path.

See Also

- [Capture Vector](#)
- [Tail of the Path](#)

Capture Clock Edge (Capture Edge)

The *capture clock edge* (or capture edge) is the clock edge used to capture the final value resulting from V2 at the tail of the path.

See Also

- [Capture Vector](#)
- [Tail of the Path](#)

Capture Vector

A *capture vector* refers to the set of values on PIs and sequential devices during the second of the two vectors required for a test for a path delay fault. The shorthand notation V2 is often used to refer to this set of values.

See Also

- [Test for a Path Delay Fault](#)

Circuit Path

A *circuit path* is a series of gates in a circuit where each gate in the circuit path (with the possible exception of the gate at the tail of the path) has its output pin connected to the input pin of another gate in the circuit path, and each gate in the circuit path (with the possible exception of the gate at the head of the path) has one of its input pins connected to the output pin of another gate in the circuit path.

See Also

- [Head of the Path](#)
- [Tail of the Path](#)

Clock

A *clock* is any primary input that can affect the stored values of sequential devices, including flip-flops, latches, and RAMS. The following principles apply to this definition:

- RAM write-control is considered a clock.
- Synchronous set or reset is considered a clock.
- Asynchronous set or reset is considered a clock

In addition to these conditions, a clock must be defined using the `add_clocks` command or the DRC Test Procedure File. Also see the `set_drc -allow_unstable_set_resets` command.

Clock Cone

A *clock cone* is the cone of combinational logic fanning out from a single clock input port and extending to one or more sequential device input pins or primary output ports.

Comment Lines

Command files can contain comments in the form of:

- blank lines
- lines beginning with "###"
- lines beginning with "#"
- text following any "###"

For example,

```
this is a comment that transcripts
# this is a comment that does not transcript
add_clocks 0 clk
## this is a comment

## the preceding blank line was a comment
```

Repeating Commands

To repeat execution of the most recent command, enter `!!` in the command line.

To repeat execution of a previous command beginning with a particular text string, enter an exclamation character followed by the first few characters of that string. For example, `!rep` repeats execution of the most recent command beginning with "rep" (probably a `report` command).

Note that both `!!` and `!xyz` work only for interactive command entry (not from within a command file), and searches are performed only on commands stored in the command history buffer.

Continuation Character

If you have a long command, it can be continued over multiple lines by ending each successive line with a backslash character ("`\`"). If the backslash is the last character in a line, it indicates that the command is continued on the following line.

The following example of the `add_atpg_primitives` command defines an ATPG primitive connected to four design pins. Each pin pathname is presented on a separate line using the continuation character.

```
DRC> add_atpg_primitives my_atpg_primitive1 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 }
```

Delay Path

A *delay path* is a circuit path that:

- begins with either a sequential device or a circuit PI, and
- ends with either a sequential device or a circuit PO

With an indicated parity (relative to the head of the path) for each gate, such that there are no sequential devices in the path other than the beginning and ending gates (possibly excepting one or more transparent latches).

See Also

- [Circuit Path](#)
- [Head of the Path](#)

Effect Cone

An *effect cone* is the cone of combinational logic fanning out from the output pin of a sequential device (clocked by "CLK1", for example) and extending through all combinational logic until it reaches other sequential devices or primary output ports. An effect cone is always relative to a specific clock attached to a clock/set/reset port of the originating sequential cell (in other words, "CLK1" in this example).

Empty Box

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, TestMAX ATPG 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.

See Also

- [Black Box](#)
- [Null Module](#)

False Path

A *false path* is a delay path that does not affect the functionality of the circuit, either because it is impossible to propagate a transition down the path (topologically false path) or because the design of the circuit does not make use of transitions down the path (functionally false path).

See Also

- [Delay Path](#)

Fanin Number

A *fanin number* can be used in the [backward](#) command to specify from which input to trace back to obtain a report on a connected primitive.

You can use the `report_primitives` command to find out what is connected to the inputs and outputs of a specified primitive. After you get such a report, using the `backward` command (just entering "b" is sufficient) traverses back along the first input pin listed to find the connected primitive, and displays a report on that primitive, just like using the `report_primitives` command.

To traverse backward from an input pin other than the first one listed, specify a fanin number in the `backward` command. For example, specify 2 for the second input or 3 for the third input.

Consider the following example:

```
DRC> report_primitives 130
/i22/reg2/lat1 (130) DLAT (P_LAT_RS)
!SB I (/TIE_1)
  RB I 28-
  CK I 17-
  D I 105-/i22/reg2/MX1/Q
  Q O 131-/i22/reg2/r/D
DRC> backward 2
/i22/reg2/MX1 (105) OR (SCANINP_UDP_1)
  --- I 45-
  --- I 104-
  Q O 130-/i22/reg2/lat1/D
```

The first command generates a report on primitive ID 130. The `backward 4` command traverses back from the fourth input listed, which is the "D" input pin, to find the connected device, primitive ID 50, and generates a similar report on that primitive.

See Also

- [Primitive ID](#)
- [report_primitives](#)

backward

Overview

This command displays the gate connected to the selected input of the last reported gate.

Syntax

```
backward [fanin_number]
```

Arguments

fanin_number

Indicates the input number (1-based) of the previously reported gate whose connecting gate is to be reported. The default is the first input.

Allowed Command Modes

DRC, Test

Description

This command displays the gate connected to the selected input of the last reported gate.

Examples

The following example uses the `report_primitives` command to display information associated with primitive ID 135, a DLAT primitive. Then the `backward 3` command is issued which navigates backward through the third input. In this example, this is the "CK" pin, which is driven from ID = 14. So the `backward` command does a "report_primitive" on ID 14. A second `backward` command navigates backward one more time to ID = 2, a primary input port called "/CLK".

```
TEST> report_primitives 135
reg4/r (135) DLAT (N_LATCH)
!SB I (TIE_1)
!RB I (TIE_1)
CK I 14-
D I 134-reg4/lat1/Q
Q O 125-
scan_behavior: MASTER(LS/-) chain=c1 cell_id=0 invert_data=IN
obs=noproc
TEST> backward 3
reg4 (14) BUF (DFFP)
CK I 2-CLK
--- O 135-reg4/r/CK
      19-
TEST> backward
CLK (2) PI (_PI)
CLK O 10-reg0/CK
      11-reg1/CK ...
PI usage: clock(off=0, master, shift)
TEST>
```

Fanout Number

A *fanout number* can be used in the `forward` command to specify from which output to trace forward to obtain a report on a connected primitive.

You can use the `report_primitives` command to find out what is connected to the inputs and outputs of a specified primitive. After you get such a report, using the `forward` command (just entering "f" is sufficient) traverses forward along the first output pin listed to find the connected primitive, and displays a report on that primitive, just like using the `report_primitives` command.

To traverse forward from an output pin other than the first one listed, specify a fanout number in the `forward` command. For example, specify 2 for the second output or 3 for the third output.

Consider the following example:

```
TEST> report_primitives /i22/reg0
/i22/reg0 (48) INV (DFRLP)
  --- I 127-/i22/reg0/r/Q
  Q 0 49-/i22/reg4/D
  QB 0 50-/i22/reg1/MX1/SDI

TEST> forward 2
/i22/reg1/MX1 (50) AND (SCANINP_UDP_1)
  SDI I 48-/i22/reg0/QB
  --- I 40-
  --- O 85-
```

The first command generates a report on instance "/i22/reg0." The `forward 2` command traverses forward from the second output listed, which is the "QB" output pin, to find the connected device, [Primitive ID 50](#), and generates a similar report on that primitive.

See Also

- [Primitive ID](#)
- [forward](#)

forward

This command displays the gate connected to the selected fanout of the last reported gate.

Syntax

```
forward [fanout_number]
```

Arguments

fanout_number

Indicates the fanout number (1 based) of the previously reported gate whose connecting gate is reported. The default is the first fanout.

Allowed Command Modes

DRC, Test

Description

This command displays the gate connected to the selected fanout of the last reported gate.

Examples

In this example the `report_primitives` command is used to display information about gate ID = 12, a BUF primitive. Then the `f` command is issued that navigates forward through the first (and only) output pin, which is connected to primitive ID 131. The result of the `f` command is a `report_primitives` on primitive ID = 131, a DLAT primitive.

```
TEST> report_primitives 12
reg2 (12) BUF (DFFRLP)
  CK I 0-CLK
  --- O 131-reg2/r/CK
      17-

TEST> forward
reg2/r (131) DLAT (N_LATCH)
  !SB I (TIE_1)
  RB I 29-
  CK I 12-
  D I 130-reg2/lat1/Q
  Q O 87-
      86-
scan_behavior: MASTER(LS/-) chain=c1 cell_id=2 invert_data=NI obs=noproc
```

report_primitives

Use this command to report data on ATPG primitives in the model created by the run `build_model` command.

Syntax

```
report_primitives
< id | instance_name | net_name | pin_pathname
| -ports | -pis | -pos | -pios | -type type
| -summary | -all > [-max d]
```

Arguments

id | instance_name | net_name | pin_pathname

Selects display of pin data for the specified object. The type of pin data is specified by a prior `set_pindata` command. An object is identified by its primitive ID number, its instance name, a net name connected to the object, or a pin

pathname of a pin of the instance. You can use a wildcard character with an instance name. However, when TestMAX ATPG reads an image file, the object cannot be identified by its instance name.

-all

Reports all primitives. The pin data, as specified by a prior `set_pindata` command, is displayed for each primitive.

-ports

Reports all external (top-level) ports; equivalent to `-pis -pos -pios`.

-pis

Reports all input ports (Primary Inputs).

-pos

Reports all output ports (Primary Outputs).

-pios

Reports all bidirectional ports (Primary I/O ports).

-type *type*

Reports all primitives of a specific classification type. The pin data, as specified by a prior `set_pindata` command, is displayed for each primitive.

Priority indicator "p" on an input shows that the input (which must be the set or reset of a DLAT or DFF) has priority over the other set or reset of the same primitive. Priority indicator "P" on an input shows that the input (which must be the set, reset, or clock input of a DLAT or DFF) has priority over other clocks non-set/reset) of the same primitive.

These indicators allow correlating the attributes of gate pins in the flattened model with the attributes of module ports in the netlist data (see the `report_modules` command).

The recognized types include simulation primitives and ATPG functions as well as learned behavior analysis:

Simulation and ATPG Function Types:

ADRBUS, AND, BUF, BUFZ, BUS, BUSK, DATABUS, DFF, DLAT, EQUIV, INV, MEMORY, MOUT, MUX, NAND, NOR, OR, PI, PIO, PO, RPORT, SEL01, SEL1, SW, TIE0, TIE1, TIEX, TIEZ, TSD, WBUF, WIRE, XNOR, XOR

Learned Behavior Types:

blocked, common_input, common_tied_input, conblocked, constrained, equivalences, implications, invert_inputs, learn_buf, learn_inv, learn_tied_gate, tied, weak.

-summary

Displays a summary of primitives used in the ATPG simulation model. This summary includes the total number of primitives as well as a count of each primitive type used in the ATPG model.

-max d

Limits the number of primitives reported to specified maximum. This option does not work with the `-summary` or `-all` options. To report a specific number out of all primitives, use the `report_primitives * -max d` command.

Allowed Command Modes

DRC, Test

Description

Use this command to report data on ATPG primitives in the model created by the `run build_model` command.

Examples

```
TEST> report primitives -summary
      Primitive Summary Report
-----
#primitives          20201
#primary_inputs      49
#primary_outputs     41
#primary_bidis       40
#DFFs                1713
  #nonscan            201
  #scan              1512
#BUSs                40
#BUFs                1573
#INVs                1963
#ANDs                2384
#NANDs               5381
#ORs                 3298
#NORs                972
#XORs                208
#XNORs               141
#TIE0s               55
#TIE1s                3
#MUXs                2259
#TSDs                81
```

```
-----  
  
# by instance pathname  
DRC> report_primitives reg0  
reg0 (48) INV (DFFRLP)  
---      I      127-reg0/r/Q  
QB       O      50-reg1/MX1/SDI  
  
# by pin pathname  
DRC> report_primitives /reg1/MX1/SDI  
reg1/MX1 (50) AND (SCANINP_UDP_1)  
SDI      I 48-reg0/QB  
---      I 40-  
---      O 85-  
  
# by primitive ID number  
DRC> report_primitives 128  
reg1/lat1 (128) DLAT (P_LAT_RS)  
!SB      I (TIE_1)  
RB       I 26-  
CK       I 16-  
D        I 85-reg1/MX1/Q  
Q        O 129-reg1/r/D  
  
# by type of modeling primitive  
DRC> report_primitives -type XOR  
-----  
List of XOR gates  
-----  
adder (53) XOR (INC4)  
---      I 51-  
---      I 47-  
S0       O 56-mux0/B  
adder (71) XOR (INC4)  
---      I 69-  
---      I 54-  
S1       O 74-mux1/B  
adder (91) XOR (INC4)  
---      I 89-  
---      I 72-  
S2       O 94-mux2/B  
adder (110) XOR (INC4)  
---      I 107-  
---      I 92-  
S3       O 113-mux3/B  
Total number of reported XOR gates = 4  
  
DRC> report_primitives -type dlat  
-----
```

List of dlat gates

```
-----  
(6) DLAT ( _DLAT)  
--- pP I 4-SET  
!--- P I 5-RESET  
--- I 0-CLK1  
--- I 2-DATA1  
--- I 1-CLK2  
--- I 3-DATA2  
--- O 7-LOUT  
Total number of reported dlat gates = 1
```

```
TEST> report_primitives -pis  
gate_id type port_name (behavior_data)  
-----  
0 PI CLK clock(off=0, master, shift)  
1 PI RSTB clock(off=1, master, reset)  
2 PI SDI2 scanin(c1)  
3 PI SDI1 scanin(c2)  
4 PI INC  
5 PI SCAN
```

```
TEST> report_primitives -pos  
gate_id type port_name (behavior_data)  
-----  
62 PO SDO2 scanout(c1)  
63 PO COUT
```

```
TEST> report_primitives -pios  
gate_id type port_name (behavior_data)  
-----  
6 PIO D0  
7 PIO D1 scanout(c2)  
8 PIO D2  
9 PIO D3
```

```
TEST> report_primitives -ports  
gate_id type port_name (behavior_data)  
-----  
0 PI CLK clock(off=0, master, shift)  
1 PI RSTB clock(off=1, master, reset)  
2 PI SDI2 scanin(c1)  
3 PI SDI1 scanin(c2)  
4 PI INC  
5 PI SCAN  
6 PIO D0  
7 PIO D1 scanout(c2)  
8 PIO D2  
9 PIO D3
```

```

62 PO      SDO2 scanout (c1)
63 PO      COUT

# for a RAM or ROM
TEST> report_primitives u4
u4 (149)    MEMORY (ram512x8)
  ---      I (TIE_0)
  ---      I (TIE_0)
wclk       I 15-
  ---      I (TIE_1)
  ---      I 102-
  ---      I 120-
  ---      O 103-

Memory data: #data_lines=8, #address_lines=9/0, range=0-499,
file=ram512x8.dat
Read port usage: #read_ports=1, #cam_ports=0, read_off=X
Write port usage: #write_ports=1, edge_trigger=yes, type=stable_low
Stability results: clock_off=yes, load=yes, read_only_ability=no
Conflict behavior: write_write=XBIT, read_write=READ_NEW,
read_read=READ_NORMAL

```

Primitive ID

The *primitive ID* is a unique identification number assigned to each primitive in the design. TestMAX ATPG assigns these numbers to all primitives, starting with 0, during the build process.

The primitive ID (sometimes called the Gate ID) is displayed in many of the messages that the ATPG tool generates, especially rule violation messages. In the graphical schematic viewer (GSV), the primitive ID is displayed below each primitive symbol.

For the following example, the `report_atpg_primitives` command displays information associated with instance `/i22/reg0/r`. The report shows that the primitive ID associated with this instance is 127, and that pins RB, CK, and D are connected to primitives 25, 10, and 126, respectively. Pin Q has two connections, to primitives 48 and 49.

```

DRC> report_atpg_primitives /i22/reg0/r
/i22/reg0/r (127)
  DLAT (N_LAT_RS) !SB I (/TIE_1)
  RB I 25-
  CK I 10-
  D I 126-
/i22/reg0/lat1/Q
  Q O
  48- 49-
DRC>

```

See Also

- [report_atpg_primitives](#)

Gray Box

A *gray box* is a module or block whose function and internal contents are partially known. This include access to the internal data structures and algorithms. It is a combination of a white box model and a black box model.

Verification coverage can be greatly increased when using a gray box.

Head of the Path

The *head of the path* is the gate at the beginning of a circuit path.

See Also

- [Circuit Path](#)

How to Copy and Paste

To copy text, select the range of text with the left mouse button. The selected text becomes reverse-highlighted. Then use *Ctrl-C* to copy the selected text, or use the right mouse button to display the popup menu and select Copy.

To paste text, use *Ctrl-V* if you are pasting a single line, or use the right mouse button to display the popup menu and select `Paste`.

To paste multiple lines of text into the command input field, use either the popup menu from the right mouse button or *Ctrl-V*, then paste the multiple lines into the command input field.

Cut/Paste between X11 window and TestMAX ATPG GUI window

The TestMAX ATPG GUI window environment is based on a third party product called Qt from a company called TrollTech.

Inside the TestMAX ATPG window the TrollTech user model applies your X11 window environment and *Ctrl-C* and *Ctrl-V* are the keys programmed for cut/paste.

Outside of the TestMAX ATPG window, your X11 window environment applies and whatever is standard for your hot keys or mouse keys is controlled by your customizations to X11 defaults.

The Qt cut/paste buffer and the X11 cut/paste buffer are the same buffer.

Instance Name

An *instance name* uniquely identifies an instance in the design. Each instance name consists of a sequence of hierarchical block names separated by slashes. It does not include pin path names. Here are some examples:

```
/i22/reg4/lat1  
/top_asic/ADDER2/bitreg/reg[2]  
/I$1/I$672/I$4/I$2
```

In the graphical schematic viewer (GSV), you can choose whether to display instance names. By default, no instance names are shown. To make instance names visible, use the Edit > Environment > Viewer menu command and check the box next to "Display Instance Names". Alternatively, you can enter the `set_environment_viewer -instance_names` command at the shell prompt.

See Also

- [set_environment_viewer](#)
-

Launch Clock

A *launch clock* is the clock used to change from V1 to V2.

See Also

- [Capture Vector](#)
 - [Setup Vector](#)
-

Launch Clock Edge (Launch Edge)

The `launch clock edge` (or launch edge) is the clock edge used to change from V1 to V2.

See Also

- [Capture Vector](#)
- [Setup Vector](#)

Feedback Path ID

When TestMAX ATPG recognizes a combinational feedback path during analysis of the design, it assigns an identification number to the loop and records the instances included in the feedback path. To obtain a list of the feedback path IDs, use the `report_feedback_paths-all` command. See the following example.

```
TEST> report_feedback_paths -all
id# #gates #sources sensitization_status
-----
0      2      1      pass
1     10      1      pass
2      5      1      pass
3      8      1      pass
```

See Also

- [report_feedback_paths](#)

Majority Gate

A *majority gate* is a circuit structure identified during the learning process that has the majority function — for example, a three-input structure with an output that has the same value as the majority of the inputs. In this case, if two or three inputs have a value of 0, then the output is 0. Majority gates can enhance the ATPG process by enabling better test coverage and using fewer patterns. *Cascaded majority gates* are several majority gates implemented in a row, with only buffers or inverters in between each gate.

Measure Scan Chain Output

Also *measure_sco*. Refers to strobing of the output port of a scan chain.

Modifying Timing Data in an Existing STL Procedures File

If you want to modify timing data in an existing STIL procedure file, you must use the following command sequence:

```
read_drc original_STL procedure_file
update_wft (with desired options)
update_clock (with desired options)
update_scale (with desired options)
```

Module Name

```
write_drc_file new_STL procedure_file
```

```
...
```

```
run_drc new_STL procedure_file
```

After specifying the timing modification commands, it is critical that you use the `write_drc_file` command to write out the modified data. It is also critical that you specify the `run_drc` command with this newly produced file to incorporate these changes.

Module Name

A *module name* is the name assigned to a hierarchical unit in a design hierarchy. A module can represent the lowest-level function such as a buffer, a very complex function such as the entire circuit, or a hierarchical unit somewhere between these levels.

Module naming conventions are flexible and each netlist format such as Verilog, EDIF, or VHDL has its own set of rules. Here are some examples of module names:

F318N1

NandTree

portadr

alu_cntrl

ALU_test_1

bus_cntrl_test_1

rd_wr_test_1

ad_counter_DW01_cmp2_6_1

JTAG_TAP

my_asic

Module Pin Name

A *module_pinname* is the name of an input, output, or bidirectional port as defined in the original module definition. To obtain a list of pins for a specific module, use the `report_modules-verbose` command.

In the following example, H01 through H10, and N01 through N04 are pin names.

```
TEST> report_modules L572 -verbose
      pins
```



```

module name      tot( i/ o/ io)  inst refs(def'd) used
-----
L572              14( 10/ 4/ 0)    9  0  (Y)    0
  Inputs : H01 ( ) H02 ( ) H03 ( ) H04 ( ) H05 ( ) H06 ( ) H07 ( )
          : H08 ( ) H09 ( ) H10 ( )
  Outputs: N01 ( ) N02 ( ) N03 ( ) N04 ( )
  U1      : _NAND conn=( I:n1 I:n2 O:N04 )
  U2      : _NAND conn=( I:n3 I:n2 O:N03 )
  U3      : _NAND conn=( I:n4 I:n2 O:N02 )
  U4      : _NAND conn=( I:n5 I:n2 O:N01 )
  U5      : _INV  conn=( I:H10 O:n2 )
  U6      : _MUX  conn=( I:H09 I:H07 I:H08 O:n1 )
  U7      : _MUX  conn=( I:H09 I:H05 I:H06 O:n3 )
  U8      : _MUX  conn=( I:H09 I:H03 I:H04 O:n4 )
  U9      : _MUX  conn=( I:H09 I:H01 I:H02 O:n5 )
-----

```

See Also

- [report_modules](#)

Net Name

A *net name* is the name assigned to a set of interconnected wires in a design. A net name might require an instance name, even when referring to a net in the topmost module. TestMAX ATPG uses the net name defined in the original module, when possible. When a module consists of a truth table that is converted into a gate-equivalent representation, TestMAX ATPG constructs its own net names.

Here are some typical net names:

```

n56
\pla
\mux_cnt[0]
my_chip/reset_b

```

See Also

- [report_nets](#)

Non-robust Detection of a Path Delay Fault

A path delay fault is said to be non-robustly detected by a pattern if that pattern provides a non-robust test for the fault.

See Also

- [Path Delay Fault](#)

Non-robust Test (For a Path Delay Fault)

A *nonrobust test* is a test for a path delay fault that is guaranteed to detect excessive delay on the delay path if no other timing faults exist in the circuit.

See Also

- [Delay Path](#)

Nonscan Behavior: C0

The nonscan cell behavior *C0* stands for Constant 0. This means that the nonscan cell attains a value of 0 by the end of the load procedure and remains stable with this value, independent of any clocking activity in the design.

Nonscan Behavior: C1

The nonscan cell behavior *C1* stands for Constant 1. This means that the nonscan cell attains a value of 1 by the end of the load procedure and remains stable with this value, independent of any clocking activity in the design.

Nonscan Behavior: CU

The nonscan cell behavior *CU* stands for Clock Unstable. This means that the cell is clocked by a signal whose state is not known (such as the output of a sequential element, a nonclock primary input, or some logical combination of these signals), or has a state that might be disturbed during scan shift; and the cell does not behave as a transparent latch (behavior TLA). The Full-Sequential ATPG algorithm can sometimes control and observe this type of nonscan cell.

See Also

- [Nonscan Behavior: TLA](#)

Nonscan Behavior: L0

The nonscan cell behavior *L0* stands for Load 0. This means that the nonscan cell attains a value of 0 by the end of the load procedure and remains stable with this value during the application of the next FORCE PIs event and up to the next clock event.

Nonscan Behavior: L1

The nonscan cell behavior *L1* stands for Load 1. This means that the nonscan cell attains a value of 1 by the end of the load procedure and remains stable with this value during the application of the next FORCE PIs event and up to the next clock event.

Nonscan Behavior: LE

The nonscan cell behavior *LE* stands for Leading Edge. This means that the nonscan cell is a flip-flop that is stable when all top-level clocks are at the "off" state and the clock at the DFF primitive is also at the "off" state. The cell captures data on the leading edge of a clock pulse at the DFF primitive.

Nonscan Behavior: LS

The nonscan cell behavior *LS* stands for Level Sensitive. This means that the nonscan cell is a latch whose clock ports are inactive when the defined top-level clock ports of the design are at the "off" state.

Nonscan Behavior: RAM_out

The nonscan cell behavior *RAM-out* stands for Macro Out bit of a memory cell. This is a placeholder representing an output pin of a memory cell. When a memory instance has multiple outputs, each output is reported as a separate *RAM_out*, but all have the same instance name. The *RAM_out* can be *load_stable* or *load_unstable*, depending on whether the memory it represents is load stable or not.

Nonscan Behavior: TE

The nonscan cell behavior *TE* stands for Trailing Edge. This means that the nonscan cell is a flip-flop that is stable when all top-level clocks are at the "off" state and the clock at the DFF primitive is at the "on" state. The cell captures data on the trailing edge of a clock pulse at the DFF primitive.

Nonscan Behavior: TLA

The nonscan cell behavior *TLA* stands for Transparent Latch. This means that the nonscan cell is a latch with the ability to become transparent when all top-level clock ports of the design are at the "off" state. In other words, the latch might be unstable with all clocks off. A latch is not classified as TLA if it has a clock path to a data input.

Null Module

A *Null Module* is a module definition with no internal gates. Its primary purpose is to define the list of module pins and their proper direction of input/output/inout.

An example null module:

```
module PLL (clkin, enable, testmode, vco_out,
phasedet, out);
  input clkin, enable, testmode;
  input phasedet;
  output vco_out, out;
  # no gates
endmodule
```

Off-path Input

An *off-path input*, also called side input, is any input to a gate on a delay path that is not an on-path input.

See Also

- [Delay Path](#)
- [On-path Input](#)

Off State

The *off state* of a clock is the value, either 0 or 1, at which the sequential cell driven by the clock is stable. The sequential cell can be a latch or a flip-flop.

A "clock" is any pulsed port, including a port used for asynchronous set or reset. See the following examples.

```
DRC> add_clocks 0 CLK # rising-edge clock
```

```
DRC> add_clocks 1 RESETB # asynchronous reset
```

See Also

- [add_clocks](#)

On-path Input

An *on-path input* is the input to a gate on a delay path that comes from another gate on the delay path.

See Also

- [Delay Path](#)

Output Redirection

You can save the output of a command to a file, instead of having the output displayed in the TestMAX ATPG window. This feature is called output redirection. You can use this feature with manually entered commands and in command files. To invoke output redirection, end the command with one of the following:

```
> filename  
>> filename  
>? filename
```

> - Replace mode. The command output replaces the contents of the specified file if the file already exists and is not write protected. If the file does not already exist, a new file is created.

>> - Append mode. The command output is appended to the specified file if the file already exists. If the file does not already exist, a new file is created.

>? - Safe mode. The output is placed in a new file with the specified name. If the file already exists, it is left unchanged and you get an error message.

Note: THERE MUST BE A SPACE CHARACTER BEFORE AND AFTER the redirection symbols (after the command and before the file name).

Output redirection can be used with almost all types of commands, except for commands that already produce an output file such as the write commands. See the following examples.

```
TEST> analyze_faults -class AN > an_faults.txt  
TEST> report_memory -all -contents all >? memory_images.txt  
TEST> report_modules ADDER -verbose > module_dump.txt  
TEST> report_modules MULT -verbose >> module_dump.txt  
TEST> report_modules PHADE -verbose >> module_dump.txt
```

Path Delay Fault

A *path delay fault* consists of a delay path, along with a direction of transition for the sequential device at the head of the path.

See Also

- [Delay Path](#)
- [Head of the Path](#)

Pin Pathname

A *pin pathname* specifies a particular pin in the design. It can be a [port_name](#) representing a top-level port of the model, or it could be the path to a pin buried within the hierarchy of the design.

The following example uses the pin pathname `/i22/reg2/r/D` in an add cell constraint command. This corresponds to pin "D" of instance `/i22/reg2/r`.

```
DRC> report_primitives 130
/i22/reg2/lat1 (130) DLAT (P_LAT_RS)
!SB I (/TIE_1)
RB I 28-
CK I 17-
D I 105-/i22/reg2/MX1/Q
Q O 131-/i22/reg2/r/D
DRC> add_cell_constraints ox /i22/reg2/r/D
```

Port Name

A *port name* is the name of a top-level port (also called a pin pathname) of the module that is having ATPG performed. Each port name is usually a simple string, sometimes preceded with a forward slash character (/). Only rarely does a port name have any hierarchy.

To get a list of port names, use the `report_primitives -ports` command, as in the following example.

```
DRC> add_clocks 0 /CLK
DRC> report_primitives -ports
0 PI /CLK
1 PI /RSTB
2 PI /SDI2
3 PI /SDI1
4 PI /INC
```

```
5 PI /SCAN
6 PIO /D0
7 PIO /D1
8 PIO /D2
9 PIO /D3
136 PO /SDO2
137 PO /COUT
```

See Also

- [Pin Pathname](#)
- [report_primitives](#)

Primitive ID

The *primitive ID* is a unique identification number assigned to each primitive in the design. TestMAX ATPG assigns these numbers to all primitives, starting with 0, during the build process.

The primitive ID (sometimes called the Gate ID) is displayed in many of the messages that the ATPG tool generates, especially rule violation messages. In the graphical schematic viewer (GSV), the primitive ID is displayed below each primitive symbol.

For the following example, the `report_atpg_primitives` command displays information associated with instance `/i22/reg0/r`. The report shows that the primitive ID associated with this instance is 127, and that pins RB, CK, and D are connected to primitives 25, 10, and 126, respectively. Pin Q has two connections, to primitives 48 and 49.

```
DRC> report_atpg_primitives /i22/reg0/r
/i22/reg0/r (127)
  DLAT (N_LAT_RS) !SB I (/TIE_1)
    RB I 25-
    CK I 10-
    D I 126-
/i22/reg0/lat1/Q
  Q O
  48- 49-
DRC>
```

See Also

- [report_atpg_primitives](#)

Sequential Model Port Priorities

The `priority_setreset` is an attribute that can appear on an input port used for set or reset in a sequential model. It indicates that the port has priority over the other reset/set port. In UDP syntax, this means that if the set/reset with priority is ON (1 or 0 if inverted input), and the other reset/set is ? (any value), and all other clocks are OFF (0, or 1 if inverted input), then the set or reset action with priority takes effect.

The `priority_otherclocks` is an attribute that can appear on an input port used for set or reset, or for the clock. It indicates that the port has priority over other non-set/reset clocks. This means that if the clock with priority is ON, and all other set/resets are OFF, and all other clocks are ?, and all other data are ?, then the action of the clock with priority takes effect.

Note the following:

- The `priority_setreset` and `priority_otherclocks` attributes are independent, so the set and reset ports can have any combination of the two properties.
- You cannot ATTACH a `priority_setreset` or `priority_otherclocks` attribute. They are only inferred from an analysis of the UDP table.

Reconverging Path

A path in which the transition from some on-path node can reconverge to the off-path input of a later node on the path. The `-allow_reconverging_paths` option of the `set_delay` command makes allowance for this; as a result, (for example) two falling transitions on the inputs to a NAND gate don't count as blocking the detection of the path delay fault if both come from the path itself.

Robust Detection of a Path Delay Fault

A path delay fault is said to be robustly detected by a pattern if that pattern provides a robust test for the fault.

See Also

- [Path Delay Fault](#)

Robust Test (For a Path Delay Fault)

A *robust test* is a test for a path delay fault that is guaranteed to detect excessive delay on the http://popup_delay_path.htm/ regardless of the timing of other signals in the circuit.

Scan Clock

A *scan clock* is the clock applied to shift scan chains. Typically, this clock is applied at a slower rate than the specified operating frequency of the design.

SCOAP

SCOAP is an acronym for "Sandia Controllability and Observability Analysis Program." The `-scoap_data` option of the `set_pindata` command displays a set of SCOAP numbers that provide a limited metric to identify the difficulty for controlling or observing a pin.

In ideal circumstances (that is, with no reconvergent fanout) SCOAP numbers provide an estimate of the number of circuit inputs that must be controlled to produce a particular value on a pin or to observe its value at an output. However, in many situations the number of inputs is irrelevant because they would all require the same effort set to their required values. Also, SCOAP numbers do not account for dependencies between the input and output side of a pin, which can make test generation more difficult. Because of these factors, SCOAP is better described as a random pattern testability measure – even when it is accurate. Random patterns are more likely to detect a fault that requires 5 inputs to be set than a fault that requires 10 inputs.

The inevitable presence of reconvergent fanout means that the SCOAP numbers are misleading. Reconvergent fanout also means that a pin that requires only 2 inputs to be set might actually be more difficult to control than a pin that requires 10 inputs. The primary dependency is the effort required to control those inputs and the extent to which the required values conflict with other required values in the circuit. Generally, computing accurate testability is an NP-complete problem. SCOAP is a linear computation, so it cannot provide accurate testability estimates.

The key issue for the applicability of this metric is to determine the level of inaccuracy of SCOAP numbers. In the presence of reconvergent fanout, SCOAP numbers can be highly misleading. However, SCOAP numbers can be useful as a general guidance mechanism for *test generation* – their primary use in the test world.

For additional information on SCOAP, you can order published papers from IEEE. Related papers can be found under the VLSI Test topic. This is the IEEE web site: <http://www.ieee.org/>

Setup Vector (Launch Vector)

A *setup vector* (or launch vector) refers to the set of values on PIs and sequential devices during the first of the two vectors required for a test for a path delay fault. The shorthand notation V1 is often used to refer to this set of values.

Shift Position

A *shift position* is considered a row of scan cells. A position exists for every scan chain that is shifted out. For a compressed block, if the chains contain 40 scan cells and there are 100 chains, then there are 40 shift positions.

set_simulation

Simulation Events

The simulation of test procedures and patterns produces one or more simulation events for each test cycle. The number of events per test cycle depends on the complexity of the simulation data.

For example, consider the following sequence of test cycles defined in a test_setup macro. These four test cycles are translated into eight simulation events.

```
test_setup {  
  V { clk=0; data=Z; reset=0;}  
  V { reset=P;}  
  V { data=1; reset=0;}  
  V { clk=P;}  
}
```

If you look at internal design nets or pins attached to any of the input ports following a set_pindata command, you should see the following test setup data sequence:

```
clk: 0-000-0-010  
  
reset: 0-010-0-000  
  
data: Z-ZZZ-1-111
```

Note: The "-" separators have been added here to show how the test cycles are translated into simulation events. These separators do not appear in normal pin data displays.

Tail of the Path

The *tail of the path* is the gate at the end of a [circuit path](#).

See Also

- [Circuit Path](#)

Test For A Path Delay Fault

A *test for a path delay fault* is a sequence of circuit inputs that result in an incorrect value appearing at a specified circuit observation point if the transition specified by the path delay fault is late arriving at the tail of the path. A test for a path delay fault involves at least two vectors, since the head of the path must undergo a transition.

See Also

- [Head of the Path](#)
- [Path Delay Fault](#)
- [Tail of the Path](#)

Unstable Set / Resets

An unstable set/reset condition exists when the control net is internally generated. The default is to consider an unstable set/reset a DRC violation. Using internally generated set/resets allows Fast-Sequential ATPG to use set/reset-unstable nonscan cells to further improve test coverage. This is especially useful in at least two situations:

- 1) When a primary input is used as both a synchronous and asynchronous set or reset, and
- 2) When the scan-enable line is used to gate internal set/reset lines during scan load.

In either case, for better test coverage, the scan-enable input and the primary input acting as both asynchronous and synchronous reset should not be defined as clocks.

WFCMap

A *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 assigned to `_wfc` instead.

Ungated Circuitry

Ungated circuitry refers to a signal path that is not constrained through a logic element. Instead, the signal passes through the logic unaffected by the logic due to the state of all other pins on that logic element. For example, for an AND gate, if all the other pins are a logic-1, the signal passes through that logic element on the one input that isn't held at a logic-1.

37

Limitations

This section describes all known limitations and possible workarounds in TestMAX ATPG.

LIM-01: Full-Sequential ATPG Features Not Supported

- Fault models: slack-based transition, cell-ware, static and dynamic bridging, IDDQ, and hold-time
- Clocking: synchronous OCC and internal clocking procedures
- SDC: stuck-at fault model using SDC with `set_simulation -timing_exceptions_for_stuck_at`
- Masking: PO masks in affect using the `add_po_masks` command are ignored during the generation of Full-Sequential ATPG patterns
- Commands
 - `set_simulation` command options `-measure` and `-verbose`
 - `set_buses` command with the `-fault_contention` option
 - `set_contention` command options:
 - `-retain_bidi`
 - `-nocapture` (always `-capture`)
 - `-nopreclock` (always `-preclock`)
 - `report_power` command or power-aware settings
- Codecs: DFTMAX and DFTMAX Ultra
- Diagnosis: invoked with the `run_diagnosis` command
- Full-sequential ATPG and full-sequential simulation cannot be run using multithreading. Process switches to non-threaded.

LIM-02: Read New Behavior Of Synchronous RAMS Supported For ATPG Only

TestMAX ATPG now supports read/write contention behavior of `read_new` for RAMS with edge triggered read and write ports. This allows the modeling of a RAM with clocked read and write ports to have the write data appear on the read port outputs during the write

cycle (write-through). However, this behavior is supported for ATPG pattern generation only and not for sequential fault simulation (`run_simulation-sequential`, `run_fault_simulation-sequential`). This means that attempting to fault simulate functional patterns which rely on the read new (write through) behavior give misleading results.

LIM-03: TestMAX ATPG Not a Formal Language Syntax Checker

Although TestMAX ATPG does identify many errors and inconsistencies found in the incoming design and library files, it should not be considered a Verilog, EDIF, or VHDL syntax checker. In some cases, the tool can ignore illegal syntax rather than report it.

LIM-04: Behavioral Syntax Not Supported

With the exception of the limited behavioral syntax used to describe RAMs and ROMs, TestMAX ATPG does not support behavioral syntax in Verilog or VHDL input. TestMAX ATPG does not understand analog blocks modeled behaviorally for simulators (that is VCS), such as analog or behavioral Verilog used to model a PLL. PLL modules are typically modeled with behavioral techniques and TestMAX ATPG does not support behavioral modeling outside of its RAM/ROM syntax.

LIM-05: Not All Verilog Structural Syntax Supported

TestMAX ATPG does not support every variation and nuance of the Verilog hardware description language. Rather, the tool supports the syntax most commonly produced by the Synopsys synthesis product flows. If your design methodology includes hand-generated Verilog, you can encounter syntax that gets rejected as a "parse error". Contact your Synopsys support center if you encounter unsupported Verilog variations.

LIM-06: Partial Support of Designs Using Multibit Library Cells

The support of designs that contain multibit library cells is dependent on the required pattern output format. You can generate STIL serial and Unified STIL format patterns for direct test application. You can perform parallel simulation of modeled multibit cells, but the models must be reviewed. Without this review, problematic multibit cells might cause incomplete fault coverage. MAX Testbench does not support multi bit cells when there are inversions inside the multi bit cell.

Serial simulation is enforced if parallel forms of MAX Testbench are not supported.

Note that parallel STIL flows with multibit support are currently being tested. If a particular unsupported format is critical for your application, you should contact Synopsys. The recommended option for optimal quality of results is to review the modeling of multibit cells.

LIM-07: No VHDL Library Support

TestMAX ATPG does not support reading libraries in VHDL format using either generics or VITAL syntax. However, it does support reading designs in VHDL format as long as the description is structural and simple signal types of either `std_logic` or `std_ulogic` are used.

LIM-08: TestMAX ATPG is Not a Functional Pattern Reformatting Tool

TestMAX ATPG does not officially support reading functional patterns in one format such as WGL, and writing them out in the same or another format such as STIL. You can try this, however there is no guarantee that patterns created are correct or what you expect. However, you can reformat TestMAX ATPG-generated ATPG patterns.

LIM-09: TestMAX ATPG is Not a Design Netlist Translation Tool

TestMAX ATPG does not officially support reading a design in one netlist format, such as EDIF, and writing them out in another format, such as Verilog or VHDL. You can try this, however there is no guarantee that the design created is correct.

LIM-10: GUI Version Has Limitations

The GUI version of TestMAX ATPG has a number of limitations. It is intended that the GUI version be used to develop command scripts and to debug DRC violations. The non-GUI (shell) version is then used to run a final command script when the GUI is not required. The GUI command pull-downs are often missing options that are supported from the command line.

LIM-11: Two-dimensional Arrays Not Supported

Use of 2 dimensional arrays of nets in Verilog, VHDL, or EDIF are not supported. Nets are allowed to be 1 dimensional arrays (busses), however arrays of arrays are not supported.

LIM-12: Cross-module Net Referencing Not Supported

In Verilog, referencing nets in other modules without passing those nets through the port list is not supported. This syntax involves referencing the net pathname using the period as a hierarchical separator: and I1 (temp1, in1, top.blockA.vdd); and I2 (temp2, in2, top.blockB.gnd);

LIM-13: Read Netlist Order Wrong For Multifile Browse

When selecting multiple files to read using the file browser via the Ctrl+LMB approach, the read order of the files is random. This is related to a Microsoft coding problem, and there is currently no workaround available. If the order with which you are reading files is important, you should read them one at a time.

LIM-14: Reading Encrypted Verilog Not Supported

TestMAX ATPG cannot read encrypted Verilog modules. So when it encounters the protected directive it issues an N1 violation.

LIM-15: DEL Key on Numeric Keypad Cannot Work

When using the native display on some UNIX platforms the DEL key on the numeric keypad does not function properly. This is not reported to occur if TestMAX ATPG is run from an Xterm session to a remote CPU.

LIM-16: 2.1GB File Limit

Limits testing has shown that many 32-bit UNIX operating systems impose a limit on the maximum file size that can be created. When writing a pattern file on a design with a large number of scan cells or a large number of patterns, this 2.1GB limit can be easily reached. Writing the file in compressed gzip format provides some relief with 32-bit executables. Switch to a 64-bit executable if you continue to have problems.

LIM-17: Pattern Timing Limitation

The definition of pattern timing in the STIL procedure file supports timing units in ms (milliseconds), us (microseconds), ns (nanoseconds), ps (picoseconds), and fs (femtoseconds), but you must not mix units and the values must be expressed as integers. This means that you must pick one time unit and use it consistently throughout the DRC file.

LIM-18: MUXClock Options

MUXClock options (D,E WFCs) are not applied with scan compression designs.

LIM-19: Verilog TRANIF, RTRAN* Primitives Not Supported*

The TRANIF* and RTRAN* primitives represent a bidirectional data flow that is not supported by TestMAX ATPG . In some cases, the actual usage is a unidirectional flow and the cell might be remodeled with a unidirectional Verilog primitive such as NMOS. For other situations you need to create a custom ATPG model.

LIM-20: Dynamic and Disturb Clock Grouping

Dynamic and disturb clock grouping is not supported by Full-Sequential ATPG.

LIM-21: Paths Through Memories in DSMTest Not Supported

Paths that write through memories, including those modeled using TestMAX ATPG , are not supported and are declared untestable. Paths to or from memories are supported.

LIM-22: Differential Scan Inputs Not Supported

The TDL91, FDDL, TSTL2, and VHDL pattern outputs do not support differential scan inputs.

LIM-23: Utility ltran_shell Limitations (write_patterns -format tdl91 | tstl2 | fddl | wgl_flat)

- Output from the options `-compress <Gzip | Binary>` are not supported. This behavior follows the tdl91, tstl2, and fddl proprietary formats, which do not support compressed formats.
- Differential scan-in designs and relationships are not supported.
- The `wgl_flat` format does not support Scan-in pad information. Because this format is generated from a STIL file, it follows STIL padding behavior.

- A log file is not generated when generating stil2wgl vectors.
- The `-config` option is not supported for writing the `wgl_flat` format.
- Be careful when using 3-cycle capture procedures; single-cycle captures minimize translation issues.
- Reading `wgl_flat` patterns results in full-sequential pattern data which cannot be used by the `run_diagnosis` command.

LIM-24: Reporting Violations in Tcl GUI Mode

If you run the `report_violations -all` command in Tcl GUI mode, you cannot stop the reporting of warnings by clicking the "Stop" button, or by entering Ctrl-C.

LIM-25: Three-Cycle Generic Capture Procedures Limitation

The WGL format does not support three-cycle generic capture procedures.

LIM-26: Reading Parallel Patterns

TestMAX ATPG cannot read back parallel patterns because the format for this data is not the native format generated for normal scan compression operation.

LIM-27: Reading MAX Testbench-Produced Data in TestMAX ATPG

TestMAX ATPG cannot read files produced from MAX Testbench or files generated from the `write_testbench` command.

LIM-28: Generating Parallel Patterns From Image Files

You must use either or both the `-netlist_data` and `-design_view` options of the `write_image` command to create image files that generate parallel patterns. Otherwise, the image file incurs a loss of design information that affects the patterns, and TestMAX ATPG issues an M269 message.

LIM-29: Fault Analysis Not Supported for DFTMAX Ultra

The `analyze_faults` command is not supported when using DFTMAX Ultra with threaded or non-threaded ATPG.

LIM-30: Cell Constraint Ignored with Shared Scan-In Ports

The `add_cell_constraints` on a specific cell is not honored unless the same shift position for shared chains is also constrained.

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 TestMAX 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.

The following sections of this appendix describe test concepts:

- [Why Perform Manufacturing Testing?](#)
- [Understanding Fault Models](#)
- [Coverage Calculations](#)
- [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 the following problems:

- 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.

Understanding 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 transition fault model, or IDDQ, which is the quiescent current fault model.

The following sections describe fault models:

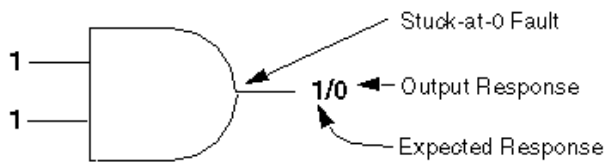
- [Stuck-At Fault Models](#)
- [Detecting Stuck-At Faults](#)
- [Transition Delay Fault Models](#)
- [Detecting Transition Delay Faults](#)
- [Using Fault Models to Determine Test Coverage](#)
- [IDDQ Fault Model](#)
- [Fault Simulation](#)
- [Automatic Test Pattern Generation](#)
- [Translation for the Manufacturing Test Environment](#)

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, the following figure 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 204 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 perform the following steps:

- 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.

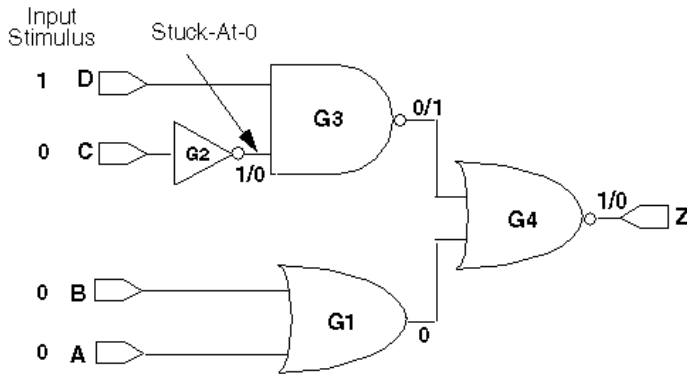
The set of logic 0s and 1s applied to the primary inputs of a design is called the input stimulus. The set of resulting values at the primary outputs, assuming a fault-free design, is called the expected response. The set of actual values measured at the primary outputs is called the output response.

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 the two-input AND gate shown in the preceding figure, 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 might need to control all other nodes to ensure observability of a particular target node. The following figure shows a circuit with a detectable stuck-at-0 fault at the output of cell G2.

Figure 205 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 the following goals:

- 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.

Transition Delay Fault Models

A transition delay fault is a delay defect concentrated at one logical node; any signal transition that passes through this node is delayed beyond the intended clock period. Two types of transition faults are associated with each node:

- A slow-to-rise (str) fault represents a node with a 0-to-1 transition that is delayed so much it is not captured by the next clock cycle.
- A slow-to-fall (stf) fault represents a node with a 1-to-0 transition that is delayed so much it is not captured by the next clock cycle.

This definition of a transition delay fault does not assume a particular magnitude of the delay defect and does not take into account the timing of the circuit or the clocks. The delay defect is assumed to be large enough to be detected along any logical path.

Slack-based transition fault testing further enhances the use of transition delay fault models by using timing information for both targeting and detecting faults. For more information, see [Slack-Based Transition Fault Testing](#).

Detecting Transition Delay Faults

Detecting transition delay faults requires two steps:

1. This step sets the fault location to its initial value. For a slow-to-rise fault, the node is set to 0.
2. This step tests the node for the stuck-at fault to its initial value. For a slow-to-rise fault, this is a test for the stuck-at-zero (sa0) fault.

Because testing the stuck-at-zero fault requires that the node is set to 1, a transition must take place at that node. However, there is no requirement for the transition to propagate beyond that node. This is different than the requirement for path delay faults (see [Path Delay Fault Theory](#)). This means the signal conditioning requirements for transition delay faults are also different than those for path delay faults. For example, off-path signals along the propagation paths of transition delay faults are care bits after the transition, but can be at any value before the transition.

In transition delay ATPG terminology, the launch cycle causes the transition and the capture cycle stores the result.

In a two-cycle transition test, the initial state is the scan-in value, the launch cycle causes the transition, and the capture cycle stores the result in a scan cell which is then scanned out. Non-scan registers and memories might require additional cycles before the launch cycle to set up the transition; additional cycles might be required after the capture cycle if the result is stored in a non-scan register and must be propagated to a scan cell.

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 TestMAX ATPG, see [Test Coverage](#).

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 process 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.

TestMAX ATPG 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 performs several logic simulations concurrently: one simulation represents the fault-free circuit (a good machine simulation) and several simulations represent the circuits containing single stuck-at faults (a faulty machine simulation). 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 205](#), it is apparent 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 [Internal Scan](#). 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, TestMAX ATPG 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 process 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 process.

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 [Internal Scan](#).

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.

Coverage Calculations

Coverage calculations are used to measure the effectiveness of test patterns and test generation for a given set of faults. The coverage calculations are for either the collapsed or uncollapsed faults. This is controlled with the `-report` option of the `set_faults` command.

Coverage formulas are given using fault counts for fault classes identified by their standard class code. `PT_credit` is the selectable detection credit given to possible detected faults and `AU_credit` is the selectable credit given to ATPG untestable faults. These credits are adjustable using the `set_faults` command.

Test Coverage

The Test Coverage (TC) is the percentage of detected faults for all detectable faults, and gives the most meaningful measure of test pattern quality. It is calculated by using the following formula:

$$TC = (DT + (NP + AP) * PT_credit) / (all_faults - UD - (AN * AU_credit) - (AX * AX_credit))$$

Fault Coverage

The Fault Coverage (FC) is the percentage of detected faults for all faults. It gives no credit for undetectable faults, and is calculated by using the following formula:

$$FC = (DT + (NP + AP) * PT_credit) / all_faults$$

Note: By default, unused gates in the design are deleted to save RAM usage and their corresponding faults are missing from the fault population. As a result, the UD class of faults is nearly empty. If you wish to see an accurate fault coverage you must also turn off unused gate deletion during circuit flattening. See the `set_build` command for more details.

ATPG Effectiveness

The ATPG Effectiveness (AE) is the percentage of faults that are resolvable by the ATPG process. Full credit is given to faults which are detected and faults which are proven to be untestable. It is calculated by using the following formula:

$$AE = (DT + UD + AN + AX + (AP + NP) * PT_credit) / all_faults$$

See Also

- [Fault Classes](#)

Internal Scan

Internal scan design is the most common 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).

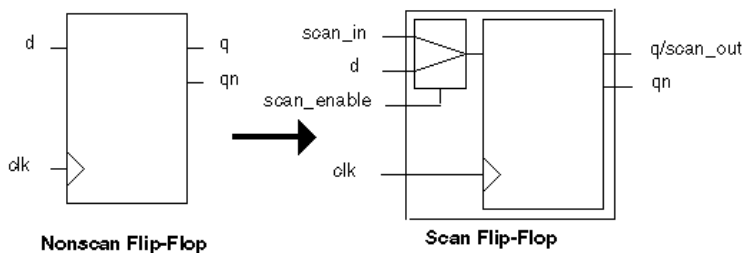
The following sections describe internal scan:

- [Example](#)
- [Applying Test Patterns](#)
- [Scan Design Requirements](#)
- [Full-Scan Design](#)
- [Partial-Scan ATPG Design](#)

Example

The following figure 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 206 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 the preceding figure.

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 TestMAX DFT, the Synopsys test-enabled synthesis configuration. For information about how TestMAX DFT minimizes the impact of inserting scan logic in your design, see the *TestMAX DFT Scan 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 206](#), 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 required 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.

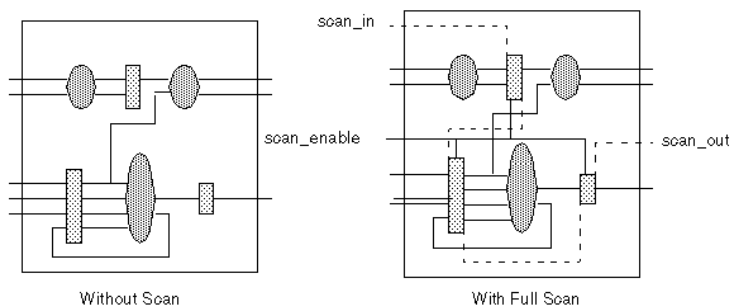
This process is similar to scan-in. The additional requirement is that the shift registers pass data reliably to the output ports.

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 the following figure. Ovals represent combinational logic; rectangles represent sequential logic. The full-scan diagram shows the scan path through the design.

Figure 207 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 TestMAX ATPG to achieve high test coverage results on a full-scan design.

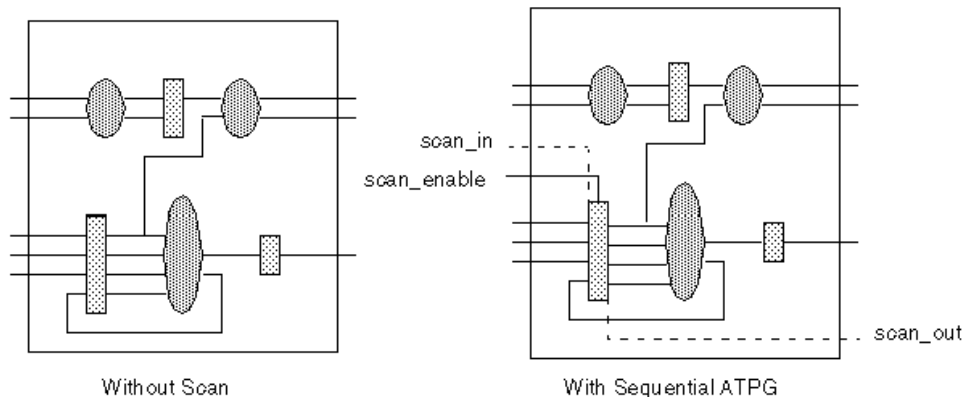
Partial-Scan ATPG Design

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 TestMAX ATPG, 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 processes perform propagation of faults through nonscan elements. For more information, see [ATPG Modes](#).

Partial scan divides a complex sequential design into simpler sequential blocks as shown in the following figure. 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 208 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 process.

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:

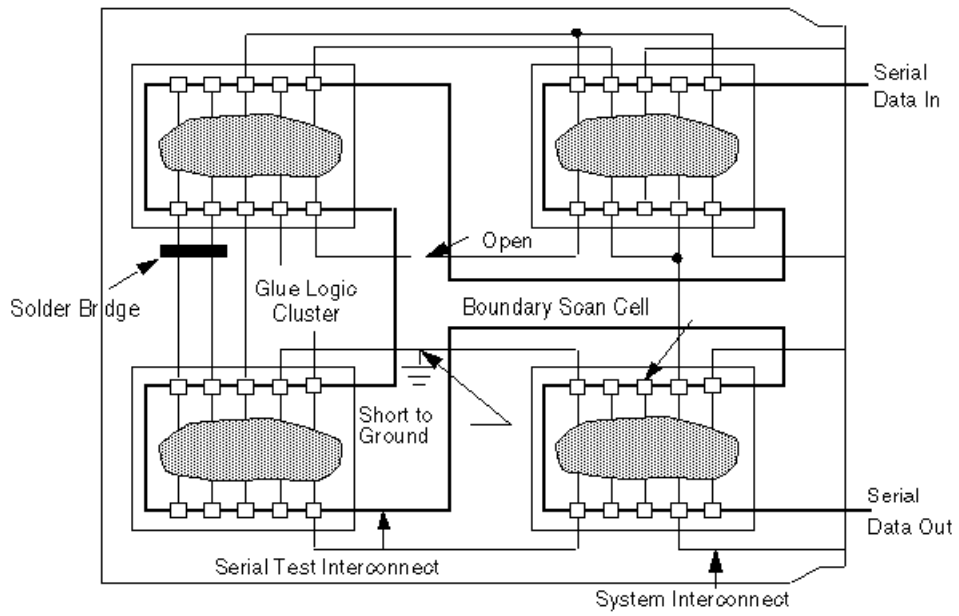
- Testing of the interconnect wiring on a printed circuit board for shorts, opens, and bridging faults
- Testing of clusters of non-boundary-scan logic
- Identification of missing, misoriented, or wrongly selected components
- Identification of fixture problems
- Limited testing of individual chips on a board

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.

The following figure shows a simple printed circuit board with several boundary scan ICs and illustrates some of the failures that boundary scan can detect.

Appendix A: Test Concepts
What Is Boundary Scan?

Figure 209 Board Testing With IEEE Std 1149.1 Boundary Scan



B

ATPG Design Guidelines

This section 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.

The following topics describe the ATPG design guidelines:

- [ATPG Design Guidelines](#)
- [Checklists for Quick Reference](#)

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.

Guidelines are provided for the following design entities:

- [Internally Generated Pulsed Signals](#)
- [Clock Control](#)
- [Pulsed Signals to Sequential Devices](#)
- [Multidriver Nets](#)
- [Bidirectional Port Controls](#)
- [Clocking Scan Chains: Clock Sources, Trees, and Edges](#)
- [Protection of RAMs During Scan Shifting](#)
- [RAM and ROM Controllability During ATPG](#)
- [Pulsed Signal to RAMs and ROMs](#)
- [Bus Keepers](#)
- [Bus Keepers](#)

Internally Generated Pulsed Signals

While TestMAX ATPG 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 210](#) in ATPG test mode. If your design contains clock gating, constrain the control side of the gating element while in ATPG test mode.

[Figure 211](#) and [Figure 212](#) show two solutions. In [Figure 211](#), the TEST input blocks the path from CLK to register B. However, B cannot be used in a scan chain.

Figure 210 Gated Clock: A Problem

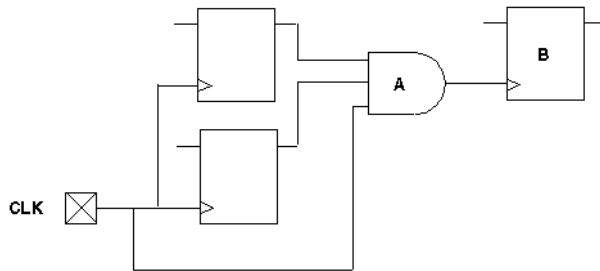
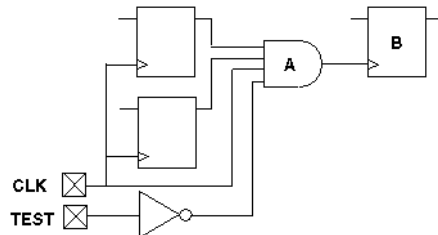
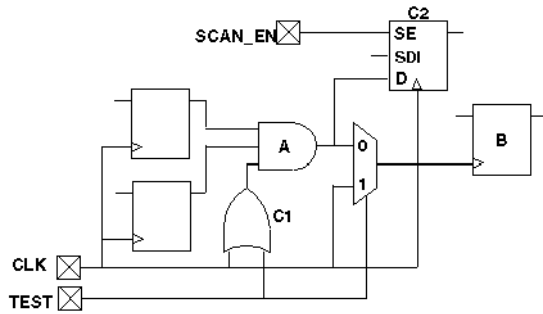


Figure 211 Gated Clock: Solution 1



In the following figure, 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 212 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 213](#). 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 214](#).

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 213 Pulse Generators: A Problem

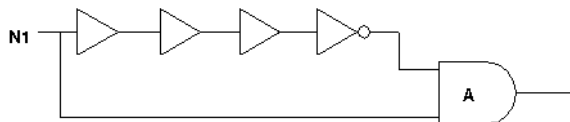
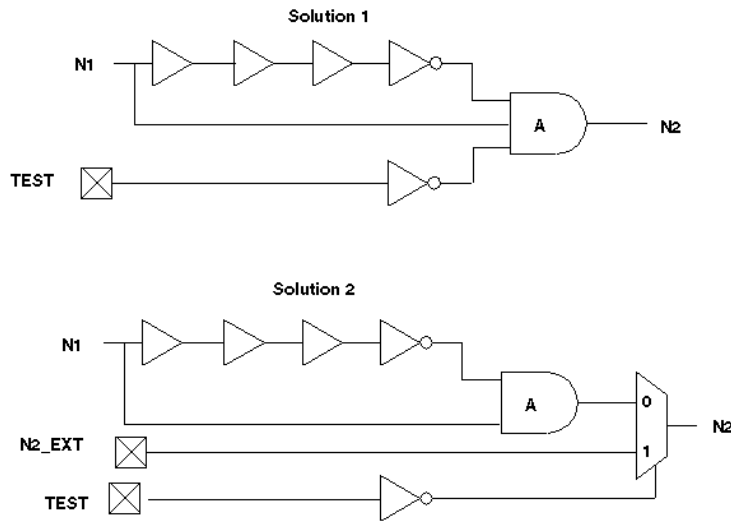


Figure 214 Pulse Generators: Two Solutions



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 215](#).

To prevent a power-on reset circuit from operating during test, you can perform either of the following steps:

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 216](#).

Use the test mode control signal to block the power-on reset source so that it has no effect during test. See [Figure 217](#).

The first of these two methods is usually better because it is less likely to cause a reduction in test coverage.

Figure 215 Power-On Reset Circuit Configuration to Avoid



Figure 216 Power-On Reset Circuit Test Method 1

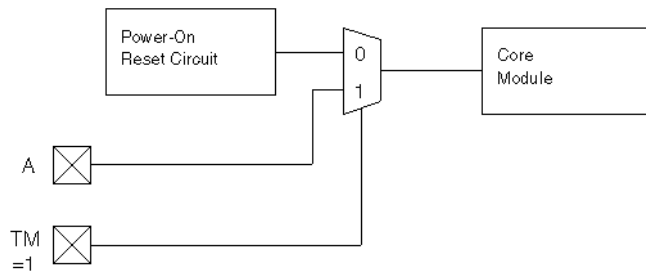
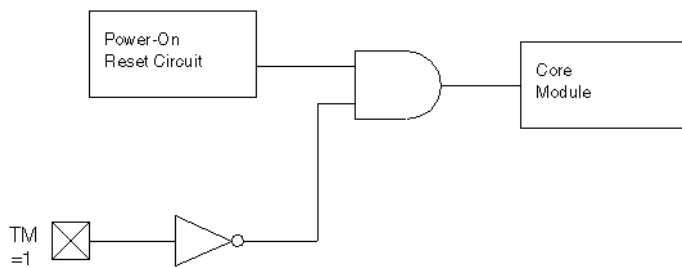


Figure 217 Power-On Reset Circuit Test Method 2



Clock Control

While TestMAX ATPG 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 218](#) and [Figure 219](#).

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 220](#) shows a JTAG input cell with a MUX through which the signal passes; it is difficult to hold the MUX control constant. [Figure 221](#) 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 222](#) and [Figure 223](#).

Figure 218 Clock Paths Through Combinational Gates

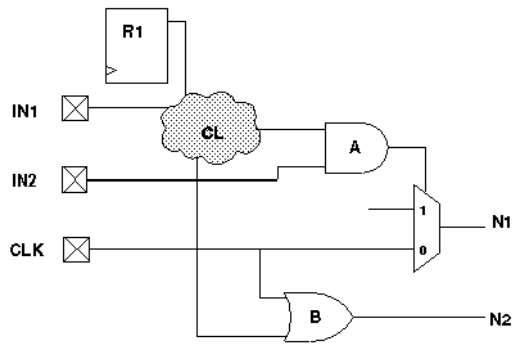


Figure 219 Clock Paths Through Combinational Gates

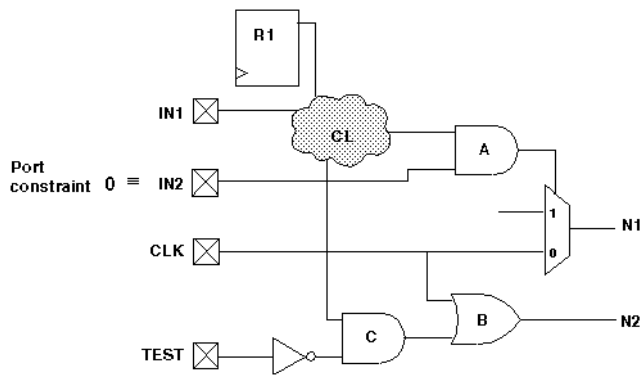


Figure 220 Clock/Set/Reset Inputs and JTAG I/O Cells

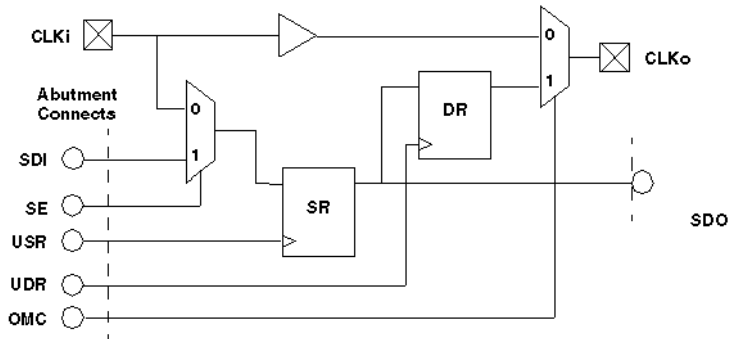


Figure 221 Clock/Set/Reset Inputs and JTAG I/O Cells

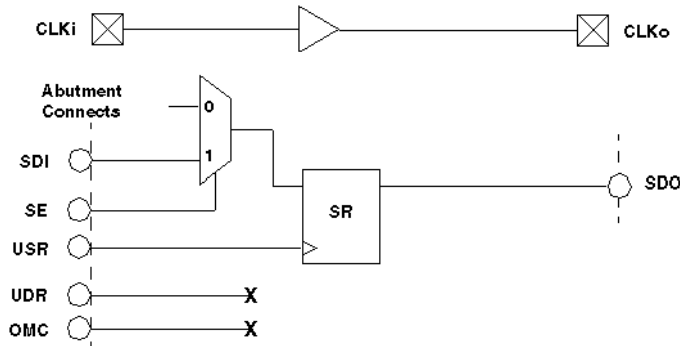


Figure 222 Bidirectional Clock/Set/Reset

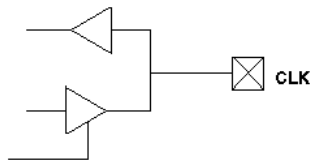
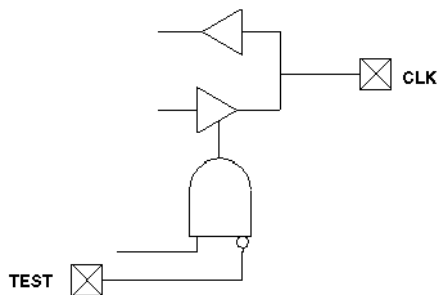


Figure 223 Bidirectional Clock/Set/Reset



Pulsed Signals to Sequential Devices

While TestMAX ATPG 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 TestMAX ATPG is in ATPG test mode. As shown in [Figure 224](#), 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 225](#). 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 TestMAX ATPG 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 224 Sequential Device Pulsed Data Inputs

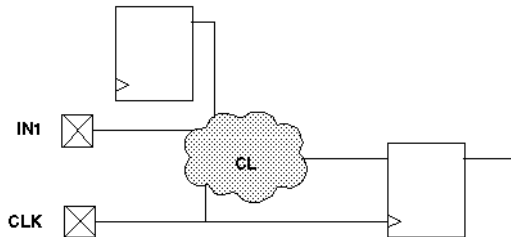
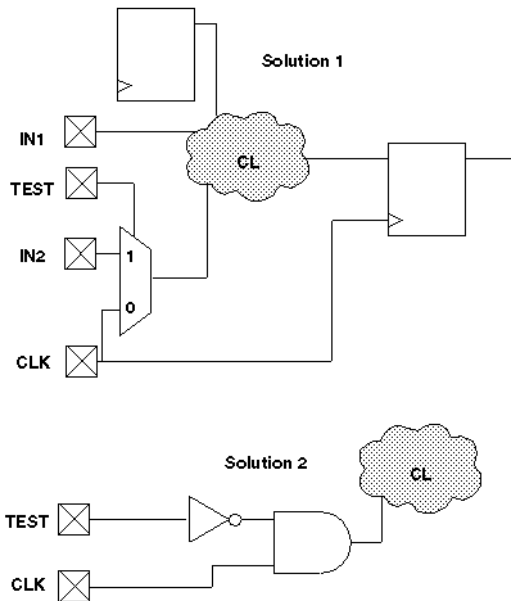


Figure 225 Sequential Device Pulsed Data Inputs



Multidriver Nets

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 226](#).

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 227](#). This primary input port should be asserted during the scan chain load and unload operation. This design guideline is supported by TestMAX DFT and is the default behavior of TestMAX DFT.
- 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 228](#). 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 226 Multidriver Nets

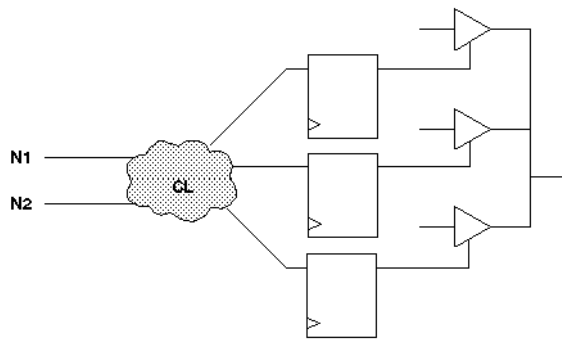


Figure 227 Multidriver Nets: Global Override Input

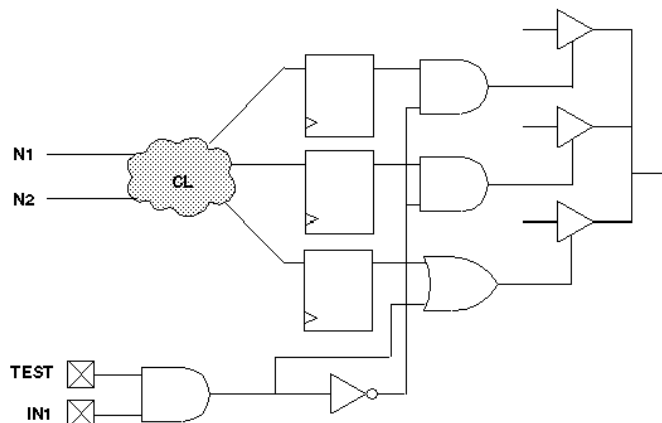
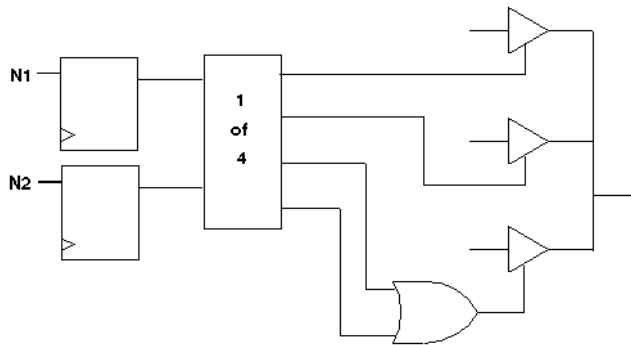


Figure 228 Multidriver Nets: Deterministic Decoding



Bidirectional Port Controls

Force all bidirectional ports to input mode while shifting scan chains in ATPG test mode, using a top-level port as control. See [Figure 229](#) and [Figure 230](#). In [Figure 230](#), 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 process more flexibility in generating patterns.

Figure 229 Bidirectional Port Controls

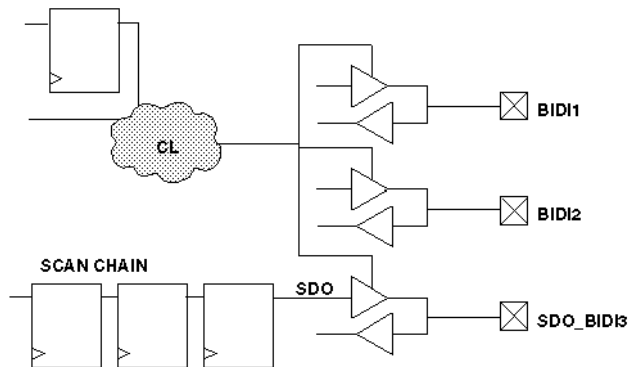
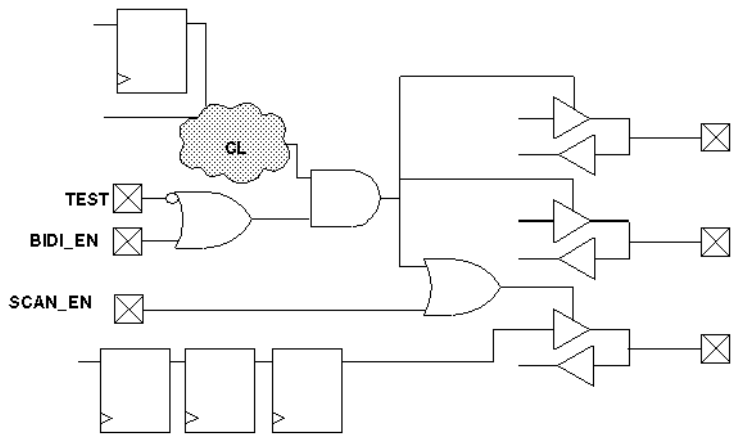


Figure 230 Solution: Bidirectional Port Controls



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 TestMAX DFT (using a single pin) and is the default behavior.

Exception

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 230](#).

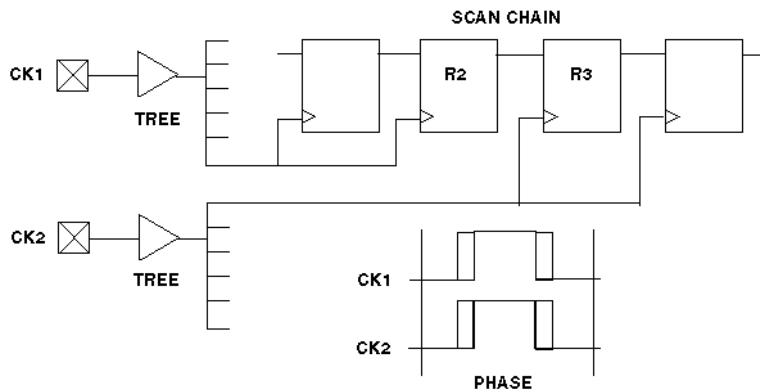
This guideline is the exception to Guideline 5 and is automatically supported by TestMAX DFT if you specify a bidirectional port for use as a scan chain output.

Clocking Scan Chains: Clock Sources, Trees, and Edges

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 the following figure, 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.

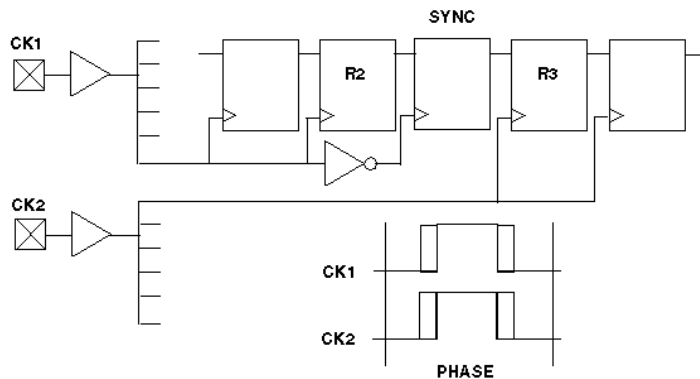
Figure 231 Problem: Multiple Clock Trees



In the following figure, 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 TestMAX DFT and is the default behavior of TestMAX DFT.

Figure 232 Solution: Multiple Clock Trees



Clock Trees

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 233](#). 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 234](#).

This design style is supported by TestMAX DFT but is not the default method of TestMAX DFT.

Figure 233 Problem: Single Clock With Multiple Clock Trees

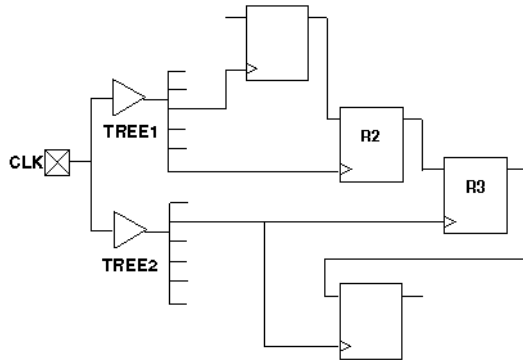
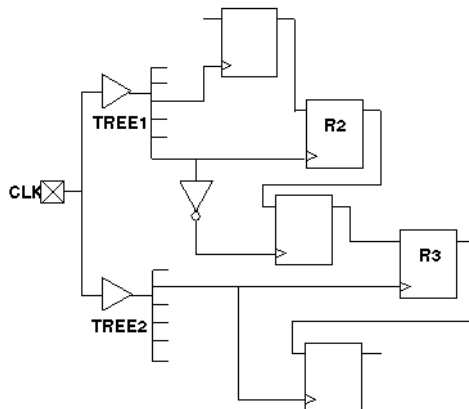


Figure 234 Solution: Single Clock With Multiple Clock Trees



Clock Flip-Flops

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 235](#), 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 236](#), 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 235 Problem: Mixed Clock Edges on a Scan Chain

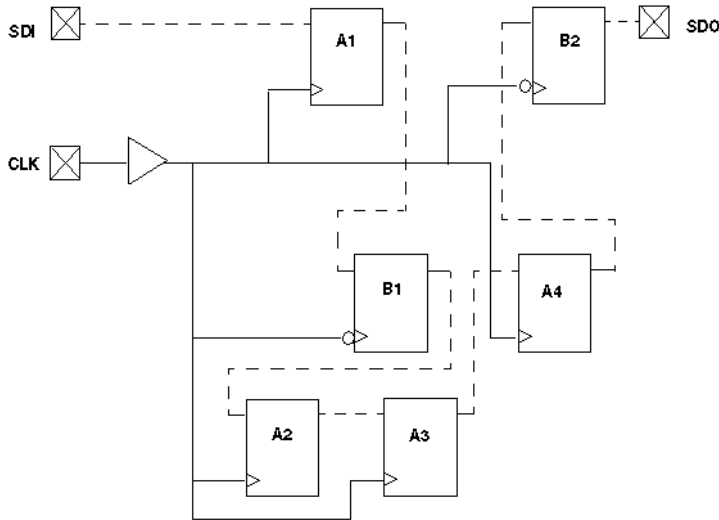
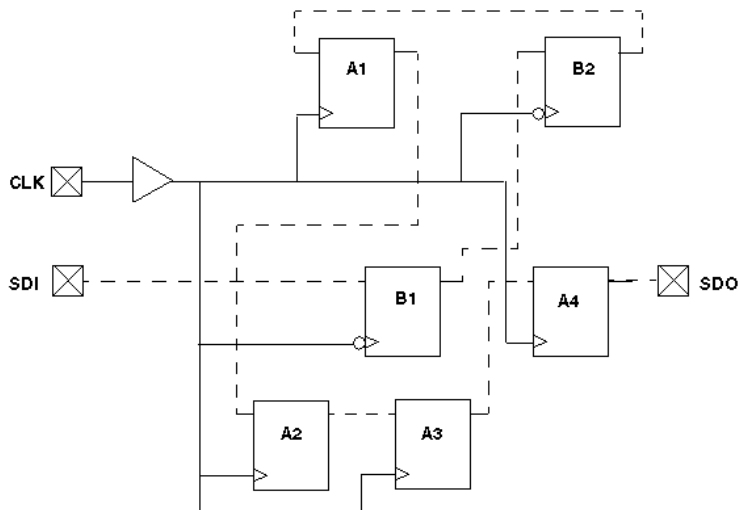


Figure 236 Solution: Mixed Clock Edges on a Scan Chain



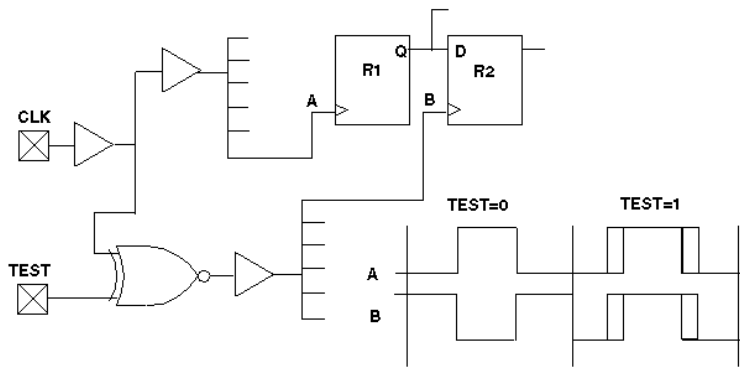
XNOR Clock Inversion and Clock Trees

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.

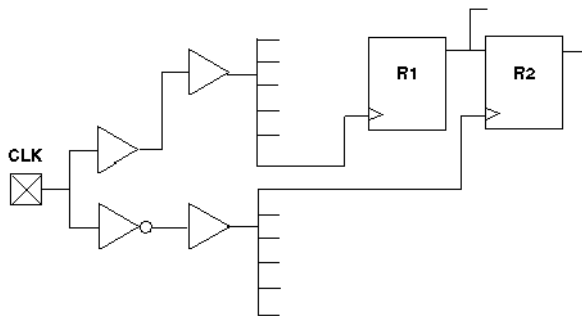
This technique is not advisable unless you can analyze the timing during test mode to ensure that no timing violations can occur during the application of any clocks. While in normal operation, there are essentially two clock zones of opposite phase. The phasing of the two clocks is such that reasonable 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 the following figure.

Figure 237 Problem: XNOR Clock Inversion and Clock Trees



To prevent this problem, replace the XNOR gate with an inverter, as shown in the following figure. 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 238 Solution: XNOR Clock Inversion and Clock Trees



Protection of RAMs During Scan Shifting

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 simultaneously usually presents problems that can be avoided by using separate ports.

RAM and ROM Controllability During ATPG

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.

In [Figure 239](#), 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.

In [Figure 240](#), 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 239 Problem: RAM/ROM Control

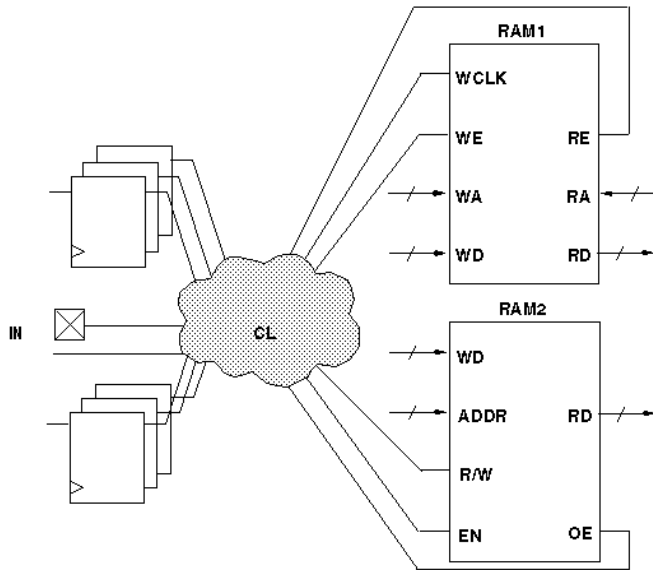
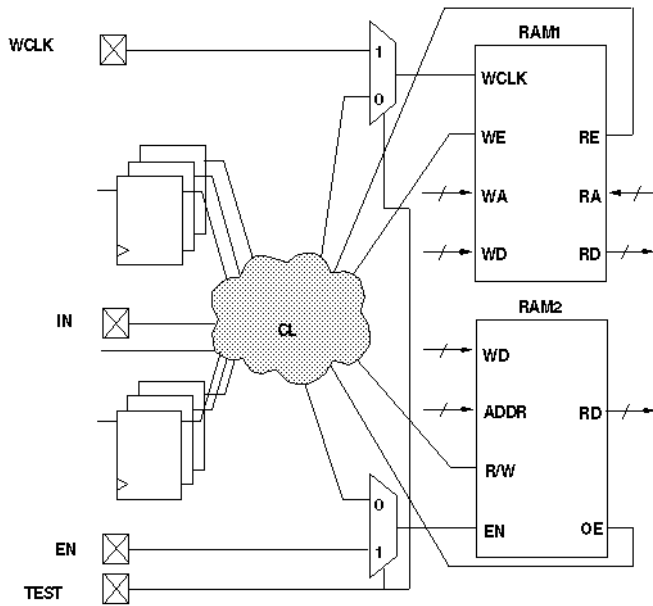


Figure 240 Solution: RAM/ROM Control



Pulsed Signal to RAMs and ROMs

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 process 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 241](#), 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 242](#), the TEST input disables pulsed paths during ATPG test mode.

Figure 241 Problem: RAMs/ROMs and Pulsed Signals

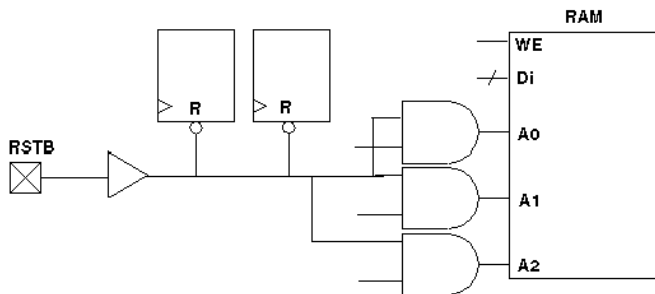
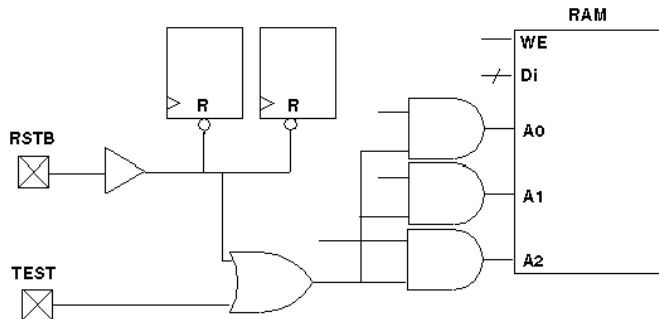


Figure 242 Solution: RAMs/ROMs and Pulsed Signals



Bus Keepers

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 243. In Figure 244, 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 243 Problem: Bus Keepers

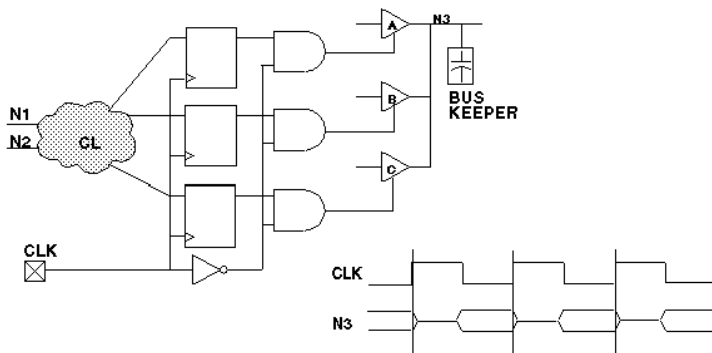
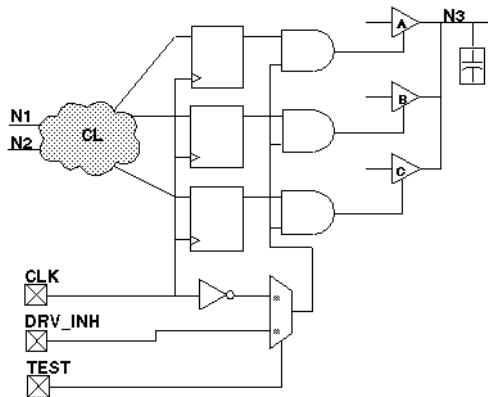


Figure 244 Solution: Bus Keepers



Non-Z State on a Multidriver Net

When using bus keepers, ensure a non-Z state on a multidriver net by the end of the `load_unload` procedure.

Non-Clocked Events

When using bus keepers, do not allow the non-clocked 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.

In [Figure 245](#), 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 246](#), the TEST input disables pulsed paths during ATPG test mode.

Figure 245 Problem: RAMs/ROMs and Pulsed Signals

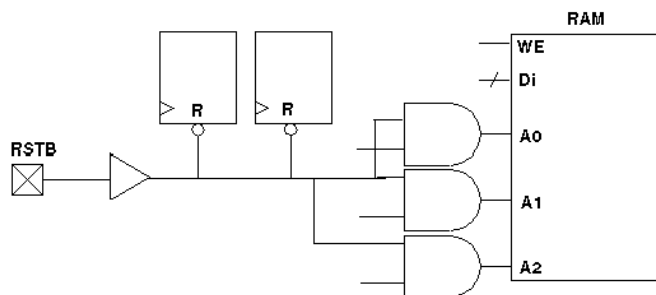
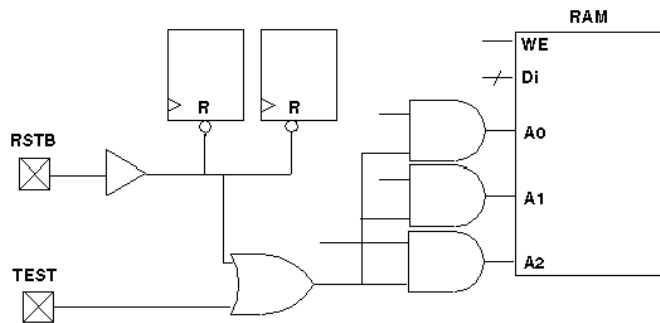


Figure 246 Solution: RAMs/ROMs and Pulsed Signals



Bus Keepers

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 247. In Figure 248, 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 247 Problem: Bus Keepers

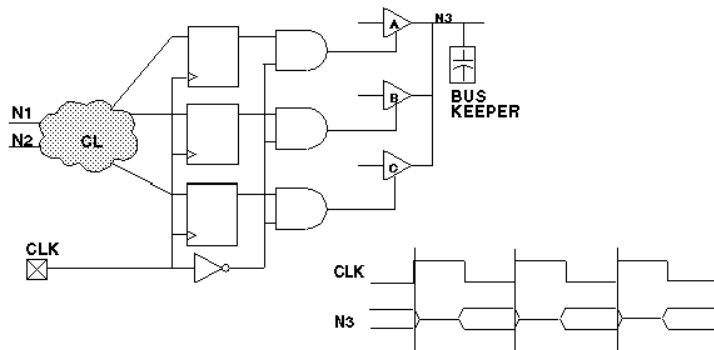
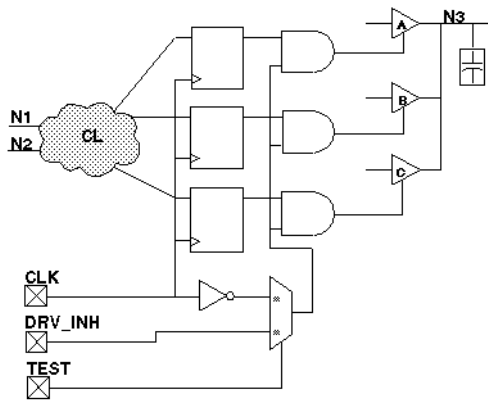


Figure 248 Solution: Bus Keepers



Non-Z State on a Multidriver Net

When using bus keepers, ensure a non-Z state on a multidriver net by the end of the `load_unload` procedure.

Non-Clocked Events

When using bus keepers, do not allow the non-clocked 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.

Checklists for Quick Reference

This section provides checklists of the design guidelines and port redefinition suggestions. Use the following checklists as a convenient reminder as you implement your design:

- [ATPG Design Guideline Checklist](#)
- [Ports for Test I/O Checklist](#)

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 TestMAX DFT

This appendix provides a brief overview of how to take a design from TestMAX DFT to TestMAX ATPG to generate ATPG patterns.

Before you begin, you should be aware of the differences between TestMAX DFT and TestMAX ATPG 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 TestMAX DFT commands in TestMAX ATPG

These differences are explained in detail in the section “Exporting a Design To TestMAX ATPG” in the *TestMAX DFT Scan User Guide* (provided with the TestMAX DFT tool).

Before you import a design from TestMAX DFT to TestMAX ATPG, 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 TestMAX DFT (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 TestMAX ATPG, 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 TestMAX ATPG, 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 STL procedure file.

After you perform these steps, you can read in the design with TestMAX ATPG. For more information on performing these steps, see the section “Exporting a Design To TestMAX ATPG” in the *DFT Compiler Scan User Guide*.

D

Utilities

The following sections of this appendix describe the utility programs supplementing TestMAX ATPG:

- [Ltran Translation Utility](#)
- [Generating PrimeTime Constraints](#)
- [Converting Timing Violations Into Timing Exceptions](#)
- [Importing PrimeTime Path Lists](#)
- [stilgen Utility and Configuration Files](#)

Ltran Translation Utility

When you use the Write Patterns dialog box or the `write_patterns` command to write patterns in the FTDL, TDL91, TSTL2, or WGL_FLAT format, TestMAX ATPG 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.

The following sections provide basic information on starting Ltran in the shell mode, and specifying and modifying Ltran configuration files:

- [Ltran in the Shell Mode](#)
- [FTDL, TDL91, and TSTL2 Configuration Files](#)
- [Understanding the Configuration File](#)
- [Configuration File Syntax](#)

If there is a problem with your Ltran installation, the following error message might appear:

```
sh: /stil2wgl.sh: No such file or directory
```

In this case, you should make sure the following files are in your `$installation/$platform/syn/ltran` directory:

- Ltran
- Ltran.sh
- gzip stil2wgl
- stil2wgl.sh
- vread.bin
- vread3.bin
- vread5.bin

If these files are not in this directory, you should go to the SolvNet Download Center, obtain the download instructions for TestMAX ATPG, and perform the installation.

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 TestMAX ATPG stops until Ltran finishes. This is different than the xterm version that kept TestMAX ATPG 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 might appear in the UNIX transcript in which TestMAX ATPG was started, but that output will not be captured in the TestMAX ATPG 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 can write out TDL91, TSTL2, and FTDL pattern formats from the TestMAX ATPG Linux shell. This is especially true if you receive a `"sh: xterm: command not found"` message.

FTDL, TDL91, and 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, a separate Ltran translation process is executed. This process begins as an independent operation in the new window. You can perform other tasks while the translation process is 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 TestMAX ATPG Installation](#).

The configuration files contained in this directory are:

- `stil2ftdl` : STIL to FTDL
- `stil2tdl91` : STIL to TDL91
- `stil2tstl2` : STIL to TSTL2
- `wgl2ftdl` : WGL to FTDL
- `wgl2tdl91` : WGL to TDL91
- `wgl2tstl2` : WGL 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 an intermediate STIL format file. Ltran then translates the STIL patterns to the target format using the conversion parameters specified in the `stil2ftdl`, `stil2tdl91`, or `stil2tstl2` configuration file.

You can also use the `-wgl` option to generate an intermediate WGL format file, and Ltran will use the provided WGL-related configuration files. However, in most cases, writing a STIL intermediate file is much faster than writing WGL file; this can save minutes of time for large pattern files.

The default files are adequate for most translations. However, 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 the following example.

Example 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](#).

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 required }
```

- Designer name

```
DESIGNER = "Designer", { edit "Designer" as required }
```

- Test vector function

```
TNAME = "FUNC", { edit "FUNC" as required }
```

- Test vector name

```
CNAME = "TEST", { edit "TEST" as required }
```

- Date of design file creation

```
DATE = "99/10/05" ; { edit DATE as required }
```

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 required }
```

- Customer

```
CUSTOMER = "Customer", { edit "Customer" as required }
```

- Part number

Appendix D: Utilities

Ltran Translation Utility

```
TI_PART_NUMBER = "PartNum", { edit "PartNum" as required }
```

- **Pattern set name**

```
PATTERN_SET_NAME = "SetName", { edit "SetName" as required }
```

- **Pattern set type**

```
PATTERN_SET_TYPE = "SetType", { edit "SetType" as required }
```

- **Revision number**

```
REVISION = "1.00", { edit REVISION as required }
```

- **Date of design file creation**

```
DATE = "10/5/2009" ; { edit DATE as required }
```

You can also do the following:

- Specify the following general Ltran configuration commands:
 - `-AUTO_GROUP` — Enables Ltran to algorithmically identify similar signals and group them in the TDL_91 pattern output file.
 - `SD_PORT = "SD"` — Enables you to specify a port name to be added to the end of each scan cell name to form the scan cell shift input pin name. The default port name is "SD". If you set this to a null string, then no text is added.
- Reference your custom configuration file when creating patterns with the `write_patterns` command. Exclude the scan chain test when writing TDL91 patterns, for example:

```
TEST-T> write_patterns <pattern_file_name> -format tdl91 \
-config_file spec_CUSTOM_FILE -exclude chain_test -replace
```

- When translating STIL/WGL files an additional flag can be set in the `TABULAR_FORMAT` statement which instructs Ltran to look for Header information at the beginning of the STIL/WGL file and pass it through to the TDL_91 output file. This flag is `-TDL91_INFO` and is used as follows:

```
TABULAR_FORMAT stil -cycle, -scan, -include_cells, -TDL91_INFO ;
```

OR

```
TABULAR_FORMAT wgl -cycle, -scan, -include_cells, -keep_annotations,
-TDL91_INFO ;
```

Note that this only applicable to translations from STIL/WGL files generated by TestMAX ATPG to TDL_91 format.

Customizing the TSTL2 Configuration File

For TSTL2 output, the `write_patterns` command uses the `wgl2tstl2` configuration file. You can customize this configuration file by editing the following parameters:

- Title

```
TITLE = "TITLE", { edit "TITLE" as required }
```

- Function Test

```
FUNCTEST = "FC1" { edit "FC1" as required }
```

- Scale

```
scale 1000;
```

Place the `scale` statement in the `PROC_BLOCK` section of the `stil2tstl2` or `wgl2tstl2` configuration file. The `scale 1000` statement in the previous example adjusts the scaling and resolution from the default, in nanoseconds (ns), to picoseconds (ps).

For example, take a signal defined as follows:

```
"rst" { P { '0ps' U; '50006ps' D; '52400ps' U; } } "rst" { P { '0ps' U; '50001ps' D; '52600ps' U; } } "rst" { P { '0ps' U; '45000ps' D; '55000ps' U; } }
```

With the `scale` value set to 1000, the TSTL2 output is as follows:

```
TIMESSET(2) NP, 50006, 2394 ; TIMESSET(2) NP, 50001, 2599 ; TIMESSET(2) NP, 45000, 10000 ;
```

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 based 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` 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.

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.

Appendix D: Utilities

Ltran Translation Utility

```
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 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 before 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 before 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.

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.

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 Fundamentals User Guide* and the *PrimeTime Advanced Timing Analysis User Guide* for these details.

The following sections describe how to generate PrimeTime Constraints:

- [Input Requirements](#)
- [Starting the Tcl Command Parser Mode](#)
- [Setting Up TestMAX ATPG](#)
- [Making Adjustments for OCC Controllers](#)
- [Performing an Analysis for Each Mode](#)
- [Implementation](#)

Input Requirements

The TestMAX ATPG input data requirements are:

- Netlists
- Library
- STIL procedure file
- Tcl command script for `build`, `run_drc` commands, and so on.
- An image file can only be used if it is written using the command `write_image -netlist_data`.

The PrimeTime input data requirements are:

- Netlists
- Technology library (.db files)
- Command scripts to read design, link, and so on
- Timing models
- Layout data (for example, SDF)

Starting the Tcl Command Parser Mode

To use this flow, you must run the tool in Tcl command parser mode, which is the default mode for TestMAX ATPG starting with the C-2009.06 release.

The command files must be in Tcl format and not in the native format. You can use the TestMAX ATPG command translation script, `native2tcl.pl` to convert native mode TestMAX ATPG command scripts into Tcl command scripts. For instructions on how to download this script, see [Converting TestMAX ATPG Command Files to Tcl](#).

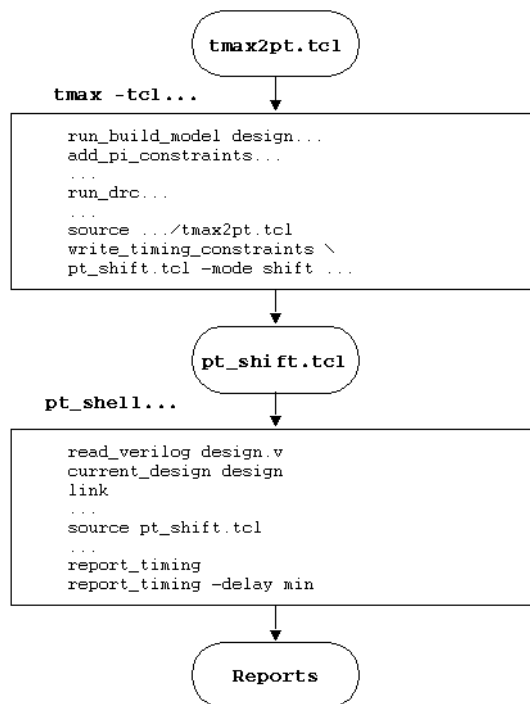
Setting Up TestMAX ATPG

The normal flow of configuring the design and TestMAX ATPG for ATPG is required. However, ATPG does not have to be run. After you run DRC and set the configuration, TestMAX ATPG has enough data to support generating the PrimeTime script.

The `tmax2pt.tcl` script is located in the `$SYNOPTSYS/auxx/syn/tmax` directory. This script must be sourced from TestMAX ATPG (see the following figure); for example:

```
TEST-T> source $env(SYNOPTSYS)/auxx/syn/tmax/tmax2pt.tcl
```

Figure 249 Shift Mode Analysis Example



The `write_timing_constraints` procedure is part of the `tmax2pt.tcl` script. Use this procedure to create a PrimeTime Tcl script. For example:

```
TEST-T> write_timing_constraints pt_shift.tcl -mode shift
...

```

The syntax for the `write_timing_constraints` procedure is as follows:

```
write_timing_constraints output_pt_script_file [-debug] [-man] [-mode shift |
capture | last_shift | update] [-no_header] [-only_constrain_scanouts]
[-replace] [-wft wft_name | default | launch | capture] launch_capture
```

```
[-wft wft_name | default | launch | capture | launch_capture] ] [-unit ns  
| ps]
```

Argument	Description
-debug	Writes additional debug data into the output file. This is useful if you are attempting to modify this script.
-man	Displays a detailed description of the <code>write_timing_constraints</code> options.
-mode <i>mode_name</i>	Specifies the mode in which to perform timing analysis.
-no_header	Suppresses header information in the output file. This is useful for comparing the results of different versions.
-only_constraint_scanouts	Sets output delay constraints only on scanout ports. By default, all outputs are constrained. This option is only compatible with the <code>-mode shift</code> option.
-replace	Overwrites the output PrimeTime script file, if it exists.
-wft <i>wft_name</i>	Specifies the WaveformTable as defined in the STIL protocol file from which the timing data is gathered. If well-known WFT names are defined, they can be abbreviated as follows: <code>default</code> (<code>_default_WFT_</code>), <code>launch</code> (<code>_launch_WFT_</code>), <code>capture</code> (<code>_capture_WFT_</code>), <code>launch_capture</code> (<code>_launch_capture_WFT_</code>). This option can be specified two times, if necessary.
-unit <i>unit</i>	Specifies ps (picoseconds) if the protocol uses ps. The default is ns (nanoseconds).

The `write_timing_constraints` procedure and options accept abbreviations.

The `mode_name` can be either `shift`, `capture`, `last_shift`, or `update`. Shift mode uses the constraints from the `load_unload` procedure and configures the design to analyze timing during scan chain shifting. Capture mode (the default) uses constraints from the capture procedures and configures the design to analyze timing during the capture cycles. Last_shift mode analyzes the timing of the last shift cycle and the subsequent capture cycle. This is normally used for analyzing the last shift launch transition pattern timing. Update mode analyzes the timing of the last shift cycle, capture cycle, and first shift cycle to determine the timing of the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches.

The `-wft` option causes the timing used for the analysis to be specified separately from the mode specification. The argument to the `-wft` option must be a valid WaveformTable in the SPF. Well-known WFT names can be abbreviated. You can use the `-wft` option one or two times in a single command. If two WFTs are specified, two cycles are timed. The first WFT is used for the first cycle timing and the second WFT is used for the second cycle. Two-cycle analysis is done by superimposing two cycles, offset by a period, for each clock. The default WFT name is `._default_WFT_`.

You should call the `write_timing_constraints` procedure for each mode. A separate script is created for each mode, and sourced in PrimeTime during separate sessions.

The following examples show the usage of the `-mode` and `-wft` options.

To validate shifting:

```
-mode shift -wft _slow_WFT_
```

To validate stuck-at capture cycles:

```
-mode capture -wft default
```

To validate system clock launch capture cycles for transition faults:

```
-mode capture -wft launch -wft capture
```

To validate the timing between shift and capture for transition faults:

```
-mode last_shift -wft default -wft _fast_WFT_
```

Note the following:

- The 'force PI' and 'measure PO' times are relative to virtual clocks in PrimeTime. The 'force PI' virtual clock rises at 0, and the 'measure PO' clock falls at the earliest PO measure time. Input and output delays are specified relative to these clocks.
- For the two WFT modes, all the clock ports will have two superimposed clocks representing the two cycles that need to be analyzed.
- The end-of-cycle measures produce cycle times of double the normal cycles to account for the expansion of vectors into multiple vectors.
- You should carefully review the generated PrimeTime script to ensure the static timing analysis configuration is as expected.
- 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 TestMAX ATPG is sourced in PrimeTime; for example:

```
pt_shell> source pt_shift.tcl
```

Making Adjustments for OCC Controllers

If you source the script written by the `write_timing_constraints` procedure inside the `pt_shell`, and an internal clock source (for example, `OCC_controller_clock_root`) is included, the following message is echoed:

```
TMAX2PT WARNING: Internal clock OCC_controller_clock_root timing is
defaulted
```

Adjust this timing to correct values before checking.

In this case, the script written by the `write_timing_constraints` procedure does not include all of the information required to perform the clock gating check in PrimeTime. The clock gating check is important and should be done for both maximum and minimum timing.

The following steps show you how to create a clock gating check script from the script written by the `write_timing_constraints` procedure:

1. Locate the `create_clock` commands for each OCC clock, and change the `source_object` to the PLL source for the OCC.
2. In each corresponding `create_generated_clock` command, change the `-source` argument to match the PLL source.
3. Add the following command to the clock gating check script:

```
set_clock_gating_check -high OCC_clock_inst
```

In this case, `OCC_clock_inst` is the instance name (without the pin name) of the OCC clock source. This step is required for OCC controllers that use multiplexors or combinational gating. However, you must skip this step for OCC controllers that use integrated clock-gating latches, since they already have clock gating checks defined for them in the library.

4. Add the following commands to enable the clock gating check to verify the slow (shift) clock gating in addition to the fast (capture) clock gating:

```
remove_case_analysis scan_enable
set_false_path -from scan_enable -to [get_clocks OCC_clock ]
```

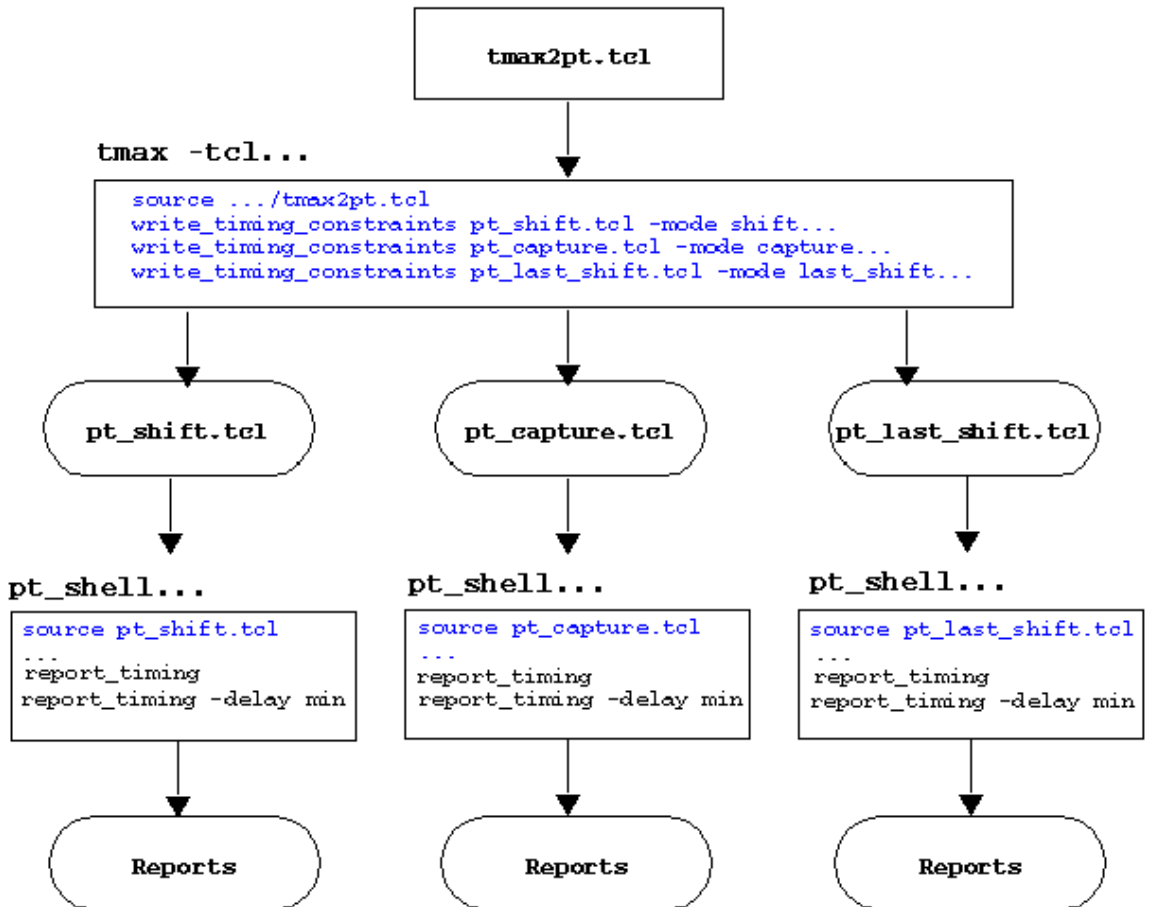
In this case, `OCC_clock` refers to all OCC clocks defined by the `create_clock` or `create_generated_clock` commands.

After you make these changes, the clock gating check is performed when you run the `report_timing` command. For more discussion of static timing analysis with OCC clocks, see [SolvNet Article #022490: Static Timing Analysis Constraints for On-Chip Clocking Support](#).

Performing an Analysis for Each Mode

As discussed previously, the flow involves performing a separate timing analysis for each mode. This is illustrated in the following figure. Example usages for various common modes are given in the following paragraphs.

Figure 250 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 STL procedure file 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 STL procedure file 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 STL procedure file 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 STL procedure file 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 TestMAX ATPG 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.

You can use the `-mode update` option to analyze timing for the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches. The suggested usage is as follows:

```
TEST-T> write_timing_constraints pt_update.tcl -mode update
```

With this mode, the constraints file defines either a `$dftmax_ultra_cache_cells` variable or a `$spcc_cache_cells` variable, depending on the compression architecture. In PrimeTime, use this variable to check the cache register timing, as shown in the following example:

```
pt_shell> report_timing -to $dftmax_ultra_cache_\  
cells -delay min_max  
pt_shell> report_timing -from $dftmax_ultra_cache_cells \  
-delay min_max
```

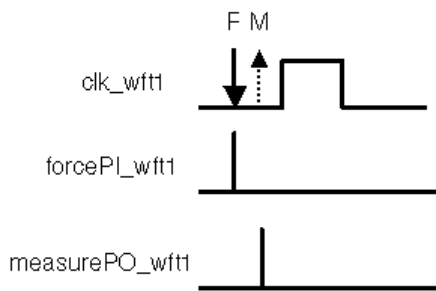
This analysis mode is intended only for analyzing the DFTMAX Ultra cache registers or the DFTMAX shift power groups control chain latches. You should still perform full-design analysis with constraints generated for the shift and capture modes.

If you are generating separate constraints for transition and stuck-at timing, you do not need to do this for update mode because the stuck-at timing is the worst case for updating the cache registers due to the shorter capture cycle. If on-chip clocking is used, the constraints should not be used for any other purpose than checking timing to and from the variables, because the clock definitions must be modified in this case.

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 the following figure). 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 that used for the TestMAX ATPG DRC run.

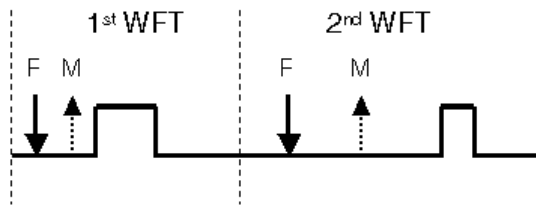
Figure 251 Waveforms Used For Timing



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 might 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 the following figure. The WFTs used are based on the order specified.

Figure 252 Superimposed Cycles For Two WFTs



Note:

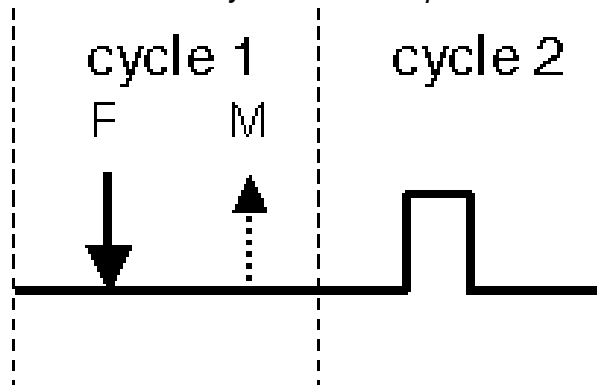
The `create_generated_clocks` command is used to allow clock reconvergence pessimism reduction to work on the two pulses.

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.

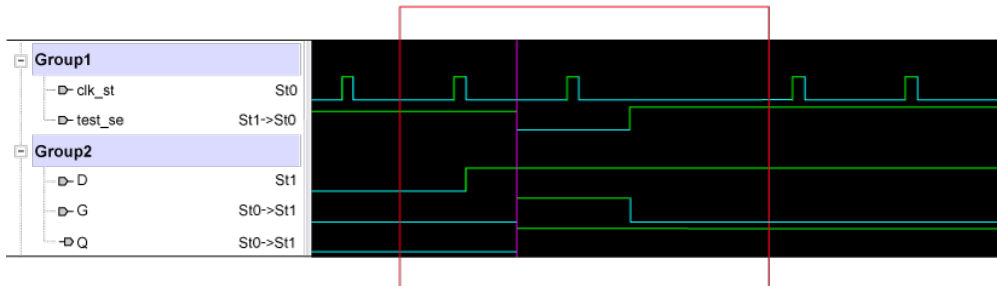
For capture cycles of end-of-cycle measures, the waveforms are expanded into a two-cycle period to adjust for the expansion of each capture vector into multiple vectors (see the following figure). Shift cycles remain single cycle.

Figure 253 End of Cycle Pattern Expansion



Another check is the update mode timing requirement. In the update mode, the cache registers in the decompressor must capture data at the conclusion of the scan shift operation. When the next scan shift operation starts, the data from the cache registers must meet setup time requirements at their destinations. This means that the cache register data propagates in the entire capture operation. The time window for this check is the last scan shift of one pattern, followed by the capture operation, and completed by the first scan shift of the next pattern.

Figure 254 Update Mode Timing Requirement



Along with the scan clocks, the scan enable (test_se in the preceding figure) is defined as a clock. This is required because the scan enable is used as the clock of the cache registers, which means the update mode can't be combined with other static timing analysis modes. The lower waveforms show a typical bit of the cache register and the position in which the bit is updated during the check.

Converting Timing Violations Into Timing Exceptions

Timing exceptions can negatively impact ATPG efficiency. You can ensure ATPG efficiency by using only those timing exceptions that apply directly to the current ATPG environment and ignoring others that are irrelevant to tester timing. The following flow describes how to convert timing violations into timing exceptions:

1. Set up TestMAX ATPG in the appropriate mode.
2. Follow the steps described in [Generating PrimeTime Constraints](#).
3. Use the `write_exceptions_from_violations` PrimeTime Tcl procedure described in this section.
4. Read the results into TestMAX ATPG using the `read_sdc` command. (Don't read the results of step 2 at this time, since they are implicit in the TestMAX ATPG setup and will reduce efficiency if applied again.)

The `write_exceptions_from_violations` procedure is part of the `$$SYNOPTSYS/auxx/syn/tmax/pt2tmax.tcl` script. To use this Tcl script, source it into `pt_shell`, set up the timing environment, and then run it. This script converts all timing violations into timing exceptions, then applies them to ensure that timing is clean. When the timing isn't clean, the newly found timing violations are converted. Each update check conversion process is considered an iteration. The new exceptions are written in SDC format.

The syntax for the `write_exceptions_from_violations` procedure is as follows:

```
write_exceptions_from_violations  
[-output filename]
```

Appendix D: Utilities

Converting Timing Violations Into Timing Exceptions

```
[-specific_start_pin]
[-max_iterations number]
[-delay_type <max | min | min_max>]
[-full_update_timing]
[-pba]
[-slack crit_slack]
[-man]
```

Argument	Description
- output <i>filename</i>	Writes the output to a specific file name. The default is <code>tmax_exceptions.sdc</code> .
- specific_start_pin	Writes separate exceptions for different outputs of a violating cell. The default is one exception per startpoint cell. This switch can improve the efficiency of the timing exceptions on some designs, especially designs in which some memory paths violate timing but other paths with the same memories do not. However, this analysis requires multiple iterations, which can dramatically increase the runtime.
- max_iterations <i>number</i>	Iterates the specified <code>number</code> of times before placing blanket exceptions on endpoints to ensure that timing is met. The default is 40.
- delay_type <max min min_max>	Specifies which violations to convert to timing exceptions. The <code>max</code> argument converts setup time violations to exceptions. The <code>min</code> argument converts hold time violations to exceptions. The <code>min_max</code> argument (the default) converts both setup and hold time violations to exceptions.
- full_update_timing	Forces a full timing update for the second iteration, and all later iterations, of the update check conversion process. You should use this option when violating paths cause excessive runtime during timing updates.
-pba	Runs timing analysis using the "Path" mode of path-based analysis. In most cases, this option reduces the number of violating paths. However, this additional analysis affects the runtime.

Argument	Description
<code>-slack</code> <code>crit_slack</code>	Sets the minimum non-violating slack. The critical slack can be positive or negative. The default is 0.0. You can use this option to reduce the number of timing exceptions when testing several paths with small timing violations. In this case, use a small negative number as the critical slack. This option can be used for other purposes since the critical slack can be positive or negative.
<code>-man</code>	Prints the syntax message.

Importing PrimeTime Path Lists

The `pt2tmax.tcl` file included with TestMAX ATPG consists of a Tcl procedure, called `write_delay_paths`, which is used for both internal and I/O path selection. This Tcl procedure generates a list of critical paths in the required DSMTest format according to the criteria you specify.

You will need to set the case analysis in PrimeTime to correspond with the device in test mode and operating on the tester. This can be done automatically using the script written by the `write_timing_constraints` command in the `tmax2pt.tcl` utility (see [Appendix D](#)).

You should never use negative-slack (failing) paths for hold time ATPG. For path delay ATPG, you might want to exclude negative-slack paths, depending on the application of the patterns. To prevent negative-slack paths from being included in the path file, you should first run the `write_exceptions_from_violations` `pt2tmax.tcl` command, then run the `write_delay_paths` command. One of the effects of the `write_exceptions_from_violations` command is that it sets all violating paths as false paths in PrimeTime, so they will not be considered by the `write_delay_paths` command. When running the `write_exceptions_from_violations` command, make sure to use either the `-delay_type min` or `-delay_type min_max` switches for hold paths, and either the `-delay_type max` or `-delay_type min_max` switches for delay paths.

The `write_delay_paths` procedure is used for both delay paths and hold time paths. Note that it isn't necessary to write out hold time paths to or from I/O ports, or use different launch and capture clocks, since ATPG will not attempt to detect these paths.

The syntax for `write_delay_paths` procedure is as follows:

```
write_delay_paths  
filename  
[-capture clock_name]  
[-cell pin_name]  
[-clock clock_name]
```

Appendix D: Utilities

Importing PrimeTime Path Lists

```
[-delay_type <max | min>]
[-group group_name]
[-help]
[-IO [-each]]
[-launch clock_name]
[-man]
[-max_paths num_paths]
[-net pin_name]
[-noZ]
[-nworst num_per]
[-pba]
[-slack crit_time]
[-version]
```

Argument	Definition
<i>filename</i>	Name of the file where the paths are written.
<code>-capture <i>clock_name</i></code>	Selects the paths ending at the <i>clock_name</i> domain. The <code>-capture</code> option is incompatible with the <code>-clock</code> , <code>-group</code> , and <code>-IO</code> options.
<code>-cell <i>pin_name</i></code>	Selects the path(s) for each input of a cell connected to the <i>pin_name</i> . The <code>-cell</code> option is incompatible with the <code>-each</code> and <code>-net</code> options.

Argument	Definition
<code>-clock <i>clock_name</i></code>	Selects the paths in <i>clock_name</i> domain.
<code>-delay_type <max min></code>	The <code>max</code> argument (the default) writes paths suitable for path delay ATPG. The <code>min</code> switch writes paths suitable for hold time ATPG.
<code>-each</code>	Selects the path(s) for each I/O. The <code>-IO</code> option must also be specified. The <code>-each</code> option is incompatible with the <code>-cell</code> and <code>-net</code> options.
<code>-group <i>group_name</i></code>	Selects paths from the existing <i>group_name</i> or selects a list of path groups and writes delay paths for every path group in the list. -The <code>-max_paths</code> option is applied separately to each group. The <code>-group</code> option is incompatible with the <code>-capture</code> , <code>-clock</code> , <code>-IO</code> , and <code>-launch</code> options.
<code>-help</code>	Prints syntax information for this procedure.
<code>-IO</code>	Writes I/O paths. The default is to only write internal paths.

Argument	Definition
<code>-launch <i>clock_name</i></code>	Selects paths starting from the specified clock domain. The <code>-launch</code> option is incompatible with the <code>-clock</code> , <code>-group</code> , and <code>-IO</code> options.
<code>-man</code>	Prints help information for this procedure.
<code>-max_paths <i>num_paths</i></code>	Specifies the maximum number of paths to be written. The default is 1.
<code>-net <i>pin_name</i></code>	Selects the path(s) for each fanout connected to the specified <i>pin_name</i> . The <code>-net</code> option is incompatible with the <code>-cell</code> and <code>-each</code> options.
<code>-noZ</code>	Suppresses paths through three-state enables. Note that this option is case-sensitive.
<code>-nworst <i>num_per</i></code>	Specifies the number of paths to each endpoint. The default is 1.
<code>-pba</code>	Uses the exhaustive-effort level of path-based analysis to gather paths.
<code>-slack <i>crit_time</i></code>	Writes paths with a slack less than the specified <i>crit_time</i> . The default is 1,000,000.
<code>-version</code>	Reports the version number.

Note that the `write_delay_paths` procedure and options accept abbreviations. The `pt2tmax.tcl` file, found under `$SYNOPTSYS/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

Appendix D: Utilities

Importing PrimeTime Path Lists

- `$slack` - Available time margin between the `$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 first node in the path 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 can 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 {
. ;
. ;
. ;
}
```

```
}
```

stilgen Utility and Configuration Files

The `stilgen` utility can be used for either [pattern porting](#) or protocol generation. You can access this utility at the following location:

```
$SYNOPSIS/linux64/syn/bin/stilgen
```

The `stilgen` syntax and configuration file syntax for pattern porting and protocol generation are different and are described in the following sections:

- [Using stilgen for Pattern Porting](#)
- [Using stilgen for Protocol Generation](#)
- [Pattern Porting Example](#)
- [Protocol Generation Notes](#)
- [Supported Configurations](#)
- [Limitations](#)

Using stilgen for Pattern Porting

The following `stilgen` syntax is used for pattern porting:

```
stilgen -config_file config.txt [-top_spf file_name] [-core_stil file_name]  
[-output_ported_stil file_name] [-verbose] [-check_only] [-print_include]  
[-exclude option_name] [-gzip] [-help]
```

Argument	Description
<code>-config_file</code> <i>config.txt</i>	This is a required configuration file. For details, see stilgen Configuration File Syntax for Pattern Porting .
<code>-top_spf</code> <i>file_name</i>	Overwrites the top-level protocol file specified in the configuration file.
<code>-core_stil</code> <i>file_name</i>	Overwrites the core patterns file specified in the configuration file.
<code>-</code> <code>output_porte</code> <code>d_stil</code> <i>file_name</i>	Specifies the file name or overwrites the file name in the configuration file for output ported patterns file.
<code>-verbose</code>	Prints verbose information on processed sections.

Argument	Description
<code>-check_only</code>	Checks the accuracy of the configuration file information.
<code>-print_include</code>	Prints the sections identified by the include files.
<code>-exclude option_name</code>	Excludes printing the information pointed to by an option in the pattern section. Accepted options: setup, patterns and all.
<code>-gzip</code>	Saves the output file in .gz format.
<code>-help</code>	Prints command usage and configuration file syntax.

stilgen Configuration File Syntax for Pattern Porting

The configuration file for pattern porting describes the input files and output files, and the relationship between the core-level test ports and top-level test ports. The syntax for this file is as follows:

Field	Description
<code>.TOP_SPF</code>	Specifies the top-level STIL protocol file with the top-level design name and the pattern_exec to be used. Example: <code>.TOP_SPF top_tmax.spf des_unit wrp_if</code>
<code>.CORE_STIL</code>	Specifies the core-level STIL-based pattern file and its port mapping file, the instance name of the core, the core design, and the pattern exec in the same order for all cores. Example: <code>.CORE_STIL U1_core.stil u1_portmap.txt U1 CORE1 wrp_if U2_core.stil u2_portmap.txt U2 CORE2 wrp_if U2_core.stil u2_portmap.txt U3 CORE3 wrp_if</code>
<code>.OUTPUT_PORTED_STIL</code>	Specifies the output top-level ported STIL-based patterns. Example: <code>.OUTPUT_PORTED_STIL ported.stil</code>
<code>.MAP_FORMAT</code>	Specifies the mapping format: <code>core2top</code> or <code>top2core</code> . The default is <code>top2core</code> . This format defines the scan-in ports, scan-out ports, clocks, and scan-enable signals. Example: <code>.MAP_FORMAT top2core</code>

Field	Description
<code>.TMAX_SPF</code>	Specifies the source of the top-level protocol. Use 1 if generated by TestMAX ATPG and 0 otherwise. Example: <code>.TMAX_SPF 1</code>

Port-Mapping File Syntax

The port-mapping file for pattern porting describes the top-level and corresponding core-level scan-in, scan-out, scan clock, and scan-enable/wrapper-shift ports:

Field	Description
<code>.SCAN_IN</code>	Specifies the mapping list or file names of top-level and corresponding core-level scan-in ports.
<code>.SCAN_OUT</code>	Specifies the mapping list or file names of top-level and corresponding core-level scan-out ports.
<code>.CLOCK</code>	Specifies the mapping list or file names of top-level and corresponding core-level clock ports.
<code>.SCAN_EN</code>	Specifies the mapping list or file names of top-level and corresponding core-level scan-enable/wrapper-shift ports.

Using stilgen for Protocol Generation

The following stilgen syntax is used for protocol generation:

```
stilgen -config_file config.txt -protocol
```

Argument	Description
<code>-config_file</code> <code>config_file</code> <code>config.txt</code>	This is a required configuration file. For details, see stilgen Configuration File Syntax for Protocol Generation .
<code>-protocol</code>	This option is required for protocol generation.

stilgen Configuration File Syntax for Protocol Generation

The syntax for the configuration file used for protocol generation is as follows:

INPUT_SPF	<p>Specifies the input protocol file with optional port mapping file and pattern_exec. Example: INPUT_SPF input.spf port_map -patternexec Internal_scan</p>
OUTPUT_SPF	<p>Specifies the name of the generated protocol file. Example: OUTPUT_SPF output.spf</p>
DFTMAX_SPF	<p>Specifies the name of DFTMAX CODEC protocol files to be ported into the output protocol. Syntax: DFTMAX_SPF <i>SPF_file</i> [<i>port_map_file</i>] [-shared_si] [-uniquify_string <i>string</i>] -exclude_chain <i>chain_name</i> Excludes specified chain that exists in DFTMAX_SPF -uniquify_string <i>string</i> Specifies the string used to uniquify each codec. -shared_si Specify this option when using shared inputs and dedicated outputs protocol generation flows.</p>
LOAD_UNLOAD_SPF	<p>Replaces the load_unload section of INPUT_SPF file with the load_unload specified protocol file. Syntax: LOAD_UNLOAD_SPF <i>SPF_file</i> [-patternexec <i>name</i>]</p>
TEST_SETUP_SPF	<p>Replaces the test_setup section of INPUT_SPF file with the test_setup section in the specified protocol file. Syntax: TEST_SETUP_SPF <i>SPF_file</i> [-patternexec <i>name</i>]</p>
CLOCKSTRUCTURES_SPF	<p>Replaces the ClockStructures section of the INPUT_SPF file with the clockstructures section in the specified protocol file. Syntax: CLOCKSTRUCTURES_SPF <i>SPF_file</i> [-patternexec <i>name</i>] [-patternexec <i>name</i>] [-prefix <i>name</i>]</p>
ADD_CLOCK_PORT	<p>Adds the specified port in the clock list with user-specified waveform structure. Syntax: ADD_CLOCK_PORT <i>port_name</i> -period <i>float_value</i> -rise_time <i>float_value</i> -fall_time <i>float_value</i> -signalgroup <i>name</i></p>

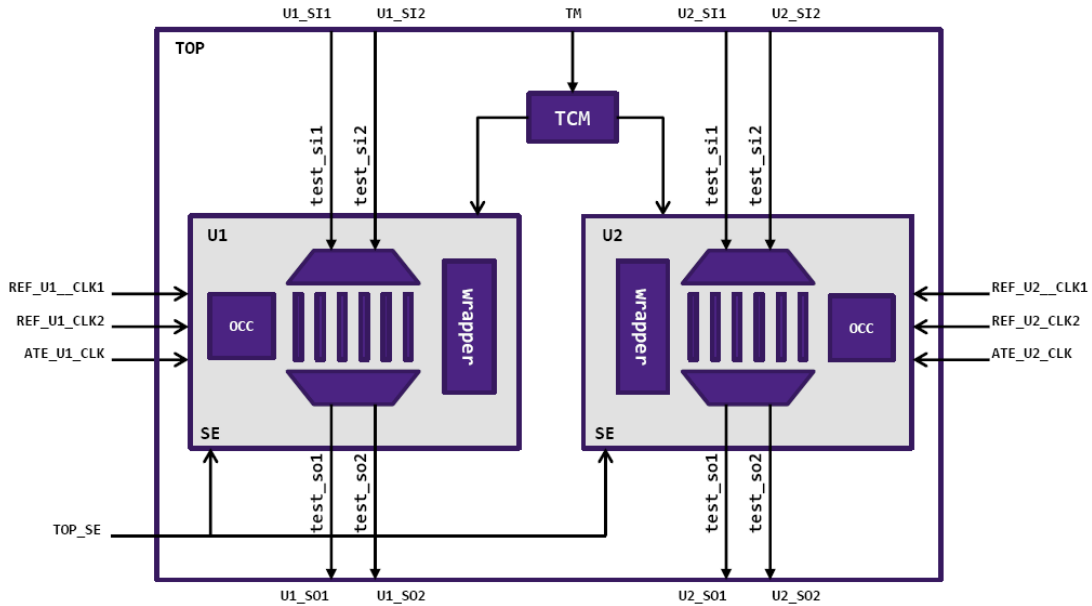
Appendix D: Utilities stilgen Utility and Configuration Files

ADD_PORT	<p>Adds the port in the generated protocol file.</p> <p>Syntax:</p> <pre>ADD_PORT <i>port_name</i> -direction <in out inout> [-signalgroup<_si _so>]</pre>
REMOVE_PORT	<p>Removes the port from the protocol file from all sections, including constraints.</p> <p>Syntax:</p> <pre>REMOVE_PORT <i>port_name1 port_name2</i></pre>
TOGGLE_CONSTRAINT	<p>Toggles the specified port constraint in the specified sections.</p> <p>Syntax:</p> <pre>TOGGLE_CONSTRAINT <i>port_name</i> [-section <all (default) capture load_unload master_observe test_setup>]</pre>
LOAD_PIPELINE_STAGES	<p>Specifies the total load pipeline stages in the design.</p> <p>Syntax:</p> <pre>LOAD_PIPELINE_STAGES <i>integer_value</i></pre>
UNLOAD_PIPELINE_STAGES	<p>Specifies the total unload pipeline stages in the design.</p> <p>Syntax:</p> <pre>UNLOAD_PIPELINE_STAGES <i>integer_value</i></pre>
MAP_FORMAT	<p>Specifies the format used in the port mapping file. Accepted strings are <code>core2top</code> or <code>top2core</code></p>
SIMILAR_PORT	<p>Specifies the file with the list of ports which are similar. The file format uses two columns:</p> <ul style="list-style-type: none"> • The first column is the new port name listed in ADD_PORT field or ADD_CLOCK_PORT field. • The second column is the old port name of the INPUT_SPF field. <p>This will assign the old port values to the new port in TIMING, PROCEDURES, MACRO sections.</p> <p>Syntax:</p> <pre>SIMILAR_PORT <i>file_name</i></pre>
CHANGE_SIGNAL_GROUP	<p>Changes either:</p> <ul style="list-style-type: none"> • The port from one signal group to another • Adds a port to a specified signal group • Removes a port from a specified signal group <p>Syntax:</p> <pre>CHANGE_SIGNAL_GROUP <i>port_name</i> [-from <i>signal_group_name</i>] [-to <i>signal_group_name</i>]</pre>
TEST_SETUP_WAVEFORM	<p>Specifies a waveform definition to be used with TEST_SETUP_SPF.</p>

Appendix D: Utilities stilgen Utility and Configuration Files

CAPTURE_PROC _SPF	Replaces the capture procedure from the file specified by the INPUT_SPF field to the capture procedure from the specified protocol file. Syntax: CAPTURE_PROC_SPF <i>SPF_file</i> [-patternexec <i>name</i>]
UNIQUIFY_STR ING	Specifies a uniquifying string to uniquify the DFTMAX codecs. Syntax: UNIQUIFY_STRING <i>name</i>
CHANGE_CONST RAINT	Changes the constraint of a specified port in the user-specified sections. Syntax: CHANGE_CONSTRAINT <i>01XNZP port_name</i> [-section <all (default) capture load_unload master_observe test_setup>]
REMOVE_CONST RAINT	Removes the constraint of a specified port in the user-specified sections. Syntax: REMOVE_CONSTRAINT <i>port_name</i> [-section <all (default) capture load_unload master_observe test_setup>]
ADD_CONSTRAI NT	Adds the constraint of a specified port in the user-specified sections. Syntax: ADD_CONSTRAINT <i>01XNZPport_name</i> [-section <all (default) capture load_unload master_observe test_setup>]
ADD_SCAN_CHA IN	Adds a scan chain with the specified scan signals in the protocol file. Syntax: ADD_SCAN_CHAIN <i>chain_name scan_in_port scan_out_port</i>

Pattern Porting Example



The example figure shows two cores U1 and U2 and their scan-clocks, scan-inputs, and scan-outputs. Each core has dedicated scan-ins and dedicated scan-outs.

The following is an example of a configuration file for pattern porting:

```
.OUTPUT_PORTED_STIL top_ported.stil
.TOP_SPF top_tmax.spf TOP
.CORE_STIL
core1.stil u1_portmap.txt U1 core1 ScanCompression_mode_occ_bypass
core2.stil u2_portmap.txt U2 core2 ScanCompression_mode_occ_bypass
.TMAX_SPF 1
```

The following is an example of port-mapping files of cores U1 and U2.


```

• u1_portmap.txt      • u2_portmap.txt
.SCAN_IN              .SCAN_IN
test_si1 U1_SI1      test_si1 U2_SI1
test_si2 U1_SI2      test_si2 U2_SI2
.SCAN_OUT             .SCAN_OUT
test_sol U1_S01       test_sol U2_S01
test_so2 U1_S02       test_so2 U2_S02
.SCAN_EN              .SCAN_EN
SE TOP_SE             SE TOP_SE
.CLOCK                .CLOCK
ate_clk ATE_U1_CLK    ate_clk ATE_U2_CLK
ref_clk1 REF_U1_CLK1  ref_clk1 REF_U2_CLK1
ref_clk2 REF_U1_CLK2  ref_clk2 REF_U2_CLK2

```

Protocol Generation Notes

Note the following when using stilgen for generating a STIL protocol file:

- When a STIL protocol file is used for updating procedures, you must use the complete protocol and cannot use sections of the protocol file.

Example:

```
TEST_SETUP_SPF spf_file
```

- There are cases when using DFTMAX serializer or DFTMAX Ultra. The following example is not supported by the stilgen utility:

```
"CORE_0_U_deserializer_ScanCompression_mode[26]" = 1
```

To correct the protocol, do one of the following steps:

- Use the following syntax:

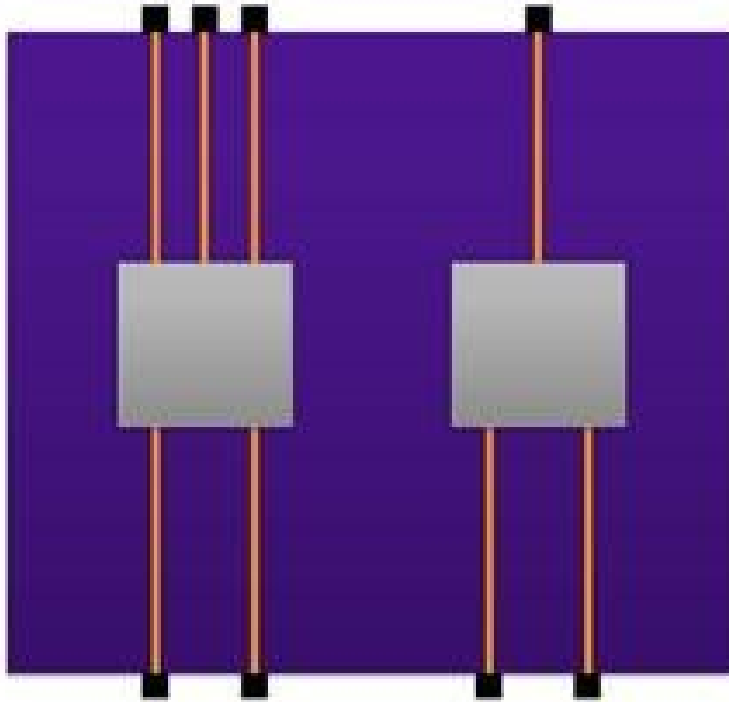
```
"CORE_0_U_deserializer_ScanCompression_mode"[26] = 1
```

- Use the stilgen utility to generate the protocol.
- Specify `set test_serializer_pseudo_signals_change true` in the TestMAX DFT scripts to regenerate correct protocol files.

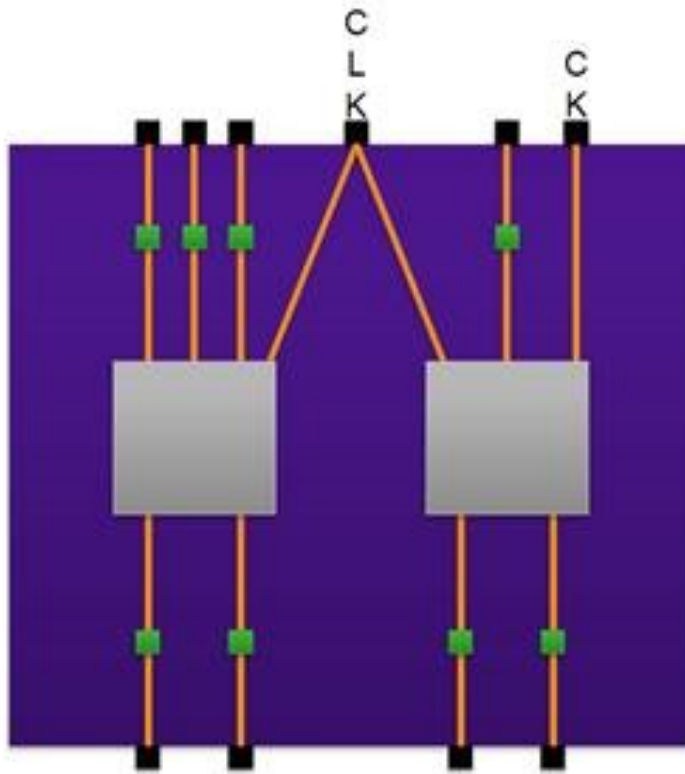
Supported Configurations

The following are some of the configurations supported by STILGen:

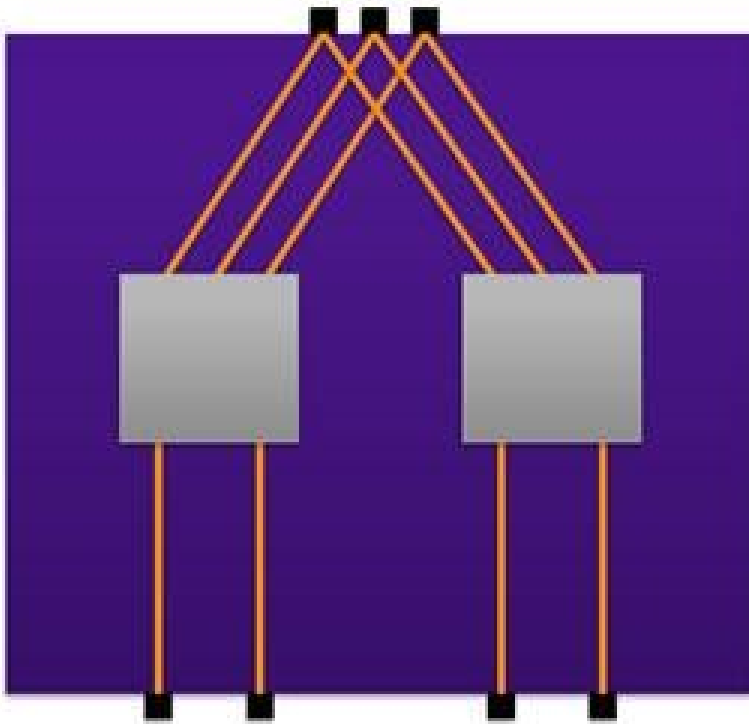
1. Dedicated scan-ins and dedicated scan-outs.



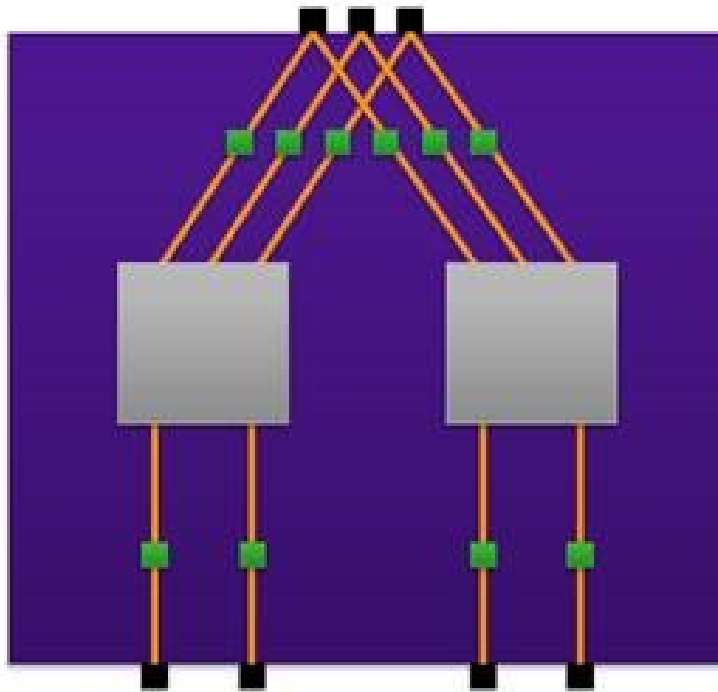
2. Dedicated scan-ins and dedicated scan-outs with pipelines at the top-level.



3. Shared scan-in and dedicated scan-outs.



4. Shared scan-in and dedicated scan-outs with pipelines at the top-level.



Limitations

STILGen has the following limitations:

1. OCC insertion is not supported at the top-level.
2. Mixing modes like uncompressed mode with compressed mode is not supported.
3. The top-level pipeline number needs to be the same across the cores.

E

STIL Language Support

The following sections of this appendix provide a brief overview of the Standard Test Interface Language (STIL), and identifies how TestMAX ATPG uses the STIL constructs.

- [STIL Overview](#)
- [TestMAX ATPG and STIL](#)
- [STIL Conventions in TestMAX ATPG](#)
- [IEEE Std. 1450.1 Extensions Used in TestMAX ATPG](#)
- [Elements of STIL Not Used by TestMAX ATPG](#)
- [Testing the STIL Procedure File](#)

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. TestMAX ATPG 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 TestMAX ATPG 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](#)
- [IEEE Std. 1450.1 Design Extensions to STIL](#)

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 might 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 might be capable of restructuring the data to support that capability on the tester; in other circumstances, separate tools might 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 rule checking (and tester rules checking) and drives the test generation process. ATPG-generated STIL patterns might be structured such that intra cycle timing, cyclization of test and raw data are separated into, respectively, Timing, Procedure, 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

TestMAX ATPG 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 TestMAX ATPG users and subsequently proposed to the 1450.1 working group. While these extensions are used by TestMAX ATPG, 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 is available through the normal IEEE channels. Until the ballot is complete, copies might 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>

TestMAX ATPG and STIL

TestMAX ATPG uses STIL in several different contexts. Design information can be provided to TestMAX ATPG through the STIL procedure file . TestMAX ATPG supports a subset of STIL syntax for input to describe scan chains, clocks, constrained ports, and pattern/response data as part of the STL procedure file definitions. Complete test sets can be written out in STIL format. Also, Tester Rules Checking is provided through STIL-formatted files.

In constructing a STL procedure file, you can define the minimum information needed by TestMAX ATPG. However, any STIL files written as TestMAX ATPG output contain an expanded form of the minimum information and may also contain pattern/response data that the ATPG process produces. TestMAX ATPG reads and writes in STIL, so after a STIL file is generated for a design, TestMAX ATPG can read it again at a later time to recover the clock/constraint/chain data, the pattern/response data, or both.

TestMAX ATPG can read some constructs that it does not generate. For example, an external pattern source can be read into TestMAX ATPG for fault simulation but it cannot be written out with the same constructs as were 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 TestMAX ATPG

STIL supports a very flexible data representation. TestMAX ATPG has defined conventions in the use of STIL to represent data in a uniform manner yet maintain this flexibility. These conventions are discussed in the following sections:

- [Use of STIL Procedures](#)
- [Context of Partial Signal Sets in Procedure Definitions](#)
- [Use of STIL SignalGroups](#)
- [WaveFormCharacter Interpretation](#)

Use of STIL Procedures

When possible, TestMAX ATPG generates calls to STIL procedures from the pattern body. STIL procedures are used in general by TestMAX ATPG 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 might 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 might 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 standalone 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 TestMAX ATPG’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, TestMAX ATPG supports the definition of a `load_unload` procedure in the STL procedure file that references signals later in the procedure (for example, during the shift block) that might 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 TestMAX ATPG, the `load_unload` procedure is “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 TestMAX ATPG leverages the flexibility of an incomplete definition for generating other test formats, in particular WGL.

TestMAX ATPG 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 might 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 might be valuable to consider the “default” handling of unused signals in the procedure to allow generation of a test that behaves similarly to the TestMAX ATPG-generated WGL test.

Use of STIL SignalGroups

TestMAX ATPG makes use of STIL `SignalGroups` to simplify creation of STIL protocol information. The STIL procedures file might be a complete set of information for certain sections of the final STIL file, or it might be an incomplete file completed by TestMAX ATPG 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 STL procedure file creation. In order to support this operation, however, TestMAX ATPG must assume certain naming conventions. Also, the grouping conventions that TestMAX ATPG 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 TestMAX ATPG 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.

TestMAX ATPG 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 TestMAX ATPG is maintained in the output patterns. This supports

direct correspondence of the test set with the internal operations performed by TestMAX ATPG, which in turn reduces confusion between TestMAX ATPG-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 might be written dependent on the capture procedure constructs, and the capture procedures might not be changed without changing the functionality of the pattern.

WaveFormCharacter Interpretation

TestMAX ATPG 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 the following table for a description of the WFC interpretations supported by TestMAX ATPG.

Table 26 Supported WaveFormCharacter Interpretations

W	Interpretation
F	
C	
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 is 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.

Table 26 Supported WaveFormCharacter Interpretations (Continued)

W Interpretation
F
C

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.
X	This waveform is used by TestMAX ATPG to indicate a no-measure operation. From a STIL perspective, the contents of this waveform can be empty; the absence of activity might imply ATE operations to inhibit previous output measures. This waveform assumes the previous drive state continues to be asserted when used in bidirectional contexts; TestMAX ATPG will define a drive state before 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 TestMAX ATPG

The IEEE Std. 1450.1 extensions used by TestMAX ATPG are identified and described in the following sections:

- [Vector Data Mapping Using \m](#)
- [Vector Data Mapping Using \j](#)
- [Signal Constraints Using Fixed and Equivalent](#)
- [ScanStructures Block](#)

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 waveform characters 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 example protocol might not be required to avoid bidi contentions. However, ATPG tools might 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 is 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 (HL), 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 specWFT; // 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 specWFT;  
  
"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](#)” 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; }  
  
}
```

The following table shows examples of “two data” conditions on an InOut.

Table 27 “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 might 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 required 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 might “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 might be seen by these two groups. It might have a WFCMap declaration like

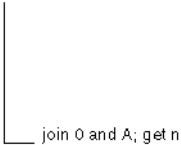
```
WFCMap 0x -> 0;
WFCMap 1x -> 1;
... etc. ...
```

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 the following figure presents:

Figure 1 WFC Example

```
WFCMap 0A -> n;
WFCMap cn -> m;
V { b = 0; b = \jA; }
```



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 previous examples 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 previous example 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 waveform characters are assigned to the same signal in a given vector, all but the last one are ignored. Thus, the previous example 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 application 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 waveform character assignment. Thus, the 1450.1 procedure would be written as:

```
Procedures {  
  
  "capture" {  
  
    V { "_pi" = \j ### ; "_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 previous example:

```
V { i=0; b1=Z; b2=1; o=H; b1=H; b2=X; }
```

to the required:

```
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 might 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 TestMAX ATPG

The following sections list the STIL output and input constructs that are not used in this version of TestMAX ATPG:

- [TestMAX ATPG STIL Output](#)
- [TestMAX ATPG STIL Input](#)

Note that this list is provided to you as a guide for tools that are designed to read the STIL output file generated from TestMAX ATPG.

The only elements of 1450.1 used by TestMAX ATPG are identified previously in 1450.1 Extensions Used in TestMAX ATPG; all other elements of 1450.1 are not used by TestMAX ATPG.

This information is provided specifically from the context of TestMAX ATPG as a standalone tool. TestMAX ATPG-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 TestMAX ATPG alone.

TestMAX ATPG STIL Output

Here is a list of output constructs that TestMAX ATPG 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.

The TestMAX ATPG internal pattern source will not write or produce STIL with the constructs described in this section. Future versions might make use of these constructs as necessary.

11. UserKeywords

TestMAX ATPG does not generate any UserKeywords. All keywords used are those defined in the standard.

12. UserFunctions

TestMAX ATPG does not generate any UserFunctions. All timing expressions use expressions that are defined in the standard.

17. PatternBurst > SignalGroups, MacroDefs, Procedures, ScanStructures (named domains)

TestMAX ATPG 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

TestMAX ATPG 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)

TestMAX ATPG 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 can be specified from other STIL contexts

18. Timing > WaveformTable > Inherit...

TestMAX ATPG does not use the Inherit statement within WaveformTables. All waveform tables are completely self-contained.

18. Timing > WaveformTable > SubWaveforms

TestMAX ATPG 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

TestMAX ATPG 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]

TestMAX ATPG does not use 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

TestMAX ATPG 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

TestMAX ATPG does not generate any expect events. Only drive and compare events are used.

19. Spec, Selector

TestMAX ATPG does not generate either Spec or Selector blocks. All timing values are specified as absolute numbers.

21.1 Cyclized data > \d

TestMAX ATPG does not generate data using the decimal notation, which is selected by use of the \d escape sequence.

21.2 Multiple-bit cyclized data

TestMAX ATPG does not generate any multiple bit vector information. All vectors contain only one wfc per signal.

21.3 Non-cyclized data

TestMAX ATPG 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

TestMAX ATPG 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

TestMAX ATPG does not generate any MatchLoop conditions. All patterns vectors are executed in a straight-line sequence.

22.8 Goto statement

TestMAX ATPG does not generate any Goto statements. All patterns vectors are executed in a straight-line sequence.

22.9 Breakpoint

TestMAX ATPG does not generate any Breakpoint statements. It is assumed that all vectors will fit into available ATE memory.

22.11 Stop

TestMAX ATPG 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

TestMAX ATPG does not generate the TimeUnit statement. This statement is only used with uncyclized data, which is not generated by TestMAX ATPG.

TestMAX ATPG STIL Input

Here is a list of constructs that TestMAX ATPG 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.

Testing the STIL Procedure File

After creating or editing an SPF, you can reread the file and perform syntax checking of your procedures by executing the following `run_drc` command:

```
DRC-T> run_drc filename.spf
```

During DRC, TestMAX ATPG simulates each procedure and checks it against a number of rules. If you encounter a DRC violation that requires editing the STIL procedure 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-T> run_drc
```

TestMAX ATPG rereads the STIL procedure file each time the `run_drc` command is executed.

Not all DRC violations are severe enough to stop TestMAX ATPG from entering TEST command mode. To rerun DRC when in TEST command mode, first return to SETUP command mode, as follows:

```
TEST-T> drc  
DRC-T> run_drc
```


F

STIL99 Versus STIL

This appendix provides an overview of the differences between the STIL99 and STIL pattern formats.

Table 28 *STIL 99 Versus STIL*

STIL 99	STIL
<pre>statement STIL 1.0; STIL 1.0 { TRC 2006; } (only if <<>>resource_tags present) Header { Title " TestMAX ATPG ..."; Date "Tue Feb ..."; Source "comment"; History { Ann {* ...previous Annotations in the History section *} Ann {* ...fault, pattern, drc reports, clocks and constrained pins *} } }</pre>	<pre>statement STIL 1.0 { Design 2005; } STIL 1.0 { TRC 2006; } (only if <<>>resource_tags present) Header { Title " TestMAX ATPG ..."; Date "Tue Feb ..."; Source "comment"; History { Ann {* ...previous Annotations in the History section *} Ann {* ...fault, pattern, drc reports, clocks and constrained pins *} } }</pre>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<p>UserKeywords ScanChainGroups (conditionally) ActiveScanChains (conditionally) ; Conditionally means with respect to the type of the design being parsed through TestMAX ATPG. ObserveValue (seq-comp only) ScanChainPartition (seq-comp only) SeqCompressorStructures (seq-comp only) SerializerStructures (dftmax with serializer only)</p> <p>Ann {* ANNOTATION *} Used only in the Header History section STL procedure file flow has options to preserve Ann in output STIL for top-section of the STIL data (not patterns). Special blocks for Ann {* load_unload <var> <cnt> *} and Ann{* end load_unload *}, reseed_partial_overlap found in stil99 patterns for sequential compressor.</p>	<p>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 TestMAX ATPG. ObserveValue (seq-comp only) ScanChainPartition (seq-comp only) SeqCompressorStructures (seq-comp only) SerializerStructures (dftmax with serializer only)</p> <p>Ann {* ANNOTATION *} Used only in the Header History section STL procedure file flow has options to preserve Ann in output STIL for top-section of the STIL data (not patterns).</p> <p>Variables { (seq-comp only, -stil only) Integer "var_name"; (seq-comp only) } (seq-compr only)</p>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>Signals { "sig": 1. Always quoted 2. Does not use [] array notation; used for Pseudo only in seq-comp 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)</pre>	<pre>Signals { "sig": 1. Always quoted 2. Does not use [] array notation; used for Pseudo only in seq-comp 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) WFCMap { 0X->0; 1X->1; ZX->Z; NX->N; PX->P; P[0 1]->P; }) * // end WFCMap] } See Appendix E for more details on WFCMap. The mapping operation is specified in either the Signals or the SignalGroups.</pre>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<p>SignalGroups { (No signalgroups domain name) Supports user names and generates specific groups: _pi lists all inputs+bidirections, _po lists all outputs+bidirectionals. { } format used if the following attributes are present: ScanIn; (no integer number) ScanOut; (no integer number)</p>	<p>SignalGroups "user_name" { (No signalgroups domain name) Supports user names and generates specific groups: _pi lists all inputs+bidirections, _po lists all outputs+bidirectionals. { } 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; (_pi _po, -stil only)PX->P; } FreeRunning{ Period time; LeadingEdge time; TrailingEdge time; OffState <D U>; }</p>
<p>TestMAX ATPG will accept the following Predefined SignalGroups:</p> <ul style="list-style-type: none"> • _in = input pins • _out = output pins • _io = bidirectional pins • _pi = inputs + bidirectional pins • _po = outputs + bidirectional pins • _si = scan chain inputs • _so = scan chain output 	<p>PatternExec { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</p> <p>PatternExec "user_name" { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</p>
<p>PatternExec { (optionally named) Patternburst "_burst_"; (Fixed burst name) }</p>	<p>Default generation if no input patternexecs is a single unnamed patternexec. If named patternexecs are input, you must identify one patternexec to be used by TestMAX ATPG, and only this patternexec is written out.</p>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<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) (ParallelPatList (SyncStart Independent LockStep) { (PAT_NAME_OR_BURST_NAME { (Extend;) }) * // end ParallelPatList} // end of PatternBurst </pre>	<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_) fixed names for features: "_launch_WFT_", and so forth. << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. Period integer_time_units; </pre>	<p data-bbox="976 1014 1419 1098">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_) fixed names for features: "_launch_WFT_", and so forth. << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. Period integer_time_units; </pre> <p data-bbox="976 1440 1419 1493">Note current environment supports integer units of time only.</p>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>Waveforms { groups_or_signal_names { << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. WFC usage in tmax is fixed to the following: inputs: 01 Z N outputs: H L T X clocks: PD E (Always single-WFC references, separated)</pre>	<pre>Waveforms { groups_or_signal_names { << resource_tag >> preserved and passed through. No generation. Supported in 2008.09-sp2. WFC usage in tmax is fixed to the follo wing: inputs: 01 Z N outputs: H L T X clocks: PD 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 ; } ObserveValue { vectored_pseudo_sig_assignment } (userkeyword statement, seq-comp only)</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 ; } ObserveValue { vectored_pseudo_sig_assignment } (userkeyword statement, seq-comp only)</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. } ScanChainPartition "name" { ... } (userkeyword statement, seq-comp only) }</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. } ScanChainPartition "name" { ... } (userkeyword statement, seq-comp only) }</pre>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>sigref_expr = vec_data; //STIL Cyclized Pattern data LIST OF WFCs — for example "_po" = HHHH 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" = HHHH 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>
<p>TestMAX ATPG 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 WaveFormCharacter Interpretation.</p>	<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>
<pre>sigref_expr = serial_data; (LABEL :) "precondition all signals": on initial C in Pattern"pattern N": used in patterns. User labels allowed in procedures and macros V(ector) { (cyclized data) V { cyclized_data_assignments_only } }W(aveformTable) NAME ; W name ;</pre>	<pre>sigref_expr = serial_data; variable_expr := variable_data; (-stil only, seq-comp only) (LABEL :) "precondition all signals": on initial C in Pattern"pattern N": used in patterns. User labels allowed in procedures and macros V(ector) { (cyclized data) V { cyclized_data_assignments_onl y } }W(aveformTable) NAME ; W name ;</pre>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
C { cyclized_data_assignments_only }	C { cyclized_data_assignments_onl y }
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; also used in seq-comp -stil99 Pattern blocks	Loop integer { ... } Allowed in setup procedures & some BIST procs; also used in seq-comp -stil99 Pattern blocks
Vector statements only with constant WFC assignments }	LoopData { ... } (-stil only, seq-comp only) Loop "var_name" { ... } (-stil only, seq-comp only) Vector statements only with constant WFC assignments }
IDDQ TestPoint;	IDDQ TestPoint;
ScanChain CHAINNAME ; ActiveScanChains group_or_chain_names ; Used around Shift blocks; also in seq-comp load_unload procedures (without Shift)	ScanChain CHAINNAME ; ActiveScanChains group_or_chain_names ; Used around Shift blocks; also in seq-comp load_unload procedures (without Shift)
	F(ixed) { (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"

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>Pattern "_pattern_" { Standard pattern structure: <pre>"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>	<pre>Pattern user_specified_pattern_name { -or- Pattern "_pattern_" { (fixed name by default) Standard pattern structure: <pre>"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>
<pre>Procedures { (Unnamed Procedures block) Procedures "diagnosis" { (In some BIST contexts) (PROCEDURE_NAME { TestMAX ATPG 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 { TestMAX ATPG 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>MacroDefs { (Unnamed MacroDefs block) (MACRO_NAME { TestMAX ATPG flow 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 { TestMAX ATPG flow uses fixed name to identify application test_setup, bist_setup macros do not use parameters W { }, C { }, V { } statements } } }</pre>

Table 28 STIL 99 Versus STIL (Continued)

STIL 99	STIL
<pre>PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters can be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters can be specified after the Shift. }</pre>	<pre>PROCEDURE_OR_MACRO_NAME { "load_unload" { (Scan procedure has fixed name) W { }, C { }, V { } statements Scan parameters can be specified before the Shift. Shift { W { }, C { }, V { } statements Scan parameters applied. } W { }, C { }, V { } statements Scan parameters can be specified after the Shift. }</pre>

Parameters Supported in Specific Contexts Only

In TestMAX ATPG 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, TestMAX ATPG 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.

TestMAX ATPG will accept the following predefined SignalGroups that:

- `_in` = input pins
- `_out` = output pins
- `_io` = bidirectional pins
- `_pi` = inputs + bidirectional pins
- `_po` = outputs + bidirectional pins

- `_si` = scan chain inputs
- `_so` = scan chain outputs

If your STIL DRC description defines a symbolic group with the same name as the predefined TestMAX ATPG groups, then your definition supersedes the predefined definition.

There is not a predefined signal group called `_clks`. TestMAX ATPG 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.

G

Defective Chain Masking for DFTMAX

The following sections of this appendix describe the flow for masking defective scan chains in DFTMAX compression:

- [Introduction](#)
- [Running the Flow](#)
- [Examples](#)
- [Limitation](#)

Introduction

Prior to the introduction of this feature, the flow for masking defective scan chains in DFTMAX compression was extremely inefficient. For example, if you found a scan hold violation on a chain from a chip returned from fabrication, you would want to generate patterns as if the entire compression chain was masked. The old flow for masking the defective chain used the `add_cell_constraints` command to place a constraint of “XX” on all the cells in the chain. However, this flow was problematic when the chain contained padding bits (the additional shift cycles required for every pattern; this situation occurs when the chain is either shorter than the longest compression mode chain or if the chain contains pipeline stages). The existing cells of the chain can be easily masked using the `add_cell_constraints` command. However, there’s no simple way to mask the padding bits. These additional bits, which also require masking, had to be manually identified. In order to resolve the patterns, the constraints were externally read in via a separate file.

In the solution described in this appendix, the external file is not required. Instead, TestMAX ATPG identifies defective scan chains based on cell constraints during DRC. If every scan cell of a particular scan chain has a cell constraint of X or XX or OX, then the scan chain is treated as a defective scan chain when you specify both the `run_simulation` and `run_atpg` commands. As a result, padding measures originating from the defective chain is masked.

Running the Flow

The following sections describe the various processes for masking defective scan chains in DFTMAX compression:

- [Placing Constraints on the Defective Chain](#)
- [Generating Patterns](#)
- [Regenerating Patterns](#)

Both the `run_simulation` and `run_atpg` commands support this flow.

Placing Constraints on the Defective Chain

Before running DRC, you need to use the `add_cell_constraints` command to place the constraints on the defective chain. This solution uses a more relaxed condition for identifying defective chains during DRC. Before, every cell of the chain had to have cell constraint “XX” in order for it to completely mask it out. Now, to identify a chain as defective, every cell in a chain needs to have a cell constraint of “X,” “OX,” or “XX.”

Some examples are as follows:

Example 1:

```
add_cell_constraints XX c2 -all
```

Note that the whole chain is constrained at once.

Example 2:

```
add_cell_constraints X c3 -position {0 2}  
add_cell_constraints XX c3 -position {3}  
add_cell_constraints OX c3 -position {4 20}
```

Note that in Example 2, different positions get different constraints and complete chain is constrained by using position fields.

It is important to note that any new `add_cell_constraints` commands you specify do not override the previous `add_cell_constraints` commands you specified if cells overlap between the two commands. A S16 error is issued in this case:

```
Error: Multiple cell constraints defined for chain1-2. (S16-1)
```

During the next step, DRC and TestMAX ATPG identifies one or more defective scan chains, and the following message appears:

```
Scan chain c2 has been identified as a defective chain
```

There are several different flow options available for generating or regenerating patterns. These flows are described in the following sections.

Generating Patterns

For generating patterns, you can use any of the following four flows:

- *Option 1:*

```
set_patterns external <pat_file> run_simulation -store write_patterns  
<new_pat_file>
```

- *Option 2 (Note that the simulation model needs to be complete):*

```
set_patterns external <pat_file> run_simulation -resolve |  
-override write_patterns <new_pat_file> -external
```

- *Option 3:*

```
set_patterns external <pat_file> run_simulation -failure_file  
<fail_file> set_patterns external <pat_file> -resolve  
<fail_file> write_patterns <new_pat_file> -external
```

- *Option 4:*

```
set_patterns external <pat_file> run_atpg -resolve write_patterns  
<new_pat_file> -external
```

Regenerating Patterns

For regenerating patterns, use the following set of commands:

```
set_patterns -external <old_patterns>  
add_faults -all  
run_atpg -auto  
write_patterns <new_pat_file> -internal
```

After generating or regenerating the patterns, the following report message will appear:

```
<number> scan chains have been completely masked
```

Examples

This section shows several flow examples.

Figure 255 *Generating Patterns*

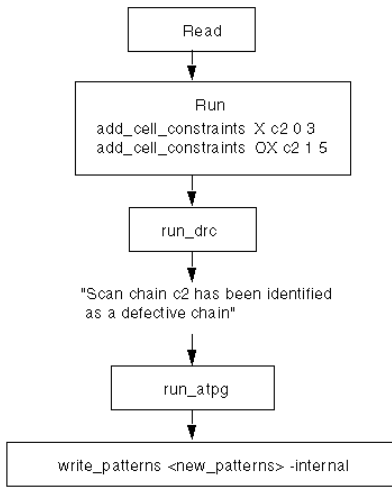


Figure 256 *Regenerating Patterns*

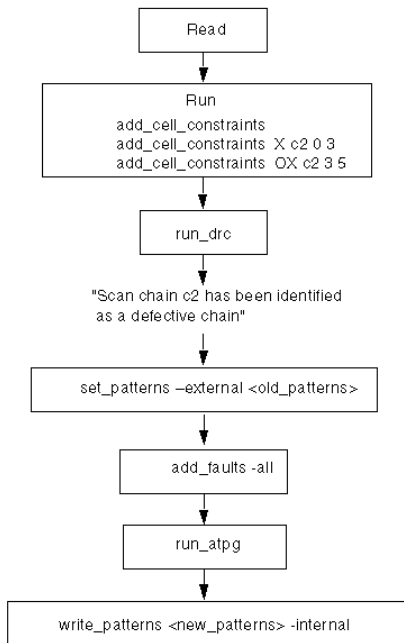
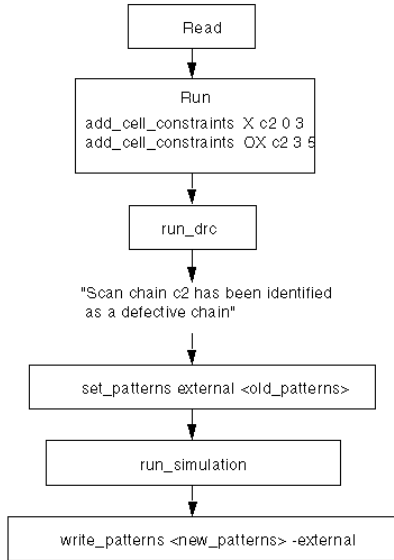


Figure 257 Re-Simulation and Updating Pattern Values

Note: In this case, scan cell 3 of chain c2 has a hold time violation and needs to be



Limitation

This flow does not include support for Full-Sequential patterns.

H

Simulation Debug Using MAX Testbench and Verdi

Verdi is an advanced automated open platform for debugging designs. It offers a full-featured waveform viewer and enables you to quickly process and debug simulation data.

When you use combine MAX Testbench, VCS, and Verdi for simulation debug, you can perform a variety of tasks, including displaying the current pattern number, the cycle count, and the active STIL statement, and adding input and output signals.

The following topics describe the process for setting up and running Verdi with MAX Testbench and VCS:

- [Setting the Environment](#)
- [Preparing MAX Testbench](#)
- [Linking Novas Object Files to the Simulation Executable](#)
- [Running VCS and Dumping an FSDB File](#)
- [Running Verdi](#)

Setting the Environment

To set up the install path for Verdi, specify the following settings:

```
setenv NOVAS_HOME path_to_verdi_installation
set path = ($NOVAS_HOME/bin $path)
```

To set up the license file for Verdi, use one of the following environment variables:

```
setenv NOVAS_LICENSE_FILE license_file:$NOVAS_LICENSE_FILE
setenv SPS_LICENSE_FILE license_file:$SPS_LICENSE_FILE
setenv SNPSLMD_LICENSE_FILE license_file
```

The license search priority is as follows:

1. SPS_LICENSE_FILE
2. NOVAS_LICENSE_FILE
3. LM_LICENSE_FILE

To set up the install path and license file for VCS, specify the following:

```
setenv VCS_HOME path_to_vcs_installation
set path=($VCS_HOME/bin $path)
setenv SNPSLMD_LICENSE_FILE license_file:
```

Preparing MAX Testbench

To prepare MAX Testbench to run with VCS and Verdi, you need to add a series of FSDB dump tasks to the testbench file. Some of the common FSDB dump tasks include:

- `$fsdbDumpfile` – Specifies the filename for the FSDB database.
- `$fsdbDumpvars` – Dumps signal value changes of specified instances and depth. To use this command, specify the FSDB file name. The default file name is `novas.fsdb`. You can specify several different FSDB file names in each `fsdbDumpvars` command
- `$fsdbDumpvarsByFile` - Uses a text file to select which scopes and signals to dump to the FSDB file. The contents of the file can be modified for each simulation without recompiling the simulation database.

The following example sets the `$fsdbDumpfile` and `$fsdbDumpvarsByFile` tasks in the MAX Testbench file (make sure you insert the `'ifdef WAVES` statement just before the `'ifdef tmax_vcde` statement):

```
`ifdef WAVES
    $fsdbDumpfile("../patterns/(YourPatternFileName).fsdb");
    $fsdbDumpvars(0);
`endif
```

For complete information on all FSDBdump tasks, refer to the following document:

`$NOVAS_HOME/doc/linking_dumping.pdf`

Linking Novas Object Files to the Simulation Executable

When you compile the VCS executable, you need to add a pointer to the Novas object files. You can do this using either of the following methods:

- Use the `-fsdb` option to automatically point to the `novas.tab` and `pli.a` files

```
% vcs [design_files] [other_desired_vcs_options] -fsdb
```

- Use the `-P` option to point to the `novas.tab` and `pli.a` files provided by Verdi, as shown in the following example:

```
% vcs -debug_pp \ -P $NOVAS_HOME/share/PLI/VCS/$PLATFORM/novas.tab
\ $NOVAS_HOME/share/PLI/VCS/$PLATFORM/pli.a \ +vcsd +vpi +memcbk
[design_files] [other_desired_vcs_options]
```

For interactive mode, you need to add the `-debug_all` option. If you need to include model-driven architecture signals (MDAs) or SystemVerilog assertions (SVAs), use the `-debug_pp` option or the `+vcsd+vpi+memcbk` option.

Running VCS and Dumping an FSDB File

The following example shows how to use VCS to compile a simulation executable with links to Novas object files, run the simulation, and dump an FSDB file:

```
LIB_FILES=" -v ../design/class.v" DEFINES="+define+WAVES"
DEBUG_OPTIONS="-debug_pp -P $NOVAS_HOME/share/PLI/VCS/LINUX64/novas.tab
$NOVAS_HOME/share/PLI/VCS/LINUX64/pli.a"
OPTIONS="-full64 +tetramax +delay_mode_zero +notimingcheck +nospecify
${DEBUG_OPTIONS}" NETLIST_FILES="../design/snps_micro_dftmax_net.v.sal"
TBENCH_FILE="../patterns/pats.v" SIMULATOR="vcs"
${SIMULATOR} -R ${DEFINES} ${OPTIONS} ${TBENCH_FILE} ${NETLIST_FILES}
${LIB_FILES} -l parallel_sim_verdiwv.log
```

Running Verdi

The following example shows how to set up and run Verdi:

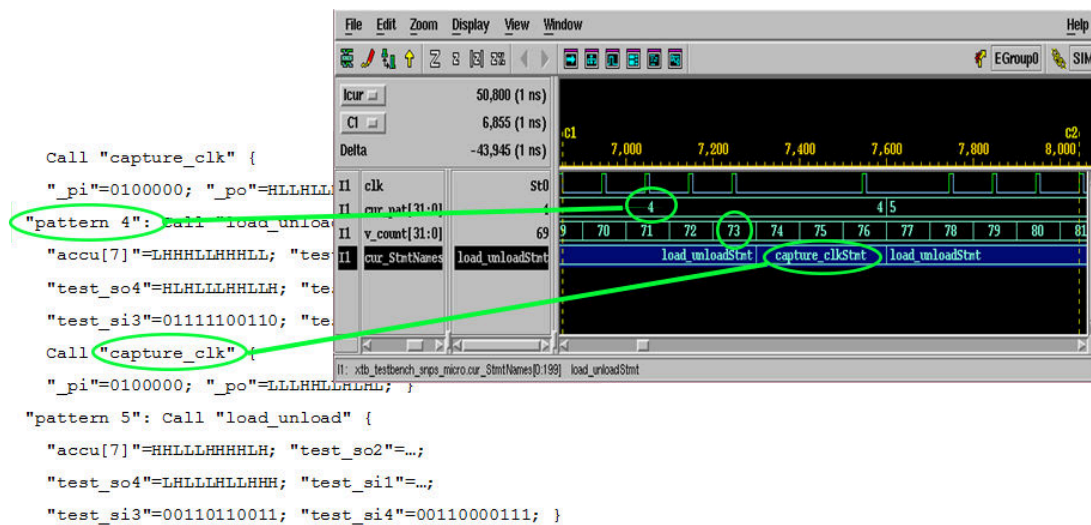
```
LIB_FILES=" -v ../design/class.v"
DEFINES=""
ANALYZE_OPTIONS=""
GUI_OPTIONS="-top snps_micro_test -ssf ../patterns/pats.fsdb"
NETLIST_FILES="../design/snps_micro_dftmax_net.v.sal"
TBENCH_FILE="../patterns/pats.v"
ANALYZER="verdi"
GUI="verdi"
${ANALYZER} ${DEFINES} ${ANALYZE_OPTIONS} ${TBENCH_FILE} ${NETLIST_FILES}
${LIB_FILES}
```

The following topics describe several scenarios for using Verdi for debugging:

- [Debugging MAX Testbench and VCS](#)
- [Changing Radix to ASCII](#)
- [Displaying the Current Pattern Number](#)
- [Displaying the Vector Count](#)
- [Using Search in the Signal List](#)

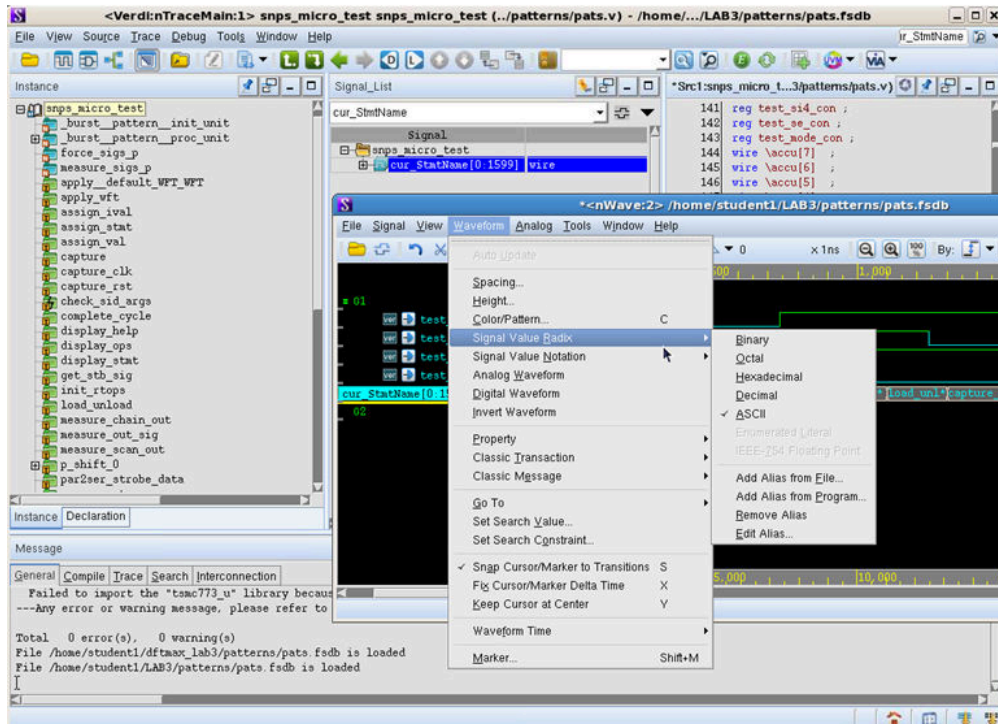
Debugging MAX Testbench and VCS

The following figure shows an example of how to use Verdi to debug MAX Testbench and VCS.



Changing Radix to ASCII

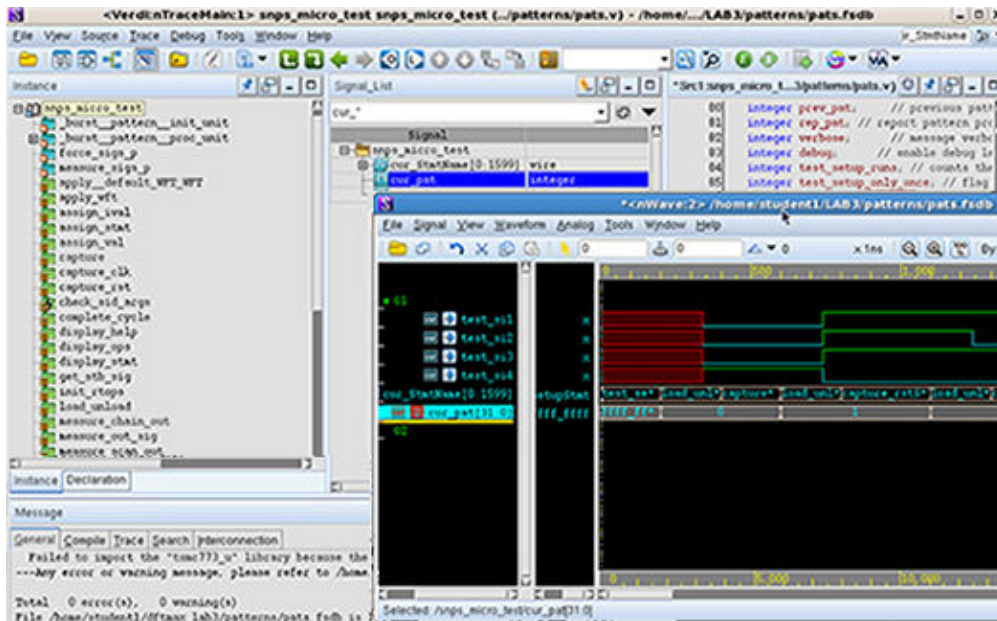
The following example shows how to change Radix-formatted signal values to an ASCII format:



Displaying the Current Pattern Number

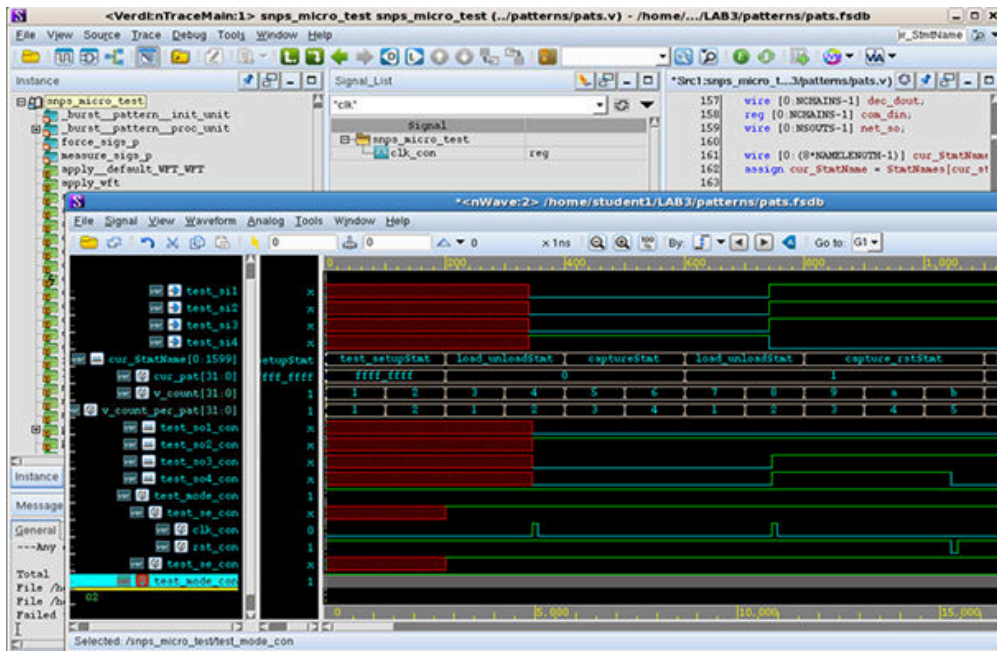
The following example shows how to display the current pattern number.

Appendix H: Simulation Debug Using MAX Testbench and Verdi Running Verdi



Displaying the Vector Count

The following example shows how to display the vector count.



Using Search in the Signal List

The following example shows how to add input and output signals by searching the signal list.

