



SIEMENS EDA

FlexTest™ User's and Reference Manual

Software Version 2021.2 and Later

Unpublished work. © 2021 Siemens

This material contains trade secrets or otherwise confidential information owned by Siemens Industry Software, Inc., its subsidiaries or its affiliates (collectively, "Siemens"), or its licensors. Access to and use of this information is strictly limited as set forth in Customer's applicable agreement with Siemens. This material may not be copied, distributed, or otherwise disclosed outside of Customer's facilities without the express written permission of Siemens, and may not be used in any way not expressly authorized by Siemens.

This document is for information and instruction purposes. Siemens reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Siemens to determine whether any changes have been made. Siemens disclaims all warranties with respect to this document including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement of intellectual property.

The terms and conditions governing the sale and licensing of Siemens products are set forth in written agreements between Siemens and its customers. Siemens' **End User License Agreement** may be viewed at: www.plm.automation.siemens.com/global/en/legal/online-terms/index.html.

No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Siemens whatsoever.

TRADEMARKS: The trademarks, logos, and service marks ("Marks") used herein are the property of Siemens or other parties. No one is permitted to use these Marks without the prior written consent of Siemens or the owner of the Marks, as applicable. The use herein of third party Marks is not an attempt to indicate Siemens as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A list of Siemens' trademarks may be viewed at: www.plm.automation.siemens.com/global/en/legal/trademarks.html. The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Support Center: support.sw.siemens.com

Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

Chapter 1

Using FlexTest	15
Non- to Full-Scan ATPG With FlexTest	17
Testing Methodologies	17
FlexTest Fault Modeling	19
Test Types and Associated Fault Models	19
Fault Locations	19
Fault Collapsing	20
Supported Fault Model Types	20
FlexTest Fault Detection	22
FlexTest Fault Classes	22
Fault Class Hierarchy	27
Fault Reporting	27
Testability Calculations	28
FlexTest Scan Terminology	29
Scan Cells	29
Master Element	30
Slave Element	30
Shadow Element	31
Copy Element	32
Extra Element	33
Scan Chains	33
Scan Groups	34
Scan Clocks	34
Test Procedure Files	35
Model Flattening	36
Understanding Design Object Naming	36
The Flattening Process	37
Simulation Primitives of the Flattened Model	38
Learning Analysis	42
Equivalence Relationships	42
Logic Behavior	42
Implied Relationships	43
Forbidden Relationships	44
Dominance Relationships	44
ATPG Design Rules Checking	45
General Rules Checking	45
Procedure Rules Checking	46
Bus Mutual Exclusivity Analysis	46
Scan Chain Tracing	47
Shadow Latch Identification	47
Data Rules Checking	48

Clock Rules Checking	48
RAM Rules Checking	48
Extra Rules Checking	49
Constrained/Forbidden/Block Value Calculations	49
Combinational Loop Handling	50
Sequential Loop Handling	52
Tri-State Devices	52
Non-Scan Cell Handling	53
FlexTest Basic Tool Flow	53
FlexTest Inputs and Outputs	55
Understanding FlexTest's ATPG Method	57
Cycle-Based Timing Circuits	57
Cycle-Based Timing Model	58
Cycle-Based Test Patterns	60
Performing Basic Operations	61
Invoking the Application	61
Invoking the FlexTest Fault Simulation Version	61
FlexTest Interrupt Capabilities	62
Setting the System Mode	62
Setting the Circuit Timing	64
Setting the Test Cycle Width	64
Defining the Cycle Behavior of Primary Inputs	64
Defining the Strobe Time of Primary Outputs	65
Fault Simulation on Simulation Derived Vectors	66
Executing Good Machine Simulation	66
Debugging the Good Machine Simulation	67
Resetting Circuit Status	67
Setting Up the Fault Information for ATPG	68
Changing to the ATPG System Mode	68
Setting the Fault Type	68
Creating the Faults List	69
Adding Faults to an Existing List	69
Loading Faults From an External List	70
Writing Faults to an External File	70
Setting Self-Initialized Test Sequences	70
Setting the Hypertrophic Limit	71
Setting DS Fault Handling	71
Setting the Possible-Detect Credit	71
Defining ATPG Constraints	72
Creating Patterns With Default Settings	73
Compressing Patterns	73
Reporting on ATPG Untestable Faults	73
Creating a Selective IDDQ Test Set	75
Setting the External Pattern Set	75
Determining When to Perform the Measures	75
Selecting the Best IDDQ Patterns	76
Selective IDDQ Example	76
Generating a Supplemental IDDQ Test Set	77
Specifying Leakage Current Checks	78

Table of Contents

Creating Instruction-Based Test Sets	80
Instruction-Based Fault Detection	80
Instruction File Format	81
Verifying Test Patterns	83
Simulating the Design With Timing	83
Test Pattern Formatting and Timing	85
Test Pattern Timing Overview	86
Timing Terminology	86
General Timing Issues	86
Generating a Procedure File	87
Defining and Modifying Timeplates	88
Saving Timing Patterns	91
Pattern Formatting Issues	92
Serial Versus Parallel Scan Chain Loading	92
Parallel Scan Chain Loading	92
FlexTest Text	94
Comparing FlexTest Text Formats With Other Test Data Formats	95
Verilog	97
Wave Generation Language (ASCII)	97
Standard Test Interface Language (STIL)	98
Saving in ASIC Vendor Data Formats	99
TI TDL 91	99
Fujitsu FTDL-E	99
Mitsubishi TDL	100
Toshiba TSTL2	100
Chapter 2	
FlexTest Command Dictionary	101
Inputs and Outputs	106
Command Line Syntax Conventions	106
Command Summary	107
Command Descriptions	117
Abort Interrupted Process	123
Add Atpg Constraints	124
Add Atpg Functions	128
Add Black Box	132
Add Cell Constraints	135
Add Clocks	138
Add Cone Blocks	140
Add Contention Free_bus	142
Add Faults	143
Add Initial States	146
Add Lists	147
Add Nofaults	149
Add Nonscan Handling	152
Add Output Masks	154
Add Pin Constraints	155
Add Pin Equivalences	158

Add Pin Strokes	159
Add Primary Inputs	160
Add Primary Outputs	162
Add Read Controls	163
Add Scan Chains	164
Add Scan Groups	165
Add Scan Instances	166
Add Scan Models	167
Add Tied Signals	168
Add Write Controls	170
Alias	171
Analyze Atpg Constraints	174
Analyze Bus	176
Analyze Contention	178
Analyze Control Signals	180
Analyze Fault	183
Analyze Race	186
Compress Patterns	188
Delete Atpg Constraints	190
Delete Atpg Functions	192
Delete Black Box	194
Delete Cell Constraints	195
Delete Clocks	196
Delete Cone Blocks	197
Delete Contention Free_bus	198
Delete Faults	199
Delete Initial States	201
Delete Lists	202
Delete Nofaults	203
Delete Nonscan Handling	206
Delete Output Masks	208
Delete Pin Constraints	209
Delete Pin Equivalences	210
Delete Pin Strokes	211
Delete Primary Inputs	212
Delete Primary Outputs	214
Delete Read Controls	216
Delete Scan Chains	217
Delete Scan Groups	218
Delete Scan Instances	219
Delete Scan Models	220
Delete Tied Signals	221
Delete Write Controls	223
Dofile	224
Exit	226
Find Design Names	227
Flatten Model	232
Help	233
History	234

Table of Contents

Load Faults	236
Read Modelfile	238
Read Procfile.	241
Report Aborted Faults.	246
Report Atpg Constraints	248
Report Atpg Functions	249
Report Au Faults.	250
Report Black Box	253
Report Bus Data	255
Report Cell Constraints.	257
Report Clocks	259
Report Cone Blocks	260
Report Contention Free_bus	261
Report Core Memory	262
Report Drc Rules	263
Report Environment	267
Report Faults.	269
Report Feedback Paths	272
Report Flatten Rules	274
Report Gates	276
Report Initial States	289
Report Lists.	290
Report Loops.	291
Report Nofaults.	293
Report Nonscan Cells	295
Report Nonscan Handling.	297
Report Output Masks	298
Report Pin Constraints	299
Report Pin Equivalences.	300
Report Pin Strokes	301
Report Primary Inputs.	302
Report Primary Outputs	304
Report Procedure	306
Report Pulse Generators	307
Report Read Controls	308
Report Scan Cells	309
Report Scan Chains.	311
Report Scan Groups	313
Report Scan Instances.	314
Report Scan Models	315
Report Statistics	316
Report Test Stimulus	320
Report Testability Data.	323
Report Tied Signals	325
Report Timeplate	326
Report Version Data	327
Report Write Controls.	328
Reset Au Faults.	329
Reset State.	330

Resume Interrupted Process	331
Run	332
Save History	335
Save Patterns	336
Select Iddq Patterns	342
Set Abort Limit	346
Set Atpg Limits	348
Set Atpg Window	350
Set Bus Handling	352
Set Capture Clock	355
Set Capture Limit	357
Set Checkpoint	359
Set Clock Restriction	360
Set Contention Check	361
Set Contention_bus Reporting	364
Set Display	366
Set Dofile Abort	367
Set Drc Handling	368
Set Driver Restriction	373
Set Fails Report	375
Set Fault Dropping	376
Set Fault Mode	379
Set Fault Sampling	381
Set Fault Type	382
Set File Compression	384
Set Flatten Handling	386
Set Gate Level	388
Set Gate Report	389
Set Gzip Options	394
Set Hypertrophic Limit	396
Set Iddq Checks	397
Set Iddq Strobe	401
Set Instruction Atpg	403
Set Internal Fault	404
Set Internal Name	405
Set Interrupt Handling	406
Set Learn Report	408
Set List File	410
Set Logfile Handling	411
Set Loop Handling	413
Set Net Dominance	414
Set Net Resolution	416
Set Nonscan Model	417
Set Output Comparison	419
Set Output Masks	421
Set Pattern Source	422
Set Possible Credit	426
Set Procedure Cycle_checking	427
Set Pulse Generators	428

Table of Contents

Set Race Data	429
Set Rail Strength	430
Set Random Atpg	431
Set Redundancy Identification	432
Set Screen Display	433
Set Self Initialization	434
Set Sensitization Checking	436
Set Sequential Learning	437
Set Shadow Check	438
Set Static Learning	439
Set Stg Extraction	441
Set System Mode	442
Set Test Cycle	444
Set Trace Report	445
Set Transient Detection	446
Set Unused Net	448
Set Z Handling	449
Setenv	451
Setup Checkpoint	452
Setup Pin Constraints	454
Setup Pin Strokes	456
Setup Tied Signals	457
Step	458
System	459
Unsetenv	460
Update Implication Detections	461
Write Core Memory	462
Write Environment	463
Write Faults	465
Write Initial States	468
Write Loops	469
Write Modelfile	470
Write Netlist	471
Write Primary Inputs	473
Write Primary Outputs	475
Write Procfile	477
Write Statistics	479
Shell Command Description	482
flextest	483

Appendix A

Test Pattern File Formats	487
ASCII Pattern Format	488
Setup_Data	488
Functional_Chain_Test	491
Test_Data	492
Scan_Cell	493
Example Circuit	495

Table Pattern Format	495
VCD Support Using VCD Plus	502
VCD Reader Control File Commands	503
Required Versions For Use of VCD Patterns	505
Example of Using VCD Reader	505
 Appendix B	
Using the FlexTest Tcl Interface	513
Using Tcl Within FlexTest	513
Modifying Existing Dofiles for Use with Tcl	516
Dollar Sign	516
Quotation Marks	516
Set Command	517
Optional Single Quotes	517
Environment Variables	517
Special Tcl Characters	518
Tcl Comments Can Be Tricky	519
The Dofile Command and Tcl Source Command Are Different	520
Tcl Resources	520
 Index	
 Third-Party Information	

List of Figures

Figure 1-1. Internal Faulting Example.	20
Figure 1-2. Generic Scan Cell	29
Figure 1-3. Generic Mux-DFF Scan Cell Implementation	30
Figure 1-4. LSSD Master/Slave Element Example	31
Figure 1-5. Dependently-clocked Mux-DFF/Shadow Element Example	31
Figure 1-6. Independently-clocked Mux-DFF/Shadow Element Example	32
Figure 1-7. Mux-DFF/Copy Element Example	32
Figure 1-8. Generic Scan Chain.	33
Figure 1-9. Generic Scan Group	34
Figure 1-10. Scan Clocks Example	35
Figure 1-11. Design Before Flattening	37
Figure 1-12. Design After Flattening.	37
Figure 1-13. 2x1 MUX Example	39
Figure 1-14. LA, DFF Example.	39
Figure 1-15. TSD, TSH Example	40
Figure 1-16. PBUS, SWBUS Example	40
Figure 1-17. Equivalence Relationship Example	42
Figure 1-18. Example of Learned Logic Behavior	43
Figure 1-19. Example of Implied Relationship Learning	43
Figure 1-20. Forbidden Relationship Example	44
Figure 1-21. Dominance Relationship Example	44
Figure 1-22. Bus Contention Example	46
Figure 1-23. Bus Contention Analysis.	47
Figure 1-24. Constrained Values in Circuitry	49
Figure 1-25. Forbidden Values in Circuitry.	49
Figure 1-26. Blocked Values in Circuitry	50
Figure 1-27. Delay Element Added to Feedback Loop	51
Figure 1-28. Fake Sequential Loop	52
Figure 1-29. Overview of FlexTest Usage.	54
Figure 1-30. FlexTest Inputs and Outputs	56
Figure 1-31. Cycle-Based Circuit with Single Phase Clock	57
Figure 1-32. Cycle-Based Circuit with Two Phase Clock.	58
Figure 1-33. Example Test Cycle	59
Figure 1-34. Example Instruction File.	82
Figure 1-35. Defining Basic Timing Process Flow	85
Figure 2-1. How the -Instance Switch Determines Fault Counts	317
Figure 2-2. Set Atpg Window Example.	350
Figure A-1. Example Scan Circuit.	495

List of Tables

Table 1-1. Test Type/Fault Model Relationship	19
Table 1-2. Pin Value Requirements for ADD Instruction	80
Table 2-1. Conventions for Command Line Syntax	106
Table 2-2. Command Summary	107
Table 2-3. Example Cell Value Changes with Different Constraints for Scan Patterns	135
Table 2-4. Example Scan Cell Value Changes with Different Constraints for Chain Patterns 136	
Table 2-5. Named Capture Procedure Merging Under Different Conditions	241
Table 2-6. Available Information Displayed and Arguments	263
Table 2-7. Fault Class Codes and Names	270
Table 2-8. Reportable Gate Types	285
Table 2-9. FlexTest Learned Gate Types	286
Table 2-10. Untestable Faults that are Reclassified by Reset Au Faults	329
Table 2-11. WIRE Bus Contention Truth Table	414
Table 2-12. AND Bus Contention Truth Table	414
Table 2-13. OR Bus Contention Truth Table	414
Table 2-14. DRC Non-scan Cell Classifications	417
Table 2-15. Shell Command Summary	482
Table B-1. Common Tcl Characters	518

Chapter 1

Using FlexTest

FlexTest is the Siemens EDA non-scan to full-scan ATPG solution. This chapter provides an overview and describes the basic operation of FlexTest.

Non- to Full-Scan ATPG With FlexTest	17
Testing Methodologies	17
FlexTest Fault Modeling	19
Test Types and Associated Fault Models	19
Fault Locations	19
Fault Collapsing	20
Supported Fault Model Types	20
FlexTest Fault Detection	22
FlexTest Fault Classes	22
Fault Class Hierarchy	27
Fault Reporting	27
Testability Calculations	28
FlexTest Scan Terminology	29
Scan Cells	29
Master Element	30
Slave Element	30
Shadow Element	31
Copy Element	32
Extra Element	33
Scan Chains	33
Scan Groups	34
Scan Clocks	34
Test Procedure Files	35
Model Flattening	36
Understanding Design Object Naming	36
The Flattening Process	37
Simulation Primitives of the Flattened Model	38
Learning Analysis	42
Equivalence Relationships	42
Logic Behavior	42
Implied Relationships	43
Forbidden Relationships	44
Dominance Relationships	44
ATPG Design Rules Checking	45

General Rules Checking	45
Procedure Rules Checking	46
Bus Mutual Exclusivity Analysis	46
Scan Chain Tracing	47
Shadow Latch Identification	47
Data Rules Checking	48
Clock Rules Checking	48
RAM Rules Checking	48
Extra Rules Checking	49
Constrained/Forbidden/Block Value Calculations	49
Combinational Loop Handling	50
Sequential Loop Handling	52
Tri-State Devices	52
Non-Scan Cell Handling	53
FlexTest Basic Tool Flow	53
FlexTest Inputs and Outputs	55
Understanding FlexTest's ATPG Method	57
Cycle-Based Timing Circuits	57
Cycle-Based Timing Model	58
Cycle-Based Test Patterns	60
Performing Basic Operations	61
Invoking the Application	61
Invoking the FlexTest Fault Simulation Version	61
FlexTest Interrupt Capabilities	62
Setting the System Mode	62
Setting the Circuit Timing	64
Setting Up the Fault Information for ATPG	68
Setting Self-Initialized Test Sequences	70
Setting the Hypertrophic Limit	71
Setting DS Fault Handling	71
Setting the Possible-Detect Credit	71
Defining ATPG Constraints	72
Creating Patterns With Default Settings	73
Compressing Patterns	73
Reporting on ATPG Untestable Faults	73
Creating a Selective IDDQ Test Set	75
Generating a Supplemental IDDQ Test Set	77
Specifying Leakage Current Checks	78
Creating Instruction-Based Test Sets	80
Instruction-Based Fault Detection	80
Instruction File Format	81
Verifying Test Patterns	83

Simulating the Design With Timing	83
Test Pattern Formatting and Timing	85
Test Pattern Timing Overview	86
Timing Terminology	86
General Timing Issues	86
Generating a Procedure File	87
Defining and Modifying Timeplates	88
Saving Timing Patterns	91
Pattern Formatting Issues	92
Saving in ASIC Vendor Data Formats	99

Non- to Full-Scan ATPG With FlexTest

FlexTest has many features.

- **Flexibility of design styles.** You can use FlexTest on designs with a wide-range of scan circuitry—from no internal scan to full scan.
- **Tight integration in the legacy top-down design flow.** FlexTest is tightly coupled with DFTAdvisor in the legacy top-down design flow.
- **Additions to scan ATPG.** FlexTest provides easy and flexible scan setup using a test procedure file. FlexTest also provides DFT rules checking (before you generate test patterns) to ensure proper scan operation.
- **Support for use in external tool environments.** You can also use FlexTest as a point tool in many third-party design flows, including Verilog and Synopsys.
- **Versatile test hardware structure support.** FlexTest supports a wide range of test hardware structures.
- **Flexible packaging.** The standard FlexTest package, flextest, operates in both graphical and non-graphical modes. FlexTest also has a fault simulation-only package, which you install normally but which licenses only the setup, good, and fault simulation capabilities of the tool; that is, you cannot run ATPG and scan identification.

Testing Methodologies

FlexTest support both supplemental and selective IDDQ test methodologies.

FlexTest uses sequential ATPG algorithms and is thus effective over a wider range of design styles. However, FlexTest works most effectively on primarily sequential designs; that is, those containing a lower percentage of scan circuitry. FlexTest currently support only the ideal IDDQ test methodology for fully static, resistive, and some dynamic CMOS circuits. The tools can also perform IDDQ checks during ATPG to ensure the vectors they produce meet the ideal

requirements. For information on creating IDDQ test sets, refer to “[Creating a Selective IDDQ Test Set](#)”.

FlexTest Fault Modeling

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

Test Types and Associated Fault Models.....	19
Fault Locations.....	19
Fault Collapsing.....	20
Supported Fault Model Types	20

Test Types and Associated Fault Models

The table associates test types, fault models, and the types of manufacturing defects targeted for detection.

Table 1-1. Test Type/Fault Model Relationship

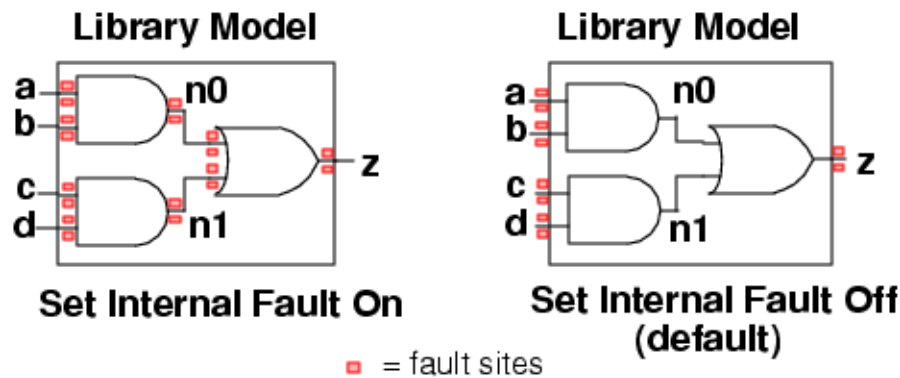
Test Type	Fault Model	Examples of Mfg. Defects Detected
Functional	Stuck-at, toggle	Some opens/shorts in circuit interconnections
IDDQ	Pseudo stuck-at	CMOS transistor stuck-on/some stuck-open conditions, resistive bridging faults, partially conducting transistors
At-speed	Transition	Partially conducting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

Fault Locations

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

Figure 1-1 shows the fault sites for both cases.

Figure 1-1. Internal Faulting Example



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

Fault Collapsing

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

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

Supported Fault Model Types

FlexTest supports stuck-at, pseudo stuck-at, toggle, and transition fault models.

Functional Testing and the Toggle Fault Model

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

FlexTest uses the following fault collapsing rules for the toggle fault model:

- Buffer - a fault on the input is equivalent to the same fault value at the output.
- Inverter - a fault on the input is equivalent to the opposite fault value at the output.

- Net between single output pin and multiple input pin - all faults of the same value are equivalent.

IDDQ Testing and the Pseudo Stuck-At Fault Model

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

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

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

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

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

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

At-Speed Testing and the Transition Fault Model

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

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

FlexTest Fault Detection

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

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

FlexTest Fault Classes	22
Fault Class Hierarchy	27
Fault Reporting	27

FlexTest Fault Classes

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

Note



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

Untestable (UT)

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

The following list discusses each of the untestable fault classes.

- **Unused (UU)**


The unused fault class includes all faults on circuitry unconnected to any circuit observation point and faults on floating primary outputs.

- **Tied (TI)**

The tied fault class includes faults on gates where the point of the fault is tied to a value identical to the fault stuck value. The tied circuitry could be due to:

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

Note

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


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

- **Blocked (BL)**

The blocked fault class includes faults on circuitry for which tied logic blocks all paths to an observable point. The tied circuitry could be due to:


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

Note

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

This class also includes faults on selector lines of multiplexers that have identical data lines.

Note

 Tied faults and blocked faults can be equivalent faults.

- **Redundant (RE)**

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

Testable (TE)

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

- **Detected (DT)**

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

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

The `det_implication` subclass normally includes faults in the scan path circuitry, as well as faults that propagate ungated to the shift clock input of scan cells. The scan chain functional test, which detects a binary difference at an observation point, guarantees detection of these faults.

FlexTest provides the Update Implication Detections command, which lets you specify additional types of faults for this category. Refer to the [Update Implication Detections](#) command description.

- **Posdet (PD)**

The posdet, or possible-detected, fault class includes all faults that fault simulation identifies as possible-detected but not hard detected. A possible-detected fault results from a 0-X or 1-X difference at an observation point. The posdet class contains two subclasses:

- `posdet_testable` (PT) - potentially detectable posdet faults. PT faults result when the tool cannot prove the 0-X or 1-X difference is the only possible outcome. A higher abort limit may reduce the number of these faults.
- `posdet_untestable` (PU) - proven ATPG_untestable and hard undetectable posdet faults.

By default, the calculations give 50% credit for posdet faults. You can adjust the credit percentage with the [Set Possible Credit](#) command.

Note



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


- **Oscillatory (OS)**

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

The oscillatory fault class contains two subclasses:

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


Note

 These faults may stabilize after a long simulation time.

- **Hypertrophic (HY)**

The hypertrophic fault class includes all faults whose effects spread extensively throughout the design, causing divergence from good state machine status for a large percentage of the design. These differences force the tool to do a large number of calculations, slowing down the simulation. Hypertrophic faults require a large amount of memory and CPU time to calculate their circuit status. To maintain fault simulation performance, the tool drops hypertrophic faults from the simulation. The tool calculates fault coverage, test coverage, and ATPG effectiveness by treating hypertrophic faults as posdet faults.

Note

 Because these faults affect the circuit extensively, even though the tool may drop them from the fault list (with accompanying lower fault coverage numbers), hypertrophic faults are most likely detected.

The hypertrophic fault class contains two subclasses:

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

FlexTest defines hypertrophic faults with the internal state difference between each faulty machine and good machine. You can use the [Set Hypertrophic Limit](#) command to specify the percentage of internal state difference required to classify a fault as hypertrophic. The default difference is 30%; if the number of hypertrophic faults exceeds 30%, the tool drops them from the list. If you reduce the limit, the tool will drop them more quickly, speeding up the simulation. Raising the limit will slow down the simulation.

- **Uninitialized (UI)**

The uninitialized fault class includes faults for which the test generator is unable to:

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

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

- **ATPG_untestable (AU)**

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

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

Note



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

- **Undetected (UD)**

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

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

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

Note

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

Fault Class Hierarchy

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

Example 1-1. Fault Class Hierarchy

```
1. Full (FU)
  1.1 TEstable (TE)
    a. DETected (DT)
      i. DET_Simulation (DS)
      ii. DET_Implication (DI)
    b. POSDET (PD)
      i. POSDET_Untestable (PU)
      ii. POSDET_Testable (PT)
    c. OSCillatory (OS)
      i. OSC_Untestable (OU)
      ii. OSC_Testable (OT)
    d. HYPertrophic (HY)
      i. HYP_Untestable (HU)
      ii. HYP_Testable (HT)
    e. Uninitializable (UI)
    f. Atpg_untestable (AU)
    g. UNDetected (UD)
      i. UNControlled (UC)
      ii. UNObserved (UO)
  1.2 UNTestable (UT)
    a. UNUsed (UU)
    b. Tied (TI)
    c. Blocked (BL)
    d. Redundant (RE)
```

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

Fault Reporting

When reporting faults with the Report Faults command, FlexTest identifies each fault by three ordered fields.

- fault value (0 for stuck-at-0 or “slow-to-rise” transition faults; 1 for stuck-at-1 or “slow-to-fall” transition faults)
- two-character fault class code

- pin pathname of the fault site

If the tools report uncollapsed faults, they display faults of a collapsed fault group together, with the representative fault first followed by the other members (with EQ fault codes).

Testability Calculations

Given the fault classes explained in the previous sections, FlexTest makes the following calculations.

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

FlexTest calculates it using the formula:

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

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

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

FlexTest calculates it using the formula:

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

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

FlexTest calculates it using the formula:

$$\frac{\#DT + \#UT + \#AU + \#UI + \#PU + \#OU + \#HU + ((\#PT + \#OT + \#HT) * \text{posdet_credit})}{\#full} \times 100$$

FlexTest Scan Terminology

FlexTest not only works toward a common goal (to improve test coverage), it shares common terminology, internal processes, and other tool concepts, such as how to view the design and the scan circuitry.

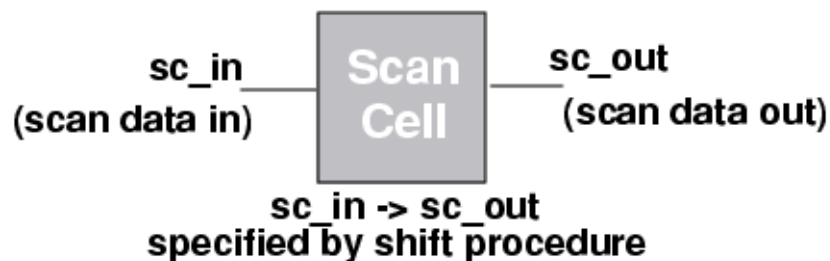
Scan Cells	29
Master Element	30
Slave Element	30
Shadow Element	31
Copy Element	32
Extra Element	33
Scan Chains	33
Scan Groups	34
Scan Clocks	34

Scan Cells

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

Because scan cell operation depends on an external procedure, scan cells are tightly linked to the notion of test procedure files. “[Test Procedure Files](#)” on page 35 discusses test procedure files in detail. [Figure 1-2](#) illustrates the black box concept of a scan cell and its reliance on a test procedure.

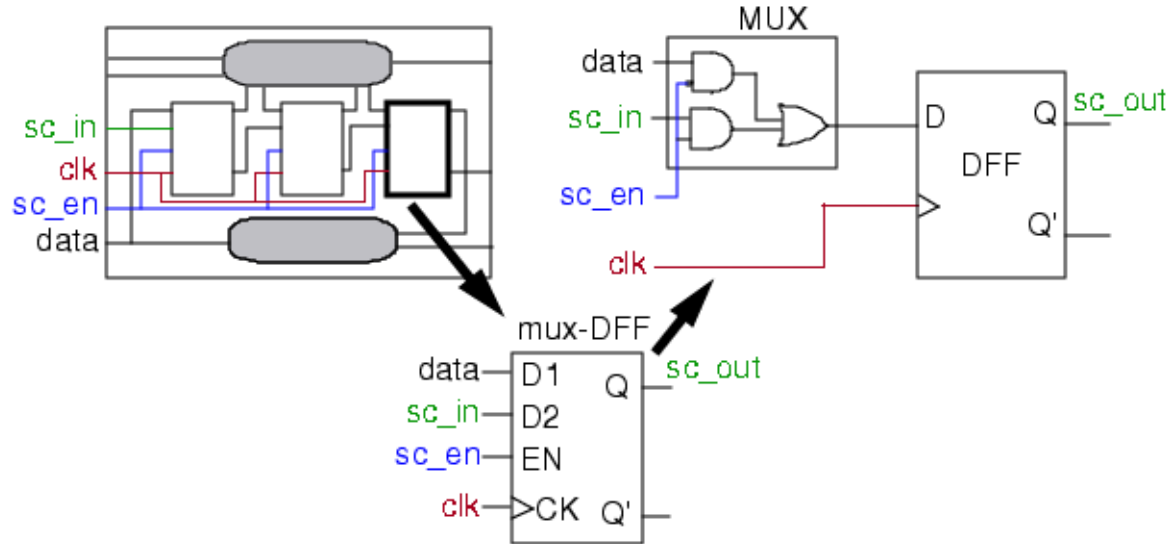
Figure 1-2. Generic Scan Cell



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

[Figure 1-3](#) gives one example of a scan cell implementation (for the mux-DFF scan type).

Figure 1-3. Generic Mux-DFF Scan Cell Implementation



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

Master Element

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

For example, Figure 1-3 shows a mux-DFF scan cell, which contains only a master element. However, scan cells can contain memory elements in addition to the master. Figures 1-4 through 1-7 illustrate examples of master elements in a variety of other scan cells.

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

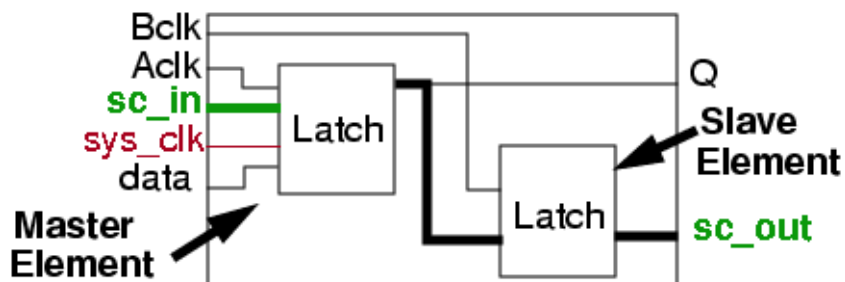
Slave Element

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

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

Figure 1-4 shows a slave element within a scan cell.

Figure 1-4. LSSD Master/Slave Element Example



In the example of Figure 1-4, Aclk controls scan data input. Activating Aclk, with sys_clk (which controls system data) held off, shifts scan data into the scan cell. Activating Bclk propagates scan data to the output.

Shadow Element

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

Figure 1-5 gives an example of a scan cell with a dependently-clocked, non-observable shadow element with a non-inverted value.

Figure 1-5. Dependently-clocked Mux-DFF/Shadow Element Example

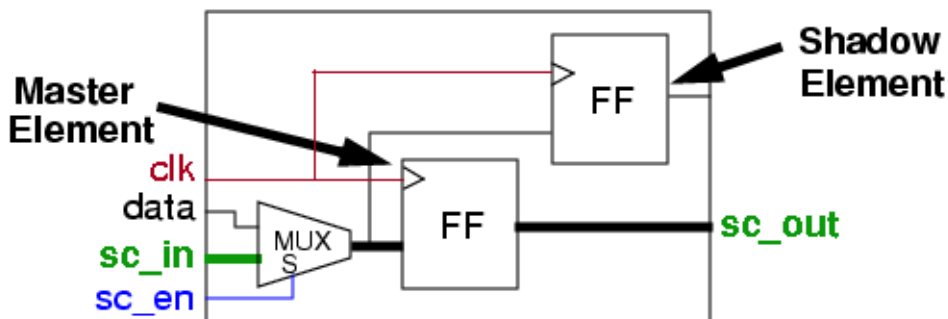
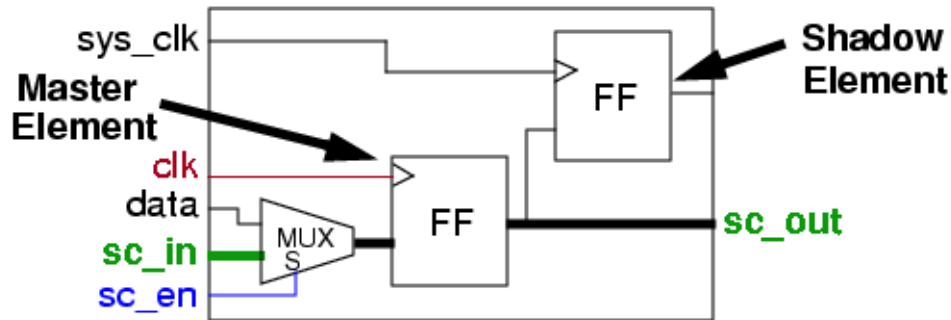


Figure 1-6 shows a similar example where the shadow element is independently-clocked.

Figure 1-6. Independently-clocked Mux-DFF/Shadow Element Example



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

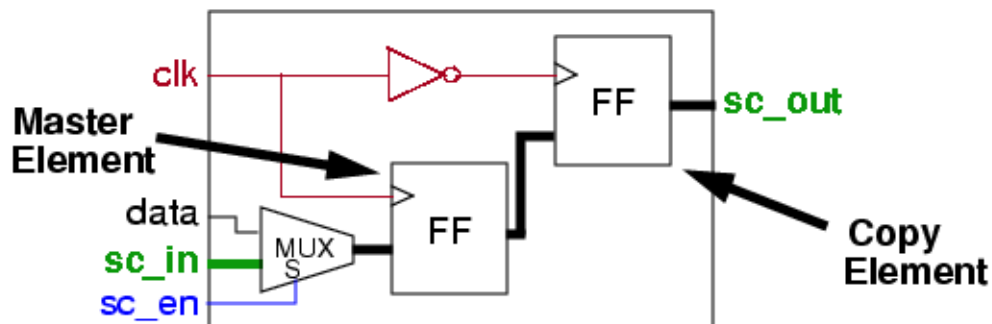
The definition of a shadow element is based on the shadow having the same (or inverse) value as the master element it shadows. A variety of interconnections of the master and shadow will accomplish this. In [Figure 1-5](#), the shadow's data input is connected to the master's data input, and both FFs are triggered by the same clock edge. The definition would also be met if the shadow's data input was connected to the master's output and the shadow was triggered on the trailing edge, the master on the leading edge, of the same clock.

Copy Element

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

[Figure 1-7](#) gives an example of a copy element within a scan cell in which a master element provides data to the copy.

Figure 1-7. Mux-DFF/Copy Element Example



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

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

Extra Element

The extra element is an additional, independently-clocked memory element of a scan cell. An extra element is any element that lies in the scan chain path between the master and slave elements.

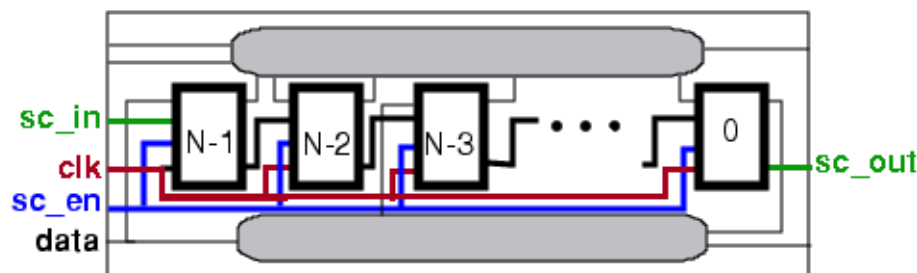
The **shift** procedure controls data capture into the extra elements. These elements are not observable. Scan cells can contain multiple extras. Extras can contain inverted data with respect to the master element.

Scan Chains

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

Figure 1-8 shows a scan chain, with scan input "sc_in" and scan output "sc_out".

Figure 1-8. Generic Scan Chain

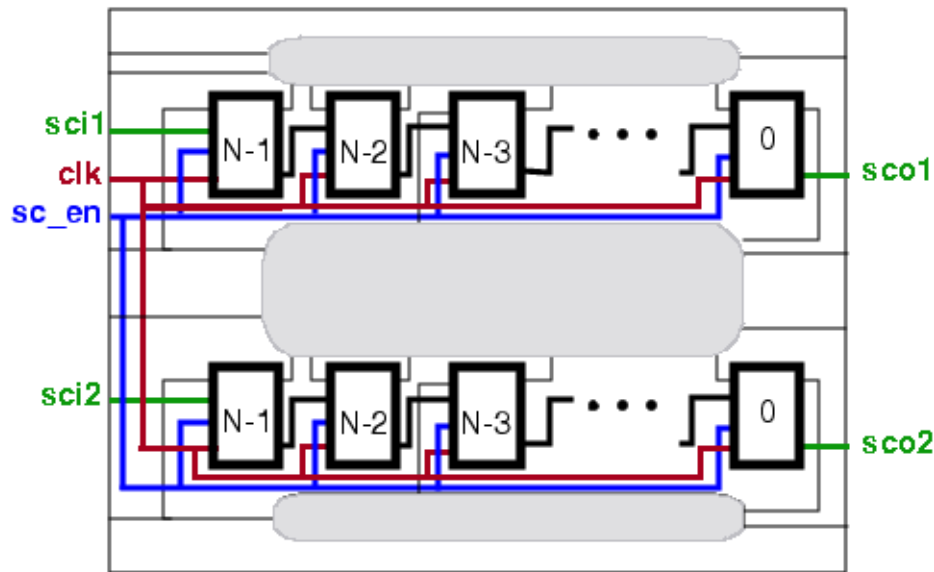


The scan chain length (N) is the number of scan cells within the scan chain. By convention, the scan cell closest to the external output pin is number 0, its predecessor is number 1, and so on. Because the numbering starts at 0, the number for the scan cell connected to the external input pin is equal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

Scan Groups

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

Figure 1-9. Generic Scan Group



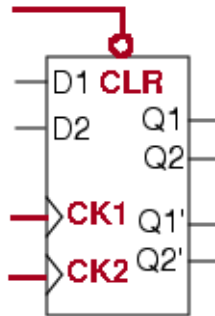
You may have two clocks, A and B, each of which clocks different scan chains. You often can clock, and therefore operate, the A and B chains concurrently, as shown in [Figure 1-9](#). However, if two chains share a single scan input pin, these chains cannot be operated in parallel. Regardless of operation, all defined scan chains in a circuit must be associated with a scan group. A scan group is a concept used by Siemens EDA DFT and ATPG tools.

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

Scan Clocks

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

[Figure 1-10](#) shows a scan cell whose scan clock signals are shown in bold.

Figure 1-10. Scan Clocks Example

In addition to capturing data into scan cells, scan clocks, in their off state, ensure that the cells hold their data. Design rule checks ensure that clocks perform both functions. A clock's off-state is the primary input value that results in a scan element's clock input being at its inactive state (for latches) or state prior to a capturing transition (for edge-triggered devices). In the case of [Figure 1-10](#), the off-state for the CLR signal is 1, and the off-states for CK1 and CK2 are both 0.

Test Procedure Files

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

In order to utilize the scan circuitry in your design, you must:

- Define the scan circuitry for the tool.
- Create a test procedure file to describe the scan circuitry operation.
- Perform DRC process. This occurs when you exit from Setup mode.

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

If your design contains scan circuitry, FlexTest require a test procedure file. You must create one before running ATPG with FlexTest.

Model Flattening

To work properly, FlexTest must use its own internal representations of the design. The tool created these internal design models by flattening the model and replacing the design cells in the netlist (described in the library) with its own primitives.

The tool flattens the model when you initially attempt to exit the Setup mode (see the [Set System Mode](#) command), just prior to design rules checking. FlexTest also provides the [Flatten Model](#) command, which allows flattening of the design model while still in Setup mode.

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

- [Flatten Model](#) — Creates a primitive gate simulation representation of the design.
- [Report Flatten Rules](#) — Displays either a summary of all the flattening rule violations or the data for a specific violation.
- [Set Flatten Handling](#) — Specifies how the tool handles flattening violations.

This section contains the following topics:

Understanding Design Object Naming	36
The Flattening Process	37
Simulation Primitives of the Flattened Model	38

Understanding Design Object Naming

FlexTest uses special terminology to describe different objects in the design hierarchy.

The following list describes the most common:

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

The Flattening Process

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

Figure 1-11 shows an example of circuitry containing an AND-OR-Invert cell and an AND gate, before flattening.

Figure 1-11. Design Before Flattening

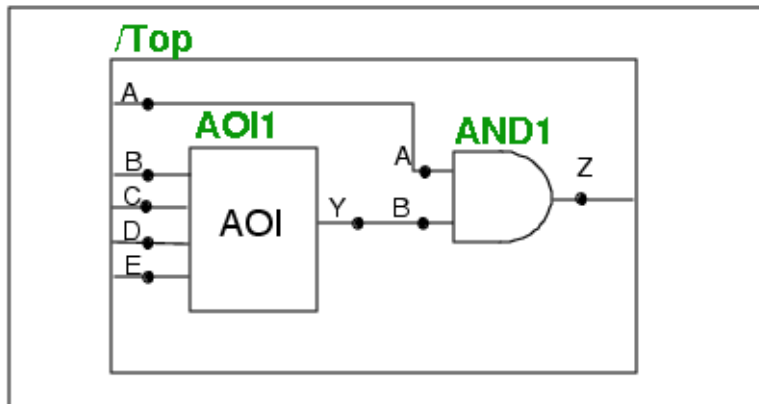
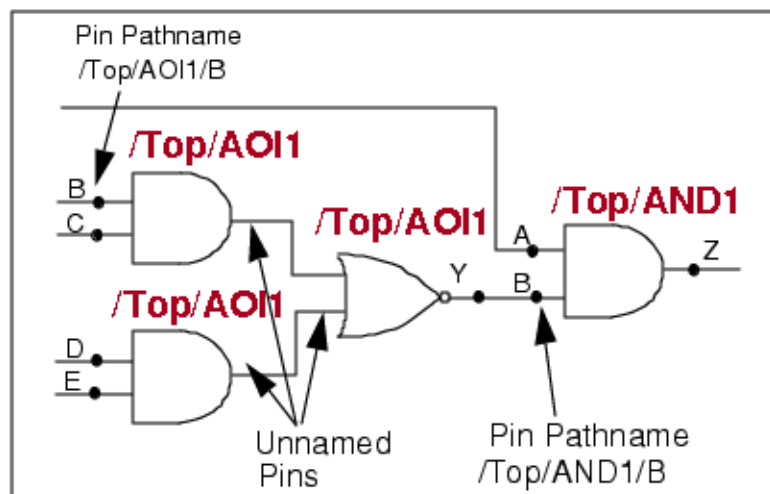


Figure 1-12 shows this same design once it has been flattened.

Figure 1-12. Design After Flattening



After flattening, only naming preserves the design hierarchy; that is, the flattened netlist maintains the hierarchy through instance naming. Figures 1-11 and 1-12 show this hierarchy preservation. /Top is the name of the hierarchy's top level. The simulation primitives (two AND gates and a NOR gate) represent the flattened instance AOI1 within /Top. Each of these flattened gates retains the original design hierarchy in its naming—in this case, /Top/AOI1.

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

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

Simulation Primitives of the Flattened Model

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

The following list describes these simulation primitives:

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

Note



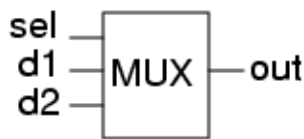
There can be one or more loops in a feedback path. In Atpg mode, you can display the loops with the [Report Loops](#) command. In Setup mode, use [Report Feedback Paths](#).

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

- **ZVAL** - a single-input gate that acts as a buffer unless Z is the input value. When a Z is the input value, the output is an X. You can modify this behavior with the Set Z Handling command.


- **INV** - a single-input gate whose output value is the opposite of the input value. The INV gate cannot accept a Z input value.
- **AND, NAND** - multiple-input gates (two to four) that act as standard AND and NAND gates.
- **OR, NOR** - multiple-input (two to four) gates that act as standard OR and NOR gates.
- **XOR, XNOR** - 2-input gates that act as XOR and XNOR gates, except that when either input is an X, the output is an X.
- **MUX** - a 2x1 mux gate whose pins are order dependent, as shown in [Figure 1-13](#).

Figure 1-13. 2x1 MUX Example



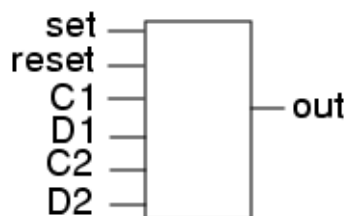
The sel input is the first defined pin, followed by the first data input and then the second data input. When sel=0, the output is d1. When sel=1, the output is d2.

Note

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

- **LA, DFF** - state elements, whose order dependent inputs include set, reset, and clock/data pairs, as shown in [Figure 1-14](#).

Figure 1-14. LA, DFF Example

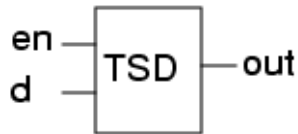


Set and reset lines are always level sensitive, active high signals. DFF clock ports are edge-triggered while LA clock ports are level sensitive. When set=1, out=1. When reset=1, out=0. When a clock is active (for example C1=1), the output reflects its associated data line value (D1). If multiple clocks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

- **TLA, STLA, STFF** - special types of learned gates that act as, and pass the design rule checks for, transparent latch, sequential transparent latch, or sequential transparent flip-flop. These gates propagate values without holding state.

- **TIE0, TIE1, TIEX, TIEZ** - zero-input, single-output gates that represent the effect of a signal tied to ground or power, or a pin or state element constrained to a specific value (0,1,X, or Z). The rules checker may also determine that state elements exhibit tied behavior and replace them with the appropriate tie gates.
- **TSD, TSH** - a 2-input gate that acts as a tri-stateTM driver, as shown in [Figure 1-15](#).

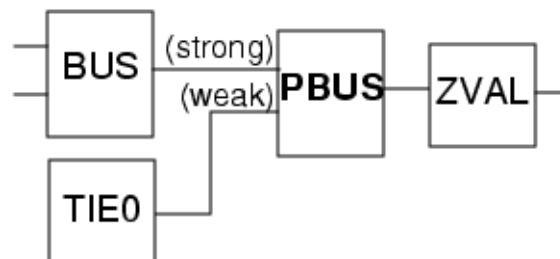
Figure 1-15. TSD, TSH Example



When en=1, out=d. When en=0, out=Z. The data line, d, cannot be a Z.

- **SW, NMOS**- a 2-input gate that acts like a tri-state driver but can also propagate a Z from input to output. FlexTest uses the NMOS gate for the same purpose.
- **BUS** - a multiple-input (up to four) gate whose drivers must include at least one TSD or SW gate. If you bus more than four tri-state drivers together, the tool creates cascaded BUS gates. The last bus gate in the cascade is considered the dominantbus gate.
- **WIRE** - a multiple-input gate that differs from a bus in that none of its drivers are tri-statable.
- **PBUS, SWBUS** - a 2-input pull bus gate, for use when you combine strong bus and weak bus signals together, as shown in [Figure 1-16](#).

Figure 1-16. PBUS, SWBUS Example



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

- **ZHOLD**- a single-input buskeeper gate (see [page 49](#) for more information on buskeepers) associated with a tri-state network that exhibits sequential behavior. If the input is a binary value, the gate acts as a buffer. If the input value is a Z, the output

depends on the gate's hold capability. There are three ZHOLD gate types, each with a different hold capability:

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

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

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

Learning Analysis

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

For more information on turning learning analysis off, refer to the [Set Static Learning](#) command or the [Set Sequential Learning](#) command.

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

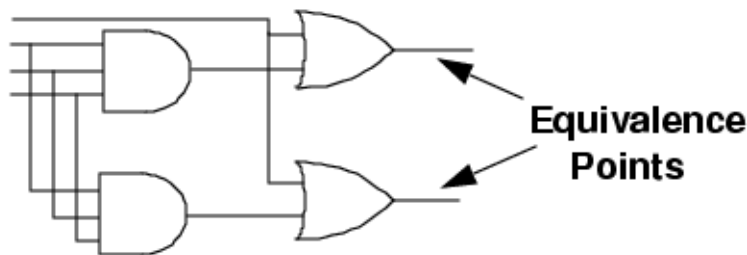
Equivalence Relationships	42
Logic Behavior	42
Implied Relationships	43
Forbidden Relationships	44
Dominance Relationships	44

Equivalence Relationships

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

[Figure 1-17](#) shows an example of two of these equivalence points within some circuitry.

Figure 1-17. Equivalence Relationship Example

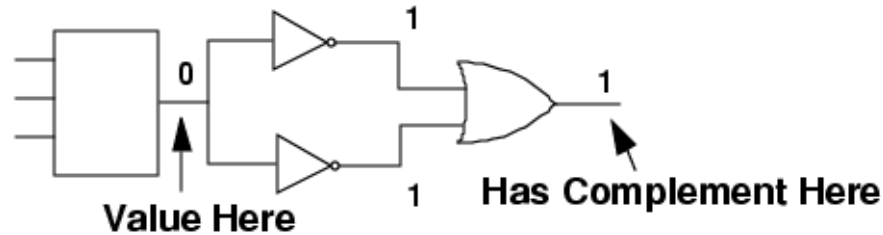


Logic Behavior

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

For example, [Figure 1-18](#) shows some circuitry that, according to the analysis, acts as an inverter.

Figure 1-18. Example of Learned Logic Behavior



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

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

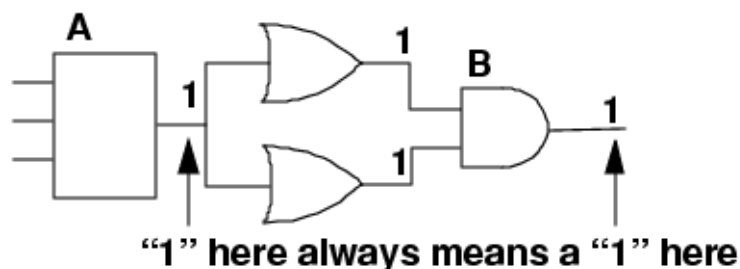
If the analysis process yields no information for a particular category, it does not issue the corresponding message.

Implied Relationships

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

[Figure 1-19](#) shows the implied learning the analysis derives from a piece of circuitry.

Figure 1-19. Example of Implied Relationship Learning



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

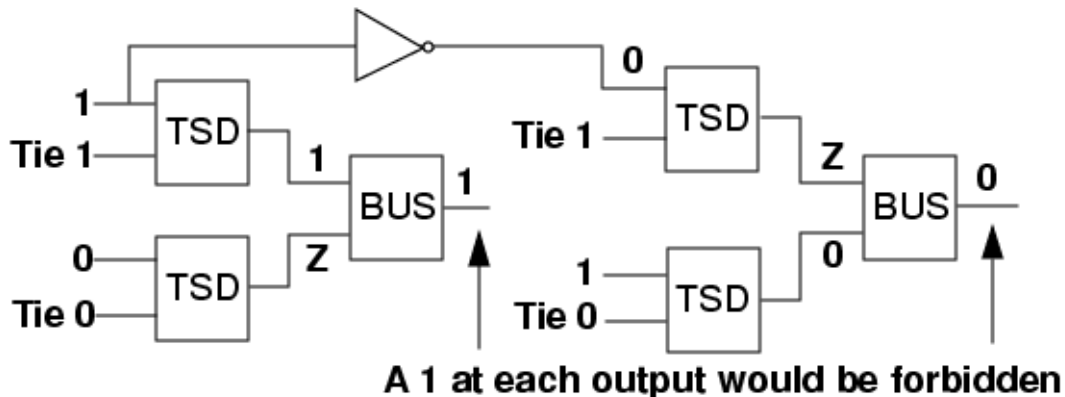
obstacles for ATPG. Thus, implied relationship learning significantly reduces the number of bad ATPG decisions.

Forbidden Relationships

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

Figure 1-20 shows an example of such behavior.

Figure 1-20. Forbidden Relationship Example

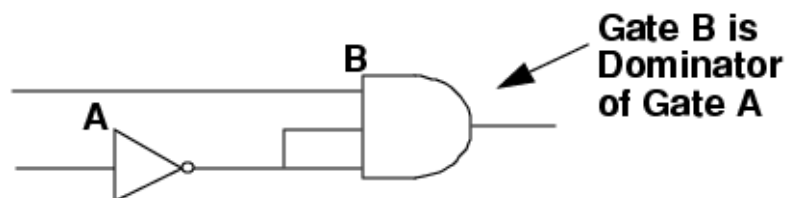


Dominance Relationships

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

Figure 1-21 shows an example of this relationship.

Figure 1-21. Dominance Relationship Example



ATPG Design Rules Checking

FlexTest performs design rules checking (DRC) after design flattening.

Design rules checking generally consists of the following processes, done in the order shown:

1. [General Rules Checking](#)
2. [Procedure Rules Checking](#)
3. [Bus Mutual Exclusivity Analysis](#)
4. [Scan Chain Tracing](#)
5. [Shadow Latch Identification](#)
6. [Data Rules Checking](#)
7. [Clock Rules Checking](#)
8. [RAM Rules Checking](#)
9. [Extra Rules Checking](#)
10. [Constrained/Forbidden/Block Value Calculations](#)

General Rules Checking	45
Procedure Rules Checking	46
Bus Mutual Exclusivity Analysis	46
Scan Chain Tracing.....	47
Shadow Latch Identification.....	47
Data Rules Checking	48
Clock Rules Checking	48
RAM Rules Checking	48
Extra Rules Checking	49
Constrained/Forbidden/Block Value Calculations.....	49

General Rules Checking

General rules checking searches for very-high-level problems in the information defined for the design.

For example, it checks to ensure the scan circuitry, clock, and RAM definitions all make sense. General rules violations are errors and you cannot change their handling.

Procedure Rules Checking

Procedure rules checking examines the test procedure file.

These checks look for parsing or syntax errors and ensure adherence to each procedure's rules. Procedure rules violations are errors and you cannot change their handling.

Bus Mutual Exclusivity Analysis

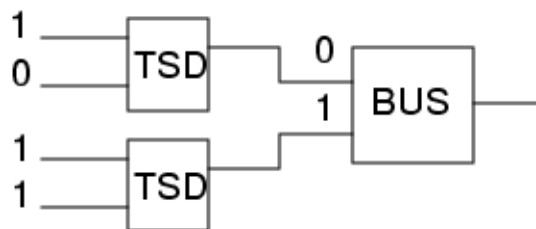
This section addresses the first concern, that ATPG must place buses in a non-contending state. For information on how to handle testing of tri-state devices, see "[Tri-State Devices](#)".

Buses in circuitry can cause the following problems for ATPG:

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

[Figure 1-22](#) shows a bus system that can have contention.

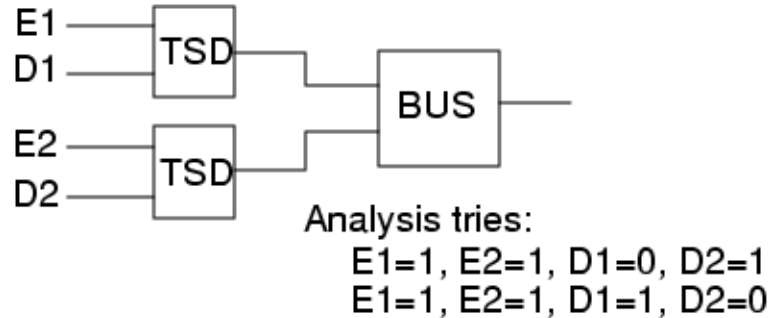
Figure 1-22. Bus Contention Example



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

Buses in a Z-state network can be classified as dominant or non-dominant and strong or weak. Weak buses and pull buses are allowed to have contention. Thus the process only analyzes strong, dominant buses, examining all drivers of these gates and performing full ATPG analysis of all combinations of two drivers being forced to opposite values. [Figure 1-23](#) demonstrates this process on a simple bus system.

Figure 1-23. Bus Contention Analysis



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

You should resolve bus mutual-exclusivity before ATPG. Extra rules E4, E7, E9, E10, E11, E12, and E13 perform bus analysis and contention checking.

Scan Chain Tracing

The purpose of scan chain tracing is for the tool to identify the scan cells in the chain and determine how to use them for control and observe points. Using the information from the test procedure file (which has already been checked for general errors during the procedure rules checks) and the defined scan data, the tool identifies the scan cells in each defined chain and simulates the operation specified by the **load_unload** procedure to ensure proper operation.

Scan chain tracing takes place during the trace rules checks, which trace back through the sensitized path from output to input. Successful scan chain tracing ensures that the tools can use the cells in the chain as control and observe points during ATPG.

Trace rules violations are either errors or warnings, and for most rules you cannot change the handling.

Shadow Latch Identification

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

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

The DRC process identifies shadow latches under the following conditions:

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

Data Rules Checking

Data rules checking ensures the proper transfer of data within the scan chain.

Data rules violations are either errors or warnings, however, you can change the handling.

Clock Rules Checking

After the scan chain trace, clock rules checking is the next most important analysis. Clock rules checks ensure data stability and capturability in the chain.

Clock rules violations are either errors or warnings, however, you can change the handling.

RAM Rules Checking

RAM rules checking ensures consistency with the defined RAM information and the chosen testing mode.

RAM rules violations are all warnings, however, you can change their handling.

Extra Rules Checking

Excluding rule E10, which performs bus mutual-exclusivity checking, most extra rules checks do not have an impact on FlexTest processes. However, they may be useful for enforcing certain design rules.

By default, most extra rules violations are set to ignore, which means they are not even checked during DRC. However, you may change the handling.

Constrained/Forbidden/Block Value Calculations

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

Figure 1-24 gives an example of a tie value gate that constrains some surrounding circuitry.

Figure 1-24. Constrained Values in Circuitry

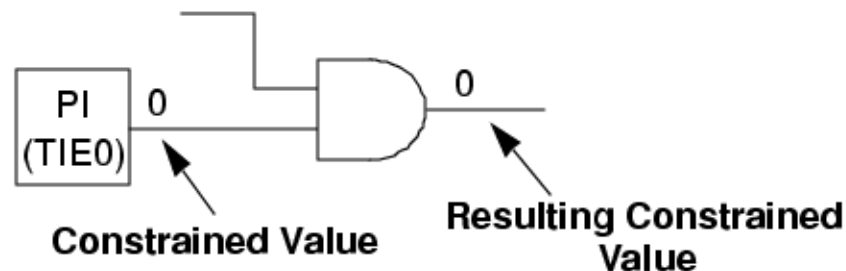


Figure 1-25 gives an example of a tied gate, and the resulting forbidden values of the surrounding circuitry.

Figure 1-25. Forbidden Values in Circuitry

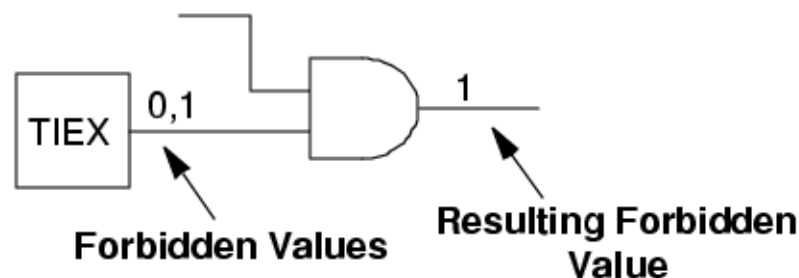
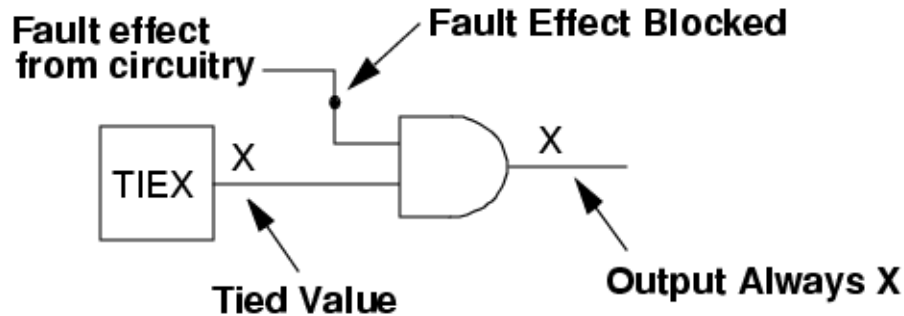


Figure 1-26 gives an example of a tied gate that blocks fault effects in the surrounding circuitry.

Figure 1-26. Blocked Values in Circuitry



Combinational Loop Handling

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

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

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

- **Simulation Method**

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

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

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

To improve performance, for each faulty machine during fault simulation, this loop cutting buffer does not start with an unknown value. Instead, the good machine value is

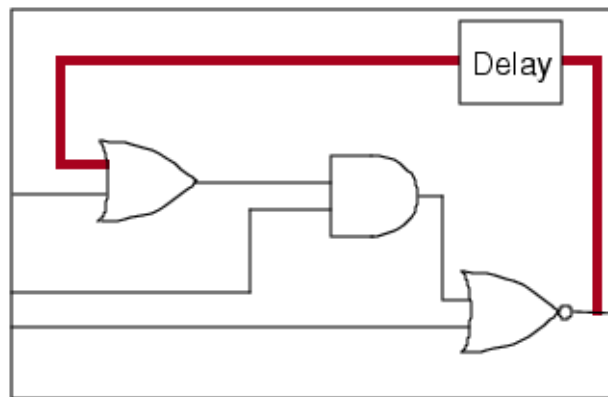
the initial value. However, if the value is changed to the opposite value, an unknown value is then used the first time to ensure loop behavior is properly simulated.

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

- **TIEX or DELAY gate insertion**


Because of its sequential nature, FlexTest can insert a DELAY element, instead of a TIE-X gate, as a means to break loops. The DELAY gate retains the new data for one timeframe before propagating it to the next element in the path. Figure 1-27 shows a DELAY element inserted to break a feedback path.

Figure 1-27. Delay Element Added to Feedback Loop



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

Note

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

- **Turning gate duplication on**

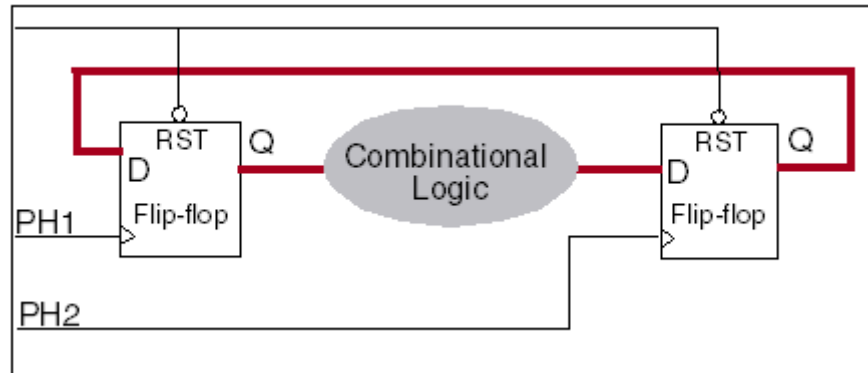
Gate duplication reduces the impact of the TIE-X or DELAY gates that the tool places to break combinational loops. You can turn this option on only when using the Tiex or Delay settings. By default, the gate duplication option is off because FlexTest performs the simulation method upon invocation of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

Sequential Loop Handling

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

Similar to fake combinational loops, fake sequential loops do not exhibit loop behavior. For example, [Figure 1-28](#) shows a fake sequential loop.

Figure 1-28. Fake Sequential Loop



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

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

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

Tri-State Devices

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

FlexTest lets you specify the fault effect of bus contention on tri-state nets. This capability increases the testability of the enable line of the tri-state drivers. Refer to the [Set Net Dominance](#) command.

Non-Scan Cell Handling

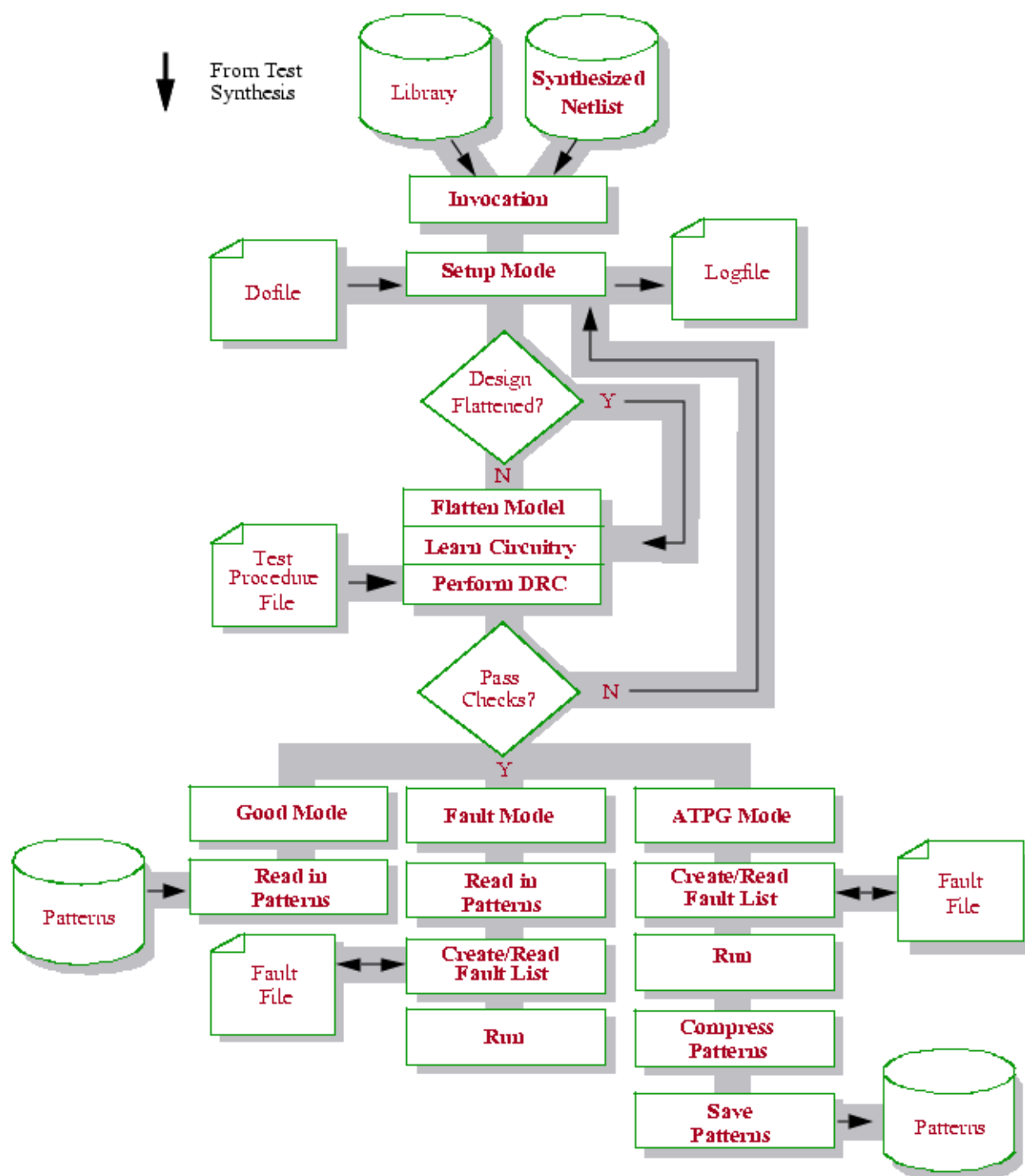
During rules checking and learning analysis, FlexTest learns the behavior of all state elements that are not part of the scan circuitry. This learning involves how the non-scan element behaves after the scan loading operation.

As a result of the learning analysis, FlexTest categorizes each of the non-scan cells.

FlexTest Basic Tool Flow

The following figure shows the basic tool flow for FlexTest.

Figure 1-29. Overview of FlexTest Usage



The following list describes the basic process for using FlexTest:

1. FlexTest requires a structural (gate-level) design netlist and a DFT library. “[FlexTest Inputs and Outputs](#)” describes which netlist formats you can use with FlexTest. Every element in the netlist must have an equivalent description in the specified DFT library.

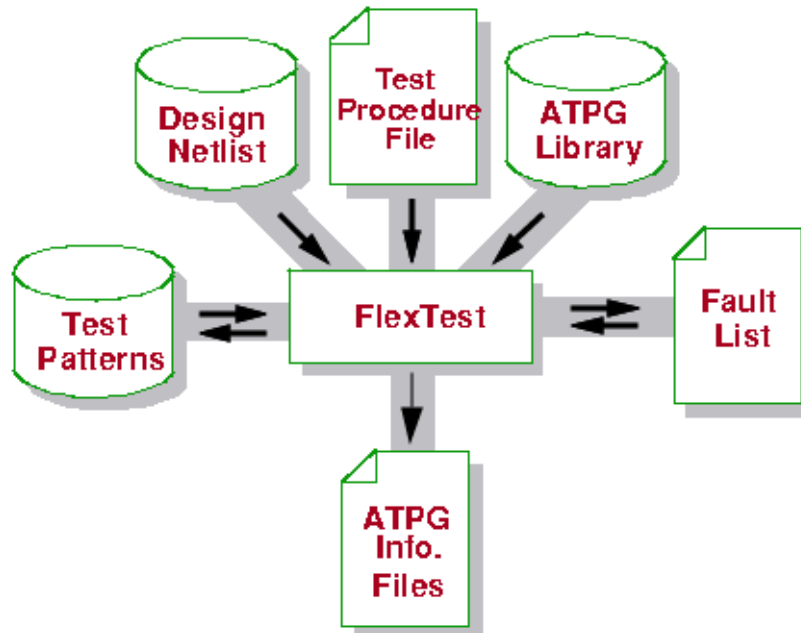
The *Tessent Cell Library Manual* gives information about the DFT library. At invocation, the tool first reads in the library and then the netlist, parsing and checking each. If the tool encounters an error during this process, it issues a message and terminates invocation.

2. After a successful invocation, the tool goes into Setup mode. Within Setup mode, you perform several tasks, using commands either interactively or through the use of a dofile. You can set up information about the design and the design's scan circuitry. "[Setting the System Mode](#)" documents this setup procedure. Within Setup mode, you can also specify information that influences simulation model creation during the design flattening phase.
3. After performing all the desired setup, you can exit the Setup mode. Exiting Setup mode triggers a number of operations. If this is the first attempt to exit Setup mode, the tool creates a flattened design model. This model may already exist if a previous attempt to exit Setup mode failed or you used the Flatten Model command. "[Model Flattening](#)" provides more details on design flattening.
4. Next, the tool performs extensive learning analysis on this model. "[Learning Analysis](#)" explains learning analysis in more detail.
5. Once the tool creates a flattened model and learns its behavior, it begins design rules checking.
6. Once the design passes rules checking, the tool enters either Good, Fault, or Atpg mode. While typically you would enter the Atpg mode, you may want to perform good machine simulation on a pattern set for the design. "[Executing Good Machine Simulation](#)" describes this procedure.
7. You may also just want to fault simulate a set of external patterns. "[Executing Good Machine Simulation](#)" documents this procedure.
8. At this point, you may typically want to create patterns. However, you must perform some additional setup steps, such as creating the fault list. "[Setting Up the Fault Information for ATPG](#)" details this procedure. You can then run ATPG on the fault list. During the ATPG run, the tool also performs fault simulation to verify that the generated patterns detect the targeted faults.
9. After generating a test set with FlexTest, you should apply timing information to the patterns and verify the design and patterns before handing them off to the vendor. "[Verifying Test Patterns](#)" documents this operation.

FlexTest Inputs and Outputs

The following figure shows the inputs and outputs of the FlexTest application.

Figure 1-30. FlexTest Inputs and Outputs



FlexTest utilizes the following inputs:

- **Design** — The supported design data format is gate-level Verilog. Other inputs also include a cell model from the design library.
- **Test Procedure File** — Defines the operation of the scan circuitry in your design. You can generate this file by hand.
- **Library** — Contains descriptions of all the cells used in the design. FlexTest uses the library to translate the design data into a flat, gate-level simulation model for use by the fault simulator and test generator.
- **Fault List** — FlexTest can read in an external fault list. They can use this list of faults and their current status as a starting point for test generation.
- **Test Patterns** — FlexTest can read in externally generated test patterns and use those patterns as the source of patterns to be simulated.

FlexTest produce the following outputs:

- **Test Patterns** — FlexTest generates files containing test patterns. The tool can generate these patterns in a number of different simulator and ASIC vendor formats. [“Saving in ASIC Vendor Data Formats”](#) discusses the test pattern formats in more detail.
- **ATPG Information Files** — These consist of a set of files containing information from the ATPG session. For example, you can specify creation of a log file for the session.
- **Fault List** — This is an ASCII-readable file that contains internal fault information in a proprietary Siemens EDA fault format.

Understanding FlexTest's ATPG Method

Some sequential ATPG algorithms must go forward and backward in time to generate a test. These algorithms are not practical for large and deep sequential circuits, due to high memory requirements. FlexTest uses a general sequential ATPG algorithm, called the BACK algorithm, that avoids this problem. The BACK algorithm uses the behavior of a target fault to predict which primary output (PO) to use as the fault effect observe point. Working from the selected PO, it sensitizes the path backward to the fault site. After creating a test sequence for the target fault, FlexTest uses a parallel differential fault simulator for synchronous sequential circuits to calculate all the faults detected by the test sequence. To facilitate the ATPG process, FlexTest first performs redundancy identification when exiting the Setup mode.

This is typically how FlexTest performs ATPG. However, FlexTest can also generate functional vectors based on the instruction set of a design. The ATPG method it uses in this situation is significantly different from the sequential-based ATPG method it normally uses. For information on using FlexTest in this capacity, refer to "[Creating Instruction-Based Test Sets](#)".

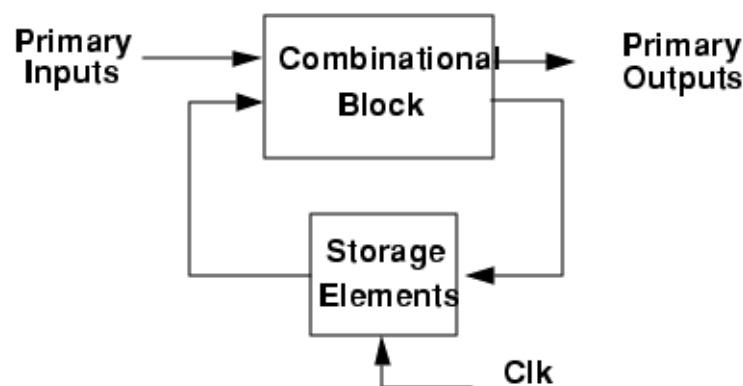
Cycle-Based Timing Circuits	57
Cycle-Based Timing Model.....	58
Cycle-Based Test Patterns.....	60

Cycle-Based Timing Circuits

Circuits have cycle-based behavior if their output values are always stable at the end of each cycle period. Most designers of synchronous and asynchronous circuits use this concept.

[Figure 1-31](#) gives an example of a cycle-based circuit.

Figure 1-31. Cycle-Based Circuit with Single Phase Clock



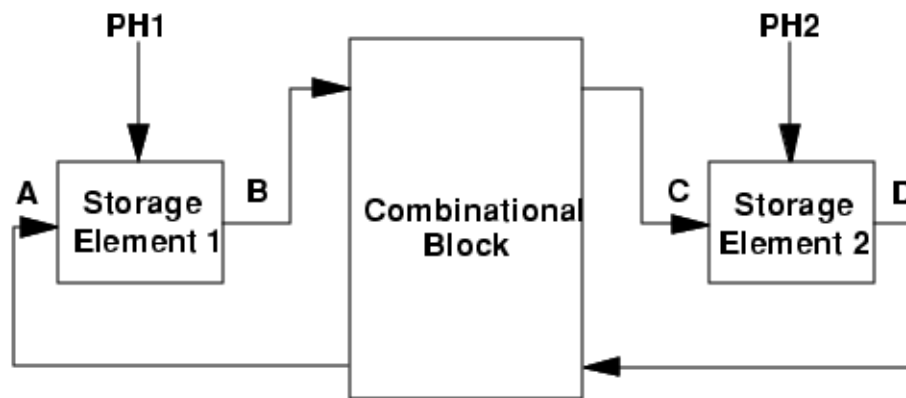
In [Figure 1-31](#), all the storage elements are edge-triggered flip-flops controlled by the rising edge of a single clock. The primary outputs and the final values of the storage elements are always stable at the end of each clock cycle, as long as the data and clock inputs of all flip-flops do not change their values at the same time. The clock period must be longer than the longest

signal path in the combinational block. Also, stable values depend only on the primary input values and the initial values on the storage elements.

For the multiple-phase design, relative timing among all the clock inputs determines whether the circuit maintains its cycle-based behavior.

In [Figure 1-32](#), the clocks PH1 and PH2 control two groups of level-sensitive latches which make up this circuit's storage elements.

Figure 1-32. Cycle-Based Circuit with Two Phase Clock



When PH1 is on and PH2 is off, the signal propagates from point D to point C. On the other hand, the signal propagates from point B to point A when PH1 is off and PH2 is on. Designers commonly use this cycle-based methodology in two-phase circuits because it generates systematic and predictable circuit behavior. As long as PH1 and PH2 are not on at the same time, the circuit exhibits cycle-based behavior. If these two clocks are on at the same time, the circuit can operate in an unpredictable manner and can even become unstable.

Cycle-Based Timing Model

All automatic test equipment (ATE) are cycle-based, unlike event-based digital simulators. A test cycle for ATE is the waveform (stored pattern) applied to all primary inputs and observed at all primary outputs of the device under test (DUT). Each test cycle has a corresponding timing definition for each pin.

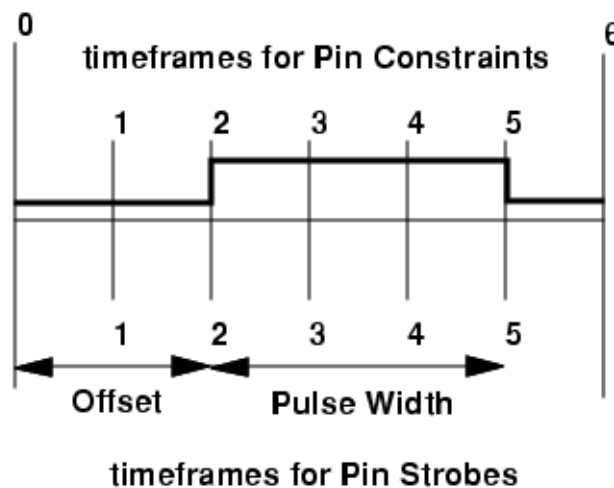
In FlexTest, you must specify the timing information for the test cycles. FlexTest provides a sophisticated timing model that you can use to properly manage timing relationships among primary inputs—especially for critical signals, such as clock inputs.

FlexTest uses a test cycle, which is conceptually the same as an ATE test cycle, to represent the period of each primary input. If the input cycle of a primary input is longer (for example, a signal with a slower frequency) than the length you set for the test cycle, then you must represent its period as a multiple of test cycles.

A test cycle further divides into timeframes. A timeframe is the smallest time unit that FlexTest can simulate. The tool simulates whatever events occur in the timeframe until signal values stabilize. For example, if data inputs change during a timeframe, the tool simulates them until the values stabilize. The number of timeframes equals the number of simulation processes FlexTest performs during a test cycle. At least one input must change during a defined timeframe. You use timeframes to define the test cycle terms offset and the pulse width. The offset is the number of timeframes that occur in the test cycle before the primary input goes active. The pulse width is the number of timeframes the primary input stays active.

Figure 1-33 shows a primary input with a positive pulse in a six timeframe test cycle. In this example, the period of the primary input is one test cycle. The length of the test cycle is six timeframes, the offset is two timeframes, and the width of its pulse is three timeframes.

Figure 1-33. Example Test Cycle



In this example, if other primary inputs have periods longer than the test cycle, you must define them in multiples of six timeframes (the defined test cycle period). Time 0 is the same as time 6, except time 0 is treated as the beginning of the test cycle, while time 6 is treated as the end of the test cycle.

Note



To increase the performance of FlexTest fault simulation and ATPG, you should try to define the test cycle to use as few timeframes as possible.

For most automatic test equipment, the tester strobes each primary output only once in each test cycle and can strobe different primary outputs at different timeframes. In the non-scan environment, FlexTest strobes primary outputs at the end of each test cycle by default.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the

strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

In the scan environment, if any scan memory element capture clock is on, the scan-in values in the scan memory elements change. Therefore, in the scan test, right after the scan load/unload operation, no clocks can be on. Also, the primary output strobe should occur before any clocks turn on. Thus, in the scan environment, FlexTest strobes primary outputs after the first timeframe of each test cycle by default.

If you strobe a primary output while the primary inputs are changing, FlexTest first strobes the primary output and then changes the values at the primary inputs. To be consistent with the boundary of the test cycle (using [Figure 1-33](#) as an example), you must describe the primary input's value change at time 6 as the change in value at time 0 of the next test cycle. Similarly, the strobe time at time 0 is the same as the strobe time at time 6 of the previous test cycle.

Cycle-Based Test Patterns

Each primary input has its own signal frequency and cycle. Test patterns are cycle-based if each individual input either holds its value or changes its value at a specific time in each of its own input cycle periods. Also, the width of the period of every primary input has to be equal to or a multiple of test cycles used by the automatic test equipment.

Cycle-based test patterns are easy to use and tend to be portable among the various automatic test equipment. For most ATE, the tester allows each primary input to change its value up to two times within its own input cycle period. A constant value means that the value of the primary input does not change. If the value of the primary input changes only once (generally for data inputs) in its own cycle, then the tester holds the new value for one cycle period. A pulse input means that the value of the primary input changes twice in its own cycle. For example, clock inputs behave in this manner.

Performing Basic Operations

This section describes the most basic operations you may need to perform FlexTest.

Invoking the Application.....	61
Invoking the FlexTest Fault Simulation Version	61
FlexTest Interrupt Capabilities	62
Setting the System Mode	62
Setting the Circuit Timing	64
Setting Up the Fault Information for ATPG.....	68
Setting Self-Initialized Test Sequences	70
Setting the Hypertrophic Limit	71
Setting DS Fault Handling	71
Setting the Possible-Detect Credit	71
Defining ATPG Constraints	72
Creating Patterns With Default Settings.....	73
Compressing Patterns	73
Reporting on ATPG Untestable Faults	73
Creating a Selective IDDQ Test Set	75
Generating a Supplemental IDDQ Test Set	77
Specifying Leakage Current Checks	78

Invoking the Application

You invoke the FlexTest application.

Procedure

1. You can invoke FlexTest using the following invocation:
<mgcdft tree>/bin/flextest arguments
2. You must enter all required arguments at the shell command line.
3. When the tool is finished invoking, the design and library are also loaded. The tool is now in Setup mode and ready for you to begin working on your design.

Invoking the FlexTest Fault Simulation Version

Similarly, FlexTest is available in a fault simulation only package called FlexTest FaultSim. This version of the tool has only the Setup, Drc, Good, and Fault system modes. An error condition occurs if you attempt to enter the Atpg system mode.

Procedure

You invoke this version of FlexTest using the -Faultsim switch, which checks for the fault simulation license, and if found, invokes the fault simulation package.

FlexTest Interrupt Capabilities

Instead of terminating the current process, FlexTest optionally allows you to interrupt a process. An interrupted process remains in a suspended state.

While in a suspended state, you may execute any of the following commands:

- Help
- all Report commands
- all Write commands
- Set Abort Limit
- Set Atpg Limits
- Set Checkpoint
- Set Fault Mode
- Set Gate Level
- Set Gate Report
- Set Logfile Handling
- Save Patterns

You may find these commands useful in determining whether or not to resume the process. By default, interrupt handling is off, thus aborting interrupted processes. If instead of aborting, you want an interrupted process to remain in a suspended state, you can issue the [Set Interrupt Handling](#) command. After you turn interrupt handling on and interrupt a process, you can either terminate the suspended process using the [Abort Interrupted Process](#) command or continue the process using the [Resume Interrupted Process](#) command.

Setting the System Mode

When FlexTest invokes, the tools assumes the first thing you want to do is set up circuit behavior, so they automatically put you in Setup mode.

The entire set of system modes includes:

- SETUP — Use to set up circuit behavior.
- DRC — Use to retain the flattened design model for design rules checking.

- ATPG — Use to run test pattern generation.
- FAULT — Use to run fault simulation.
- GOOD — Use to run good simulation.

To change the system mode, you use the [Set System Mode](#) command.

Setting the Circuit Timing

To create reliable test patterns with FlexTest, you need to provide proper timing information for certain primary inputs. The following subsections describe how to set circuit timing.

If you need to better understand FlexTest timing, you should refer to “[Test Pattern Formatting and Timing](#)”.

Setting the Test Cycle Width	64
Defining the Cycle Behavior of Primary Inputs	64
Defining the Strobe Time of Primary Outputs	65
Fault Simulation on Simulation Derived Vectors	66
Executing Good Machine Simulation	66
Debugging the Good Machine Simulation	67
Resetting Circuit Status	67

Setting the Test Cycle Width

When you set the test cycle width, you specify the number of timeframes needed per test cycle. The larger the number you enter for timeframes, the better the resolution you have when adding pin constraints. The smaller the number of timeframes you specify per cycle, the better the performance FlexTest has during ATPG.

By default, FlexTest assumes a test cycle of one timeframe. However, typically you will need to set the test cycle to two timeframes. And if you define a clock using the [Add Clocks](#) command, you must specify at least two timeframes. In a typical test cycle, the first timeframe is when the data inputs change (forced and measured) and the second timeframe is when the clock changes. If you have multi-phased clocks, or want certain data pins to change when the clock is active, you should set three or more timeframes per test cycle.

At least one input or set of inputs should change in a given timeframe. If not, the timeframe is unnecessary. Unnecessary timeframes adversely affect FlexTest performance. When you attempt to exit Setup mode, FlexTest checks for unnecessary timeframes, just prior to design flattening. If the check fails, FlexTest issues an error message and remains in Setup mode.

To set the number of timeframes in a test cycle, you use the [Set Test Cycle](#) command.

Defining the Cycle Behavior of Primary Inputs

As discussed previously, testers are naturally cyclic and the test patterns FlexTest generates are also cyclic. Events occur repeatedly, or in cycles. Cycles further divide into timeframes. Clocks exhibit cyclic behavior and you must define this behavior in terms of the test cycle. Thus, after setting the test cycle width, you need to define the cyclic behavior of the circuit’s primary inputs.

There are three components to describing the cyclic behavior of signals. A pulse signal contains a period (that is equal to or a multiple of test cycles), an offset time, and a pulse width. Constraining a pin lets you define when its signal can change in relation to the defined test cycle. To add pin constraints to a specific pin, you use the [Add Pin Constraints](#) command. The only way to define a constant value signal is by using the constant constraint formats. And for a signal with a hold value, the definition includes a period and an offset time.

There are eleven constraint formats from which to choose. The constraint values (or waveform types) further divide into the three waveform groups used in all automatic test equipment:

- **Group 1: Non-return waveform (Signal value changes only once)** — These include hold (NR <period> <offset>), constant zero (C0), constant one (C1), constant unknown (CX), and constant Z (CZ).
- **Group 2: Return-zero waveform (Signal may go to a 1 and then return to 0)** — These include one positive pulse per period (R0 <period> <offset><width>), one suppressible positive pulse (SR0 <period><offset> <width>), and no positive pulse during non-scan (CR0 <period> <offset> <width>).
- **Group 3: Return-one waveform (Signal may go to a 0 and then return to 1)** — These include one negative pulse per cycle (R1 <period> <offset><width>), one suppressible negative pulse (SR1 <period><offset> <width>), and no negative pulse during non-scan (CR1 <period> <offset> <width>).

Pins not specifically constrained with Add Pin Constraints adopt the default constraint format of NR 1 0. You can change the default constraint format using the [Setup Pin Constraints](#) command.

- [Delete Pin Constraints](#) — Deletes the specified pin constraints.
- [Report Pin Constraints](#) — Displays cycle behavior of the specified inputs.

Defining the Strobe Time of Primary Outputs

After setting the cyclic behavior of all primary inputs, you need to define the strobe time of primary outputs. Each primary output has a strobe time—the time at which the tool measures its value—in each test cycle. Typically, all outputs are strobed at once, however different primary outputs can have different strobe times.

To specify a unique strobe time for certain primary outputs, you use the [Add Pin Strokes](#) command. You can also optionally specify the period for each pin strobe.

Any primary output without a specified strobe time uses the default strobe time. To set the default strobe time for all unspecified primary output pins, you use the [Setup Pin Strokes](#) command.

FlexTest groups all primary outputs with the same pin strobe time in the same output bus array, even if the outputs have different pin strobe periods. At each test cycle, FlexTest displays the

strobed values of all output bus arrays. Primary outputs not strobed in the particular test cycle receive unknown values.

- [Delete Pin Strobes](#) — Deletes the specified pin strobes.
- [Report Pin Strobes](#) — Displays the strobe time of the specified outputs.

Fault Simulation on Simulation Derived Vectors

In many cases, you begin test generation with a set of vectors previously derived from a simulator. You can read in these external patterns in a compatible format (FlexTest Table format for example), and have FlexTest perform fault simulation on them.

FlexTest uses these existing patterns to initialize the circuit and give some initial fault coverage. Then you can perform ATPG on the remaining faults. This method can result in more efficient test pattern sets and shorter test generation run times.

Running Fault Simulation on the Functional Vectors

To run fault simulation on the vectors that are in FlexTest table format, use the following commands:

```
SETUP> set system mode atpg
ATPG> set pattern source external table.flex -table
ATPG> add faults -all
ATPG> run
ATPG> set pattern source internal
ATPG> run
```

First, set the system mode to Atpg if you are not already in that system mode. Next, you must specify that the patterns you want to simulate are in an external file (named table.flex in this example). Then generate the fault list including all faults, and run the simulation. You could then set the pattern source to be internal and run the basic ATPG process on the remaining undetected faults.

Executing Good Machine Simulation

During good machine simulation, the tool compares good machine simulation results to an external pattern source, primarily for debugging purposes.

Procedure

1. To set up good circuit simulation comparison within FlexTest, use the [Set Output Comparison](#) command from the Good system mode.
2. By default, the output comparison of good circuit simulation is off. FlexTest performs the comparison if you specify ON. The -X_ignore options will allow you to control whether X values, in either simulated results or reference output, should be ignored when output comparison capability is used.

3. To execute the simulation comparison, enter the [Run](#) command at the Good mode prompt as follows:

GOOD> run

Debugging the Good Machine Simulation

You can debug your good machine simulation in several ways.

Procedure

1. If you want to run the simulation and save the values of certain pins in batch mode, you can use the [Add Lists](#) and [Set List File](#) commands.
2. The Add Lists command specifies which pins to report. The Set List File command specifies the name of the file in which you want to place simulation values for the selected pins.
3. If you prefer to perform interactive debugging, you can use the [Run](#) and [Report Gates](#) commands to examine internal pin values. If using FlexTest, you can use the -Record switch with the Run command to store the internal states for the specified number of test cycles.

Resetting Circuit Status

In Good mode, you can reset the circuit status.

Procedure

To reset the status use the [Reset State](#) command.

Setting Up the Fault Information for ATPG

Prior to performing test generation, you must set up a list of all faults the application has to evaluate. The tool can either read the list in from an external source, or generate the list itself. The type of faults in the fault list vary depending on the fault model and your targeted test type.

For more information on fault modeling and the supported models, refer to “[FlexTest Fault Modeling](#)”.

After the application identifies all the faults, it implements a process of structural equivalence fault collapsing from the original uncollapsed fault list. From this point on, the application works on the collapsed fault list. The results, however, are reported for both the uncollapsed and collapsed fault lists. Executing any command that changes the fault list causes the tool to discard all patterns in the current internal test pattern set due to the probable introduction of inconsistencies. Also, whenever you re-enter the Setup mode, it deletes all faults from the current fault list. The following subsections describe how to create a fault list and define fault related information.

Changing to the ATPG System Mode	68
Setting the Fault Type	68
Creating the Faults List.....	69
Adding Faults to an Existing List.....	69
Loading Faults From an External List	70
Writing Faults to an External File	70

Changing to the ATPG System Mode

You can enter the fault list commands from the Good, Fault, or Atpg system modes.

Procedure

1. In the context of running ATPG, you must switch from Setup to the Atpg mode using the [Set System Mode](#) command.
2. For example, assuming your circuit passes rules checking with no violations, you can exit the Setup system mode and enter the Atpg system mode as follows:

```
SETUP> set system mode atpg
```

Setting the Fault Type

By default, the fault type is stuck-at. If you want to generate patterns to detect stuck-at faults, you do not need to issue this command.

Procedure

1. If you wish to change the fault type to toggle, pseudo stuck-at (IDDQ), transition, you can issue the [Set Fault Type](#) command.
2. Whenever you change the fault type, the application deletes the current fault list and current internal pattern set.

Creating the Faults List

The application creates the internal fault list the first time you add faults or load in external faults. Typically, you would create a fault list with all possible faults of the selected type, although you can place some restrictions on the types of faults in the list.

Procedure

1. To create a list with all faults of the given type, enter the [Add Faults](#) command using the -All switch as in the following example:

ATPG> add faults -all
2. If you do not want all possible faults in the list, you can use other options of the Add Faults command to restrict the added faults. You can also specify no-faulted instances to limit placing faults in the list. You flag instances as “Nofault” while in Setup mode. For more information, refer to the [Add Nofaults](#) command.
3. When the tool first generates the fault list, it classifies all faults as uncontrolled (UC).
 - [Delete Faults](#)— Deletes the specified faults from the current fault list.
 - [Report Faults](#) — Displays the specified types of faults.

Adding Faults to an Existing List

You can add new faults to the existing fault list.

Procedure

1. To add new faults to the current fault list, enter the [Add Faults](#) command.
2. You must enter either a list of object names (pin pathnames or instance names) or use the -All switch to indicate the pins whose faults you want added to the fault list. You can use the -Stuck-at switch to indicate which stuck faults on the selected pins you want added to the list. If you do not use the Stuck-at switch, the tool adds both stuck-at-0 and stuck-at-1 faults. FlexTest initially places faults added to a fault list in the undetected-uncontrolled (UC) fault class.

Loading Faults From an External List

You can place faults from a previous run (from an external file) into the internal fault list.

Procedure

1. To load faults from an external file into the current fault list, enter the [Load Faults](#) command.
2. The applications support external fault files in the 3, 4, or 6 column formats. The only data they use from the external file is the first column (stuck-at value) and the last column (pin pathname)—unless you use the -Restore option.
3. The -Retain option (-Restore in FlexTest) causes the application to retain the fault class (second column of information) from the external fault list. The -Delete option deletes all faults in the specified file from the internal faults list. The -Column option, in FlexTest, specifies the column format of the fault file.

Writing Faults to an External File

You can write all or only selected faults from a current fault list into an external file. You can then edit or load this file to create a new fault list.

Procedure

To write faults to a file, enter the [Write Faults](#) command. You must specify the name of the file you want to write.

Setting Self-Initialized Test Sequences

FlexTest generates test sequences for target faults that are self-initialized. With the knowledge of self-initialized test sequences, static vector compaction by reordering is possible, as well as splitting the test set without losing test coverage. Some pattern compaction routines also rely on the self-initializing properties of sequences. Each self-initialized test sequence is defined as a test pattern.

The [Set Self Initialization](#) command allows you to turn this feature on or off. By default, self-initializing behavior is on.

If the self-initializing property is enabled during ATPG:

- Self-initializing boundaries in the test set will be determined.
- During fault simulation, all state elements (except the ones with TIED properties) at self-initializing boundaries are set to X. Therefore, the reported fault coverage is actually the lower bound to the real fault coverage if state information were maintained between self-initializing sequences (the reported coverage will be close to or equal to the real fault coverage).

The self-initializing results can be saved by issuing the [Save Patterns -Ascii](#) command.

Note

Only the ASCII pattern format includes this test pattern information.

Setting the Hypertrophic Limit

To improve fault simulation performance, you can reduce or eliminate hypertrophic faults with little consequence to the accuracy of the fault coverage. In fault simulation, hypertrophic faults require additional memory and processor time. These type of faults do not occur often, but do significantly affect fault simulation performance.

Procedure

1. To set the hypertrophic limit, enter the [Set Hypertrophic Limit](#) command.
2. You can specify a percentage between 1 and 100, which means that when a fault begins to cause more than that percent of the state elements to deviate from the good machine status, the simulator will drop that fault from simulation. The default is a 30% difference (between good and faulty machine status) to classify a fault as hypertrophic. To improve performance, you can reduce the percentage number.

Setting DS Fault Handling

To facilitate fault diagnosis, you can set FlexTest to carry out fault simulation without dropping faults. The Set Fault Dropping command enables or disables the dropping of DS faults when FlexTest is in Fault mode.

Procedure

1. To set DS fault handling, enter the [Set Fault Dropping](#) command.
2. When carrying out fault simulation, enabling fault dropping sets FlexTest to drop DS faults; this is the default behavior of the tool.

Setting the Possible-Detect Credit


Before reporting test coverage, fault coverage, and ATPG effectiveness, you should specify the credit you want given to possible-detected faults.

Procedure

1. To set the credit to be given to possible-detected faults, use the [Set Possible Credit](#) command.

2. The selected credit may be any positive integer less than or equal to 100, the default being 50%.

Note

 If you are using FlexTest and you set the possible detection credit to 0, it does not place any faults in the possible-detected category. If faults already exist in these categories, the tool reclassifies PT faults as UO and PU faults as AU.

Defining ATPG Constraints

ATPG constraints are similar to pin constraints and scan cell constraints. Pin constraints and scan cell constraints restrict the values of pins and scan cells, respectively. ATPG constraints place restrictions on the acceptable kinds of values at any location in the circuit. For example, you can use ATPG constraints to prevent bus contention or other undesirable events within a design. Additionally, your design may have certain conditions that can never occur under normal system operation. If you want to place these same constraints on the circuit during ATPG, use ATPG constraints.

During deterministic pattern generation, only the restricted values on the constrained circuitry are allowed. Unlike pin and scan cell constraints, which are only available in Setup mode, you can define ATPG constraints in any system mode after design flattening. If you want to set ATPG constraints prior to performing design rules checking, you must first create a flattened model of the design using the Flatten Model command.

ATPG constraints are useful when you know something about the way the circuit behaves that you want the ATPG process to examine. For example, the design may have a portion of circuitry that behaves like a bus system; that is, only one of various inputs may be on, or selected, at a time. Using ATPG constraints, combined with a defined ATPG function, you can specify this information to FlexTest. ATPG functions place artificial Boolean relationships on circuitry within your design. After defining the functionality of a portion of circuitry with an ATPG function, you can then constrain the value of the function as desired with an ATPG constraint. This is more useful than just constraining a point in a design to a specific value.

FlexTest allows you to specify temporal ATPG functions by using a Delay primitive to delay the signal for one timeframe. Temporal constraints can be achieved by combining ATPG constraints with the temporal function options.

Setting ATPG Limits

Normally, there is no need to limit the ATPG process when creating patterns. There may be an occasional special case, however, when you want FlexTest to terminate the ATPG process if CPU time, test coverage, or pattern (cycle) count limits are met. To set these limits, use the [Set Atpg Limits](#) command.

FlexTest Only — The last test sequence generated by an ATPG process is truncated to make sure the total test cycles do not exceed cycle limit.

Creating Patterns With Default Settings

Use the default settings to create patterns in FlexTest.

Procedure

1. An ATPG process is initiated with the [Run](#) command:
ATPG> run
2. To analyze the results if pattern creation fails, use the [Add Atpg Constraints](#) command.


Compressing Patterns

Because a tester requires a relatively long time to apply each scan pattern, it is important to create as small a test pattern set as possible while still maintaining the same test coverage. Static pattern compression minimizes the number of test patterns in a generated set.

Patterns generated early on in the pattern set may no longer be necessary because later patterns also detect the faults detected by these earlier patterns. Thus, you can compress the pattern set by rerunning fault simulation on the same patterns, first in reverse order and then in random order, keeping only those patterns necessary for fault detection. This method normally reduces an uncompressed original test pattern set by 30 to 40 percent with very little effort.

To apply static compression to test patterns, you use the [Compress Patterns](#) command.

Note

 The tool only performs pattern compression on independent test blocks; that is, for patterns generated for combinational or scan designs. Thus, FlexTest first does some checking of the test set to determine whether it can implement pattern compression.

Reporting on ATPG Untestable Faults

FlexTest has the capability to report the reasons why a fault is classified as ATPG_untestable (AU). This fault category includes AU, UI, PU, OU, and HU faults.

For more information on these fault categories, refer to “[FlexTest Fault Classes](#)”. You can determine why these faults are undetected by using the [Report Au Faults](#) command.

Setting the Abort Limit

If the fault list contains a number of aborted faults, the tools may be able to detect these faults if you change the abort limit. You can increase the abort limit for the number of backtracks, test cycles, or CPU time and recreate patterns.

The default for combinational ATPG is 30. The clock sequential abort limit defaults to the limit set for combinational. Both the [Report Environment](#) command and a message at the start of deterministic test generation indicate the combinational and sequential abort limits. If they differ, the sequential limit follows the combinational abort limit.

For FlexTest, the [Set Abort Limit](#) command's initial defaults are 30 backtracks, 300 test cycles, and 300 seconds per target fault. If your fault coverage is too low, you may want to re-issue this command using a larger integer (500 is a reasonable choice for a second pass) with the -Backtrack switch. Use caution, however, because if the numbers you specify are too high, test generation may take a long time to complete.

The application classifies any faults that remain undetected after reaching the limits as aborted faults—which it considers undetected faults.

Creating a Selective IDDQ Test Set

The following subsections discuss basic information about selecting IDDQ patterns from an existing set, and also present an example of a typical IDDQ pattern selection run.

Setting the External Pattern Set.....	75
Determining When to Perform the Measures.....	75
Selecting the Best IDDQ Patterns.....	76
Selective IDDQ Example.....	76

Setting the External Pattern Set

In order to create a selective IDDQ test set, you must have an existing set of test patterns. These patterns must reside in an external file, and you must change the pattern source so the tool works from this external file.

You specify the external pattern source using the [Set Pattern Source](#) command. This external file must be in FlexTest Text format.

Determining When to Perform the Measures

The pre-existing external test set may or may not target IDDQ faults. For example, you can run ATPG using the stuck-at fault type and then select patterns from this set for IDDQ testing. If the pattern set does not target IDDQ faults, it will not contain statements that specify IDDQ measurements. IDDQ test patterns must contain statements that tell the tester to make an IDDQ measure.

In FlexTest Text format, this IDDQ measure statement, or label, appears as follows:

```
measure IDDQ ALL <time>;
```

By default, FlexTest places these statements at the end of patterns (cycles) that can contain IDDQ measurements. You can manually add these statements to patterns (cycles) within the external pattern set.

When you want to select patterns from an external set, you must specify which patterns can contain an IDDQ measurement. If the pattern set contains no IDDQ measure statements, you can specify that the tools assume the tester can make a measurement at the end of each pattern or cycle. If the pattern set already contains IDDQ measure statements (if you manually added these statements), you can specify that simulation should only occur for those patterns that already contain an IDDQ measure statement, or label. To set this measurement information, use the [Set Iddq Strobe](#) command.

Selecting the Best IDDQ Patterns

Generally, ASIC vendors have restrictions on the number of IDDQ measurements they allow. The expensive nature of IDDQ measurements typically restricts a test set to a small number of patterns with IDDQ measure statements.

Additionally, you can set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. “[Specifying Leakage Current Checks](#)” discusses these IDDQ restrictions. To specify the IDDQ pattern selection criteria and run the selection process, use [Select Iddq Patterns](#).

The `Select Iddq Patterns` command fault simulates the current pattern source and determines the IDDQ patterns that best meet the selection criteria you specify, thus creating an IDDQ test pattern set. If working from an external pattern source, it reads the external patterns into the internal pattern set, and places IDDQ measure statements within the selected patterns or cycles of this test set based on the specified selection criteria.

Note



FlexTest supplies some additional arguments for this command. Refer to [Select Iddq Patterns](#) for details.

Selective IDDQ Example

The following list demonstrates a common situation in which you could select IDDQ test patterns using FlexTest.

1. Invoke FlexTest on the design, set up the appropriate parameters for ATPG run, pass rules checking, and enter the ATPG mode.

```
...  
SETUP> set system mode atpg
```

This example assumes you set the fault type to stuck-at, or some fault type other than IDDQ.

2. Run ATPG.

```
ATPG> run
```

3. Save generated test set to external file named *orig.pats*.

```
ATPG> save patterns orig.pats
```

4. Change pattern source to the saved external file.

```
ATPG> set pattern source external orig.pats
```

5. Set the fault type to IDDQ.

```
ATPG> set fault type iddq
```

6. Add all IDDQ faults to the current fault list.

ATPG> add faults -all

7. Assume IDDQ measurements can occur within each pattern or cycle in the external pattern set.

ATPG> set iddq strobe -all

8. Specify to select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

Note



You could use the Set Iddq Checks command prior to the ATPG run to place restrictions on the selected patterns.

ATPG> select iddq patterns -max_measure 15 -threshold 10

9. Save these IDDQ patterns into a file.

ATPG> save patterns iddq.pats

Generating a Supplemental IDDQ Test Set

The following subsections discuss the basic IDDQ pattern generation process and provide an example of a typical IDDQ pattern generation run.

Generating the Patterns

Prior to pattern generation, you may want to set up restrictions that the selection process must abide by when choosing the best IDDQ patterns. “[Specifying Leakage Current Checks](#)” on page 78 discusses these IDDQ restrictions. As with any other fault type, you issue the [Run](#) command within ATPG mode. This generates an internal pattern set targeting the IDDQ faults in the current list.

Selecting the Best IDDQ Patterns

Issuing the [Run](#) command results in an internal IDDQ pattern set. Each pattern generated automatically contains a “measure IDDQ ALL” statement, or label. If you use FlexTest to generate the IDDQ patterns, you do not need to use the [Set Iddq Strobe](#) command, because (by default) the tool only simulates IDDQ measures at each label.

The generated IDDQ pattern set may contain more patterns than you want for IDDQ testing. At this point, you just set up the IDDQ pattern selection criteria and run the selection process using the [Select Iddq Patterns](#) command.

Supplemental IDDQ Example

1. Invoke FlexTest on design, set up appropriate parameters for ATPG run, pass rules checking, and enter ATPG mode.

```
...  
SETUP> set system mode atpg
```

2. Set the fault type to IDDQ.

```
ATPG> set fault type iddq
```

3. Add all IDDQ faults to the current fault list.

```
ATPG> add faults -all
```

Instead of creating a new fault list, you could load a previously-saved fault list. For example, you could write the undetected faults from a previous ATPG run and load them into the current session with Load Faults, using them as the basis for the IDDQ ATPG run.

4. Run ATPG, generating patterns that target the IDDQ faults in the current fault list.

Note



You could use the [Set Iddq Checks](#) command prior to the ATPG run to place restrictions on the generated patterns.

```
ATPG> run
```

5. Select the best 15 IDDQ patterns that detect a minimum of 10 IDDQ faults each.

```
ATPG> select iddq patterns -max_measure 15 -threshold 10
```

Note



You did not need to specify which patterns could contain IDDQ measures with [Set Iddq Strobe](#), as the generated internal pattern source already contains the appropriate measure statements.

6. Save these IDDQ patterns into a file.

```
ATPG> save patterns iddq.pats
```

Specifying Leakage Current Checks

For CMOS circuits with pull-up or pull-down resistors or tri-state buffers, the good circuit should have a nearly zero IDDQ current. FlexTest allows you to specify various IDDQ measurement checks to ensure that the good circuit does not raise IDDQ current during the measurement.

Use the [Set Iddq Checks](#) command to specify these options.

By default, the tool does not perform IDDQ checks. Both ATPG and fault simulation processes consider the checks you specify.

Creating Instruction-Based Test Sets

FlexTest can generate a functional test pattern set based on the instruction set of a design. You would typically use this method of test generation for high-end, non-scan designs containing a block of logic, such as a microprocessor or ALU. Because this is embedded logic and not fully controllable or observable from the design level, testing this type of functional block is not a trivial task. In many such cases, the easiest way to approach test generation is through manipulation of the instruction set.

Given information on the instruction set of a design, FlexTest randomly combines these instructions and determines the best data values to generate a high test coverage functional pattern set. You enable this functionality by using the [Set Instruction Atpg](#) command.

By default, FlexTest turns off instruction-based ATPG. If you choose to turn this capability on, you must specify a filename defining information on the design's input pins and instruction set. The following subsections discuss the fault detection method and instruction information requirements in more detail.

Instruction-Based Fault Detection	80
Instruction File Format	81

Instruction-Based Fault Detection

The instruction set of a design relates to a set of values on the control pins of a design. Given the set of control pin values that define the instruction set, FlexTest can determine the best data pin (and other non-constrained pin) values for fault detection.

For example, [Table 1-2](#) shows the pin value requirements for an ADD instruction which completes in three test cycles.

Note


 An N value indicates the pin may take on a new value, while an H indicates the pin must hold its current value.

Table 1-2. Pin Value Requirements for ADD Instruction

	Ctrl 1	Ctrl 2	Ctrl 3	Ctrl 4	Data1	Data2	Data3	Data4	Data5	Data6
Cycle1	1	0	1	0	N	N	N	N	N	N
Cycle2	H	H	H	H	H	H	H	H	H	H
Cycle3	H	H	H	H	H	H	H	H	H	H

As [Table 1-2](#) indicates, the value 1010 on pins Ctrl1, Ctrl2, Ctrl3, and Ctrl4 defines the ADD instruction. Thus, a vector to test the functionality of the ADD instruction must contain this value on the control pins. However, the tool does not constrain the data pin values to any

particular values. That is, FlexTest can test the ADD instruction with many different data values. Given the constraints on the control pins, FlexTest generates patterns for the data pin values, fault simulates the patterns, and keeps those that achieve the highest fault detection.

Instruction File Format

The following list describes the syntax rules for the instruction file format.

- The file consists of three sections, each defining a specific type of information: control inputs, data inputs, and instructions.
- You define control pins, with one pin name per line, following the “Control Input:” keyword.
- You define data pins, with one pin name per line, following the “Data Input:” keyword.
- You define instructions, with all pin values for one test cycle per line, following the “Instruction” keyword. The pin values for the defined instructions must abide by the following rules:
 - You must use the same order as defined in the “Control Input:” and “Data Input:” sections.
 - You can use values 0 (logic 0), 1 (logic 1), X (unknown), Z (high impedance), N (new binary value, 0 or 1, allowed), and H (hold previous value) in the pin value definitions.
 - You cannot use N or Z values for control pin values.
 - You cannot use H in the first test cycle.
- You define the time of the output strobe by placing the keyword “STROBE” after the pin definitions for the test cycle at the end of which the strobe occurs.
- You use “/” as the last character of a line to break long lines.
- You place comments after a “//” at any place within a line.
- All characters in the file, including keywords, are case insensitive.

During test generation, FlexTest determines the pin values most appropriate to achieve high test coverage. It does so for each pin that is not a control pin, or a constrained data pin, given the information you define in the instruction file.

[Figure 1-34](#) shows an example instruction file for the ADD instruction defined in [Table 1-2](#) on page 80, as well as a subtraction (SUB) and multiplication (MULT) instruction.

Figure 1-34. Example Instruction File

```
Control input:
Ctrl1
Ctrl2
Ctrl3
Ctrl4
Data Input:
Data1
Data2
Data3
Data4
Data5
Data6
Instruction: ADD
1010NNNNNN //start of 3 test cycle ADD Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE//strobe after last test cycle
Instruction: SUB
1101NNNNNN //start of 3 test cycle SUB Instruction
HHHHHHHHHH
HHHHHHHHHH
STROBE //strobe after last test cycle
Instruction: MULT
1110NNNNNN //start of 6 test cycle MULT Instruction
HHHHHHHHHH
1001NNNNNN //next part of MULT Instruction
HHHHHHHHHH
0101HHHHHH //last part of MULT, hold values
STROBE//strobe after 5th test cycle
HHHHHHHHHH
```

This instruction file defines four control pins, six data pins, and three instructions: ADD, SUB, and MULT. The ADD and SUB instructions each require three test cycles and strobe the outputs following the third test cycle. The MULT instruction requires six test cycles and strobes the outputs following the fifth test cycle. During the first test cycle, the ADD instruction requires the values 1010 on pins Ctrl1, Ctrl2, Ctrl3, Ctrl4, and allows FlexTest to place new values on any of the data pins. The ADD instruction then requires that all pins hold their values for the remaining two test cycles. The resulting pattern set, if saved in ASCII format, contains comments specifying the cycles for testing the individual instructions.

Verifying Test Patterns

After testing the functionality of the circuit with a simulator, and generating the test vectors with FlexTest, you should run the test vectors in a timing-based simulator and compare the results with predicted behavior from the ATPG tools. This run will point out any functionality discrepancies between the two tools, and also show timing differences that may cause different results. The following subsections further discuss the verification you should perform.


Simulating the Design With Timing. 83

Simulating the Design With Timing

At this point in the design process, you should run a full timing verification to ensure a match between the results of golden simulation and ATPG. This verification is especially crucial for designs containing asynchronous circuitry.

You should have already saved the generated test patterns with the [Save Patterns](#) command in FlexTest. The tool saved the patterns in parallel unless you used the -Serial switch to save the patterns in series. You can reduce the size of a serial pattern file by using the -Sample switch; the tool then saves samples of patterns for each pattern type, rather than the entire pattern set (except MacroTest patterns, which are not sampled nor included in the sampled pattern file). This is useful when you are simulating serial patterns because the size of the sampled pattern file is reduced and thus, the time it takes to simulate the sampled patterns is also reduced.

Note

 Using the -Start and -End switches will limit file size as well, but the portion of internal patterns saved will not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns will more closely approximate the results you would obtain from the entire pattern set.

For example, assume you saved the patterns generated in Flextest as follows:

```
ATPG> save patterns pat_parallel.v -verilog -replace
```

The tool writes the test patterns out in one or more pattern files and an enhanced Verilog test bench file that instantiates the top level of the design. These files contain procedures to apply the test patterns and compare expected output with simulated output.

Be sure to simulate parallel patterns and at least a few serial patterns. Parallel patterns simulate relatively quickly, but do not detect problems that occur when data is shifted through the scan chains. One such problem, for example, is data shifting through two cells on one clock cycle due to clock skew. Serial patterns can detect such problems. Another reason to simulate a few serial patterns is that correct loading of shadow or copy cells depends on shift activity. Because parallel patterns lack the requisite shift activity to load shadow cells correctly, you may get simulation mismatches with parallel patterns that disappear when you use serial patterns.

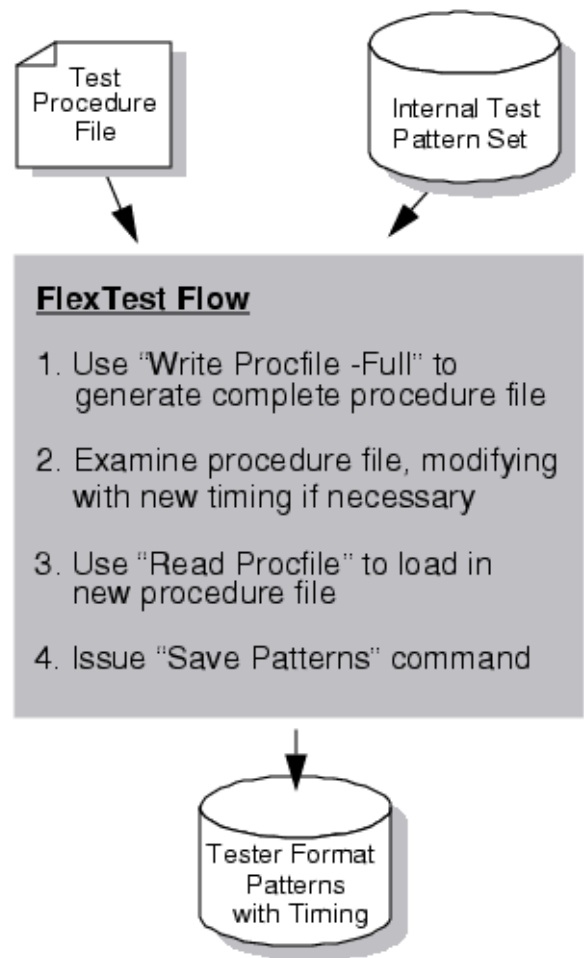
Therefore, always simulate at least the chain test or a few serial patterns in addition to the parallel patterns.

For a detailed description of the differences between serial and parallel patterns, refer to the first two subsections under “[Pattern Formatting Issues](#)”. See also “[Sampling to Reduce Serial Loading Simulation Time](#)” for information on creating a subset of sampled serial patterns. Serial patterns take much longer to simulate than parallel patterns (due to the time required to serially load and unload the scan chains), so typically only a subset of serial patterns is simulated.

Test Pattern Formatting and Timing

The following figure shows a basic process flow for defining test pattern timing.

Figure 1-35. Defining Basic Timing Process Flow



Test Pattern Timing Overview	86
Timing Terminology	86
General Timing Issues.....	86
Generating a Procedure File.....	87
Defining and Modifying Timeplates.....	88
Saving Timing Patterns.....	91
Pattern Formatting Issues.....	92
Saving in ASIC Vendor Data Formats	99

Test Pattern Timing Overview

Test procedure files contain both scan and non-scan procedures. All timing for all pattern information, both scan and non-scan, is defined in this procedure file.

While the ATPG process itself does not require test procedure files to contain real timing information, automatic test equipment (ATE) and some simulators do require this information. Therefore, you must modify the test procedure files you use for ATPG to include real timing information. “[General Timing Issues](#)” on page 86 discusses how you add timing information to existing test procedures.

After creating real timing for the test procedures, you are ready to save the patterns. You use the Save Patterns command with the proper format to create a test pattern set with timing information. For more information, refer to “[Saving Timing Patterns](#)”.

Test procedures contain groups of statements that define scan-related events.

Timing Terminology

The following list defines some timing-related terms.

- **Non-return Timing** — Primary inputs that change, at most, once during a test cycle.
- **Offset** — The timeframe in a test cycle in which pin values change.
- **Period** — The duration of pin timing—one or more test cycles.
- **Return Timing** — Primary inputs, typically clocks, that pulse high or low during every test cycle. Return timing indicates that the pin starts at one logic level, changes, and returns to the original logic level before the cycle ends.
- **Suppressible Return Timing** — Primary inputs that can exhibit return timing during a test cycle, although not necessarily.

General Timing Issues

ATEs require test data in a cycle-based format. The patterns you apply to such equipment must specify the waveforms of each input, output, or bidirectional pin, for each test cycle.

Within a test cycle, a device under test must abide by the following restrictions:

- At most, each non-clock input pin changes once in a test cycle. However, different input pins can change at different times.
- Each clock input pin is at its off-state at both the start and end of a test cycle.
- At most, each clock input pin changes twice in a test cycle. However, different clock pins can change at different times.

- Each output pin has only one expected value during a test cycle. However, the equipment can measure different output pin values at different times.
- A bidirectional pin acts as either an input or an output, but not both, during a single test cycle.

To avoid adverse timing problems, the following timing requirements satisfy some ATE timing constraints:

- **Unused Outputs** — By default, test procedures without measure events (all procedures except **shift**) strobe unused outputs at a time of $\text{cycle}/2$, and end the strobe at $3*\text{cycle}/4$. The **shift** procedure strobcs unused outputs at the same time as the scan output pin.
- **Unused Inputs** — By default, all unused input pins in a test procedure have a force offset of 0.
- **Unused Clock Pins** — By default, unused clock pins in a test procedure have an offset of $\text{cycle}/4$ and a width of $\text{cycle}/2$, where cycle is the duration of each cycle in the test procedure.
- **Pattern Loading and Unloading** — During the **load_unload** procedure, when one pattern loads, the result from the previous pattern unloads. When the tool loads the first pattern, the unload values are X. After the tool loads the last pattern, it loads a pattern of X's so it can simultaneously unload the values resulting from the final pattern.

Generating a Procedure File

Use the basic process flow for defining test pattern timing.

1. Use the [Write Procfile](#) -Full command and switch to generate a complete procedure file.
2. Examine the procedure file, modify timeplates with new timing if necessary.
3. Use the [Read Procfile](#) command to load in the revised procedure file.
4. Issue the [Save Patterns](#) command.

There are three ways to load existing procedure file information into FlexTest:

- During SETUP mode, use the [Add Scan Groups](#) *procedure_filename* command. Any timing information in these procedure files will be used when [Save Patterns](#) is issued if no other timing information or procedure information is loaded.
- Use the [Read Procfile](#) command. This is only valid when not in SETUP mode. Using this command loads a new procedure file that will overwrite or merge with the procedure and timing data already loaded. This new data is now in effect for all subsequent [Save Patterns](#) commands.

- If you specify a new procedure file on the [Save Patterns](#) command line, the timing information in that procedure file will be used for that Save Patterns command only, and then the previous information will be restored.

Defining and Modifying Timeplates

This section gives an overview of the test procedure file timeplate syntax, to facilitate Step 2 in the process flow listed previously.

After you have used the [Write Procfile](#) -Full command and switch to generate a procedure file, you can examine the procedure file, modifying timeplates with new timing if necessary. Any timing changes to the existing TimePlates, cannot change the event order of the timeplate used for scan procedures. The times may change, but the event order must be maintained.

In the following example, there are two events happening at time 20, and both are listed as event 4. These may be skewed, but they may not interfere with any other event. The events must stay in the order listed in the comments:

```
force_pi          0; // event 1
bidi_force_pi     12; // event 3
measure_po        31; // event 7
bidi_measure_po   32; // event 8
force_InPin       9; // event 2
measure_OutPin    35; // event 9
pulse_Clk1        20 5; // event 4 & 5 respectively
pulse_Clk2        20 10; // event 4 & 6 respectively
period 50;        // no events but all events
                  // have to happen in period
```

Test procedure files have the following format:


```
[set_statement ...]
[alias_definition]
timeplate_definition [timeplate_definition]
procedure_definition [procedure_definition]
```

The timeplate definition describes a single tester cycle and specifies where in that cycle all event edges are placed. You must define all timeplates before they are referenced. A procedure file must have at least one timeplate definition. The timeplate definition has the following format:

```
timeplate timeplate_name =
    timeplate_statement
    [timeplate_statement ...]
    period time;
end;
```

The following list contains available timeplate_statement statements. The timeplate definition should contain at least the force_pi and measure_po statements.


Note

 You are not required to include pulse statements for the clocks. But if you do not “pulse” a clock, the STI code uses two cycles to pulse it, resulting in larger patterns.

```
timeplate_statement:
  offstate pin_name off_state;
  force_pi time;
  bidi_force_pi time;
  measure_po time;
  bidi_measure_po time;
  force pin_name time;
  measure pin_name time;
  pulse pin_name time width;
```


- ***timeplate_name*** — A string that specifies the name of the timeplate.
- ***offstate pin_name off_state*** — A literal and double string that specifies the inactive, off-state value (0 or 1) for a specific named pin that is not defined as a clock pin by the [Add Clocks](#) command. This statement must occur before all other timeplate_statement statements. This statement is only needed for a pin that is not defined as a clock pin by the Add Clocks command but will be pulsed within this timeplate.
- ***force_pitime*** — A literal and string pair that specifies the force time for all primary inputs.
- ***bidi_force_pitime*** — A literal and string pair that specifies the force time for all bidirectional pins. This statement allows the bidirectional pins to be forced after applying the tri-state control signal, so the system avoids bus contention. This statement overrides “force_pi” and “measure_po”.
- ***measure_potime*** — A literal and string pair that specifies the time at which the tool measures (or strobos) the primary outputs.
- ***bidi_measure_potime*** — A literal and string pair that specifies the time at which the tool measures (or strobos) the bidirectional pins. This statement overrides “force_pi” and “measure_po”.
- ***force pin_name time*** — A literal and double string that specifies the force time for a specific named pin.

Note

 This force time overrides the force time specified in force_pi for this specific pin.

- ***measure pin_name time*** — A literal and double string that specifies the measure time for a specific named pin.

Note

 This measure time overrides the measure time specified in measure_po for this specific pin.

- **pulsepin_name time width** — A literal and triple string that specifies the pulse timing for a specific named clock pin. The time value specifies the leading edge of the clock pulse and the width value specifies the width of the clock pulse. This statement can only reference pins that have been declared as clocks by the [Add Clocks](#) command or pins that have an offstate specified by the “offstate” statement. The sum of the time and width must be less than the period.
- **period time** — A literal and string pair that defines the period of a tester cycle. This statement ensures that the cycle contains sufficient time, after the last force event, for the circuit to stabilize. The time you specify should be greater than or equal to the final event time.

Example 1

```
timeplate tp1 =  
  force_pi 0;  
  pulse T 30 30;  
  pulse R 30 30;  
  measure_po 90;  
  period 100;  
end;
```

Example 2

The following example shows a shift procedure that pulses b_clk with an off-state value of 0. The timeplate tp_shift defines the off-state for pin b_clk. The b_clk pin is not declared as a clock in the ATPG tool.

```
timeplate tp_shift =  
  offstate b_clk 0;  
  force_pi 0;  
  measure_po 10;  
  pulse clk 50 30;  
  pulse b_clk 140 50;  
  period 200;  
end;
```

```
procedure shift =  
  timeplate tp_shift;  
  cycle =  
    force_sci;  
    measure_sco;  
    pulse clk;  
    pulse b_clk;  
  end;  
end;
```

Saving Timing Patterns

You can save patterns generated during the ATPG process both for timing simulation and use on the ATE.

After you create the proper timing information in a test procedure file, FlexTest uses an internal test pattern data formatter to generate the patterns in the following formats:

- FlexTest text format (ASCII)
- Wave Generation Language (WGL)
- Standard Test Interface Language (STIL)
- Verilog
- Texas Instruments Test Description Language (TDL 91)
- Fujitsu Test data Description Language (FTDL-E)
- Mitsubishi Test Description Language (MITDL)
- Toshiba Standard Tester interface Language 2 (TSTL2)

Features of the Formatter

The main features of the test pattern data formatter include:

- Generating basic test pattern data formats. FlexTest Text, Verilog, and WGL (ASCII and binary).
- Generating ASIC Vendor test data formats: TDL 91, FTDL-E, MITDL, and TSTL2.
- Supporting parallel load of scan cells (in Verilog format).
- Reading in external input patterns and output responses, and directly translating to one of the formats.
- Reading in external input patterns, performing good or faulty machine simulation to generate output responses, and then translating to any of the formats.
- Writing out just a subset of patterns in any test data format. Facilitating failure analysis by having the test data files cross-reference information between tester cycle numbers and FlexTest pattern numbers.
- Supporting differential scan input pins for each simulation data format.

Pattern Formatting Issues

The following subsections describe issues you should understand regarding the test pattern formatter and pattern saving process.

Serial Versus Parallel Scan Chain Loading	92
Parallel Scan Chain Loading	92
FlexTest Text	94
Comparing FlexTest Text Formats With Other Test Data Formats	95
Verilog.....	97
Wave Generation Language (ASCII).....	97
Standard Test Interface Language (STIL)	98

Serial Versus Parallel Scan Chain Loading

When you simulate test patterns, most of the time is spent loading and unloading the scan chains, as opposed to actually simulating the circuit response to a test pattern. You can use either serial or parallel loading, and each affects the total simulation time differently.

The primary advantage of simulating serial loading is that it emulates how patterns are loaded on the tester. You thus obtain a very realistic indication of circuit operation. The disadvantage is that for each pattern, you must clock the scan chain registers at least as many times as you have scan cells in the longest chain. For large designs, simulating serial loading takes an extremely long time to process a full set of patterns.

The primary advantage of simulating parallel loading of the scan chains is it greatly reduces simulation time compared to serial loading. You can directly (in parallel) load the simulation model with the necessary test pattern values because you have access, in the simulator, to internal nodes in the design. Parallel loading makes it practical for you to perform timing simulations for the entire pattern set in a reasonable time using popular simulators like Questa SIM that utilize the Verilog format.

Parallel Scan Chain Loading


You accomplish parallel loading through the scan input and scan output pins of scan sub-chains (a chain of one or more scan cells, modeled as a single library model) because these pins are unique to both the timing simulator model and the FlexTest internal models. For example, you can parallel load the scan chain by using Verilog force statements to change the value of the scan input pin of each sub-chain.

After the parallel load, you apply the **shift** procedure a few times (depending on the number of scan cells in the longest subchain, but usually only once) to load the scan-in value into the sub-chains. Simulating the **shift** procedure only a few times can dramatically improve timing

simulation performance. You can then observe the scan-out value at the scan output pin of each sub-chain.

Parallel loading ensures that all memory elements in the scan sub-chains achieve the same states as when serially loaded. Also, this technique is independent of the scan design style or type of scan cells the design uses. Moreover, when writing patterns using parallel loading, you do not have to specify the mapping of the memory elements in a sub-chain between the timing simulator and FlexTest. This method does not constrain library model development for scan cells.

Note

 When your design contains at least one stable-high scan cell, the **shift** procedure period must exceed the shift clock off time. If the **shift** procedure period is less than or equal to the shift clock off time, you may encounter timing violations during simulation. The test pattern formatter checks for this condition and issues an appropriate error message when it encounters a violation.

For example, the test pattern timing checker would issue an error message when reading in the following **shift** procedure and its corresponding timeplate:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 300; // Period same as shift clock off time
end;

procedure shift =
    scan_group grp1;
    timeplate gen_tp1;
    cycle =
        force_sci;
        measure_sco;
        pulse CLK; // Force shift clock on and off
    end;
end;
```

The error message would state:

```
// Error: There is at least one stable high scan cell in the design. The
shift procedure period must be greater than the shift clock off time to
avoid simulation timing violations.
```

The following modified timeplate would pass timing rules checks:

```
timeplate gen_tp1 =
    force_pi 0;
    measure_po 100;
    pulse CLK 200 100;
    period 400; // Period greater than shift clock off time
end;
```

Sampling to Reduce Serial Loading Simulation Time

When you use the [Save Patterns](#) command, you can specify to save a sample of the full pattern set by using the `-Sample` switch. This reduces the number of patterns in the pattern file(s), reducing simulation time accordingly. In addition, the `-Sample` switch allows you to control how many patterns of each type are included in the sample. By varying the number of sample patterns, you can fine-tune the trade-off between file size and simulation time for serial patterns.

Note



Using the `-Start` and `-End` switches limits file size as well, but the portion of internal patterns saved does not provide a very reliable indication of pattern characteristics when simulated. Sampled patterns more closely approximate the results you would obtain from the entire pattern set.

After performing initial verification with parallel loading, you can use a sampled pattern set for simulating series loading until you are satisfied test coverage is reasonably close to desired specification. Then, perform a series loading simulation with the unsampled pattern set only once, as your last verification step.

Test Pattern Data Support for IDDQ

For best results, you should measure current after each non-scan cycle if doing so catches additional IDDQ faults. However, you can only measure current at specific places in the test pattern sequence, typically at the end of the test cycle boundary. To identify when IDDQ current measurement can occur, FlexTest pattern files add the following command at the appropriate places:

```
measure IDDQ ALL;
```

Several test pattern data formats support IDDQ testing. There are special IDDQ measurement constructs in TDL 91 (Texas Instruments), MITDL (Mitsubishi), TSTL2 (Toshiba), and FTDL-E (Fujitsu). The tools add these constructs to the test data files. All other formats (WGL and Verilog) represent these statements as comments.

Saving Patterns in Basic Test Data Formats

The [Save Patterns](#) command saves the patterns in the basic test data formats including FlexTest text, Verilog, and WGL (ASCII and binary). The test pattern formatter can write any of these formats as part of the standard FlexTest package—you do not have to buy a separate option. You can use these formats for timing simulation.

FlexTest Text

This is the default format that FlexTest generates when you run the `Save Patterns` command. This is one of only two formats (the other being FlexTest table format) that FlexTest can read back in, so you should always generate a pattern file in this format to save intermediate results.

This format contains test pattern data in a text-based parallel format, along with cycle boundary specifications. The main pattern block calls the appropriate test procedures, while the header contains test coverage statistics and the necessary environment variable settings. This format also contains each of the scan test procedures, as well as information about each scan memory element in the design.

To create a FlexTest text format file, enter the following at the application command line:

ATPG> save patterns *filename* -ascii

The formatter writes the complete test data to the file named *filename*.

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Note

This pattern format does not contain explicit timing information. For more information on this test pattern format, refer to “[Test Pattern File Formats](#)”.

Comparing FlexTest Text Formats With Other Test Data Formats

The FlexTest text formats describe the contents of the test set in a human readable form. In many cases, you may find it useful to compare the contents of a simulation or test data format with that of the text format for debugging purposes. This section provides detailed information necessary for this task.

Often, the first cycle in a test set must perform certain tasks. The first test cycle in all test data formats turns off the clocks at all clock pins, drives Z on all bidirectional pins, drives an X on all other input pins, and disables measurement at any primary output pins.

The FlexTest test pattern sets can contain two main parts: the chain test block, to detect faults in the scan chain, and the scan test or cycle test block, to detect other system faults.

The Chain Test Block

The chain test applies the **test_setup** procedure, followed by the **load_unload** procedure for loading scan chains, and the **load_unload** procedure again for unloading scan chains. Each **load_unload** procedure in turn calls the **shift** procedure. This operation typically loads a repeating pattern of “0011” into the chains. However, if scan chains with less than four cells exist, then the operation loads and unloads a repeating “01” pattern followed by a repeating “10” pattern. Also, when multiple scan chains in a group share a common scan input pin, the chain test process separately loads and unloads each of the scan chains with the repeating pattern to test them in sequence.

The test procedure file applies each event in a test procedure at the specified time. Each test procedure corresponds to one or more test cycles. Each test procedure can have a test cycle with a different timing definition. By default, all events use a timescale of 1 ns.

The Cycle Test Block

The cycle test block in the FlexTest pattern set also starts with an application of the **test_setup** procedure. This test pattern set consists of a sequence of scan operations and test cycles. The number of test cycles between scan operations can vary within the same test pattern set. A FlexTest pattern can be just a scan operation along with the subsequent test cycle, or a test cycle without a preceding scan operation. The scan operations use the **load_unload** procedure and the **master_observe** procedure for LSSD designs. The **load_unload** procedure translates to one or more test cycles.

Using FlexTest, you can completely define the number of timeframes and the sequence of events in each test cycle. Each timeframe in a test cycle has a force event and a measure event. Therefore, each event in a test cycle has a sequence number associated with it. The sequence number's default time scale is 1 ns.

Unloading of the scan chains for the current pattern occurs concurrently with the loading of scan chains for the next pattern. For designs with sequential controllers, like boundary scan designs, each test procedure may contain several test cycles that operate the sequential scan controller.

General Considerations

During a test procedure, you may leave many pins unspecified. Unspecified primary input pins retain their previous state. FlexTest does not measure unspecified primary output pins, nor does it drive (drive Z) or measure unspecified bidirectional pins. This prevents bus contention at bidirectional pins.

Note




If you run ATPG after setting pin constraints, you should also ensure that you set these pins to their constrained states at the end of the **test_setup** procedure. The Add Pin Constraints command constrains pins for the non-scan cycles, not the test procedures. If you do not properly constrain the pins within the **test_setup** procedure, the tool does it for you, internally adding the extra force events after the **test_setup** procedure. This increases the period of the **test_setup** procedure by one time unit. This increased period can conflict with the test cycle period, potentially forcing you to re-run ATPG with the modified test procedure file.

All test data formats contain comment lines that indicate the beginning of each test block and each test pattern. You can use these comments to correlate the test data in the FlexTest text format with other test data formats.

These comment lines also contain the cycle count and the loop count, which help correlate tester pattern data with the original test pattern data. The cycle count represents the number of test cycles, with the shift sequence counted as one cycle. The loop count represents the number of

all test cycles, including the shift cycles. The cycle count is useful if the tester has a separate memory buffer for scan patterns, otherwise the loop count is more relevant.

Note

 The cycle count and loop count contain information for all test cycles—including the test cycles corresponding to test procedures. You can use this information to correlate tester failures to a FlexTest cycle for fault diagnosis.

Verilog

This format contains test pattern data and timing information in a text-based format readable by both the Verilog and Verifault simulators. This format also supports both serial and parallel loading of scan cells. The Verilog format supports all FlexTest timing definitions, because Verilog stimulus is a sequence of timed events.

To generate a basic Verilog format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* [-Parallel | -Serial] -Verilog

The Verilog pattern file contains procedures to apply the test patterns, compare expected output with simulated output, and print out a report containing information about failing comparisons. The tools write all patterns and comparison functions into one main file (*filename*), while writing the primary output names in another file (*filename.po.name*). If you choose parallel loading, they also write the names of the scan output pins of each scan sub-chain of each scan chain in separate files (for example, *filename.chain1.name*). This allows the tools to report output pins that have discrepancies between the expected and simulated outputs. You can enhance the Verilog testbench with Standard Delay Format (SDF) back annotation.

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Wave Generation Language (ASCII)

The Wave Generation Language (WGL) format contains test pattern data and timing information in a structured text-based format. You can translate this format into a variety of simulation and tester environments, but you must first read it into the Waveform database and use the appropriate translator. This format supports both serial and parallel loading of scan cells.

Some test data flows verify patterns by translating WGL to stimulus and response files for use by the chip foundry's golden simulator. Sometimes this translation process uses its own parallel loading scheme, called memory-to-memory mapping, for scan simulation. In this scheme, each scan memory element in the ATPG model must have the same name as the corresponding

memory element in the simulation model. Due to the limitations of this parallel loading scheme, you should ensure the following:

1) there is only one scan cell for each DFT library model (also called a scan subchain), 2) the hierarchical scan cell names in the netlist and DFT library match those of the golden simulator (because the scan cell names in the ATPG model appear in the scan section of the parallel WGL output), and 3) the scan-in and scan-out pin names of all scan cells are the same.

To generate a basic WGL format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* [-Parallel | -Serial] -Wgl

For more information on the Save Patterns command and its options, see [Save Patterns](#).

For more information on the WGL format, contact Integrated Measurement Systems, Inc.

Standard Test Interface Language (STIL)

To generate a STIL format test pattern file, use the following arguments with the Save Patterns command.

SAVe PATterns *filename* [-Parallel | -Serial] -STIL

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Saving in ASIC Vendor Data Formats

The ASIC vendor test data formats include Texas Instruments TDL 91, Fujitsu FTDL-E, Mitsubishi MITDL, and Toshiba TSTL2. The ASIC vendor's chip testers use these formats.

All the ASIC vendor data formats are text-based and load data into scan cells in a parallel manner. Also, ASIC vendors usually impose several restrictions on pattern timing. Most ASIC vendor pattern formats support only a single timing definition. Refer to your ASIC vendor for test pattern formatting and other requirements.

The following subsections briefly describe the ASIC vendor pattern formats.

TI TDL 91	99
Fujitsu FTDL-E	99
Mitsubishi TDL	100
Toshiba TSTL2	100

TI TDL 91

This format contains test pattern data in a text-based format.

FlexTest supports features of TDL 91 version 3.0 and of TDL 91 version 6.0. The version 3.0 format supports multiple scan chains, but allows only a single timing definition for all test cycles. Thus, all test cycles must use the timing of the main capture cycle. TI's ASIC division imposes the additional restriction that comparison should always be done at the end of a tester cycle.

To generate a basic TI TDL 91 format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* -TItDl

The formatter writes the complete test data to the file *filename*. It also writes the chain test to another file (*filename.chain*) for separate use during the TI ASIC flow.

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Fujitsu FTDL-E

This format contains test pattern data in a text-based format. The FTDL-E format splits test data into patterns that measure 1 or 0 values, and patterns that measure Z values. The test patterns divide into test blocks that each contain 64K tester cycles.

To generate a basic FTDL-E format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* -Fjtdl

The formatter writes the complete test data to the file named *filename.fjtdl.func*. If the test pattern set contains IDDQ measurements, the formatter creates a separate DC parametric test block in a file named *filename.ftjtl.dc*.

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Mitsubishi TDL

This format contains test pattern data in a text-based format.

To generate a basic Mitsubishi Test Description Language (TDL) format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* -Mitdl

The formatter represents all scan data in a parallel format. It writes the test data into two files: the program file (*filename.td0*), which contains all pin definitions, timing definitions, and scan chain definitions; and the test data file (*filename.td1*), which contains the actual test vector data in a parallel format.

For more information on the Save Patterns command and its options, see [Save Patterns](#).

Toshiba TSTL2

This format contains only test pattern data in a text-based format. The test pattern data files contain timing information. This format supports multiple scan chains, but allows only a single timing definition for all test cycles. TSTL2 represents all scan data in a parallel format.

To generate a basic Toshiba TSTL2 format test pattern file, use the following arguments with the Save Patterns command:

SAVe PATterns *filename* -TSTl2

The formatter writes the complete test data to the file named *filename*. For more information on the Save Patterns command and its options, see [Save Patterns](#).

Chapter 2

FlexTest Command Dictionary

FlexTest is a high performance sequential ATPG system that allows you to create a set of test patterns that achieve a high, accurately measured test coverage for your cycle-based circuits.

FlexTest-specific features include the following:

- Supports a wide range of DFT structures.
- Can display a wide variety of useful information—from design and debugging information to statistical reports for the generated test set.

Inputs and Outputs	106
Command Line Syntax Conventions	106
Command Summary	107
Command Descriptions	117
Abort Interrupted Process	123
Add Atpg Constraints	124
Add Atpg Functions	128
Add Black Box	132
Add Cell Constraints	135
Add Clocks	138
Add Cone Blocks	140
Add Contention Free_bus	142
Add Faults	143
Add Initial States	146
Add Lists	147
Add Nofaults	149
Add Nonscan Handling	152
Add Output Masks	154
Add Pin Constraints	155
Add Pin Equivalences	158
Add Pin Strokes	159
Add Primary Inputs	160
Add Primary Outputs	162
Add Read Controls	163
Add Scan Chains	164
Add Scan Groups	165
Add Scan Instances	166
Add Scan Models	167
Add Tied Signals	168

Add Write Controls	170
Alias	171
Analyze Atpg Constraints	174
Analyze Bus	176
Analyze Contention	178
Analyze Control Signals	180
Analyze Fault	183
Analyze Race	186
Compress Patterns	188
Delete Atpg Constraints	190
Delete Atpg Functions	192
Delete Black Box	194
Delete Cell Constraints	195
Delete Clocks	196
Delete Cone Blocks	197
Delete Contention Free_bus	198
Delete Faults	199
Delete Initial States	201
Delete Lists	202
Delete Nofaults	203
Delete Nonscan Handling	206
Delete Output Masks	208
Delete Pin Constraints	209
Delete Pin Equivalences	210
Delete Pin Strokes	211
Delete Primary Inputs	212
Delete Primary Outputs	214
Delete Read Controls	216
Delete Scan Chains	217
Delete Scan Groups	218
Delete Scan Instances	219
Delete Scan Models	220
Delete Tied Signals	221
Delete Write Controls	223
Dofile	224
Exit	226
Find Design Names	227
Flatten Model	232
Help	233
History	234
Load Faults	236
Read Modelfile	238
Read Procfile	241
Report Aborted Faults	246
Report Atpg Constraints	248

Report Atpg Functions	249
Report Au Faults	250
Report Black Box	253
Report Bus Data	255
Report Cell Constraints	257
Report Clocks	259
Report Cone Blocks	260
Report Contention Free_bus	261
Report Core Memory	262
Report Drc Rules	263
Report Environment	267
Report Faults	269
Report Feedback Paths	272
Report Flatten Rules	274
Report Gates	276
Report Initial States	289
Report Lists	290
Report Loops	291
Report Nofaults	293
Report Nonscan Cells	295
Report Nonscan Handling	297
Report Output Masks	298
Report Pin Constraints	299
Report Pin Equivalences	300
Report Pin Strokes	301
Report Primary Inputs	302
Report Primary Outputs	304
Report Procedure	306
Report Pulse Generators	307
Report Read Controls	308
Report Scan Cells	309
Report Scan Chains	311
Report Scan Groups	313
Report Scan Instances	314
Report Scan Models	315
Report Statistics	316
Report Test Stimulus	320
Report Testability Data	323
Report Tied Signals	325
Report Timeplate	326
Report Version Data	327
Report Write Controls	328
Reset Au Faults	329
Reset State	330
Resume Interrupted Process	331

Run	332
Save History	335
Save Patterns	336
Select Iddq Patterns	342
Set Abort Limit	346
Set Atpg Limits	348
Set Atpg Window	350
Set Bus Handling	352
Set Capture Clock	355
Set Capture Limit	357
Set Checkpoint	359
Set Clock Restriction	360
Set Contention Check	361
Set Contention_bus Reporting	364
Set Display	366
Set Dofile Abort	367
Set Drc Handling	368
Set Driver Restriction	373
Set Fails Report	375
Set Fault Dropping	376
Set Fault Mode	379
Set Fault Sampling	381
Set Fault Type	382
Set File Compression	384
Set Flatten Handling	386
Set Gate Level	388
Set Gate Report	389
Set Gzip Options	394
Set Hypertrophic Limit	396
Set Iddq Checks	397
Set Iddq Strobe	401
Set Instruction Atpg	403
Set Internal Fault	404
Set Internal Name	405
Set Interrupt Handling	406
Set Learn Report	408
Set List File	410
Set Logfile Handling	411
Set Loop Handling	413
Set Net Dominance	414
Set Net Resolution	416
Set Nonscan Model	417
Set Output Comparison	419
Set Output Masks	421
Set Pattern Source	422

Set Possible Credit	426
Set Procedure Cycle_checking	427
Set Pulse Generators	428
Set Race Data	429
Set Rail Strength	430
Set Random Atpg	431
Set Redundancy Identification	432
Set Screen Display	433
Set Self Initialization	434
Set Sensitization Checking	436
Set Sequential Learning	437
Set Shadow Check	438
Set Static Learning	439
Set Stg Extraction	441
Set System Mode	442
Set Test Cycle	444
Set Trace Report	445
Set Transient Detection	446
Set Unused Net	448
Set Z Handling	449
Setenv	451
Setup Checkpoint	452
Setup Pin Constraints	454
Setup Pin Strokes	456
Setup Tied Signals	457
Step	458
System	459
Unsetenv	460
Update Implication Detections	461
Write Core Memory	462
Write Environment	463
Write Faults	465
Write Initial States	468
Write Loops	469
Write Modelfile	470
Write Netlist	471
Write Primary Inputs	473
Write Primary Outputs	475
Write Procfile	477
Write Statistics	479
Shell Command Description	482
flextest	483

Inputs and Outputs

FlexTest utilizes the following inputs.

- **Design** — The supported netlist format is gate-level Verilog.
- **Test Procedure File** — This file defines the operation of the scan circuitry in your design.
- **Library** — This file contains model descriptions for all library cells used in your design.
- **Fault List** — This is an external fault list that you can use as a source of faults for the internal fault list of FlexTest.
- **Test Patterns** — This is a set of externally-generated test patterns that you can use as the pattern source for simulation.

FlexTest produces the following outputs:

- **Test Patterns** — This file set contains test patterns in one or more of the supported simulator or ASIC vendor pattern formats. For more information on the available test pattern formats, refer to the [Save Patterns](#) command reference page within this manual.
- **ATPG Information Files** — These files contain session information that you can save using various commands.
- **Fault List** — This is an ASCII file that contains internal fault information in a proprietary Siemens EDA fault format.

Command Line Syntax Conventions

This manual uses the following command usage line syntax conventions.

Table 2-1. Conventions for Command Line Syntax

Convention	Example	Usage
UPPerCase	REPort ENvironment	Required command letters are in uppercase; in most cases, you may omit lowercase letters when entering commands or literal arguments and you need not enter in uppercase. Command names and options are normally case insensitive. Commands usually follow the 3-2-1 rule: the first three letters of the first word, the first two letters of the second word, and the first letter of the third, fourth, etc. words.
Boldface	ADD INitial States { 0 1 X }	A boldface font indicates a required argument.

Table 2-1. Conventions for Command Line Syntax (cont.)

Convention	Example	Usage
[]	EXIt [-Force]	Square brackets enclose optional arguments. Do not enter the brackets.
<i>Italic</i>	DOFile <i>filename</i>	An italic font indicates a user-supplied argument.
{ }	ADD AMbiguous Paths { <i>path_name</i> -All} [-Max_paths <i>number</i>]	Braces enclose arguments to show grouping. Do not enter the braces.
	ADD AMbiguous Paths { <i>path_name</i> -All} [-Max_paths <i>number</i>]	The vertical bar indicates an either/or choice between items. Do not include the bar in the command.
Underline	SET DOfile Abort <u>ON</u> OFF	An underlined item indicates either the default argument or the default value of an argument.
...	ADD CLocks <i>off_state</i> <i>primary_input_pin</i> ... [-Internal]	An ellipsis follows an argument that may appear more than once. Do not include the ellipsis when entering commands.

Command Summary

The following table contains a summary of the FlexTest commands described in this manual.

Table 2-2. Command Summary

Command	Description
Abort Interrupted Process	Aborts a command placed in suspended state by a Control-C interrupt while the Set Interrupt Handling command is on.
Add Atpg Constraints	Specifies that the tool restrict all patterns it places into the internal pattern set according to the user-defined constraints.
Add Atpg Functions	Creates an ATPG function that you can then use when generating user-defined ATPG constraints.
Add Black Box	Defines black boxes and sets the constrained value on output or bidirectional black box pins.
Add Cell Constraints	Constrains scan cells (also non-scan cells for DX and SX constraints) to a constant value.
Add Clocks	Adds clock primary inputs to the clock list.
Add Cone Blocks	Specifies the blockage points that you want the tool to use during the calculation of the clock and effect cones.
Add Contention Free_bus	Specifies which if any contention free buses are added.
Add Faults	Adds faults to the current fault list.

Table 2-2. Command Summary (cont.)

Command	Description
Add Initial States	Specifies an initial state for the selected sequential instance.
Add Lists	Adds pins to the list of pins on which to report.
Add Nofaults	Places nofault settings either on pin pathnames, pin names of specified instances, or modules.
Add Nonscan Handling	Overrides behavior classification of non-scan elements that FlexTest learns during the design rules checking process.
Add Output Masks	Ignores any fault effects that propagate to the primary output pins you name.
Add Pin Constraints	Adds a pin constraint to a primary input pin.
Add Pin Equivalences	Adds restrictions to primary inputs so that they have equal or inverted values.
Add Pin Strobes	Adds strobe time to the primary outputs.
Add Primary Inputs	Adds primary inputs.
Add Primary Outputs	Adds primary outputs.
Add Read Controls	Defines a PI as a read control and specifies its off value.
Add Scan Chains	Adds a scan chain to a scan group.
Add Scan Groups	Adds a scan chain group to the system.
Add Scan Instances	Adds sequential instances to the scan instance list.
Add Scan Models	Adds sequential models to the scan model list.
Add Tied Signals	Adds a value to floating signals or pins.
Add Write Controls	Defines a PI as a write control and specifies its off value.
Alias	Specifies the shorthand name for a tool command, UNIX command, or existing command alias, or any combination of the three.
Analyze Atpg Constraints	Specifies for the tool to check the ATPG constraints you've created for their satisfiability or for their mutual exclusivity.
Analyze Bus	Causes the tool to analyze the specified bus gates for contention problems.
Analyze Contention	Specifies for the tool to analyze contention buses and ports.
Analyze Control Signals	Identifies and optionally defines the primary inputs of control signals.
Analyze Fault	Performs an analysis to identify why a fault is not detected.
Analyze Race	Checks for race conditions between the clock and data signals.

Table 2-2. Command Summary (cont.)

Command	Description
Compress Patterns	Compresses patterns in the current test pattern set.
Delete Atpg Constraints	Removes the state restrictions from the specified objects.
Delete Atpg Functions	Removes the specified function definitions.
Delete Black Box	Undoes the effect of the Add Black Box command.
Delete Cell Constraints	Removes constraints placed on scan cells.
Delete Clocks	Removes primary input pins from the clock list.
Delete Cone Blocks	Removes the specified output pin names from the user-created list which the tool uses to calculate the clock and effect cones.
Delete Contention Free_bus	Specifies which if any contention free buses are deleted.
Delete Faults	Removes faults from the current fault list.
Delete Initial States	Removes the initial state settings for the specified instance names.
Delete Lists	Removes pins from the pin list the tool monitors and reports on during simulation.
Delete Nofaults	Removes the nofault settings from either the specified pin or instance/module pathnames.
Delete Nonscan Handling	Removes the overriding, learned behavior classification for the specified non-scan elements.
Delete Output Masks	Removes the masking of the specified primary output pins.
Delete Pin Constraints	Removes the pin constraints from the specified primary input pins.
Delete Pin Equivalences	Removes the pin equivalence specifications for the designated primary input pins.
Delete Pin Strokes	Removes the strobe time from the specified primary output pins.
Delete Primary Inputs	Removes the specified primary inputs from the current netlist.
Delete Primary Outputs	Removes the specified primary outputs from the current netlist.
Delete Read Controls	Removes the read control line definitions from the specified primary input pins.
Delete Scan Chains	Removes the specified scan chain definitions from the scan chain list.
Delete Scan Groups	Removes the specified scan chain group definitions from the scan chain group list.

Table 2-2. Command Summary (cont.)

Command	Description
Delete Scan Instances	Removes the specified sequential instances from the scan instance list.
Delete Scan Models	Removes the specified sequential models from the scan model list.
Delete Tied Signals	Removes the assigned (tied) value from the specified floating nets or pins.
Delete Write Controls	Removes the write control line definitions from the specified primary input pins.
Dofile	Executes the commands contained within the specified file.
Exit	Terminates the application tool program.
Find Design Names	Displays design object hierarchical names matched by an input regular expression.
Flatten Model	Creates a primitive gate simulation representation of the design.
Help	Displays the usage syntax and system mode for the specified command.
History	Displays a list of previously-executed commands.
Load Faults	Updates the current fault list with the faults contained in the specified fault file.
Read Modelfile	Initializes the specified RAM or ROM gate using the memory states contained in the named modelfile or deletes an earlier modelfile assignment.
Read Procfile	Reads the specified test procedure file.
Report Aborted Faults	Displays information on testable faults that remain undetected (UD) after the ATPG process.
Report Atpg Constraints	Displays all the current ATPG state restrictions and the pins on which they reside.
Report Atpg Functions	Displays all the current ATPG function definitions.
Report Au Faults	Displays information on ATPG untestable faults.
Report Black Box	Displays information on blackboxes and undefined models.
Report Bus Data	Displays the bus data information for either an individual bus gate or for the buses of a specific type.
Report Cell Constraints	Displays a list of the constrained cells.
Report Clocks	Displays a list of all the primary input pins currently in the clock list.

Table 2-2. Command Summary (cont.)

Command	Description
Report Cone Blocks	Displays the current, user-defined output pin pathnames that the tool uses to calculate the clock and effect cones.
Report Contention Free_bus	Reports contention free buses.
Report Core Memory	Displays the amount of memory FlexTest requires to avoid paging during the ATPG and simulation processes.
Report Drc Rules	Displays either a summary of DRC violations (fails) or violation occurrence message(s).
Report Environment	Displays the current value(s) of many frequently used “Set...” commands.
Report Faults	Displays fault information from the current fault list.
Report Feedback Paths	Displays a report of currently identified feedback paths.
Report Flatten Rules	Displays either a summary of all the flattening rule violations or the data for a specific violation.
Report Gates	Displays the netlist information and simulation results for the specified gates and the simulation results for the specified user-defined ATPG functions.
Report Initial States	Displays the initial state settings of the specified design instances.
Report Lists	Displays the list of pins whose values the tool will report during simulation.
Report Loops	Displays information about circuit loops.
Report Nofaults	Displays the nofault settings for the specified pin pathnames or pin names of instances.
Report Nonscan Cells	Displays the non-scan cells whose model type you specify.
Report Output Masks	Displays a list of the currently masked primary output pins.
Report Pin Constraints	Displays the pin constraints of the primary inputs.
Report Pin Equivalences	Displays the pin equivalences of the primary inputs.
Report Pin Strokes	Displays the current pin strobe timing for the specified primary output pins.
Report Primary Inputs	Displays the specified primary inputs.
Report Primary Outputs	Displays the specified primary outputs.
Report Procedure	Displays the specified procedure.
Report Pulse Generators	Displays the list of pulse generator sink (PGS) gates.

Table 2-2. Command Summary (cont.)

Command	Description
Report Read Controls	Displays all of the currently defined read control lines.
Report Scan Cells	Displays a report on the scan cells that reside in the specified scan chains.
Report Scan Chains	Displays a report on all the current scan chains.
Report Scan Groups	Displays a report on all the current scan chain groups.
Report Scan Instances	Displays the currently defined sequential scan instances.
Report Scan Models	Displays the sequential scan models currently in the scan model list.
Report Statistics	Displays a detailed report of the design's simulation statistics.
Report Test Stimulus	Displays the stimulus necessary to satisfy the specified set, write, or read conditions.
Report Testability Data	Analyzes collapsed faults for the specified fault class and displays the analysis.
Report Tied Signals	Displays a list of the tied floating signals and pins.
Report Timeplate	Displays the specified timeplate.
Report Version Data	Displays the current software version information.
Report Write Controls	Displays the currently-defined write control lines and their off-states.
Reset Au Faults	Reclassifies the faults in certain untestable categories.
Reset State	Resets the circuit status.
Resume Interrupted Process	Continues a command that you placed in a suspended state by entering a Control-C interrupt.
Run	Runs a simulation or ATPG process.
Save History	Saves the command line history file to the specified file.
Save Patterns	Saves the current test pattern set to a file in the format that you specify.
Select Iddq Patterns	Selects the patterns that most effectively detect IDDQ faults.
Set Abort Limit	Specifies the abort limit for the test pattern generator.
Set Atpg Limits	Specifies the ATPG process limits at which the tool terminates the ATPG process.
Set Atpg Window	Lets you specify the size of the FlexTest simulation window.
Set Bus Handling	Specifies the bus contention results that you desire for the identified buses.

Table 2-2. Command Summary (cont.)

Command	Description
Set Capture Clock	Specifies the capture clock for random patterns or, optionally, for all ATPG patterns.
Set Capture Limit	Specifies the number of test cycles between two consecutive scan operations.
Set Checkpoint	Specifies whether the tool uses the checkpoint functionality.
Set Clock Restriction	Specifies whether ATPG can create patterns with more than one active capture clock.
Set Contention Check	Specifies the conditions of contention checking.
Set Contention_bus Reporting	Specifies the conditions of contention bus reporting.
Set Display	Sets the DISPLAY environment variable from the tool's command line.
Set Dofile Abort	Specifies whether the tool aborts or continues dofile execution if it detects an error condition.
Set Drc Handling	Specifies how the tool globally handles design rule violations.
Set Driver Restriction	Specifies whether the tool allows multiple drivers on buses and multiple active ports on gates.
Set Fails Report	Specifies whether the design rules checker displays clock rule failures.
Set Fault Dropping	Specifies whether FlexTest drops DS faults in Fault mode.
Set Fault Mode	Specifies whether the fault mode is collapsed or uncollapsed.
Set Fault Sampling	Specifies the fault sampling percentage.
Set Fault Type	Specifies the fault model for which the tool develops or selects ATPG patterns.
Set File Compression	Controls whether the tools read and write files with .Z or .gz extensions as compressed files (the default).
Set Flatten Handling	Specifies how the tool globally handles flattening violations.
Set Gate Level	Specifies the hierarchical level of gate reporting and displaying.
Set Gate Report	Specifies the additional display information for the Report Gates command.
Set Gzip Options	Specifies GNU gzip options to use with the GNU gzip command.
Set Hypertrophic Limit	Specifies the percentage of the original design's sequential primitives that can differ from the good machine, before the tool classifies them as hypertrophic faults.

Table 2-2. Command Summary (cont.)

Command	Description
Set Iddq Checks	Specifies the restrictions and conditions that you want the tool to use when creating or selecting patterns for detecting IDDQ faults.
Set Iddq Strobe	Specifies on which patterns (cycles) the tool will simulate IDDQ measurements.
Set Instruction Atpg	Specifies whether FlexTest generates instruction-based test vectors using the random ATPG process.
Set Internal Fault	Specifies whether the tool allows faults within or only on the boundary of library models.
Set Internal Name	Specifies whether to delete or keep pin names of library internal pins containing no-fault attributes.
Set Interrupt Handling	Specifies how FlexTest interprets a Control-C interrupt.
Set Learn Report	Specifies whether the Report Gates command can display the learned behavior for a specific gate.
Set List File	Specifies the name of the list file into which the tool places the pins' logic values during simulation.
Set Logfile Handling	Specifies for the tool to direct the transcript information to a file.
Set Loop Handling	Specifies how the tool handles feedback networks.
Set Net Dominance	Specifies the fault effect of bus contention on tri-state nets.
Set Net Resolution	Specifies the behavior of multi-driver nets.
Set Nonscan Model	Specifies how FlexTest classifies the behavior of nonscan cells with the HOLD and INITX functionality during the operation of the scan chain.
Set Output Comparison	Specifies whether FlexTest performs a good circuit simulation comparison.
Set Output Masks	Ignores any fault effects that propagate to the primary output pins you select.
Set Pattern Source	Specifies the source of the patterns for future Run commands.
Set Possible Credit	Specifies the percentage of credit that the tool assigns possible-detected faults.
Set Procedure Cycle_checking	Enables test procedure cycle timing checking to be done immediately following scan chain tracing during design rules checking.
Set Pulse Generators	Specifies whether the tool identifies pulse generator sink (PGS) gates.

Table 2-2. Command Summary (cont.)

Command	Description
Set Race Data	Specifies how FlexTest handles the output states of a flip-flop when the data input pin changes at the same time as the clock triggers.
Set Rail Strength	Specifies for FlexTest to set the strongest strength of a fault site to a bus driver.
Set Random Atpg	Specifies whether the tool uses random patterns during ATPG.
Set Redundancy Identification	Specifies whether FlexTest performs the checks for redundant logic when leaving the Setup mode.
Set Screen Display	Specifies whether the tool writes the transcript to the session window.
Set Self Initialization	Specifies whether FlexTest turns on/off self-initializing sequence behavior.
Set Sensitization Checking	Specifies whether DRC checking attempts to verify a suspected C3 or C4 rules violation.
Set Sequential Learning	Specifies whether the tool performs the learning analysis of sequential elements to make the ATPG process more efficient.
Set Shadow Check	Specifies whether the tool will identify sequential elements as a “shadow” element during scan chain tracing.
Set System Mode	Specifies whether the tool performs the learning analysis to make the ATPG process more efficient.
Set Stg Extraction	Specifies whether FlexTest performs state transition graph extraction.
Set System Mode	Specifies the system mode you want the tool to enter.
Set Test Cycle	Specifies the number of timeframes per test cycle.
Set Trace Report	Specifies whether the tool displays gates in the scan chain trace.
Set Transient Detection	Specifies whether the tool detects all zero width events on the clock lines of state elements.
Set Unused Net	Specifies whether FlexTest removes unused bus and wire nets in the design.
Set Z Handling	Specifies the simulation handling for high impedance signals on internal and external tri-state nets.
Setenv	Sets a shell environment variable within the tool environment.
Setup Checkpoint	Specifies the checkpoint file to which the tool writes test patterns or fault lists during ATPG.

Table 2-2. Command Summary (cont.)

Command	Description
Setup Pin Constraints	Changes the default cycle behavior for non-constrained primary inputs.
Setup Pin Strokes	Changes the default strobe time for primary outputs without specified strobe times.
Setup Tied Signals	Changes the default value for floating pins and floating nets which do not have assigned values.
Step	Single-steps through several cycles of a test set.
System	Passes the specified command to the operating system for execution.
Unsetenv	Unsets a shell environment variable within the tool environment.
Update Implication Detections	Performs an analysis on the undetected and possibly-detected faults to see if the tool can classify any of those faults as detected-by-implication.
Write Core Memory	Writes to a file the amount of memory that FlexTest requires to avoid paging during the ATPG and simulation processes.
Write Environment	Writes the current environment settings to the file that you specify.
Write Faults	Writes fault information from the current fault list to a file.
Write Initial States	Writes the initial state settings of design instances into the file that you specify.
Write Loops	Writes a list of all the current loops to a file.
Write Modelfile	Writes all internal states for a RAM or ROM gate into the file that you specify.
Write Netlist	Writes the current design in the specified netlist format to the specified file.
Write Primary Inputs	Writes the primary inputs to the specified file.
Write Primary Outputs	Writes the primary outputs to the specified file.
Write Procfile	Writes existing procedure and timing data to the named test procedure file.
Write Statistics	Writes the current simulation statistics to the specified file.

Command Descriptions

This section describes, in alphabetical order, each command available in FlexTest.

Abort Interrupted Process	123
Add Atpg Constraints	124
Add Atpg Functions	128
Add Black Box	132
Add Cell Constraints	135
Add Clocks	138
Add Cone Blocks	140
Add Contention Free_bus	142
Add Faults	143
Add Initial States	146
Add Lists	147
Add Nofaults	149
Add Nonscan Handling	152
Add Output Masks	154
Add Pin Constraints	155
Add Pin Equivalences	158
Add Pin Strokes	159
Add Primary Inputs	160
Add Primary Outputs	162
Add Read Controls	163
Add Scan Chains	164
Add Scan Groups	165
Add Scan Instances	166
Add Scan Models	167
Add Tied Signals	168
Add Write Controls	170
Alias	171
Analyze Atpg Constraints	174
Analyze Bus	176
Analyze Contention	178
Analyze Control Signals	180
Analyze Fault	183

Analyze Race	186
Compress Patterns.....	188
Delete Atpg Constraints	190
Delete Atpg Functions	192
Delete Black Box	194
Delete Cell Constraints	195
Delete Clocks	196
Delete Cone Blocks.....	197
Delete Contention Free_bus	198
Delete Faults.....	199
Delete Initial States	201
Delete Lists	202
Delete Nofaults	203
Delete Nonscan Handling	206
Delete Output Masks.....	208
Delete Pin Constraints.....	209
Delete Pin Equivalences.....	210
Delete Pin Strokes	211
Delete Primary Inputs.....	212
Delete Primary Outputs	214
Delete Read Controls.....	216
Delete Scan Chains.....	217
Delete Scan Groups	218
Delete Scan Instances.....	219
Delete Scan Models	220
Delete Tied Signals.....	221
Delete Write Controls	223
Dofile	224
Exit	226
Find Design Names	227
Flatten Model	232
Help	233
History.....	234
Load Faults.....	236
Read Modelfile	238

Read Procfile	241
Report Aborted Faults.....	246
Report Atpg Constraints.....	248
Report Atpg Functions	249
Report Au Faults	250
Report Black Box.....	253
Report Bus Data.....	255
Report Cell Constraints.....	257
Report Clocks.....	259
Report Cone Blocks.....	260
Report Contention Free_bus.....	261
Report Core Memory	262
Report Drc Rules	263
Report Environment	267
Report Faults	269
Report Feedback Paths	272
Report Flatten Rules	274
Report Gates.....	276
Report Initial States.....	289
Report Lists	290
Report Loops	291
Report Nofaults	293
Report Nonscan Cells	295
Report Nonscan Handling.....	297
Report Output Masks	298
Report Pin Constraints	299
Report Pin Equivalences.....	300
Report Pin Strokes.....	301
Report Primary Inputs	302
Report Primary Outputs.....	304
Report Procedure.....	306
Report Pulse Generators.....	307
Report Read Controls	308
Report Scan Cells.....	309
Report Scan Chains.....	311

Report Scan Groups	313
Report Scan Instances	314
Report Scan Models.....	315
Report Statistics.....	316
Report Test Stimulus.....	320
Report Testability Data.....	323
Report Tied Signals	325
Report Timeplate.....	326
Report Version Data	327
Report Write Controls.....	328
Reset Au Faults	329
Reset State	330
Resume Interrupted Process.....	331
Run	332
Save History	335
Save Patterns	336
Select Iddq Patterns.....	342
Set Abort Limit	346
Set Atpg Limits.....	348
Set Atpg Window.....	350
Set Bus Handling	352
Set Capture Clock	355
Set Capture Limit	357
Set Checkpoint	359
Set Clock Restriction.....	360
Set Contention Check	361
Set Contention_bus Reporting	364
Set Display	366
Set Dofile Abort	367
Set Drc Handling	368
Set Driver Restriction	373
Set Fails Report	375
Set Fault Dropping.....	376
Set Fault Mode	379
Set Fault Sampling.....	381

Set Fault Type	382
Set File Compression	384
Set Flatten Handling	386
Set Gate Level.	388
Set Gate Report	389
Set Gzip Options	394
Set Hypertrophic Limit	396
Set Iddq Checks	397
Set Iddq Strobe.	401
Set Instruction Atpg.	403
Set Internal Fault.	404
Set Internal Name	405
Set Interrupt Handling	406
Set Learn Report	408
Set List File.	410
Set Logfile Handling	411
Set Loop Handling	413
Set Net Dominance.	414
Set Net Resolution	416
Set Nonscan Model.	417
Set Output Comparison.	419
Set Output Masks.	421
Set Pattern Source	422
Set Possible Credit	426
Set Procedure Cycle_checking	427
Set Pulse Generators	428
Set Race Data	429
Set Rail Strength	430
Set Random Atpg.	431
Set Redundancy Identification	432
Set Screen Display	433
Set Self Initialization	434
Set Sensitization Checking	436
Set Sequential Learning	437
Set Shadow Check	438

Set Static Learning	439
Set Stg Extraction	441
Set System Mode	442
Set Test Cycle	444
Set Trace Report	445
Set Transient Detection	446
Set Unused Net	448
Set Z Handling	449
Setenv	451
Setup Checkpoint	452
Setup Pin Constraints	454
Setup Pin Strokes	456
Setup Tied Signals	457
Step	458
System	459
Unsetenv	460
Update Implication Detections	461
Write Core Memory	462
Write Environment	463
Write Faults	465
Write Initial States	468
Write Loops	469
Write Modelfile	470
Write Netlist	471
Write Primary Inputs	473
Write Primary Outputs	475
Write Procfile	477
Write Statistics	479

Abort Interrupted Process

Scope: All modes

Prerequisites: The Set Interrupt Handling command must be on and a FlexTest command must be interrupted with a Control-C.

Aborts a FlexTest command previously interrupted by pressing the Control-C key. This removes the interrupted command from the suspended state and returns control to FlexTest.

Usage

ABOrt INterrupted Process

Arguments

None.

Examples

The following example enables the suspend-state interrupt handling, begins an ATPG run, and interrupts the run:

```
set interrupt handling on
set system mode atpg
add faults -all
run
<Control-C>
```

Now with the run suspended, the example continues by displaying all the untestable faults, and then aborting the run:

```
report faults faultlist -class ut
abort interrupted process
```

Related Topics

[Resume Interrupted Process](#)

[Set Interrupt Handling](#)

Add Atpg Constraints

Scope: Setup, Atpg, Good, and Fault modes

Prerequisites: This command can only be used after the design is flattened to the simulation model, which happens when you first exit Setup mode or when you issue the Flatten Model command.

Specifies to restrict all patterns placed into the internal pattern set according to the user-defined constraints.

Usage

```
ADD ATpg Constraints {0 | 1 | Z} {pin_pathname | net_pathname | gate_id# |  
  {function_name | {-Cell cell_name { {pin_name | net_name} ... } } } ... [-Instance  
  {object_expression...}]  
  [-MODULE {module_name...}]} ... [-DYNAMIC | -Static]
```

Description

When a simulated pattern is rejected, a message is issued that indicates the number of rejected patterns and the first gate at which the failure occurred. You can control the severity of the violation with the Set Contention Check command. If you set the checking severity to Error, the simulation is terminated if it rejects a pattern due to a user-defined constraint. You can analyze the simulation data up to the termination point by using the Report Gates command with the Error_pattern option.

User-defined constraints are used when using FlexTest to generate test patterns.

If you change an ATPG constraint for a single internal set of patterns, pattern compression continues using the new constraints, which can cause the good patterns to be rejected; therefore, you should remove all ATPG constraints before compressing the pattern set.

Note



If you constrain a pin by directly creating an ATPG constraint to the *pin_pathname*, and then create another constraint that indirectly creates a different constraint, the constraint that directly specified the *pin_pathname* (overriding the global ATPG cell constraint) is used.

The -Dynamic switch lets you change the ATPG constraints any time during the ATPG process, affecting only the fault simulation and test generation that occurs after the constraint changes. Any subsequently-simulated patterns that fail to meet the current constraints are rejected during fault simulation.

Dynamic ATPG constraints do not affect DRC because of their temporary nature. Static ATPG constraints are unchangeable in ATPG mode, ensuring that DRC must be repeated if they are changed.

In addition to the functionality mentioned above, the Add Atpg Constraints command lets you constrain a net; therefore, if the circuit structure changes and the ATPG constraints specified on

the net pathnames do not change, you do not have to identify the instance and the pin on which the ATPG constraints have to be applied. If any ATPG constraint is added to the net, the equivalent pin is found first and the function is added to that pin instead; therefore, the Report Atpg Constraints command may not show the net pathname specified. You can delete the constraints added to the net using the same net name.

Arguments

- **0 | 1 | Z**

A literal that restricts the named object to a low state, high state, or high impedance state, respectively. You must choose one of the three literals to indicate the state value to which you want the tool to constrain the specified object.

The following lists the four objects on which you can place the constraint. You can use any number of the four argument choices in any order.

- ***pin_pathname***

A repeatable string specifying the pathname of a top-level pin or a library model pin on which you are placing the constraint. Pathnames of pins on intermediate hierarchy modules are not supported.

- ***net_pathname***

A repeatable string specifying the pathname of the net on which you are placing the constraint. You cannot put ATPG constraints on a net in any library modules.

- ***gate_id#***

A repeatable integer specifying the gate identification number of the gate you want to constrain.

- ***function_name***

A repeatable string specifying the name of a function you created with the Add Atpg Functions command. If you place a constraint on an ATPG function that you generated with the Add Atpg Function -Cell command, all cells affected by that ATPG function are also constrained. You can delete all these constraints using the *function_name* argument with the Delete Atpg Constraints command.

- ***-Cell cell_name {pin_name | net_name }***

A repeatable switch and string pair specifying the name of a DFT library cell and name of a pin or specific net on that cell. You can repeat the *pin_name* or *net_name* argument if there are multiple pins or nets on a cell that you need to constrain. If you use the -Cell option, an ATPG constraint is placed on every occurrence of that cell within the design.

- ***-Instance object_expression***

An optional switch and repeatable string pair that places ATPG constraints inside the specified list of instances. You can use regular expressions, which may include any number of embedded asterisk (*) and/or question mark (?) wildcard characters. You can only use this switch when using the -Cell switch or when a function name is specified.

- **-MODULE** *module_name*
An optional switch and repeatable string pair that places the ATPG constraints inside all instances of the specified module.
- **-DYNAMIC**
An optional switch that satisfies only the ATPG constraints during the ATPG process and not during DRC. You can change these constraints during the ATPG process; therefore, DRC does not check these constraints. This is the default.
- **-STATIC**
An optional switch that satisfies the ATPG constraints you are defining during all its processes. You can only add or delete static ATPG constraints when you are in Setup mode, ensuring the static ATPG constraints are used for all ATPG analyses during design rules checking. DRC checks for any violations of ATPG constraints during the simulation of the test procedures (rule E12).

Examples

Example 1

The following example creates a user-defined ATPG function and then uses it when creating ATPG pin constraints:

```
add atpg functions and_b_in and /i$144/q /i$141/q /i$142/q
add atpg constraints 0 /i$135/q
add atpg constraints 1 and_b_in
```

Example 2

The following example creates a user-defined ATPG function and then uses it when creating ATPG constraints. The ATPG constraints are added to the instances inside core1 only.

```
add atpg functions mux_1hotse1 SELECT1 -cell imux3sux2 S2 S1 S0
add atpg constraints 1 mux_1hotsel -instance core1
```

Example 3

The following example adds ATPG constraints to all occurrences of the specified cell inside the instances with pathnames that begin with “/core1/u19/reg”.

```
add atpg constraints 1 -cell imux3sux2 S1 -instance /core1/u19/reg*
```

Example 4

The following example shows how to use wildcards in the second level of hierarchy.

```
add atpg constraints 1 -cell imux3sux2 S1 -instance /core1/*/reg*
```

Related Topics

[Add Atpg Functions](#)

[Report Atpg Constraints](#)

Delete Atpg Constraints

Add Atpg Functions

Scope: All modes

Prerequisites: This command can be used only after the tool flattens the design to the simulation model, which happens when you first attempt to exit Setup mode or when you issue the Flatten Model command.

Creates an ATPG function that you can then use when generating user-defined ATPG constraints.

Usage

ADD ATpg Functions *function_name type {pin_pathname | net_pathname | gate_id# | existing_function_name | {-Cell cell_name {pin_name | net_name}...}}...*
[-Init_state {0 | 1 | X}...]

Description

You can specify any combination of pin pathnames, gate identification numbers, and previously user-defined functions up to a maximum of 32 objects for each function. You can precede any object with the tilde (~) character to indicate an inverted input with respect to the function. If you specify an input pin pathname, the tool automatically converts it to the output pin of the gate that drives that input pin.

Temporal ATPG functions can be specified by using a Delay primitive to delay the signal for one time frame. Temporal constraints can be achieved by combining ATPG constraints with this temporal function option. The -Init_state switch lets you specify initial values when using Frame or Cycle functions.

Note



Temporal constraints cannot be used with self-initialized test sequences. FlexTest requires the first test vector of the current test sequence to satisfy the temporal constraints with the previous generated test sequence. For more information, see the [Set Self Initialization](#) command.

The Add Atpg Functions command also lets you add ATPG functions to a net. If the circuit structure changes and the ATPG functions specified on the net pathnames do not change, you do not have to identify the instance and the pin on which the ATPG functions are applied. If any ATPG function is added to the net, the equivalent pin is found first and the function is added to that pin instead. Therefore, the Report Atpg Function command may not show the net pathname specified.

Arguments

- *function_name*

A required string that specifies the name of the ATPG function you are creating. You can use this *function_name* as an argument to the Add Atpg Constraints command.

- *type*

A required argument specifying the operation that the function performs on the selected objects. The choices for the *type* argument, from which you can select only one, are as follows:

And — The output of the function is the same as for a standard AND gate.

Or — The output of the function is the same as for a standard OR gate.

Equiv — The output of the function is a high state (1) if all its inputs are at a low state (0), or if all its inputs are at a high state (1). So, the function's output is a low state if there is at least one input at a low state and at least one input at a high state.

Select — The output of the function is a high state (1) if all its inputs are at a low state (0) or if one input is at a high state and the other inputs are at a low state. So, if there are at least two inputs at a high state, the function's output is at a low state.

SELECT1 — The output of the function is a high state (1) if one input is at a high state and the other inputs are at a low state (0). So, the function's output is a low state if there are at least two inputs at a high state or all inputs are at a low state.

Frame — The output of the function is delayed by one time frame. This option is not available in Setup mode.

Cycle — The output of the function is delayed by one test cycle. This option is not available in Setup mode.

- *-Init_state 0 | 1 | X*

An optional switch that defines the initial state value of a Frame or Cycle function. You must specify this option at the end of the Add Atpg Functions command when using the Frame or Cycle function type. If this option is not given, the initial value is assumed to be X. For Frame, only one initial value is needed. For Cycle, the number of initial values specified is the same as the number of frames per cycle which is defined in Set Test Cycle command. For example, if there are 3 time frames per cycle, the corresponding command is:

```
Add Atpg Function foo_cycle cycle foo -init_state 110
```

For multiple initial values, the order specified begins with the value specified furthest on the right. In the example above, the first initial value is 0 followed by 1, and finally by 1 again.

The following lists the objects on which the function operates. You can use any number of the argument choices in any order.

- *pin_pathname*

A repeatable string that specifies the pathname to the pin on which you are placing the function. If you specify an input pin name, it is automatically replaced with the output pin of the gate that drives that input pin.

- *gate_ID#*

A repeatable integer that specifies the gate identification number.

- *existing_function_name*
A repeatable string that specifies the name of another function you created with the Add Atpg Functions command. The *existing_function_name* argument cannot be the same as any pin name in the design. This string cannot be used with the -Cell option.
- *net_pathname*
A repeatable string that specifies the pathname to the net on which you are placing the function. You cannot put ATPG functions on a net in any library modules.

Examples

Example 1

The following example creates an ATPG function and then uses it in an Add Atpg Constraints command:

```
add atpg functions and_b_in and /i$144/q /i$141/q /i$142/q
add atpg constraints 1 and_b_in
```

Example 2

The following example generates multiple levels of ATPG functions when using the -Cell switch.

For every instantiation of the OR4 cell, the following function is implemented:

```
[ ( A AND B ) OR ( C OR D ) ]
```

where A, B, C, and D are inputs of the OR4 cell. Normally, you can define two functions, f1 and f2, then add a third function, f3, that refers to f1 and f2. However, when you use the -Cell switch with the Add Atpg Functions command, this is not possible. You can use the -Function switch to define up to two levels of functions using the -Cell switch. The syntax is as follows:

```
SETUP> add atpg function top_function or -cell OR4 -function and A B
        -function or C D
```

This implemented the previously mentioned function, which can be confirmed with the Report Atpg Functions command.

Example 3

The following example assigns the two Q outputs of scan cells /u11 and /u12 to always have the same assigned state after loading.

```
SETUP> add atpg function CELL_INCLUSIVE equiv /u11/Q /u12/Q
SETUP> add atpg constraint 1 CELL_INCLUSIVE
```

Example 4

The following example causes one input to always be the opposite from the other. The tilde (~) character is the negation symbol.

```
SETUP>add atpg function CELL_EXCLUSIVE equiv ~ /u71/Q /u72/Q
SETUP>add atpg constraint 1 CELL_EXCLUSIVE
```

Related Topics

[Add Atpg Constraints](#)

[Report Atpg Functions](#)

[Delete Atpg Functions](#)

Add Black Box

Scope: Setup mode

Defines instances of Verilog modules or ATPG library models as black boxes for ATPG, and sets the constrained value on output or bidirectional black box pins.

Usage

```
ADD BLaCK Box { { {-Instance ins_pathname | -Module module_name }  
[0 | 1 | X | Z]} [-Pin pinname {0 | 1 | X | Z}]... } | {-Auto [0 | 1 | X | Z]}  
[-FAULt boundary | -NOFAULt_boundary | -NO_Boundary]
```

Description

You can blackbox a single instance, or every instance of a particular module or ATPG model you do not want included in the next ATPG run. You can also use this command to automatically blackbox instances of nonexistent modules or ATPG library models.

When reading in a Verilog netlist, a warning message is issued when an instance is found that references a model (Verilog module or ATPG library model) that is not in the netlist or ATPG library. A referenced model for which a module or ATPG library model of the same name is not found in the netlist or ATPG library is considered to be undefined. You must blackbox instances of undefined models; otherwise, an error message is issued when you attempt to leave Setup mode or enter a Flatten Model command. The warning lists the names of referenced models that were not found in the netlist or ATPG library. You can also list the names using “report black box -undefined”.

Use Add Black Box with the -Auto switch to automatically blackbox all instances of models that are considered to be undefined. By default, instances that are automatically blackboxed drive Xs on their outputs. Faults that propagate to the black box inputs are classified as ATPG_untestable (AU). You can use the Add Black Box command to change the output values.

Issuing multiple Add Black Box commands in succession, the first with the -Module switch and subsequent ones with -Instance, globally blackbox all instances of a particular module or ATPG model while selectively blackboxing particular instances of the same module or ATPG model with slightly different pin values.

Tip



: For optimal performance, issue all Add Black Box commands before issuing any other commands.

Arguments

- **-Instance *ins_pathname***

A switch and string pair that blackboxes the single netlist instance whose pathname is *ins_pathname*.

Note



Blackboxing you do with the -Instance switch always overrides blackboxing you do with the -Module switch.

- **-Module *module_name***

A switch and string pair that blackboxes every instance of the Verilog module or ATPG library model *module_name*.

- **-Auto**

A switch that blackboxes any netlist instance that references a model (Verilog module or ATPG library model) that is not in the netlist or in the ATPG library.

- **0 | 1 | X | Z**

An optional literal that specifies pin tie values. When used with the -Instance or -Module switch, it specifies the tie value for any pin not explicitly defined by the -Pin switch. When used with -Auto, it specifies the tie value for every pin. If you specify no value for this option, then the [Setup Tied Signals](#) command's value is the default.

- **-Pin *pinname*0 | 1 | X | Z**

An optional, repeatable switch, string, and literal that specifies the tie value of a particular pin. Use this switch to explicitly define the tie value for a pin, overriding for that pin the global tie value in effect for the -Instance or -Module switch.

- **-FAUlt boundary**

A switch that keeps pin pathnames at the boundaries of all blackboxed instances and allow boundary pins to become fault sites. This is the default.

- **-NOFAUlt_{boundary}**

A switch that keeps pin pathnames at the boundaries of all blackboxed instances, but not allow boundary pins to become fault sites.

- **-NO_{Boundary}**

A switch that does not keep pin pathnames at the boundaries of all blackboxed instances and to not allow boundary pins to become fault sites.

Examples

Example 1

The following example creates a black box for a module named “core”, with each output driven by a tie0 logic gate. The example then overrides, for the single instance “/core1” of the module “core”, the output value of pin1, so it is driven by a tie1 logic gate instead of a tie0. All other instances of the “core” module still have outputs driven by tie0 logic.

```
add black box -module core 0
add black box -instance /core1 -pin pin1 1
```

Example 2

The following example shows the tool's warning message for instances of undefined models. The example then creates black boxes for these instances.

```
// Compiling library ...
// Reading Verilog Netlist ...
// Reading Verilog file dfta_out/pipe_scan.v
// Finished reading file dfta_out/pipe_scan.v
// WARNING: Following modules are undefined:
//     ao21
//     and02
// Use "add black box -auto" to treat these as black boxes.
```

add black box -auto

Related Topics

[Add Tied Signals](#)

[Report Black Box](#)

[Delete Black Box](#)

Add Cell Constraints

Scope: Setup mode

Constrains scan cells to a constant value.

Usage

For FlexTest

ADD CELL Constraints {*pin_pathname* | {*chain_name cell_position*}} **C0** | **C1** | **CX** | **Ox** | **Xx**

Description

The command constrains a scan cell so that the tool loads it with a constant value during scan loading; however, scan cells may change value after scan loading.

For all tools, you can identify a particular scan cell either by specifying a scan chain name along with the cell's position in the scan chain, or by specifying an output pin pathname that connects to a scan memory element. The constraint value that you specify is placed at either the output pin or the scan cell MASTER.

The rules checker audits the correctness of the data that defines the constrained scan cells immediately after scan cell identification. The checker identifies all invalid scan cell constraints and an error condition occurs.

In the case of scan cells with improper controllability or observability, rather than rejecting these circuits you can constrain (or mask) their controllability or observability.

Table 2-3 shows, for scan patterns, examples of the different values of a cell on different cycles with different constraints applied. Each constraint is described in more detail in the Arguments section.

Table 2-3. Example Cell Value Changes with Different Constraints for Scan Patterns

Cycle: Constraint	0 Loaded Value (initialized value)	1 Cell Not Disturbed	2 Cell Disturbed (same val. captured)	3 Cell Disturbed (diff. val. captured)	4 Unloaded Value
No constraint	a	a	a	b	b
C0	0	0	0	b	b
C1	1	1	1	b	b
CX	X	X	X	b	b
OX	a	a	a	b	X
XX	X	X	a	b	X

Table 2-4 shows, for chain patterns, the different load and unload values of a scan cell with different constraints applied.

Table 2-4. Example Scan Cell Value Changes with Different Constraints for Chain Patterns

Constraint	Loaded Value	Unloaded Value
No constraint	a	a
C0	0	0
C1	1	1
CX	X	X
OX	a	X
XX	X	X

Arguments

- ***pin_pathname***

A string that specifies the name of an output pin of the scan cell or an output pin directly connected through only buffers and inverters to the output of a scan memory element. The scan memory element is set to the value that you specify such that the pin is at the constrained value.

Except as noted below, an error condition occurs if the pin pathname does not resolve to a scan memory element. Buffers and inverters may reside between the pin and the memory element.

- ***chain_name cell_position***

A string pair that specifies the name of the scan chain and the position of the cell in the scan chain. The scan chain must be a currently-defined scan chain and the position must be an integer where 0 is the scan cell closest to the scan-out pin. You can determine the position of a cell within a scan chain by using the Report Scan Cells command.

The MASTER memory element of the specified scan cell is set to the value that you specify; there is no inversion. However, the tool may invert the output pin of the scan cell if there is anything between it and the MASTER memory element if inversion exists between the MASTER and the scan output pin of the scan cell only.

- **C0**

A literal that constrains the scan cell to load value 0 only.

- **C1**

A literal that constrains the scan cell to load value 1 only.

- **CX**
A literal that specifies to simulate the loaded scan cell value as unknown (uncontrollable).
- **OX**
A literal that specifies to simulate the unloaded scan cell value as unknown (unobservable).
- **XX**
A literal that constrains the scan cell to be both uncontrollable and unobservable.

Examples

The following example applies one constraint for use during the generation of scan patterns and one constraint for use during the generation of all patterns. Then it displays a list of all the constrained scan cells:

```
add cell constraints chain1 5 c0
add cell constraint /reg_d2_/Q ox -all_patterns
report cell constraints
```

// constraint	chain	cell	cell_instance	constraint	pattern scope
// value	name	position	name	properties	
// -----	-----	-----	-----	-----	-----
// CO	chain1	5		(Dynamic)	Scan Patterns only
// OX	chain4	7	/reg_d2_/Q	(Dynamic)	Chain + Scan Patt's

Related Topics

[Delete Cell Constraints](#)

[Report Scan Cells](#)

[Report Cell Constraints](#)

[Report Scan Chains](#)

Add Clocks

Scope: Setup mode

Adds scan or non-scan clock pins to the clock list for proper scan operation. Any signal is considered to be a clock if it can change the state of a sequential element, including system clocks, sets, and resets.

Usage

ADD CLocks *off_state pin_pathname...*

Description

Pins that you add to the clock list must have an off-state. The off-state of a clock pin is the value on the pin that results in the clock inputs of sequential memory elements becoming inactive. For edge-triggered devices, the off-state is the value on the pin that results in placing their clock inputs at the initial value of a capturing transition. Set and reset lines are also considered as clock lines. You can constrain a clock pin to its off-state in order to suppress its use as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state or an error occurs. If you add an equivalence to the clock list, all of its equivalent pins are added to the clock list as well.

Arguments

- *off_state*
A required literal that specifies the pin value that inactivates the sequential memory elements. The choices are:
 - 0** — A literal specifying the off-state value is 0.
 - 1** — A literal specifying the off-state value is 1.
- *pin_pathname*
A required, repeatable string that typically lists the primary input pins to assign as clocks. The primary input pins in the list must have the same off_state. If the -Internal switch is used, *pin_pathname* lists internal pin pathnames. If both the -Internal and -Pin_name switches are used, *pin_pathname* lists internal pin pathnames to merge into a single, new primary input pin. The *pin_pathname* may include any number of asterisk (*) and/or question mark (?) wildcard characters in a name.

Examples

The following example adds a clock to the clock list with an off-state of one.

```
add clocks 1 clock1
report clocks
```

Related Topics

[Analyze Control Signals](#)

[Report Clocks](#)

[Delete Clocks](#)

[Set Clock Restriction](#)

Add Cone Blocks

Scope: Setup mode

Specifies the blockage points to use during the calculation of the clock and effect cones.

Usage

ADD COne Blocks *pin_pathname*... [-Both] [-Clock] [-Effect] [-Cell *cell_name*]

Description

It overrides the default clock or effect cone blockage points that the tool uses. For example, if you are getting a clock rules violation, you may want to change the clock cone blockage point that the tool uses in its calculations. However, you need to ensure that by changing the blockage point, you are not introducing a problem downstream in the ATPG process (for example, disturbing the scan chain during the scan operation.)

When you change the blockage point for a clock or effect cone, the tool performs rules checking on the validity of the pin that you specified during the general rules checking process. If there is a violation against the pin, the tool assigns it a rule violation identification number of G12.

Arguments

- *pin_pathname*
A required repeatable string that specifies the output pin of a cell as a blockage point.
- -Both
An optional switch that specifies that the cone blockage point is for both the clock cone and the effect cone calculations. This is the command default.
- -Clock
An optional switch that specifies the cone blockage point is only for the clock cone calculation.
- -Effect
An optional switch specifying that the cone blockage point is only for the effect cone calculation.
- -Cell *cell_name*
An optional switch and string pair that specify the name of a DFT library cell at whose *pin_pathnames* you want the tool to place clock cone block points.

Examples

The following example shows a clock that fails on the C3 rule, which says that the clock input of a scan latch is in both the clock and effect cone. If you know that it will not cause a problem downstream, you can change the blockage point the tool uses for the clock cone (or effect cone) and allow that element to pass through the rules checker.

```
// -----  
// Begin scan clock rules checking.  
// -----  
// 5 scan clock/set/reset lines have been identified.  
// All scan clocks successfully passed off-state check.  
// All scan clocks successfully passed capture ability check.  
// Error: Clock /clk failed rule C3 on input 7 of /LS0 (83).  
//      Source of violation: input 7 of /LS0 (83).  
// Error: Rules checking unsuccessful, cannot exit SETUP mode.
```

add cone blocks /ls0/q -clock
report cone blocks

clock /LS0/Q

set system mode atpg
...

Related Topics

[Delete Cone Blocks](#)

[Report Cone Blocks](#)

Add Contention Free_bus

Scope: All modes

Specifies which if any contention free buses are added.

Usage

ADD COntention Free_bus [-ALI | *net_name* | *gate_id*] [-Ignore]

Description

You can add all, specific, or ignore contention free buses. Specifying contention free buses reduces the time and effort needed for DRC to check and classify buses as contention free.

Arguments

- -ALI
A switch that sets all buses as contention free.
- *net_name*
An optional repeatable string that specifies the name of the contention free bus.
- *gate_id*
An optional repeatable string that specifies the gate identification number of the contention free bus.
- -Ignore
A switch that ignores any hard contentions found in a contention free bus you specified. If the -Ignore switch is not turned on, any patterns that cause hard contention will be rejected unless you set contention checking to check for soft contentions only.

Examples

The following example adds a contention free bus.

```
add contention free_bus bus1  
run
```

Related Topics

[Delete Contention Free_bus](#)

[Set Contention Check](#)

[Report Contention Free_bus](#)

[Set Contention_bus Reporting](#)

[Report Gates](#)

[Set Gate Report](#)

[Set Bus Handling](#)

Add Faults

Scope: Atpg, Fault, and Good modes

Adds faults to the current fault list, discards all patterns in the current test pattern set, and sets all faults to undetected (actual category is UC).

Usage

```
ADD FAults {object_pathname... | -All} [-Stuck_at {01 | 0 | 1}] [{> | >>} file_pathname]
```

Description

When you enter the Setup mode, all faults from the current fault list are deleted. Furthermore, if you change the fault type, all faults are deleted.

You cannot add faults for clock domains, and specific objects or paths within the same command instance. Use a separate command instance for each.

With the fault mode set to “uncollapsed” (the invocation default), all possible faults are added to the list. If you change the fault mode to “collapsed” with the [Set Fault Mode](#) command, only one instance of any given fault is added, ignoring any equivalent faults.

Arguments

- *object_pathname*
A required, repeatable string that specifies the instances or pins whose faults you want added to the current fault list.
- -All
A required switch that adds all faults.
- -Stuck_at {01 | 0 | 1}
An optional switch and literal pair that specifies which stuck-at or transition faults to add to the fault list.
 - 01 — A literal specifying that, for stuck-at faults, the tool add both the “stuck-at-0” and “stuck-at-1” faults; or for transition faults, the tool add both “slow-to-rise” and “slow-to-fall” faults. This is the default.
 - 0 — A literal specifying that, for stuck-at faults, the tool add only the “stuck-at-0” faults; or for transition faults, the tool add only “slow-to-rise” faults.
 - 1 — A literal specifying that, for stuck-at faults, the tool add only the “stuck-at-1” faults; or for transition faults, the tool add only “slow-to-fall” faults.
- > *file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- `>> file_pathname`

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

Example 1

The following example adds all faults to the circuit so you can run an ATPG process:

```
set system mode atpg  
add faults -all  
run
```

Example 2

The following example uses a prefix match to add faults only to pins on instances whose pathnames begin with “/u9/u2/LCT_reg”:

```
add faults /u9/u2/LCT_reg*  
  
=== Found 16 design pins ===  
/u9/u2/LCT_reg_0_/A0  
/u9/u2/LCT_reg_0_/A1  
/u9/u2/LCT_reg_0_/S0  
/u9/u2/LCT_reg_0_/Y  
/u9/u2/LCT_reg_1_/A0  
/u9/u2/LCT_reg_1_/A1  
/u9/u2/LCT_reg_1_/S0  
/u9/u2/LCT_reg_1_/Y  
/u9/u2/LCT_reg_2_/A0  
/u9/u2/LCT_reg_2_/A1  
/u9/u2/LCT_reg_2_/S0  
/u9/u2/LCT_reg_2_/Y  
/u9/u2/LCT_reg_3_/A0  
/u9/u2/LCT_reg_3_/A1  
/u9/u2/LCT_reg_3_/S0  
/u9/u2/LCT_reg_3_/Y
```

Related Topics

[Add Nofaults](#)

[Set Fault Mode](#)

[Delete Faults](#)

[Set Fault Sampling](#)

[Load Faults](#)

[Set Fault Type](#)

[Report Faults](#)

[Set Internal Fault](#)

[Report Testability Data](#)

Write Faults

Add Initial States

Scope: Setup mode

Specifies an initial state for the selected sequential instance.

Usage

ADD INitial States {**0** | **1** | **X**} *instance_pathname...*

Description

You can also initialize states using the **test_setup** procedure within the test procedure file. The problem with using the **test_setup** procedure is that it always applies a force operation (even when there is no force statement), which can destroy the initial state you just set.

If you use both the **test_setup** procedure and the Add Initial States command, FlexTest overrides the states after the **test_setup** procedure with the state you specify in the Add Initial States command.

FlexTest does not use the information that you specify with the Add Initial States command during the rules checking process.

Arguments

- **0**
A literal that initializes the instance to a low state.
- **1**
A literal that initializes the instance to a high state.
- **X**
A literal that initializes the instance to an unknown value.
- *instance_pathname*
A required repeatable string that specifies the name of a design hierarchical instance. You cannot specify a DFT library hierarchical instance name. You can specify the whole circuit by entering “/”.

Examples

The following example initializes two flip-flop instances to a low state:

```
add initial state 0 /amm/g30/ff0 /amm/g29/ff0
```

Related Topics

[Delete Initial States](#)

[Write Initial States](#)

[Report Initial States](#)

Add Lists

Scope: Atpg, Fault, and Good modes

Adds pins to the list of pins on which to report.

Usage

ADD Lists *pin_pathname*...

Description

The Add Lists command specifies cell output pins for which the tool should report the results of good and faulty simulation. In Good simulation mode, the tool displays the values of these pins for the good machine. In Fault simulation mode and Atpg mode, the tool displays the values of these pins for the good machine and their values for the faulty machine when the two values differ.

Tip



This command is a useful debugging aid, as it provides a way to check the values simulated for specific output pins of cells within the design.

To list all the pins the tool currently will report on during simulation, use the [Report Lists](#) command. To turn off reporting of a pin's simulation values, use the [Delete Lists](#) command. To direct the tool to write the pins' simulation values to a file instead of displaying them onscreen, use the [Set List File](#) command.

When switching to Setup mode, the tool discards all pins from the report list.

Arguments

- *pin_pathname*

A required, repeatable string that specifies the output pins on which to report during good and/or faulty machine simulation.

Examples

The following example writes to the file, *pinlist_goodsim.txt*, the values simulated for two output pins during good machine simulation of an external pattern source:

```
set system mode good
set pattern source external pattern1
add lists /i_1006/o /i_102/o
set list file pinlist_goodsim.txt -replace
run
```

Related Topics

[Delete Lists](#)

[Set List File](#)

[Report Lists](#)

Add Nofaults

Scope: Setup mode

Places nofault settings either on pin pathnames, pin names of specified instances, or modules.

Usage

For FlexTest

```
ADD NOfaults { { modulename... -Module } | { object_expression... [-PIN | -Instance] } }  
[-Stuck_at { 01 | 0 | 1 } ] [-Keep_boundary] [-VERBOSE] [{> | >> } file_pathname]
```

Description

By specifying pathnames of pins, instances, or modules while in Setup mode, the Add Nofaults command places a nofault setting either on the specific pins or on boundary and internal pins of the instances or modules. All added nofault pin pathnames are in the user class.

The -Stuck_at switch applies to either stuck-at faults or transition faults. For the latter, you specify 0 for “slow-to-rise” and 1 for “slow-to-fall” transition faults. If you do not specify a value, then the tool places a nofault setting on both stuck-at (transition) values. If you add faults with the Add Faults command after you issue the Add Nofaults command, the specified pin pathnames or boundary and internal pins of instances or modules cannot be sites for those added faults.

Note



When you add nofault settings, the tool deletes the flattened simulation model if it exists. This removes information you added after model flattening, such as ATPG functions. To avoid losing this information or having to wait for the tool to flatten the design again, add nofault settings prior to model flattening.

Arguments

- ***modulename***

A repeatable string that specifies the name of a module to which you want to assign nofault settings. You must include the -Module switch when you specify a module name.

- **-Module**

A switch that specifies to interpret the *modulename* argument as a module pathname. All instances of the module are affected. You can use the asterisk (*) and question mark (?) wildcards for the *modulename* argument, and the tool adds the nofault for all matching modules or library models.

- ***object_expression***

A string representing a list of pathnames of instances or pins for which you want to assign nofault settings. You can use regular expressions, which may include any number of asterisk (*) and question mark (?) wildcard characters.

Pin pathnames must be ATPG library cell instance pins, also referred to as design level pins. If the object expression specifies a pin within an instance of an ATPG library model, the tool ignores it. By default, pin pathnames are matched first. If a pin pathname match is not found, the tool next tries to match instance pathnames. You can force the tool to match only pin pathnames or only instance pathnames by including the `-Pin` or `-Instance` switch after the *object_expression*.

- **-PIN**

An optional switch that specifies to use the preceding object expression to match only pin pathnames; the tool will then assign nofault settings to all the pins matched.

- **-Instance**

An optional switch that specifies to use the preceding object expression to match only instance pathnames; the tool will then assign nofault settings to all boundary and internal pins of the instances matched (unless you use the `-Keep_boundary` switch).

- **-Stuck_at 01 | 0 | 1**

An optional switch and literal pair that specifies to which stuck-at or transition values you want to assign a nofault setting. The choices are as follows:

01 — A literal that specifies to place a nofault setting on both the “stuck-at-0” and “stuck-at-1” faults for stuck-at faults; or on both the “slow-to-rise” and “slow-to-fall” faults for transition faults. This is the default.

0 — A literal that specifies to place a nofault setting on only the “stuck-at-0” faults for stuck-at faults; or on only the “slow-to-rise” faults for transition faults.

1 — A literal that specifies to place a nofault setting on only the “stuck-at-1” faults for stuck-at faults; or on only the “slow-to-fall” faults for transition faults.

- **-Keep_boundary**

An optional switch that specifies to apply nofault settings to the inside of the specified instance/module, but allow faults at the boundary pins of these instances/modules. This option does not apply to nofaults on pin pathnames.

- **-VERbose**

By default when you specify wildcard characters for instance (`-INstance`) or pin (`-PIN`) names, the tool outputs a summary message similar to the following:

```
// Note: Adding faults for 330 fault sites.
```

When you specify the optional `-VERbose` switch, then the tool outputs the instance or pin names instead of the summary. You can optionally redirect this output to a file using the `>` or `>>` redirection operators.

If you use actual instance or pin names instead of wildcards, then this switch has no effect.

- **> *file_pathname***

An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- `>> file_pathname`

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example defines nofault settings for all the pins in a particular instance, so when you add all faults to the circuit for an ATPG run, the tool will not place faults on the pins of that instance:

```
add nofaults i_1006 -instance  
set system mode atpg  
add faults -all  
run
```

Related Topics

[Add Faults](#)

[Report Faults](#)

[Delete Faults](#)

[Report Nofaults](#)

[Delete Nofaults](#)

Add Nonscan Handling

Scope: Setup mode

Prerequisites: Your design must have scan in order to be able to add nonscan handling.

Overrides behavior classification of non-scan elements that FlexTest learns during the design rules checking process.

Usage

ADD NOnscan Handling *learned_behavior_element_pathname...*
[-Instance | -Module]

Description

When you exit the Setup mode, the design rules checker classifies each non-scan element into a type of learned behavior. FlexTest then uses this information when simulating the operation of the scan chains.

However, due to limitations on modeling and simulation capabilities, FlexTest can sometimes pessimistically classify a non-scan element. If you want to override the behavior classification of a particular non-scan element, use the Add Nonscan Handling command.

The Set Nonscan Model command continues to have the same effect on HOLD and INITX for behaviors learned from the design rules checker, or that you set with the Add Nonscan Handling command.

Arguments

- *learned_behavior*

A required literal argument that specifies the classification of learned behavior that you want to assign to the named non-scan element. The choices for the *learned_behavior* argument, from which you can select only one, are:

TIE0 — A literal that specifies for the non-scan element to always be at a low state when FlexTest operates the scan chain.

TIE1 — A literal that specifies for the non-scan element to always be at a high state when FlexTest operates the scan chain.

Hold — A literal that specifies for the state of this type of element to remain undisturbed when FlexTest operates the scan chain.

INITX — A literal specifying that the logic state of the non-scan element is unknown when FlexTest finishes operating the scan chain.

INIT0 — A literal that specifies for the output of this non-scan element to be a low state when FlexTest finishes operating the scan chain.

INIT1 — A literal that specifies for the output of this non-scan element to be a high state when FlexTest finishes operating the scan chain.

- ***element_pathname***
A required repeatable string that specifies the pathname to the non-scan element whose learned behavior you want to reclassify during the time FlexTest operates the scan chain.
- **-Instance**
An optional literal that specifies that the ***element_pathname(s)*** specified are instance pathnames. This is the default upon invocation.
- **-Module**
An optional literal that specifies that the ***element_pathname(s)*** specified are module names. All instances with the specified modules are affected by this command as well as the Delete Nonscan Handling command.

Examples

The following example specifies for FlexTest to assume that the given non-scan element is always at a high state, regardless of how the design rules checker determined its behavior:

```
add nonscan handling tie0 i_6_16
report nonscan handling

TIE0 I_6_16
```

Related Topics

[Delete Nonscan Handling](#)

[Set Nonscan Model](#)

[Report Nonscan Handling](#)

Add Output Masks

Scope: Setup mode

Ignores any fault effects that propagate to the primary output pins you name.

Usage

ADD OUtput Masks *primary_output...* | **-All**

Description

The tool uses primary output pins as the observe points during the fault detection process. When you mask a primary output pin, you inform the tool to mark that pin as an invalid observation point during the fault detection process. This command allows you the ability to flag primary output pins that do not have strobe capability. The tool classifies the faults whose effects only propagate to that observation point as ATPG_untestable (AU).

Arguments

- *primary_output*
A repeatable string that specifies the name of the primary output pin you want to mask.
- **-All**
A switch that specifies to mask all primary output pins.

Examples

The following example specifies the primary output pins that will not have the strobe capability on the hardware tester:

```
add output masks qb1 qb2 qb3
```

Related Topics

[Delete Output Masks](#)

[Set Output Masks](#)

[Report Output Masks](#)

Add Pin Constraints

Scope: Setup mode

Adds a pin constraint to a primary input pin.

Usage

ADD PIn Constraints *primary_input_pin constraint_format*

Description

The Add Pin Constraint command performs slightly differently depending on which tool you use.

The Add Pin Constraint command adds cycle behavior constraints to the specified primary input.

For every primary input for which you do not specify a constraint by using the Add Pin Constraint command, FlexTest automatically uses the default format type NR, with a period of 1 and an offset of 0. To change the default format, use the Setup Pin Constraints command.

To specify the test cycle width, use the Set Test Cycle command.

You can constrain a clock pin to its off-state to prevent its use as a capture clock during the ATPG process. The constrained value must be the same as the clock off-state or an error occurs. All clocks with a 0 off-state should have a return-zero waveform. Likewise, all clocks with a 1 off-state should have a return-one waveform. You cannot constrain an equivalent pin, with the exception of a NR format pins.

FlexTest provides 11 constraint formats from which you choose the constant value that you want to apply to a primary input. Further, these constraint formats (waveform types) group into three waveform classes that apply to all automatic test equipment:

Group 1 — Non-return waveform; the pin value may change only once. Includes the NR, C0, C1, CZ, and CX constraint formats. Group 1 waveforms require you to specify the period and offset. If not specified for C0, C1, CX, and CZ, FlexTest assumes the period is 1 and the offset is 0.

Group 2 — Return-zero waveform; the pin value may rise to a 1 and then return to a 0. Includes the R0, SR0, and CR0 constraint formats. Group 2 waveforms require you to specify the period, offset, and pulse width.

Group 3 — Return-one waveform; the pin value may fall to a 0 and then return to a 1. Includes the R1, SR1, and CR1 constraint formats. Group 3 waveforms require you to specify the period, offset, and pulse width.

The following “Arguments” subsection describes the constraint formats in more detail.

Arguments

- ***primary_input_pin***

A required string that specifies the primary input pin you want to constrain.

- ***constraint_format***

An argument that specifies the constant value with which you want to constrain the primary input pin. The constraint format choices are as follows:

NR *period offset* — A literal and two-integer triplet that specifies application of the non-return waveform value to the chosen primary input pin. The test pattern set you provide determines the actual value FlexTest assigns to the pin.

C0 — A literal that specifies application of the constant 0 to the chosen primary input pin. For FlexTest, if the value of the pin changes during the scan operation, FlexTest uses the non-return waveform. The tools do not use C0 constraints to determine tied circuitry. Therefore, faults associated with line holds resulting from these constraints are classified as ATPG_untestable (AU), as opposed to Tied (TI) or Blocked (BL). Use a C0 constraint if the constraint is valid in test mode, but not valid in system mode.

Note



If the constraint is valid in system mode as well, consider using a CT0 constraint. Faults associated with tied circuitry resulting from a CT0 constraint are not classified as ATPG_untestable, thereby improving test coverage compared to using a C0 constraint. To ensure the improvement in coverage matches reality, however, be sure to use CT0 only for valid system mode constraints.

C1 — A literal that specifies application of the constant 1 to the chosen primary input pin. For FlexTest, if the value of the pin changes during the scan operation, FlexTest uses the non-return waveform. The tools do not use C1 constraints to determine tied circuitry. Therefore, faults associated with line holds resulting from these constraints are classified as ATPG_untestable (AU), as opposed to Tied (TI) or Blocked (BL). Use C1 if the constraint is valid in test mode, but not valid in system mode.

Note



If the constraint is valid in system mode as well, consider using CT1 instead. Faults associated with tied circuitry resulting from a CT1 constraint are not classified as ATPG_untestable, thereby improving test coverage compared to using a C1 constraint. To ensure the improvement in coverage matches reality, however, be sure to use CT1 only for valid system mode constraints.

CZ — A literal that specifies application of the constant Z (high-impedance) to the chosen primary input pin. For FlexTest, if the value of the pin changes during the scan operation, FlexTest uses the non-return waveform.

CX — A literal that specifies application of the constant X (unknown) to the chosen primary input pin.

If the value of the pin changes during the scan operation, FlexTest uses the non-return waveform.

R0 *period offset width* — A literal and three-integer quadruplet that specifies application of one positive pulse per period.

SR0 *period offset width* — A literal and three-integer quadruplet that specifies application of one suppressible positive pulse during non-scan operation.

CR0 *period offset width* — A literal and three-integer quadruplet that specifies no positive pulse during non-scan operation.

R1 *period offset width* — A literal and three-integer quadruplet that specifies application of one negative pulse per specified period during non-scan operation.

SR1 *period offset width* — A literal and three-integer quadruplet that specifies application of one suppressible negative pulse.

CR1 *period offset width* — A literal and three-integer quadruplet that specifies no negative pulse during non-scan operation.

Where:

period — An integer that specifies the period in terms of the total number of test cycles. The Set Test Cycle command defines the number of timeframes per test cycle.

offset — An integer that specifies the timeframe in which values start to change in each cycle.

width — An integer that specifies the pulse width of the pulse type waveform in number of timeframes.

Examples

The following FlexTest example adds a cycle behavior constraint to a primary input. This primary input will always have one positive pulse per cycle. The rising edge is at time 0 (offset is 0), and the falling edge is at time 1 (pulse width is 1). Its cycle period is the same as one test cycle that consists of two timeframes:

```
set test cycle 2
add pin constraints ph1 r0 1 0 1
```

Related Topics

[Delete Pin Constraints](#)

[Set Test Cycle](#)

[Report Pin Constraints](#)

[Setup Pin Constraints](#)

Add Pin Equivalences

Scope: Setup mode

Adds restrictions to primary inputs so that they have equal or inverted values.

Usage

ADD PIN Equivalences *target_pin*... [-Invert] *reference_pin*

Description

The Add Pin Equivalences command specifies that all primary input pins named prior to the *reference_pin* take on the value (or the inverted value) of the *reference_pin*. You can only specify either pin equivalences or inversions in one command line. If you need to specify both pin equivalences and inversions, you need to enter the command twice.

Arguments

- *target_pin*
A repeatable string that lists the primary input pins whose values you want to either equal or invert with respect to *reference_pin*.
- -Invert
An optional switch that specifies for FlexTest to hold the *target_pin* value to the opposite state of the *reference_pin* value. If you use this switch, you must enter it immediately prior to the *reference_pin* value.
- *reference_pin*
A required string that specifies the name of the primary input pin whose value you want the tool to use when determining the state value of the other named, primary input pins.

Examples

The following is a FlexTest example:

```
add pin equivalences indata3 indata2
add pin equivalences indata4 -invert indata2
```

This example provides the following results:

```
indata3 is equivalent to indata2
indata4 is inverted with respect to indata2
Example 1
```

Related Topics

[Delete Pin Equivalences](#)

[Report Pin Equivalences](#)

Add Pin Strokes

Scope: Setup mode

Adds strobe time to the primary outputs.

Usage

ADD PIn Strokes *strobe_time primary_output_pin...* [-Period integer]

Description

The Add Pin Strokes command adds a strobe time for each test cycle of the specified primary output pins. Any primary outputs without specified strobe times use the default strobe time. For nonscan circuits, the default strobe time is the last timeframe of each test cycle. For scan circuits, FlexTest designates time 1 of each test cycle as the default strobe time for every primary output. You can change the default time frame for non-scan operations by using the -Period option.

Arguments

- *strobe_time*
A required integer that specifies the strobe time for each test cycle. This number should not be greater than the period set with the Set Test Cycle command.
- *primary_output_pin*
A required repeatable string that specifies a list of primary output pins.
- -Period integer
Specifies the number of cycles for the period of each strobe. The default is 1. This option is only available for non-scan operations.

Examples

The following example adds time 1 as the strobe time of primary output pin outdata1.

```
set test cycle 3
add pin strokes 1 outdata1
```

Related Topics

[Delete Pin Strokes](#)

[Setup Pin Strokes](#)

[Report Pin Strokes](#)

Add Primary Inputs

Scope: Setup mode

Adds a primary input (PI) to the specified nets or pins.

Usage

ADD PRimary Inputs *net_or_pin_pathname...* [-Cut] [-Module]

Description

It also designates them as user-class PIs rather than system-class PIs described in the original netlist. Use the -Cut switch to disconnect the original drivers of the net, so that the added PI becomes the only driver of the net or pin; otherwise, the net is treated as a wired net if there are other drivers besides the newly-added PI. You can display the user-class, system-class, or full-class PIs by using the Report Primary Inputs command.

Arguments

- *net_or_pin_pathname*

A required, repeatable string that specifies the pathname of a net or pin to which you want to add a primary input. If the -Internal switch is used, *net_or_pin_pathname* lists internal net pathnames. If both the -Internal and -Pin name switches are used, *net_or_pin_pathname* lists internal net or pin pathnames to merge into a single, new primary input pin. The *net_or_pin_pathname* may include any number of asterisk (*) and/or question mark (?) wildcard characters in a name.

- -Cut

An optional switch that disconnects the original drivers of each specified net or pin and makes the PI added to each net the only driver of the net. The added PI is treated like any real netlist PI; forcing it when creating patterns and adding it to the external pattern interface when saving the patterns.

Note



Before simulation, unless you add corresponding inputs to the top-level interface of the original Verilog netlist, Verilog patterns are not able to force these nodes during time-based simulation. As a result, you will likely get a simulator error when you try to perform the simulation.

- -Module

An optional switch that adds the primary input to the specified nets in all modules.

Examples

The following example adds two new primary inputs to the circuit and places them in the user class of primary inputs:

```
add primary inputs indata2 indata4
```


Related Topics

[Add Clocks](#)

[Report Primary Inputs](#)

[Add Primary Outputs](#)

[Write Primary Inputs](#)

[Delete Primary Inputs](#)

Add Primary Outputs

Scope: Setup mode

Adds primary outputs.

Usage

ADD PRimary Outputs *net_or_pin_pathname...*

Description

The Add Primary Outputs command adds an additional primary output to each specified net or pin. Once added, the tool defines them as user-class primary outputs. The tool defines the primary outputs described in the original netlist as system class primary outputs. You can display the user class, system class, or full classes of primary outputs using the Report Primary Outputs command.

Arguments

- *net_or_pin_pathname*

A required, repeatable string that specifies the net or pin to which you want to add primary outputs.

Examples

The following example adds a new primary output to the circuit and places it in the user class of primary outputs:

```
add primary outputs outdata1
```

Related Topics

[Add Primary Inputs](#)

[Report Primary Outputs](#)

[Delete Primary Outputs](#)

[Write Primary Outputs](#)

Add Read Controls

Scope: Setup mode

Defines a PI as a read control and specifies its off value.

Usage

ADD REad Controls **0** | **1** *primary_input_pin...*

Description

The Add Read Controls command defines the circuit read control lines and assigns their off-state values. The off-state value of the pins that you specify must be sufficient to keep the RAM outputs stable. You can use clocks, constrained pins, or equivalent pins as read control lines if their off-states are the same.

Arguments

- **0**
A literal specifying that 0 is the off-state value for the read control lines.
- **1**
A literal specifying that 1 is the off-state value for the read control lines.
- *primary_input_pin*
A required repeatable string that specifies the primary input pins you designate as read control lines, and to which you assign the given off-state value.

Examples

The following example assigns an off-state value of 0 to two read control lines, r1 and r2:

```
add read controls 0 r1 r2
set system mode atpg
add faults -all
run
```

Related Topics

[Analyze Control Signals](#)

[Report Read Controls](#)

[Delete Read Controls](#)

Add Scan Chains

Scope: Setup mode

Prerequisites: You must define the scan chain group with the Add Scan Groups command prior to using this command.

The Add Scan Chains command defines a scan chain that exists in the design. A scan chain references the name of a scan chain group defined prior to issuing this command.

Usage

ADD SCan Chains {*chain_name* *group_name* *input_pin* *output_pin*}...

Arguments

- ***chain_name***
A required repeatable string that specifies the name of the scan chain to add to the specified scan group.
- ***group_name***
A required repeatable string that specifies the name of the scan chain group to add the specified scan chain to.
- ***primary_input_pin***
A required repeatable string that specifies the input pin of the specified scan chain.
- ***primary_output_pin***
A required repeatable string that specifies the output pin of the specified scan chain.

Examples

The following example defines two scan chains (chain1 and chain2) that belong to the scan group (grp1):

```
add scan groups grp1 atpg.testproc
add scan chains chain1 grp1 si1 so1
add scan chains chain2 grp1 si2 so2
```

Related Topics

[Add Scan Groups](#)

[Report Scan Chains](#)

[Delete Scan Chains](#)

Add Scan Groups

Scope: Setup mode

Adds a scan chain group to the system.

Usage

ADD SCan Groups {*group_name test_procedure_filename*}...

Description

The Add Scan Groups command defines a scan chain group that contains scan chains for the design. The procedures defined in *test_procedure_filename* control the set of scan chains which make up the scan chain group.

If you specify “dummy” as the group name and provide a test procedure filename, the tool expects the test procedure file to contain only the seq_transparent and test_setup procedures. Doing so lets you run ATPG without having a scan structure currently in the design.

You can define multiple scan chain groups on one command line by repeating the argument pair for each scan chain group.

Arguments

- *group_name*
A required string that specifies the name of the scan chain group that you want to add to the system.
- *test_procedure_filename*
A required string that specifies the name of the test procedure file that contains the information for controlling the scan chains in the specified scan chain group.

Examples

The following example defines a scan chain group, group1, which loads and unloads a set of scan chains, chain1 and chain2, by using the procedures in the file, *scanfile*:

```
add scan groups group1 scanfile
add scan chains chain1 group1 indata2 testout2
add scan chains chain2 group1 indata4 testout4
```

Related Topics

[Add Scan Chains](#)

[Read Procfile](#)

[Delete Scan Groups](#)

[Write Procfile](#)

[Report Scan Groups](#)

Add Scan Instances

Scope: Setup mode

Adds sequential instances to the scan instance list.

Usage

ADD SCan Instances *instance_pathname...*

Description

The Add Scan Instances command specifies that FlexTest treat each sequential instance you name as a scan cell during the ATPG process. If an instance is a module instance, then FlexTest treats all sequential instances beneath it as scan cells during the ATPG process.

This can be used to determine the test coverage on an experimental basis.

Arguments

- *instance_pathname*
A required, repeatable string that specifies the instance pathnames that you want to add to the scan instance list.

Examples

The following example adds two user-defined sequential instances to the scan instance list, and then runs ATPG to determine the resulting test coverage:

```
set system mode setup
add scan instances i_1006 i_1007
set system mode atpg
run
```

Related Topics

[Delete Scan Instances](#)

[Report Scan Instances](#)

Add Scan Models

Scope: Setup mode

Adds sequential models to the scan model list.

Usage

ADD SCan Models *model_name...*

Description

The Add Scan Models command specifies for FlexTest to treat each sequential instance, identified by the model you name, as a scan cell during the ATPG process.

This can be used to determine the test coverage on an experimental basis.

Arguments

- *model_name*
A required, repeatable string that specifies the model names that you want to add to the scan model list. Enter the model names as they appear in the design library.

Examples

The following example treats all instances of the specified library model as scan cells, and then runs ATPG to determine the resulting test coverage:

```
set system mode atpg
add scan models d_flip_flop
set system mode atpg
run
```

Related Topics

[Delete Scan Models](#)

[Report Scan Models](#)

Add Tied Signals

Scope: Setup mode

Adds a value to floating signals or pins.

Usage

ADD Tied Signals {**0** | **1** | **X** | **Z**} *floating_object_name*... [-Pin]

Description

The Add Tied Signals command assigns a specific value to not-clearly-defined floating signals or pins. If there are floating signals or pins in the design, a warning appears when you leave the Setup mode. If you do not assign a specific value, the tool ties the signal or pin values to the default value. To change the default tied value, use the Setup Tied Signals command.

When you add tied signals or pins, the tool places them into the user class. This includes instance-based, blackbox tied signals. When the netlist ties signals or pins to a value, the tool places them into the system class.

Note



The tool will not tie a signal that is connected to bidirectional pins. For example, with a Verilog netlist that has a top level pin named Vdd of type “inout”, you cannot use “add tied signals 1 Vdd -pin” to tie Vdd high.

Arguments

- **0**
A literal that ties the floating nets or pins to logic 0 (low to ground).
- **1**
A literal that ties the floating nets or pins to logic 1 (high to voltage source).
- **X**
A literal that ties the floating nets or pins to unknown.
- **Z**
A literal that ties the floating nets or pins to high-impedance.
- *floating_object_name*
A required, repeatable string that specifies the floating nets or pins to which you want to assign a specific value. The tool assigns the tied value to all floating nets or pins in all modules that have the names that you specify.

Note



For pin names, you must include the -Pin switch on the command line; otherwise, the tool assumes the name is a net name.

If you specify a net pathname, you cannot use the -Pin option.

- -Pin

An optional switch specifying that the *floating_object_name* argument that you provide is a floating pin name.

Examples

The following example ties all floating signals in the circuit that have the net names vcc and vdd, to logic 1 (tied to high):

```
add tied signals 1 vcc vdd
```

Related Topics

[Add Black Box](#)

[Report Tied Signals](#)

[Delete Tied Signals](#)

[Setup Tied Signals](#)

Add Write Controls

Scope: Setup mode

Defines a PI as a write control and specifies its off value.

Usage

ADD Write Controls **0** | **1** *primary_input_pin...*

Description

The Add Write Controls command defines the circuit write control lines and assigns their off-state values. The off-state value of the pins that you specify must be sufficient to keep the RAM contents stable. You can use clocks, constrained pins, or equivalent pins as write control lines if their off-states are the same.

Arguments

- **0**
A literal specifying that 0 is the off-state value for the primary_input_pins.
- **1**
A literal specifying that 1 is the off-state value for the primary_input_pins.
- *primary_input_pin*
A required, repeatable string that specifies the primary input pins that are write control lines to which you want to assign an off-state value.

Examples

The following example assigns an off-state to two write control lines, w1 and w2:

```
add write controls 0 w1 w2
set system mode atpg
add faults -all
run
```

Related Topics

[Analyze Control Signals](#)

[Report Write Controls](#)

[Delete Write Controls](#)

Alias

Scope: All modes

Specifies the shorthand name for a tool command, UNIX command, or existing command alias, or any combination of the three.

Usage

Alias [*synonym* {!*unix_command*; | *tool_command*; | *alias_synonym*;}...]

Description

Issuing the Alias command with no arguments will list the current aliased commands. If you specify a shorthand name (*synonym*) and one of the command types, that shorthand name can substitute for the command and any arguments you specify. You utilize the full power of the Alias command when you take advantage of the repeatable nature of the second string, intermixing any number of command types, and separating them with semicolons.

Note



To display the commands defined for a particular *synonym*, issue the Help or Alias command with the *synonym* and no other arguments.

In addition, the command strings can be parameterized by using the formal parameters, \$1 through \$9, inserted in the command string in any order. When you issue the *synonym* as a command, you must supply the actual arguments, which are substituted into the command prior to its execution.

You can also provide an optional startup file, (.flextest_startup) that contains tool-specific commands to be executed prior to any other batch or interactive commands. The primary purpose of this file is to execute Alias commands that tailor the tool's command language to your needs. Upon invocation, the tool searches for the startup file in the following locations and order of precedence:

- The directory pointed to by the MGCDFT_STARTUP environment variable
- The local invocation directory
- Your home directory

The first startup file the tool encounters is the only one it executes if you have startup files in multiple locations. If the tool does not locate a tool-specific startup file, it searches the same locations for the generic startup file (.mgcdft_startup).

Arguments

- *synonym* { *!unix_command*; | *tool_command*; | *alias_synonym*; }

An optional string with a repeatable string that specifies a shorthand name, *synonym*, for the specified UNIX or tool command or for a previously-defined alias synonym (which has the effect of a command). Repeated commands must be separated by semicolons.

!unix_command — An optional, repeatable string that consists of any well-formed UNIX command, with its arguments, or script. You must precede this string with an exclamation point to differentiate it from a tool-specific command.

tool_command — An optional, repeatable string that consists of any well-formed FlexTest command and its arguments.

alias_synonym — An optional, repeatable string that consists of any *synonym* previously defined with the Alias command.

Examples

Example 1

The following example defines a new command with the shorthand name, *watch*, and includes a formal parameter in the definition. The next line issues the new command and supplies the actual parameter:

```
alias watch !ps -e | egrep $1;  
watch netscape
```

The result of issuing the new command (which you can think of as an alias for the otherwise lengthy command string for which it substitutes) is to list all the process ids associated with Netscape processes on the host machine.

Example 2

The following example defines the new command, *findlockup*, which searches the current directory for Verilog files and invokes *egrep* on each one in turn, looking for and displaying any “*lockup*” names:

```
alias findlockup !find . -name \*.v -print -exec egrep lockup {} \;
```

You could then use that new command within another Alias command that writes out the current design:

```
alias findit write netlist -verilog temp.v -replace; findlockup
```

Example 3

The following example defines two new command aliases (“*wibble*” and “*wobble*”), invokes them, and requests help on them:

```
alias wibble !echo arg1 arg2 $1 $2 $3 $4  
alias wobble report black box -undefined  
wibble one_1 two_2 three_3 four_4
```

```
arg1 arg2 one_1 two_2 three_3 four_4
```

wobble

```
// Undefined Modules:  
//     foo
```

alias wobble

```
// alias: wobble=report black box -undefined
```

alias

```
// List of aliased commands:  
//     wobble=report black box -undefined  
//     wibble=!echo arg1 arg2 $1 $2 $3 $4
```

help wobble

```
// alias: wobble=report black box -undefined
```

help wibble

```
// alias: wibble=!echo arg1 arg2 $1 $2 $3 $4
```

Related Topics

[Dofile](#)[System](#)[History](#)

Analyze Atpg Constraints

Scope: Atpg, Fault, and Good modes

Checks the ATPG constraints you created for their satisfiability or mutual exclusivity.

Usage

```
ANalyze Atpg Constraints {-AUto | -ALL | {pin_pathname | gate_id# | function_name}...}  
[-Bus]
```

Description

If you issue the Analyze Atpg Constraints command without any arguments, the default is -All. When the command finishes, the tool displays a message that indicates whether the analysis passed, failed, or aborted the ATPG constraint analysis.

Arguments

The following lists the three methods for naming the objects for which you want to analyze the constraints. You can use any number of the three argument choices, in any order.

- -Auto
An optional switch that automatically tries to locate the atpg constraint that cannot be satisfied. The analysis checks to see if any single constraint cannot be satisfied, then reports each constraint that cannot be satisfied (given the current abort limit and other restrictions). Sometimes, each constraint can be satisfied by itself, but some set of constraints cannot all be satisfied. In this case, -Auto switch proceeds to a second analysis where it adds atpg constraints to a set to create a minimal set that can't be satisfied.
- -ALL
An optional switch that specifies for the tools to perform the ATPG analysis simultaneously for all the current ATPG constraints. If you do not specify an object name, this is the command default.
- pin_pathname
A repeatable string that specifies the pathname to the pin on which you are analyzing the constraints.
- gate_id#
A repeatable integer that specifies the gate identification number of the gate on which you want to analyze the constraints.
- function_name
A repeatable string that specifies the name of a function you created with the Add Atpg Functions command. If you generated the ATPG function with the -Cell option and added constraints with the -Cell option, then the tool also analyzes the constraints on all the cells affected by that ATPG function.

- -Bus

An optional switch that specifies for the tool to consider bus contention prevention during the ATPG process.

Related Topics

[Add Atpg Constraints](#)

Analyze Bus

Scope: Drc mode

Analyzes the specified bus gates for contention problems.

Usage

ANalyze BUs {*gate_id#*... [-Exclusivity | -Prevention | -Zstate]} |
-Drc_check | -All | -Auto | -ANalyze_sequentially

Description

If the bus passes the analysis, the tool displays a message indicating that it did so. If the analysis aborts, the tool displays a message that identifies the tri-state drivers (TSDs) the tool was analyzing at abort time. If the bus fails the analysis, the tool displays a message that identifies the two offending TSDs (the tri-state drivers capable of being on simultaneously, while driving different values).

Note



After using the Analyze Bus command, the [Set Gate Report Parallel_pattern 0](#) command may display conflicting values if failures are found. That is, it is possible to have TIE-X gates with a 1/0 as an output. This is due to the Analyze Bus command setting values to determine contention states. This analysis continues until a conflict is found.

The commands Set Contention Check On -Atpg and Set Iddq Checks -Bus both cause ATPG to ensure that every bus is contention free during deterministic test generation. Sometimes, this requirement cannot be met for many or all of the faults targeted by ATPG, preventing you from obtaining adequate fault coverage. When this happens, the Analyze Bus command determines which bus or buses cannot be made contention free, so that you can investigate the circuit around this bus to find out what is preventing contention- free tests.

When you issue this command, you must either specify a *gate_id#* value or one of the global switches (-Drc_check or -All).

Arguments

- *gate_id#* -Exclusivity | -Prevention | -Zstate

A repeatable integer with an optional switch that specifies the identification number of the bus gate and the type of analysis you want the tool to perform.

The available switch choices are as follows:

- Exclusivity — An optional switch that specifies for the tool to analyze the bus gate to see if it has mutual exclusivity. Mutual exclusivity means that only one driver can simultaneously force a strong signal onto the bus. Exclusivity is the default behavior when you specify a *gate_id#* value without a corresponding switch.
- Prevention — An optional switch that specifies for the tool to analyze the bus gate for its ability to attain a state of non-contention.

-Zstate — An optional switch that specifies for the tool to analyze the bus gate for its ability to attain a high-impedance (Z) state.

- **-Drc_check**

A switch that specifies for the tool to run the design rule check process again to categorize all buses and display the results. This is useful if you are changing constraints or the abort limit in an attempt to pass bus checks rather than abort.

- **-ALI**

A switch that specifies for the tool to use the more extensive ATPG process to place all the fail and abort buses in a noncontentious state. The internal simulation data for this pattern is available in parallel pattern 0.

- **-AUto**

A switch that automatically tries to locate the bus that cannot be made contention free. The analysis checks to see if any single bus cannot be made contention free. Each bus that cannot be made contention free (given the current abort limit and other restrictions) is reported to the user. Sometimes, each bus can be satisfied by itself, but some set of buses cannot all be satisfied. In this case, -Auto switch proceeds to a second analysis where it creates a minimized set of buses that can't be satisfied. The final, reduced set of buses, which cannot all be made contention free, is reported.

- **-ANalyze_sequentially**

A switch that specifies to perform a sequential analysis to find bus contention problems. For each clock defined using the [Add Clocks](#) command, the sequential analysis applies a 1-cycle test, then a 2-cycle test, and so on, up to the reporting depths for which the tests fail. It then analyzes and lists the specific buses causing the failures at those depths.

Examples

The following example analyzes a bus that failed the regular bus contention checking:

```
set system mode atpg
analyze bus 493
```

Related Topics

[Report Bus Data](#)

[Set Gate Level](#)

[Set Contention Check](#)

[Set Gate Report](#)

Analyze Contention

Scope: Atpg mode

Specifies for the tool to analyze contention buses and ports.

Usage

```
ANalyze COntention [{net_path_name | cell_name | gate_id#}... | -Bus | -Port | -All]
[-AUto | -Static] [-Noconstrain] [-Xstate]
```

Description

When bus/port contention checking is turned on, FlexTest checks if it is possible to avoid bus and port contention under the current ATPG constraints. If no successful result is found, the tool generates a warning message. No information about the buses/ports that cause the problem is displayed. The Analyze Contention command helps you debug contention buses and ports. This command enables the tool to automatically identify a sub-set of buses/ports that cannot avoid contention simultaneously during ATPG.

Arguments

- **net_path_name**
The net path name of the bus to be analyzed by the tool for bus contention.
- **cell_name**
The state element specified with multiple ports to be analyzed by the tool for port contention.
- **gate_id#**
The gate ID of the bus to be analyzed by the tool.
- **-Bus**
An optional switch that specifies for the tool to analyze contention of tri-state driver buses.
- **-Port**
An optional switch that specifies for the tool to analyze contention of multiple-port flip-flops and latches.
- **-All**
An optional switch that specifies for the tool to analyze contention of both tri-state driver buses and multiple-port flip-flops and latches.
- **-AUto**
An optional switch that automatically tries to locate a subset of buses/ports that cannot avoid contention simultaneously.
- **-Static**
An optional switch that specifies for the tool to not use the ATPG engine to analyze the contention problem. Instead, the analysis is carried out by using local implication.

- **-Noconstrain**

An optional switch that specifies for the tool not to consider ATPG constraints during contention analysis. If this switch is not specified, all ATPG constraints will be considered during analysis.

- **-Xstate**

An optional switch that forces all non-scan state elements as uncontrollable during analyzing contention.

Examples

The following example analyzes contention of both tri-state driver buses and multiple-port flip-flops and latches:

```
analyze contention -all
```

```
// Contention can be avoided.
```

Related Topics

[Set Bus Handling](#)

[Set Contention Check](#)

[Report Bus Data](#)

Analyze Control Signals

Scope: All modes

Identifies and optionally defines the primary inputs of control signals.

Usage

ANalyze COntrl Signals [-Report_only | -Auto_fix] [-Verbose]

Description

The Analyze Control Signals command identifies each control signal (clocks, set, reset, write-control, read-control, and so on) of every sequential element (DFF, latch, RAM, ROM, etc.) and optionally defines its primary input as a control signal. The purpose of analyzing the control signals is to identify the primary inputs that need to be defined as a clock, read-control, or write-control. This analysis also considers pin constraints, but only traces through simple combinational gates.

If the -Verbose option is specified, the tool issues messages that indicate why certain control signals are not reported. At the end of the analysis, the tool displays statistical information such as the number of primary inputs identified as control signals, their types, and additional information.

If the -Auto_fix option is specified, all identified primary inputs of control signals are automatically defined. For example, when a clock is identified, an implicit Add Clocks command is performed to define the primary input. By default, all control signals are only reported.

Note



The tool defers to your expert knowledge when you use “add clocks” to define a clock; the -Auto_fix option will not alter a clock definition specified by a preceding [Add Clocks](#) command.

Note



This command will perform the flattening process automatically, if executed prior to performing flattening.

Arguments

- -Report_only

An optional literal that specifies to identify control signals only (does not define the primary inputs as control signals). This is the invocation default.

- -Auto_fix

An optional literal that specifies to define the primary inputs of all identified control signals as control signals. For example, when a clock is identified, an implicit Add Clocks command is performed to define the primary input.

- -Verbose

An optional literal that specifies to display information on control signals (whether they are identified or not, and why) while the analysis is performed.

Examples

The following example analyzes the control signals, then only provides a verbose report on the control signals in the design. After examining the transcript, you can then perform another analysis of the control signals to add them.

analyze control signals -verbose

```
// command: analyze control signals -reports_only -verbose
-----

// Begin control signals identification analysis.
-----
// Warning: Clock line of '/cc01/tim_cc1/add1/post_latch_29/WRITEB_reg/r/
// (7352)' is uncontrolledat '/IT12 (4)'.
...
...
...
// Identified 2 clock control primary inputs.
//      /IT23 (5) with off-state = 0.
//      /IT12 (4) with off-state = 0.
// Identified 0 set control primary inputs.
// Identified 1 reset control primary inputs.
//      /IRST (1) with off-state = 0.
// Identified 0 read control primary inputs.
// Identified 0 write control primary inputs.
-----
// Total number of internal lines is 105 (35 clocks, 35 sets , 35 resets,
// 0 reads, 0 writes).
// Total number of controlled internal lines is 25 (17 clocks, 0 sets ,
// 8 resets, 0 reads, 0 writes).
// Total number of uncontrolled internal lines is 80 (18 clocks, 35 sets,
// 27 resets, 0 reads, 0 writes).
// Total number of added primary input controls 0 (0 clocks, 0 sets ,
// 0 resets, 0 reads, 0 writes).
-----
```

analyze control signals -auto_fix

Related Topics

[Add Clocks](#)

[Report Clocks](#)

[Add Read Controls](#)

[Report Read Controls](#)

[Add Write Controls](#)

[Report Write Controls](#)

Analyze Fault

Scope: Atpg mode

Performs an analysis to identify why a fault is not detected.

Usage

For FlexTest

```
ANALyze FAult pin_pathname {-Stuck_at {0 | 1}} [-Observe gate_id#]  
[-Time integer] [-Continue]
```

Description

The Analyze Fault command performs an analysis to identify why the fault that you specify was not detected. You can use the -Observe switch to specify the observe point for the sensitization analysis.

The fault analysis performed by the Analyze Fault command consists of the following actions:

1. A message is given if the selected fault has been nofaulted.
2. A message is given that will identify if the fault is in the current fault list. If the fault is in the current fault list and the fault is the representative member, its fault classification is displayed.
3. If the fault was not identified as included in the active fault list, the basic fault analysis is performed that determines if the fault can be classified as unused, tied, blocked, or detected by implication.
4. If the fault category is detected by simulation, detected by implication, or unused, a message is given and the analysis is terminated. You can override the termination by using the -Continue switch.
5. If the fault category is tied, all sources of the tied condition are identified and the analysis is terminated.
6. If the fault category is blocked, all blockage points (100 maximum) are identified. For each blockage point, all sources of the tied conditions causing the blockage are identified. The analysis then terminates.
7. The states that result from all constrained pins and stable non-scan cells are calculated.
8. If the fault site is now prevented from attaining the necessary state, a message is given indicating the fault is tied by constrained logic, all sources of the tied condition are identified, and the analysis then terminates.
9. An analysis is made to identify all blockage points (100 maximum) and all potential detection points (25 maximum).

10. If there are no potential detection points, the blockage points are identified and for those points blocked by tied logic, all sources of the tied condition are identified. The analysis then terminates.
11. If there were potential detection points, the detection points are identified (25 maximum).
12. A controllability test generation is performed to determine if the fault site can be controlled. If successful, the test generation values are displayed using `parallel_pattern 0`. If unsuccessful, the analysis then terminates.
13. If an observe point is not selected, a complete test generation is attempted where the fault is sensitized from the fault site to any unblocked point. Potential problem points in any sensitization path are identified. The points will include tri-state driver enable lines, transparent latch data lines, `clock_pos`, wire gates, latch and flip-flop set/reset/clock lines, RAM write/read/address/data lines, and ROM read/address lines. If the test generation is successful, the test generation values are displayed using `parallel_pattern 1`.
14. If an observe point is selected, the fault is sensitized from the fault site to the observe point. Potential problem points in the sensitization path are identified. If the sensitization is successful, the test generation values are displayed using `parallel_pattern 1`.

Note



Only stuck-at, toggle, and transition fault types can be analyzed using the Analyze Fault command.

Arguments

- ***pin_pathname***

A required string that specifies the pin pathname of the fault where you want to perform the analysis.

- **`-Stuck_at 0 | 1`**

A switch and literal pair that specifies the stuck-at fault value that you want to analyze. The stuck-at values are as follows:

0 — A literal that specifies that the tool analyze the *pin_pathname* for a “stuck-at-0” fault.

1 — A literal that specifies that the tool analyze the *pin_pathname* for a “stuck-at-1” fault.

- **`-Observe gate_id#`**

An optional switch and integer pair that specifies the observe point for the sensitization analysis.

gate_id# — An integer that specifies a gate identification number whose location you want to use as the observe point for the sensitization analysis.

- -Time *integer*

An optional switch and integer pair that specifies the pin strobe time for the sensitization analysis. Pin strobes always occur at the end of a timeframe, and the time value of a pin strobe is always the timeframe number+1.

- -Continue

An optional switch that forces the tool to complete the analysis of faults which have already been detected by the pattern set. This lets you inspect the generated pattern by using the Report Faults command.

If you do not specify this switch and the tool detects the fault category by simulation, by implication, or as unused, then the tool displays a message and terminates the analysis.

Related Topics

[Report Faults](#)

[Report Testability Data](#)

Analyze Race

Scope: Atpg, Good, and Fault modes

Checks for race conditions between the clock and data signals.

Usage

ANALyze RAcE [Edge | Level | Both] [-Warning | -Error]

Description

FlexTest is a zero delay simulator, which means that to achieve accurate simulation results, the data and clock signals of each sequential device cannot simultaneously change state. You can change the data capturing default behavior for race conditions with the Set Race Data command.

You can prevent race conditions by constraining the clock and data signals to the appropriate values with the Add Pin Constraints command. When you exit Setup mode, you can check to see if your added pin constraints adequately prevent race conditions with the Analyze Race command.

Arguments

- EDge
An optional literal that performs race analysis on edge-triggered sequential devices (flip-flops). This is the command default.
- Level
An optional literal that performs race analysis on level-sensitive sequential devices (latches).
- Both
An optional literal that performs race analysis on both the edge-triggered and level-sensitive sequential devices.
- -Warning
An optional switch that specifies for FlexTest to display a warning message for each possible race contention. This is the command default.
- -ERror
An optional switch that specifies for FlexTest to display an error message for the first race condition it encounters and then stop the simulation. You can use the Report Gates command with the Set Gate Report commands Race option to investigate the cause of the race condition error.

Examples

The following example checks and displays the results of possible race conditions:

analyze race edge -warning

```
// No race conditions found at timeframe '0' with all clocks  
off  
// Warning: 'I_3_16/DFF1/(107)' with type 'DFF' may have race  
condition at port 2 at timeframe 0 with the clock 'CLK' on  
// Warning: 'I_14_16/DFF1/(141)' with the 'DFF' may have race  
condition at port 2 at timeframe 0 with the clock 'CLK' on  
// No race conditions found at timeframe '0' with clock 'CLR'  
on
```

Related Topics

[Report Gates](#)

[Set Gate Report](#)

[Set Race Data](#)

Compress Patterns

Scope: Atpg mode

Compresses patterns in the current test pattern set.

Usage

```
COMpress PATterns [passes_integer] [-Force] [-MAx_useless_passes integer]  
[-MIn_elim_per_pass number] [-EFfort {LOW | MEdium | HIgh | MAMaximum}]
```

Description

The Compress Patterns command performs static pattern compression on the current test pattern set by repeating fault simulation for the patterns in either reverse or random order and selecting only those patterns required for detection. The *passes_integer* argument specifies the number of pattern compression passes. The first pattern compression pass runs in reverse order and then alternates between random and reverse for additional passes. If you do not specify a *passes_integer* argument, the tool performs only one compression pass.

You may only use the Compress Patterns command for combinational circuits or scan circuits. For scan circuits, FlexTest assumes all the non-scan cells will not hold their values during loading. By default, FlexTest does not allow pattern compression for scan circuits that contain any non-scan cells having Hold capability during scan operation. This is because the results may change due to the reordering of the test patterns causing reduced fault coverage. You can override the default by using the -Force option to compress these patterns, but the results may change. You should understand the impact of this option when deciding whether or not to use this option.

The Compress Patterns command has a residual memory effect. The initial mode (reverse/random) is not fixed. Instead, it toggles back and forth, starting with the mode last used in the same run. You must quit and restart FlexTest to begin with the same compaction mode.

Arguments

- *passes_integer*
An optional integer that specifies the number of pattern compression passes. The default is 1 (performs only one compression pass).
- -MAx_useless_passes *integer*
An optional switch and integer pair that specifies the maximum number of consecutive, useless (no eliminated patterns) passes the tool allows before terminating the pattern compression process. This command option has no effect on the pattern set if the number of passes (*passes_integer* argument) is smaller than the *integer* value of this switch. The default is the *passes_integer* value.

- **-MIn_elim_per_pass** *integer*

An optional switch and integer pair that specifies the minimum number of eliminated patterns required in a single pass to continue the pattern compression process. If you specify this switch, you must enter a value greater than 0.

- **-E**ffort** Low** | MEdium | H**I**gh | M**A**ximum

An optional switch and literal pair that specifies which kind of compression strategy for the tool to use. If you use the Low option, the tool uses the original reverse and random strategy. Plus, you can specify other Compress Patterns parameters. If you use any other -Effort option, all other Compress Patterns parameters will be ignored. The higher the effort level selected, the more complex the strategy. The -Effort switch with the Low option is default when no compression strategy is specified.

Examples

The following example compresses the generated test pattern set with two passes; the first pass is by reverse order and the second pass is by random order:

```
set system mode atpg  
add faults -all  
run  
compress patterns 2
```

Delete Atpg Constraints

Scope: Atpg, Good, and Fault modes

Prerequisites: Pin constraints must be added with the Add Atpg Constraints command.

Removes state restrictions from the specified pins. You define pin restrictions by using the Add Atpg Constraints command.

Usage

```
DELEte ATpg Constraints {pin_pathname | net_pathname | gate_ID# | {function_name |  
  {-Cell cell_name {pin_name | net_name}...}}... [-Instance {object_expression...}]  
[-MODULE {module_name...}]]... | -All
```

Arguments

- *pin_pathname*
A repeatable string specifying the pathname of the pin from which to remove any ATPG pin constraints.
- *net_pathname*
A repeatable string specifying the pathname of the net from which to remove any ATPG net constraints.
- *gate_ID#*
A repeatable integer specifying the gate identification number of the gate from which to remove any ATPG pin constraints.
- *function_name*
A repeatable string specifying the name of a function created with the Add Atpg Functions command and from which to remove any ATPG pin constraints.
- -Cell *cell_name* {*pin_name* | *net_name*}
A repeatable switch and string pair that removes ATPG constraints from the specified pin or net on the DFT library cell. You can repeat the *pin_name* or *net_name* argument if there are multiple pins or nets on a cell that have ATPG constraints you need to remove.
- -Instance *object_expression*
An optional switch and repeatable string pair that removes ATPG constraints inside the specified list of instances. You can use regular expressions, which may include any number of embedded asterisk (*) and/or question mark (?) wildcard characters. You can only use this switch when using the -Cell option or when a function name is specified.
- -MODULE *module_name*
An optional switch and repeatable string pair that removes the ATPG constraints inside all instances of the specified module.
- -All
A switch that removes all current, user-defined ATPG constraints from all objects.

Examples

Example 1

The following example creates two user-defined ATPG pin constraints, runs the ATPG process, removes all ATPG constraints, and then compresses the pattern set:

```
set system mode atpg
add atpg functions and_b_in and /i$144/q /i$141/q /i$142/q
add atpg constraints 0 /i$135/q
add atpg constraints 1 and_b_in
add faults -all
run
delete atpg constraints -all
compress patterns
```

Example 2

The following example shows how to use wildcards.

```
add atpg constraints 1 -cell mux21 Y -ins uALU/*/ix184 .
report atpg constraints .

1 dynamic /uALU/u3/ix184/Y (4400) .
1 dynamic /uALU/u2/ix184/Y (4406) .
1 dynamic /uALU/u1/ix184/Y (4412) .

delete atpg constraints -cell mux21 Y -ins uALU/u1/* .
report atpg constraints .

1 dynamic /uALU/u3/ix184/Y (4400) .
1 dynamic /uALU/u2/ix184/Y (4406) .
```

Related Topics

[Add Atpg Constraints](#)

[Report Atpg Constraints](#)

[Add Atpg Functions](#)

Delete Atpg Functions

Scope: Atpg, Fault, and Good modes

Prerequisites: You can only delete functions added with the Add Atpg Functions command.

Removes the specified function definitions.

Usage

DELEte ATpg Functions *function_name*... | -All

Description

The Delete Atpg Functions command lets you delete ATPG functions defined with the Add Atpg Functions command. You cannot remove an ATPG function if an ATPG constraint is currently using that function. If you attempt to remove an in-use function, the tool generates an error. Therefore, if you need to delete an in-use ATPG function, you must first remove all the associated ATPG constraints using the Delete Atpg Constraints command; then you can remove the ATPG function.

You can display a list of the current ATPG functions that the tool is using as ATPG constraints by using the Report Atpg Constraints command.

Arguments

- *function_name*
A repeatable string that specifies the names of the ATPG functions that you want to delete.
- -All
A switch that removes all ATPG function definitions.

Examples

The following example creates two user-defined ATPG functions, one user-defined ATPG constraint, displays the currently-in-use ATPG constraints, and then removes one of the inactive ATPG functions:

```
add atpg functions and_b_in And /i$144/q /i$141/q /i$142/q
add atpg functions select_b_in select /i$144/q /i$142/q
add atpg constraints 0 /i$135/q
report atpg constraints

0 /$135/Q (23)
```

```
delete atpg functions and_b_in
```

Related Topics

[Add Atpg Functions](#)

[Report Atpg Constraints](#)

[Delete Atpg Constraints](#)

[Report Atpg Functions](#)

Delete Black Box

Scope: Setup mode

Undoes the effect of the Add Black Box command.

Usage

DELeTe BLaCk BoX **-Instance** [*ins_pathname*] | **-Module** [*module_name*] | **-All**

Description

For a module that was originally modeled, removing the effect of the Add Black Box command reinstates the original model. Specified tied values are no longer set on the output pins. For a module that was empty or undefined in the input netlist, the output pins revert to the default tied value of X.

Note



The tool releases the flattened model if one exists at the time you issue this command.

Arguments

- **-Instance** [*ins_pathname*]
A required switch that specifies for the tool to undo the effect of the Add Black Box command on all instance-based blackboxes. This is the default if no *ins_pathname* is given. You can optionally specify an instance pathname to undo a single instance-based blackbox.
- **-Module** [*module_name*]
A required switch that specifies for the tool to undo the effect of the Add Black Box command on all module-based blackboxes. This is the default if no *module_name* is given. You can optionally specify a module name to undo a single module-based blackbox.
- **-All**
A required switch that specifies for the tool to undo the effect of the Add Black Box command on all blackboxes.

Examples

The following example adds the black box for module core then undoes all blackboxes that were defined.

```
add black box -module core 1
delete black box -all
```

Related Topics

[Add Black Box](#)

[Report Black Box](#)

[Delete Tied Signals](#)

Delete Cell Constraints

Scope: Setup mode

Prerequisites: You can only delete constraints added with the Add Cell Constraint command.

Removes constraints placed on scan cells.

Usage

For FlexTest

DELEte Cell Constraints *pin_pathname* | {*chain_name cell_position*} | -All

Description

The Delete Cell Constraints command deletes the constraints placed on scan cells using the Add Cell Constraint command. You can specify a scan cell by using either a pin pathname or a position in a scan chain.

Arguments

- *pin_pathname*
A string that specifies the name of an output pin which directly connects to a scan memory element (you can only specify output pins of buffers and inverters). Valid in Setup mode only.
- *chain_name cell_position*
A string and integer pair that specifies the name of a currently-defined scan chain and the position of the cell in the scan chain. The *cell_position* is an integer where 0 is the scan cell closest to the scan-out pin. Valid in Setup mode only.
- -All
A switch that, in Setup mode, specifies to delete all constraints from all scan cells.

Examples

The following example deletes an incorrectly added cell constraint placed on a scan cell:

```
add clocks 1 clock1
add scan groups group1 proc.g1
add scan chains chain1 group1 scanin1 scanout1
add scan chains chain2 group1 scanin2 scanout2
add cell constraints chain1 5 c0
add cell constraints chain2 3 c1
delete cell constraints chain2 3
add cell constraints chain2 4 c1
report cell constraints
```

Related Topics

[Add Cell Constraints](#)

[Report Cell Constraints](#)

Delete Clocks

Scope: Setup mode

Prerequisites: You can only delete primary input pin names added with the Add Clocks command.

Removes primary input pins from the clock list.

Usage

DELEte CLocks *primary_input_pin...* | **-All**

Description

The Delete Clocks command deletes primary input pins from the clock list. If you delete an equivalence pin, the command also deletes all of the equivalent pins from the clock list.

Arguments

- *primary_input_pin*
A repeatable string that specifies the primary input pins to delete from the clock list.
- **-All**
A switch that deletes all pins from the clock list.

Examples

The following example deletes an incorrectly added clock from the clock list:

```
add clocks 1 clock1
add clocks 1 clock2
delete clocks clock1
```

Related Topics

[Add Clocks](#)

[Report Clocks](#)

Delete Cone Blocks

Scope: Setup mode

Prerequisites: You must add output pins names to the clock and effect cone list with the Add Cone Blocks command before you can delete them.

Removes the specified output pin names from the user-created list which the tool uses to calculate the clock and effect cones.

Usage

DELEte COne Blocks *pin_pathname*... | **-All**

Description

The Delete Cone Blocks command deletes output pins added to the tool's internal clock and effect cone list with the Add Cone Blocks command. The tool uses these output pins as blockage points for calculating clock and effect cones. You can generate a report on the current output pins in the user-defined list by using the Report Cone Blocks command.

Arguments

- *pin_pathname*
A repeatable string that specifies the output pin pathname that you want the tool to remove from the user-defined list that it uses when calculating the clock and effect cones.
- **-All**
A switch that removes all the pins from the user-defined list. Unless you create new user-specified blockages with the Add Cone Blocks command, the tool returns to using the output pins it chooses by default for the clock and effect cone calculations.

Examples

The following example shows adding and removing cone blockages:

```
add cone blocks /ls0/q
report cone blocks
```

```
both /LS0/Q
```

```
delete cone blocks /ls0/q
add cone blocks /ls0/q -clock
report cone blocks
```

```
clock /LS0/Q
```

Related Topics

[Add Cone Blocks](#)

[Report Cone Blocks](#)

Delete Contention Free_bus

Scope: All modes

Specifies which if any contention free buses are deleted.

The Delete Contention Free_bus command lets you delete all or specific contention free buses.

Usage

DELeTe COntention Free_bus [-ALl | *net_name* | *gate_id*]

Arguments

- **-ALl**
A switch that specifies for the tool to delete all contention free buses.
- *net_name*
An optional repeatable string that specifies the name of the contention free bus to delete.
- *gate_id*
An optional repeatable string that specifies the gate identification number of the contention free bus to delete.

Examples

The following example deletes a contention free bus.

```
delete contention free_bus bus1  
run
```

Related Topics

[Add Contention Free_bus](#)

[Set Contention Check](#)

[Report Contention Free_bus](#)

[Set Contention_bus Reporting](#)

[Report Gates](#)

[Set Gate Report](#)

[Set Bus Handling](#)

Delete Faults

Scope: Atpg, Fault, and Good modes

Prerequisites: Faults must be added with the [Add Faults](#) or [Load Faults](#) commands.

Removes faults from the current fault list.

Usage

```
DELEte Faults {object_pathname... | -All | -Untestable}  
[-Stuck_at {01 | 0 | 1}] [{> | >>} file_pathname]
```

Description

The Delete Faults command deletes from the fault list, faults you added using the Add Faults or Load Faults command. Alternatively, you can also delete a large number of non-protected faults using the [Load Faults](#) commands's -Delete switch.

When you issue this command, the tool discards all patterns in the current test pattern set. To save the current test patterns you must explicitly save them with the [Save Patterns](#) command prior to issuing the Delete Faults command.

You cannot use a single command to delete faults for clock domains and specific classes, objects, or paths. Use a separate command instance for each.

Arguments

- **-All**
A required switch that deletes all faults in the current fault list.
- ***object_pathname***
A required, repeatable string that specifies a list of instances or pins whose faults you want deleted from the current fault list.
- **-Untestable**
An optional switch that switch deletes all identified, untestable faults. Untestable faults are common when using random patterns. This includes faults that the tool cannot detect due either to constraints or the use of a single capture clock.
If you use actual instance or pin names instead of wildcards, then this switch has no effect.
- **-Stuck_at 01 | 1 | 0**
An optional switch and literal pair that specifies the stuck-at or transition faults to delete from the fault list.
 - 01** — A literal specifying that for stuck-at faults the tool delete both stuck-at-0 and stuck-at-1 faults; or for transition faults the tool delete both slow-to-rise and slow-to-fall faults. This is the default.
 - 0** — A literal specifying to delete only the stuck-at-0 faults (slow-to-rise faults for transition faults).

1 — A literal specifying to delete only the stuck-at-1 faults (slow-to-fall faults for transition faults).

- *>file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- *>>file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Delete Initial States

Scope: Setup mode

Prerequisites: You must add initial state settings with the Add Initial States command before you can delete them.

Removes the initial state settings for the specified instance names.

Usage

DELEte INitial States *instance_pathname...* | **-All**

Description

The Delete Initial States command deletes the initial state settings added using the Add Initial States command. You can display a list of the current initial state settings by using the Report Initial States command.

Arguments

- *instance_pathname*
A repeatable string that specifies the pathnames of the design hierarchical instances that have initial state settings that you want to remove.
- **-All**
A switch that removes all the initial states created with the Add Initial States command.

Examples

The following example creates two initial state settings and then removes one:

```
add initial states 0 /amm/g30/ff0 /amm/g29/ff0
delete initial states /amm/g30/ff0
```

Related Topics

[Add Initial States](#)

[Write Initial States](#)

[Report Initial States](#)

Delete Lists

Scope: Atpg, Fault, and Good modes

Removes pins from the pin list the tool monitors and reports on during simulation.

Usage

DELeTe Lists *pin_pathname...* | **-All**

Description

The Delete Lists command removes from the monitored pin list, pins that you previously added to the list using the [Add Lists](#) command. To review the current list of pins, use the [Report Lists](#) command. To put additional pins on the list, use the Add Lists command.

Arguments

- *pin_pathname*
A repeatable string that specifies the pins you want to remove from the monitored pin list.
- **-All**
A switch that removes all currently listed pins from the monitored pin list.

Examples

The following example removes an extra added output pin from the monitored pin list:

```
add lists /i_1006/o /i_1007/o /i_1008/o
report lists

3 pins are currently monitored.
/i_1006/o
/i_1007/o
/i_1008/o
```

```
delete lists /i_1007/o
report lists

2 pins are currently monitored.
/i_1006/o
/i_1008/o
```

Related Topics

[Add Lists](#)

[Set List File](#)

[Report Lists](#)

Delete Nofaults

Scope: Setup mode

Removes the nofault settings from either the specified pin or instance/module pathnames.

Usage

```
DELEte NOfaults {-All | {modulename... -Module} | {object_expression... [-PIN |  
-Instance]}} [-Stuck_at {01 | 0 | 1}] [-Class {User | System | Full}] [-VERbose] [{> | >>}  
file_pathname]
```

Description

The Delete Nofaults command deletes nofault settings you added using the Add Nofaults command.

You can optionally specify nofault settings that have a specific stuck-at or transition value. For the latter, you specify 0 for “slow-to-rise” and 1 for “slow-to-fall” transition faults. If you do not specify a stuck-at (transition) value when deleting a nofault setting, the command deletes both stuck-at (transition) nofault settings.

You can also optionally specify nofault settings that have a specific class code: user-defined, system netlist, or both. If you do not specify a class code, then the command deletes the nofault setting from the user class.

You can use the Report Nofaults command to display all the current nofault settings.

Arguments

- **-All**
A switch that deletes all nofault settings.
- ***modulename***
A repeatable string that specifies the name(s) of the module(s) from which you want to delete nofault settings. You must include the -Module switch when you specify a module name.
- **-Module**
A switch that specifies interpretation of the ***modulename*** argument as a module pathname. All instances of these modules are affected. You can use the asterisk (*) and question mark (?) wildcards for the ***modulename*** argument, and the tool deletes the nofault for all matching modules or library models.
- ***object_expression***
A string representing a list of pathnames of instances or pins from which you want to delete nofault settings. You can use regular expressions, which may include any number of asterisk (*) and question mark (?) wildcard characters.

Pin pathnames must be ATPG library cell instance pins, also referred to as design level pins. If the object expression specifies a pin within an instance of an ATPG library model, the tool ignores it. By default, pin pathnames are matched first. If a pin pathname match is not found, the tool next tries to match instance pathnames. You can force the tool to match only pin pathnames or only instance pathnames by including the `-Pin` or `-Instance` switch after the *object_expression*.

- `-Pin`

An optional switch that specifies to use the preceding object expression to match only pin pathnames; the tool will then delete nofault settings from all the pins matched.

- `-Instance`

An optional switch that specifies to use the preceding object expression to match only instance pathnames; the tool will then delete nofault settings from all boundary and internal pins of the instances matched.

- `-Stuck_at 01 | 0 | 1`

An optional switch and literal pair that specifies the stuck-at or transition values from which you want to remove a nofault setting. The choices are as follows:

01 — A literal specifying that for stuck-at faults the tool delete both “stuck-at-0” and “stuck-at-1” nofault settings; or for transition faults the tool delete both “slow-to-rise” and “slow-to-fall” nofault settings. This is the default.

0 — A literal that deletes only the “stuck-at-0” nofault settings (“slow-to-rise” nofault settings for transition faults).

1 — A literal that deletes only the “stuck-at-1” nofault settings (“slow-to-fall” nofault settings for transition faults).

- `-Class User | System | Full`

An optional switch and literal pair that specifies the source (or class) of the nofault settings which you want to delete. The valid literals are as follows:

User — A literal that deletes the user-entered nofault settings. This is the default.

System — A literal that deletes netlist-based nofault settings.

Full — A literal that deletes all the nofault settings in the user and system classes.

- `-VERbose`

By default when you specify wildcard characters for instance (`-Instance`) or pin (`-PIN`) names, the tool outputs a summary message similar to the following:

```
// Note: Adding faults for 330 fault sites.
```

When you specify the optional `-VERbose` switch, then the tool outputs the instance or pin names instead of the summary. You can optionally redirect this output to a file using the `>` or `>>` redirection operators.

If you use actual instance or pin names instead of wildcards, then this switch has no effect.

- *>file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- *>>file_pathname*
An optional redirection operator and pathname pair, used at the end o

Examples

The following example deletes nofault settings from the pins of instance i_1007 and then adds all faults to the circuit, thereby allowing the tool to add faults to the pins of the i_1007 instance:

```
add nofaults i_1006 i_1007 i_1008 -instance  
delete nofaults i_1007 -instance  
set system mode atpg  
add faults -all
```

Related Topics

[Add Faults](#)

[Report Faults](#)

[Add Nofaults](#)

[Report Nofaults](#)

[Delete Faults](#)

Delete Nonscan Handling

Scope: Setup mode

Removes the overriding, learned behavior classification for the specified non-scan elements.

Usage

DELeTe NOncan Handling *element_pathname*... | **-All** [-Instance | -Module]

Description

The Delete Nonscan Handling command deletes the overriding learned behavior classification created with the Add Nonscan Handling command. Any non-scan element, from which you remove the handling, reverts back to having the design rules checker classify its learned behavior.

To list the current, overriding, learned behavior classifications for the non-scan elements use the Report Nonscan Handling command.

Arguments

- *element_pathname*
A repeatable string that specifies the pathnames to the non-scan element for which you want to remove any user-defined learned behavior classifications.
- **-All**
A switch that removes all the user-defined learned behavior classifications.
- **-Instance**
An optional literal that specifies that the *element_pathname(s)* specified are instance pathnames. This is the default.
- **-Module**
An optional literal that specifies that the *element_pathname(s)* specified are module names. All instances with the specified modules are affected by this command as well as the Add Nonscan Handling command.

Examples

The following example first explicitly defines how FlexTest is to handle two non-scan elements, then removes one of those definitions, and finally reports on the current list of learned behavior overrides for the design rules checker:

```
add nonscan handling tie0 i_6_16 i_28_3
delete nonscan handling i_28_3
report nonscan handling

TIE0  I_6_16
```

Related Topics

[Add Nonscan Handling](#)

[Set Nonscan Model](#)

[Report Nonscan Handling](#)

Delete Output Masks

Scope: Setup mode

Prerequisites: You must add primary output pin masks with the Add Output Masks command before you can delete them.

Removes the masking of the specified primary output pins.

Usage

DELEte OUtput Masks *primary_output...* | **-All**

Description

The tools use primary output pins as the observe points during the fault detection process. When you mask a primary output pin with the Add Output Masks command, the tools mark that pin as an invalid primary output during the fault detection process.

Arguments

- *primary_output*
A repeatable string that specifies the names of the primary output pins that you want to unmask.
- **-All**
A switch that unmask all primary outputs masked using the Add Output Masks command.

Examples

The following example first incorrectly chooses two of the design's primary output pins to mask. The example then unmask the one primary output that was inappropriate, masks the correct primary output, and then displays the complete list of currently-masked primary output pins no longer available as observation points:

```
add output masks q1 qb3
delete output masks q1
add output masks qb1
report output masks

qb1
qb3
```

Related Topics

[Add Output Masks](#)

[Set Output Masks](#)

[Report Output Masks](#)

Delete Pin Constraints

Scope: Setup mode

Prerequisites: You must add pin constraints with the Add Pin Constraints command before you can delete them.

Removes the pin constraints from the specified primary input pins.

Usage

DELEte PIn Constraints *primary_input_pin...* | **-All**

Description

The Delete Pin Constraints command deletes pin constraints added to the primary inputs with the Add Pin Constraint command. You can delete the pin constraints for specific pins or for all pins.

Primary inputs that do not have any constraints use the default format of type NR, period 1, and offset 0. You can change the default format by using the Setup Pin Constraints command.

Arguments

- *primary_input_pin*
A repeatable string that specifies a list of primary input pins whose pin constraints you want to delete.
- **-All**
A switch that deletes the pin constraints of all primary input pins.

Examples

The following example adds two pin constraints and then deletes one of them:

```
add pin constraint indata2 c1
add pin constraint indata4 c1
delete pin constraints indata2
```

Related Topics

[Add Pin Constraints](#)

[Setup Pin Constraints](#)

[Report Pin Constraints](#)

Delete Pin Equivalences

Scope: Setup mode

Prerequisites: You must add equivalences with the Add Pin Equivalences command before you can delete them.

Removes the pin equivalence specifications for the designated primary input pins.

Usage

DELEte PIn Equivalences *primary_input_pin...* | **-All**

Description

The Delete Pin Equivalences command deletes the equivalence specifications added to the primary inputs with the Add Pin Equivalences command. You can delete pin equivalences for specific pins or for all pins.

Arguments

- *primary_input_pin*
A repeatable string that specifies a list of primary input pins whose equivalence specifications you want to delete.
- **-All**
A switch that deletes all pin equivalence effects.

Examples

The following example deletes an incorrect pin equivalence specification and adds the correct one:

```
add pin equivalences indata2 -invert indata4
delete pin equivalences indata2
add pin equivalences indata3 -invert indata4
```

Related Topics

[Add Pin Equivalences](#)

[Report Pin Equivalences](#)

Delete Pin Strobes

Scope: Setup mode

Prerequisites: You must add strobe times with the Add Pin Strobes command before you can delete them.

Removes the strobe time from the specified primary output pins.

Usage

DELEte PIn Strobes *primary_output_pin...* | **-All**

Description

The Delete Pin Strobes command deletes the strobe time added to the primary outputs using the Add Pin Strobes command. You can delete the strobe time of specific pins or of all pins.

Once you delete a primary output pin's strobe time, the pin uses the default strobe time. For nonscan circuits, the default strobe time is the last timeframe of each test cycle. For scan circuits, FlexTest designates time 1 of each test cycle as the default strobe time for every primary output. You can change the default strobe time by using the Setup Pin Strobes command.

Arguments

- *primary_output_pin*
A repeatable string that specifies a list of primary output pins whose strobe times you want to delete.
- **-All**
A switch that deletes the strobe times of all primary outputs; the pins then use the default strobe time.

Examples

The following example deletes the strobe time of a primary output pin:

```
set test cycle 3
add pin strobes 1 outdata1 outdata2 outdata3
delete pin strobes outdata2
```

The pin then takes on the default strobe time value.

Related Topics

[Add Pin Strobes](#)

[Setup Pin Strobes](#)

[Report Pin Strobes](#)

Delete Primary Inputs

Scope: Setup mode

Removes the specified primary inputs from the current netlist.

Usage

DELEte PRimary Inputs {*net_pathname...* | *primary_input_pin...* | **-All**}
[-Class {User | System | Full}]

Description

The Delete Primary Inputs command deletes from a circuit the primary inputs that you specify. You can delete either the user class, system class, or full classes of primary inputs. If you do not specify a class, the tool deletes the primary inputs from the user class.

You can display a list of any class of primary inputs by using the Report Primary Inputs command.

Note



Once you delete a primary input pin with this command, adding the pin back again with the Add Primary Inputs command will not create the same pin. Bus information is lost and the pin will be marked as user-added. Take care to use Delete Primary Inputs only if you wish to truly delete a pin from the design so it is not used.

Arguments

- ***net_pathname***
A repeatable string that specifies the circuit connections that you want to delete. You can specify the class of primary inputs to delete with the -Class switch.
- ***primary_input_pin...***
A repeatable string that specifies a list of primary input pins that you want to delete. You can specify the class of primary inputs to delete with the -Class switch.
- **-All**
A switch that deletes all primary inputs. You can specify the class of primary inputs to delete with the -Class switch.
- **-Class User | System | Full**
An optional switch and literal pair that specifies the class code of the designated primary input pins. The valid class code literal names are as follows:
 - User — A literal specifying that you added the primary inputs using the Add Primary Inputs command. This is the default class.
 - System — A literal specifying that the primary inputs derive from the netlist.

Full — A literal specifying that the primary inputs consist of both user and system classes.

Examples

The following example deletes an extra added primary input from the user class of primary inputs:

```
add primary inputs indata2 indata4 indata6  
delete primary inputs indata4 -class user
```

Related Topics

[Add Primary Inputs](#)

[Write Primary Inputs](#)

[Report Primary Inputs](#)

Delete Primary Outputs

Scope: Setup mode

Removes the specified primary outputs from the current netlist.

Usage

DELEte PRImary Outputs {*net_pathname...* | *primary_output_pin...* | **-All**}
[-Class {User | System | Full}]

Description

The Delete Primary Outputs command deletes from a circuit the primary outputs that you specify. You can delete either the user class, system class, or full classes of primary outputs. If you do not specify a class, the tool deletes the primary outputs from the user class.

You can display a list of any class of primary outputs by using the Report Primary Outputs command.

Note



Once you delete a primary output pin with this command, adding the pin back again with the Add Primary Outputs command will not create the same pin. Bus information is lost and the pin will be marked as user-added. Take care to use Delete Primary Outputs only if you wish to truly delete a pin from the design so it is not used.

Arguments

- ***net_pathname***

A repeatable string that specifies the circuit connections that you want to delete. You can specify the class of primary outputs to delete with the -Class switch.

- ***primary_output_pin***

A repeatable string that specifies a list of primary output pins that you want to delete. You can specify the class of primary outputs to delete with the -Class switch.

- **-All**

A switch that deletes all primary outputs. You can specify the class of primary outputs to delete with the -Class switch.

- **-Class User | System | Full**

An optional switch and literal pair that specifies the class code of the primary output pins that you specify. The valid literal names are as follows:

User — A literal specifying that the list of primary outputs were added using the Add Primary Outputs command. This is the default class.

System — A literal specifying that the list of primary outputs derive from the netlist.

Full — A literal specifying that the list of primary outputs consists of both the user and system class.

Examples

The following example deletes a primary output from the system class of primary outputs:

```
delete primary outputs outdata1 -class system
```

Related Topics

[Add Primary Outputs](#)

[Write Primary Outputs](#)

[Report Primary Outputs](#)

Delete Read Controls

Scope: Setup mode

Prerequisites: You must add read control lines with the Add Read Controls command before you can delete them.

Removes the read control line definitions from the specified primary input pins.

Usage

DELEte REad Controls *primary_input_pin...* | **-All**

Description

The Delete Read Controls command deletes read control lines defined with the Add Read Controls command. You can delete the read control line definitions for specific pins or for all pins.

Arguments

- *primary_input_pin*
A repeatable string that specifies a list of primary input pins from which you want to delete any read control line definitions.
- **-All**
A switch that deletes the read control line definitions for all primary input pins.

Examples

The following example deletes an incorrect read control line, then redefines that read control line with the correct off-state:

```
add read controls 0 r1 r2
delete read controls r1
add read controls 1 r1
set system mode atpg
```

Related Topics

[Add Read Controls](#)

[Report Read Controls](#)

Delete Scan Chains

Scope: Setup mode

Prerequisites: You must add scan chains with the Add Scan Chains command before you can delete them.

Removes the specified scan chain definitions from the scan chain list.

Usage

DELEte SCan Chains *chain_name...* | **-All**

Description

The Delete Scan Chains command deletes scan chains defined with the Add Scan Chains command. You can delete the definitions of specific scan chains or of all scan chains.

Arguments

- *chain_name*
A repeatable string that specifies the names of the scan chain definitions that you want to delete.
- **-All**
A switch that deletes all scan chain definitions.

Examples

The following example defines several scan chains, adding them to the scan chain list, then deletes one of the scan chains:

```
add scan chains chain1 group1 indata2 outdata4
add scan chains chain2 group1 indata3 outdata5
add scan chains chain3 group1 indata4 outdata6
delete scan chains chain2
```

Related Topics

[Add Scan Chains](#)

[Report Scan Chains](#)

Delete Scan Groups

Scope: Setup mode

Prerequisites: You must add scan chain groups with the Add Scan Groups command before you can delete them.

Removes the specified scan chain group definitions from the scan chain group list.

Usage

DELEte SCan Groups *group_name*... | **-All**

Description

The Delete Scan Groups command deletes scan chain groups defined with the Add Scan Groups command. You can delete the definitions of specific scan chain groups or of all scan chain groups.

When you delete a scan chain group, the tool also deletes all scan chains within the group.

Arguments

- *group_name*
A repeatable string that specifies the names of the scan chain group definitions that you want to delete.
- **-All**
A switch that deletes all the scan chain group definitions.

Examples

The following example defines two scan chain groups, adding them to the scan chain group list, then deletes one of the scan chain groups:

```
add scan groups group1 scanfile1
add scan groups group2 scanfile2
delete scan groups group1
```

Related Topics

[Add Scan Groups](#)

[Report Scan Groups](#)

Delete Scan Instances

Scope: Setup mode

Prerequisites: You must add sequential instances with the Add Scan Instances command before you can delete them.

Removes the specified sequential instances from the scan instance list.

Usage

DELEte SCan Instances *instance_pathname...* | **-All**

Description

The Delete Scan Instances command deletes sequential instances added to the scan instance list with the Add Scan Instances command. You can delete a specific list of instances or all the instances.

Arguments

- *instance_pathname*
A repeatable string that specifies the pathnames of the instances that you want to delete from the scan instance list.
- **-All**
A switch that deletes all instances from the scan instance list.

Examples

The following example deletes an extra sequential scan instance marked for treatment as a scan cell from the scan instance list:

```
set system mode setup
add scan instances i_1006 i_1007 i_1008
delete scan instances i_1007
```

Related Topics

[Add Scan Instances](#)

[Report Scan Instances](#)

Delete Scan Models

Scope: Setup mode

Removes the specified sequential models from the scan model list.

Usage

DELeTe SCan Models *model_name...* | **-All**

Description

The Delete Scan Models command deletes all instances of the specified sequential models. You can delete a specific list of sequential models or all the models.

To display the current scan model list use the Report Scan Models command.

Arguments

- *model_name*
A repeatable string that specifies the model names that you want to delete from the scan model list. Enter the model names as they appear in the design library.
- **-All**
A switch that deletes all models from the scan model list.

Examples

The following example deletes an extra added sequential scan model from the scan model list:

```
set system mode identification
add scan models d_flip_flop1 d_flip_flop2
delete scan models d_flip_flop2
```

Related Topics

[Add Scan Models](#)

[Report Scan Models](#)

Delete Tied Signals

Scope: Setup mode

Removes the assigned (tied) value from the specified floating nets or pins.

Usage

DELeTe Tied Signals {*floating_object_name...* | **-All**} [-Class {User | System | Full}] [-Pin]

Description

The Delete Tied Signals command deletes the tied values assigned with the Add Tied Signals command. You can delete tied values from either user class, system class, or full classes of floating nets or pins. If you do not specify a class, the tool deletes the tied values from the user class of floating nets or pins. You can display a list of any class of tied floating nets or pins by using the Report Tied Signals command.

Whenever you delete tied values from nets or pins, be sure to re-add any necessary values before performing another simulation. If you do not add required tied values to floating nets or pins, the tool displays a warning. The warning states that the design has floating nets or pins and assumes they are tied to the default value; you must set the default value using the Setup Tied Signals command.

Arguments

- *floating_object_name*

A repeatable string that specifies the names of the tied floating nets or pins whose tied values you want to delete. You can specify the class of floating nets or pins on which to delete the tied values with the -Class switch.

If you do not specify the -Pin option, the tool assumes *floating_object_name* is a net name. If you specify a full net pathname, the tool deletes only the specified instance-based blackbox tied signal. If you specify the -Pin option, it assumes the *floating_object_name* is a pin name.

- **-All**

A switch that deletes the tied values from all tied floating nets or pins in the class of tied floating nets or pins, which you specify with the -Class switch. This also includes all instance-based blackbox tied signals.

- -Class User | System | Full

An optional switch and literal pair that specifies the class code of the tied floating nets or pins that you specify. The valid literal names are as follows:

User — A literal specifying that the tied floating nets or pins were added by using the Add Tied Signals command. This is the default class.

System — A literal specifying that the tied floating nets or pins derive from the netlist.

Full — A literal specifying that the tied floating nets or pins consist of both user and system classes.

- -Pin

A switch specifying that the *floating_object_name* argument that you provide is a floating pin name.

Examples

The following example deletes the tied value from the user-class tied net “vcc”; thereby leaving “vcc” as a floating net:

```
add tied signals 1 vcc vdd
delete tied signals vcc -class user
```

Related Topics

[Add Tied Signals](#)

[Report Tied Signals](#)

[Delete Black Box](#)

[Setup Tied Signals](#)

Delete Write Controls

Scope: Setup mode

Prerequisites: You must add write control lines with the Add Write Controls command before you can delete them.

Removes the write control line definitions from the specified primary input pins.

Usage

DELEte WRite Controls *primary_input_pin...* | **-All**

Description

The Delete Write Controls command deletes write control lines defined with the Add Write Controls command. You can delete the write control line definitions for specific pins or for all pins.

Arguments

- *primary_input_pin*
A repeatable string that specifies a list of primary input pins from which you want to delete any write control line definitions.
- **-All**
A switch that deletes the write control line definitions for all primary input pins.

Examples

The following example deletes an incorrect write control line, then re-adds that write control line with the correct off-state:

```
add write controls 0 w1 w2
delete write controls w1
add write controls 1 w1
set system mode atpg
```

Related Topics

[Add Write Controls](#)

[Report Write Controls](#)

Dofile

Scope: All modes

Executes the commands contained within the specified file.

Usage

DOfile *filename* [-History]

Description

The Dofile command sequentially executes the commands contained in a specified file. This command is especially useful when you must issue a series of commands. Rather than executing each command separately, you can place them into a file in their desired order and then execute them by using the Dofile command. You can also place comment lines in the file by starting the line with a double slash (//); the tool handles these lines as comments and ignores them.

The Dofile command sends each command expression (in order) to the tool which in turn displays each command line from the file before executing it. If the tool encounters an error due to any command, the Dofile command stops its execution, and displays an error message. You can enable the Dofile command to continue regardless of errors by setting the Set Dofile Abort command to Off.

Arguments

- *filename*
A required string that specifies the name of the file that contains the commands that you want the tool to execute.
- -History
An optional switch that specifies for the tool to add the commands from a dofile to the command line history list. By default, the commands in a dofile are not inserted into the history list, but the dofile command itself is added to the list.

Examples

The following example executes, in order, all the commands from the file, *command_file*:

dofile command_file

The *command_file* may contain any application command available. An example of a *command_file* is as follows:

```
set system mode atpg
add faults -all
run
```

Related Topics

[History](#)

[Set Dofile Abort](#)

[Save History](#)

Exit

Scope: All modes

Terminates the application tool program.

Usage

EXIt [-Force]

Description

The Exit command terminates the tool session and returns to the operating system. You should either save the current test patterns before exiting the tool or specify the -Force switch to not save the test patterns.

If you are operating in interactive mode (not running a dofile) and you neither saved the current test pattern set nor used the -Force option, the tool displays a warning message, and you are given the opportunity to continue the session and save the test patterns before exiting.

Arguments

- -Force

An optional switch that explicitly specifies to not save the current test pattern set and to immediately terminate the tool session.

Examples

The following example quits the tool without saving the current test pattern set.

```
set system mode atpg
add faults -all
run
exit -force
```

Find Design Names

Scope: All modes

Displays design object hierarchical names matched by an input regular expression, which may include asterisk (*) or question mark (?) wildcard characters in the pathname string.

Usage

```
FINd DEsign Names regular_expression [-LOcal | -Hier]  
[-Design | -NETList | -Llbrary | -All]  
[-INStance | -Net | -Pin [INPut | OUtput | INOut | ALLIn | ALLOut]  
-Cell | -Module]  
[ {> | >>} file_pathname ]
```

Arguments

- ***regular_expression***
A required, regular expression, which may include asterisk (*) or question mark (?) wildcard characters.
- **-LOcal**
An optional switch that matches wildcard characters within the current hierarchy level.
- **-Hier**
An optional switch that matches the regular expression across hierarchy boundaries (for example, a* matches a1/b/c). This is the default.
- **-Design**
An optional switch that matches only pathnames to objects at the topmost library cell level.
- **-NETList**
An optional switch that matches objects from the top of the design down to the topmost library cell.
- **-Llbrary**
An optional switch that matches objects within any level of library cells.
- **-All**
An optional switch that specifies matches objects at all levels of the design. This is the default.
- **-INStance**
An optional switch that matches only instance pathnames. This is the default.
- **-Net**
An optional switch that matches only net pathnames.

- **-Pin**
An optional switch that matches only pin pathnames (any pin direction). The following optional pin filters restrict which pins are matched:
 - INPut — Match only input pin pathnames.
 - OUtput — Match only output pin pathnames.
 - INOut — Match only bidirectional pin pathnames.
 - ALLIn — Match both input and bidirectional pin pathnames.
 - ALLOut — Match both output and bidirectional pin pathnames.
- **-Cell**
An optional switch that finds all library cell (model) names matching the specified regular expression.
- **-Module**
An optional switch that finds all netlist module names matching the specified regular expression.
- **> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- **>> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following examples display object pathnames for various input wildcard expressions, given a netlist with the following instance hierarchy:

```

/
  tiny_i
    U5
  ret_i
    intreg1_reg_0 ... intreg1_reg_31
    add_20
      U1_0 ... U1_3
    add_30
      U5 ... U12
    mul_18
      U5 ... U868
      FS
        U5 ... U33
    mul_19
      FS
        U5 ... U278
        U5 ... U181
    mul_22
      U5 ... U735
      FS
        U15 ...

```

and assuming the U5 instances all reference the following library cell:

```

model LSR2BUFA(Q, QN, S, R, G, SD, RD) (
  input(S, R, G, SD, RD) ()
  output(Q) (primitive = _buf UP1 (QT, Q);)
  output(QN) (primitive = _buf UP2 (QNT, QN);)
  intern(QT_int) (instance = LSI_LSR2 UD1 (QT_int, S, R, G, SD, RD);)
  intern(QNT_int) (instance = LSI_LSR2N UD2 (QNT_int, S, R, G, SD, RD);)
  intern(QT) (instance = LSI_NOTI UD3 (QT, QT_int);)
  intern(QNT) (instance = LSI_NOTI UD4 (QNT, QNT_int);)
)

```

Example 1

SETUP> find design names /ret_i/add_2* -instance -design -hier

```

// Note: Matched 4 names
/ret_i/add_20/U1_0
/ret_i/add_20/U1_1
/ret_i/add_20/U1_2
/ret_i/add_20/U1_3

```

Example 2

SETUP> find design names /ret_i/add_2* -instance -netlist -hier

```
// Note: Matched 5 names  
/ret_i/add_20  
/ret_i/add_20/U1_0  
/ret_i/add_20/U1_1  
/ret_i/add_20/U1_2  
/ret_i/add_20/U1_3
```

Finds instance add_20 under /ret_i/, and also descends the hierarchy to find all netlist instances under /ret_i/add_20/.

Example 3

```
SETUP> find design names /ret_i/add_2* -inst -netlist -local
```

```
// Note: Matched 1 names  
/ret_i/add_20
```

This example shows that -Local does not descend the hierarchy to find more matches as the previous example does.

Example 4

```
SETUP> find design names /ret_i/add_2* -ins -design -local
```

```
// Note: Matched 0 names
```

There are no instances of a library cell under /ret_i/ with instance name starting with add_2.

Example 5

```
SETUP> find design names /ret_i/*_2? -ins -netlist -local
```

```
// Note: Matched 2 names  
/ret_i/add_20  
/ret_i/mul_22
```

Found 2 instances under /ret_i/.

Note



/ret_i/gt_68_2 did not match because the “?” in the wildcard expression requires another character after the “_2”.

Example 6

```
SETUP> find design names */U5 -inst -design -hier
```

```
// Note: Matched 7 names
/tiny_i/U5
/ret_i/add_20/U5
/ret_i/mul_18/U5
/ret_i/mul_18/FS/U5
/ret_i/mul_19/U5
/ret_i/mul_19/FS/U5
/ret_i/mul_22/U5
```

Example 7

SETUP> find design names ret_i/mul*/U5 -ins -des -local

```
// Note: Matched 3 names
/ret_i/mul_18/U5
/ret_i/mul_19/U5
/ret_i/mul_22/U5
```

Example 8

SETUP> find design names ret_i/mul*/U5 -ins -design -hier

```
// Note: Matched 5 names
/ret_i/mul_18/U5
/ret_i/mul_18/FS/U5
/ret_i/mul_19/U5
/ret_i/mul_19/FS/U5
/ret_i/mul_22/U5
```

Example 9

SETUP> find design names ret_i/mul_18/U5* -ins -library -hier

```
// Note: Matched 5 names
/ret_i/mul_18/U5
/ret_i/mul_18/U5/UD1
/ret_i/mul_18/U5/UD2
/ret_i/mul_18/U5/UD3
/ret_i/mul_18/U5/UD4
```

Example 10

SETUP> find design names ret_i/mul*/U5/* -pin output -design -local

```
// Note: Matched 6 names
/ret_i/mul_18/U5/Q
/ret_i/mul_18/U5/QN
/ret_i/mul_19/U5/Q
/ret_i/mul_19/U5/QN
/ret_i/mul_22/U5/Q
/ret_i/mul_22/U5/QN
```

Flatten Model

Scope: Setup mode

Creates a primitive gate simulation representation of the design.

Usage

FLAtten MOdel

Description

The tool automatically flattens the design hierarchy down to the logically equivalent design when you exit Setup mode. However, there may be times that you would like to access the flattened model without having to exit Setup mode. For example, you may want to add ATPG constraints and functions before you exit Setup mode.

If you exit Setup mode and then add ATPG constraints and functions, the design rule checker does not have access to those ATPG constraints during the rule checking. If you issue the Flatten Model command in Setup mode and then add those ATPG constraints, the design rule checker has access to them during the rule checking.

Arguments

None.

Examples

The following example shows flattening the design to the simulation primitives before adding constraints that the rule checker then uses when you run the design rule checker. The rule checker runs when you first attempt to exit Setup mode

```
flatten model
add atpg functions and_b_in and /i$144/q /i$141/q /i$142/q
add atpg constraints 0 /i$135/q
add atpg constraints 1 and_b_in
set system mode atpg
```

Related Topics

[Add Atpg Constraints](#)

[Set System Mode](#)

[Delete Atpg Functions](#)

Help

Scope: All modes

Displays the usage syntax and system mode for the specified command.

Usage

HElp [*command_name*] [-MANual]

Description

The Help command displays useful information for a selected command. You can display the usage and syntax of a command by typing Help and the command name. You can display a list of certain groups of commands by entering Help and a keyword such as Add, Delete, Set, and so on.

Arguments

- *command_name*
An optional string that either specifies the name of the command for which you want help or specifies one of the following keywords whose group of commands you want to list: ADD, DELEte, SET, SETUp, or WRite. If you do not supply a *command_name*, the default display is a list of all the valid command names.
- -MANual
An optional string that specifies to also display the reference manual description for the specified command.

If you type HEUp and include only the -MANual switch, the tool opens the product bookcase, giving access to all the manuals for that product group.

Examples

The following example displays the usage and system mode for the Report Primary Inputs command:

help report primary inputs

```
// Report primary inputs
//   usage: REPort PRimary Inputs [-Class <User|System|Full>]
//                                   [-All | pin_pathname...]
//   legal system modes: ALL
```

History

Scope: All modes

Displays a list of previously-executed commands.

Usage

HIStory [*list_count*] [-Nonumbers] [-Reverse]

Description

The History command is similar to the Korn shell (ksh) history command in UNIX. By default, this command displays a list of all previously-executed commands, including all arguments associated with each command, starting with the oldest.

Note



The HISTFILE and HISTSIZE ksh environment variables do not control the command history of the tool. The Save History command controls where the tool stores the history file.

You can perform command line editing if you set the VISUAL or EDITOR ksh environment variable to either emacs, gmacs, or vi editing. Please see the ksh(1) man page for specifics on the various editing modes.

A leading number precedes each command line in the history list that indicates the order in which the commands were entered.

Arguments

- *list_count*
An optional integer that specifies for the tool to display only the specified number (*list_count*) of most recent executed commands. If no *list_count* is specified, the tool displays all previously executed commands.
- -Nonumbers
An optional string that specifies for the tool to display the history list without the leading numbers. This is useful for creating dofiles. The default displays the leading numbers.
- -Reverse
An optional switch that specifies for the tool to display the history list starting with the most recent command rather than the oldest.

Examples

The following command displays the history list with leading numbers, starting with the oldest command.

history

```
1  help hist
2  dof instructor/fault.do
3  set system mode atpg
4  set fault type stuck
5  add faults -all
6  run
7  report statistics
8  report faults -class ATPG_UNTESTABLE
9  analyze fault /I$20/en -stuck_at 1
10 set system mode setup
11 set system mode atpg
12 set fault type iddq
13 add faults -all
14 run
15 report statistics
16 history
```

Related Topics

[Save History](#)

Load Faults

Scope: Atpg, Fault, and Good modes

Updates the current fault list with the faults contained in the specified fault file.

Usage

LOAd Faults *filename* [-Restore | -Delete] [-Column *integer*]

Description

The Load Faults command affects the current fault population by either adding or removing faults which you specify in an external fault file. Because you must identify the faults before performing ATPG or Fault simulation, this command is useful when you have a large number of faults to identify.

The format of the fault file data can be either in a three-, four-, or six-column standard format. Regardless of the format, the Load Faults command uses only the information in the first, second, and last columns. You may also place comment lines in the file by starting the line with a double slash (//); the tool ignores comment lines. The file follows the format illustrated below:

```
stuck fault_class (cellname) (netname) (ignore) pin_pathname
```

- **stuck** — The first column must be the stuck-at value.
- **fault_class** — The second column must be the fault class value, but only if you use the -Retain option (-Restore in FlexTest). For detailed information on fault classes, refer to the “[Fault Classes](#)” section in the *Scan and ATPG User’s Manual*.
- **cellname** — The third column is the cell name enclosed in parenthesis. When present, this column indicates the type of cell in which the fault resides.
- **netname** — The fourth column is the net name enclosed in parenthesis. When present, this column indicates the net in which the fault resides.
- **ignore** — The fifth column is ignored by both tools.
- **pin_pathname** — The last column must be the pin pathname.

Here is an example of a short fault file that provides just the data the tool uses (as described in the first, second, and last bullets above):

```
// Short fault file example
1 UC /u510/u17/A0
1 UC /u510/u17/Y
0 UU /u510/u17/A1
1 UU /u510/u17/A1
```

When you issue this command, the tool discards all patterns in the current test pattern.

Arguments

- *filename*

A required string that specifies the name of the ASCII file containing the fault list to load. Only one string can be entered.

- -Restore

An optional switch that specifies for the tool to retain the fault class of each fault that is in the fault list.

When you read in a fault class and try to maintain the fault classes within the fault file, the following rules apply:

- If the fault class is EQ, the tool uses the fault class of the previous fault in the file.
- If a fault class code is not valid, the tool considers the fault class to be UC.
- After collapsing, the tool uses only the fault class found in the second column of the individual fault. When faults collapse together, there is no checking to ensure that they have the same fault class.
- If the tool analyzes a fault to be unused, tied, blocked, or detected-by-implication, the tool places the fault in that class independent of the fault class found in the second column of the fault file.
- You may use multiple loads to create the internal fault population list. If you load a fault that already exists in the current fault population list, the command uses the new value for the fault code, and the tool does not issue a warning message.

- -Delete

An optional switch that specifies for the tool to remove all the *filename* faults except the protected faults from the current fault population. To delete protected faults, you must use the [Delete Faults](#) command.

- -Column *integer*

An optional switch that specifies the column format of the fault file. *Integer* specifies the number of columns to be read by the tool. All other columns are ignored. If this switch is not specified, the tool determines the number of columns by counting the number of words, delimited by spaces, on each line.

Read Modelfile

Scope: Setup mode

Initializes the specified RAM or ROM gate using the memory states contained in the named modelfile or deletes an earlier modelfile assignment.

Usage

REAd MOdelfile {*modelfile_name* | **-Delete**} *RAM/ROM_instance_name*

Description

The Read Modelfile command sets the initial memory states of a RAM or ROM gate using the data that you provide in a modelfile. You can create a modelfile from within the library cell or by using the Write Modelfile command. The modelfile must contain initialization data that is in a proprietary Siemens EDA modelfile format.

You specify the RAM or ROM gate that you want to initialize by using its instance name. An error condition occurs if the instance contains multiple RAM/ROM gates. When you issue the command, the flattened model may not be available, so the tool checks for the correctness of the instance name and the modelfile name during rules checking.

You may also initialize memory states of a RAM or ROM gate by specifying the modelfile from within the RAM or ROM model description. To do so, use the `init_file` attribute. When using the `init_file` attribute, you must specify the full pathname to the initialization file if the file is not located in the directory from where you invoked the tool. For more information about modeling RAMs and ROMs and the `init_file` attribute, refer to the “[RAM and ROM](#)” subsection of the *Tessent Cell Library Manual*.

Modelfile Format

A Siemens EDA modelfile contains addresses and data. You must present the addresses in hexadecimal format. You can specify a range of addresses such as 0-1f. An address range may contain an asterisk (*) wildcard character. For example, to specify that you want all addresses set to a hexadecimal F, use “* / f;”. You cannot use an X in an address.

To add comments to a RAM or ROM initialization file, precede the comment text with two forward slashes (//). For example:

```
0/3;  
// here is a comment  
1/2;
```

You can present the data in either binary or hexadecimal format; the default is hexadecimal. To specify data in binary format, you must add a ‘%’ to the beginning of the data values. If you use an X within hexadecimal data, all four bits that it represents are Xs. Therefore, to set a single bit to X, use the binary format.

The following two examples are equivalent. The first example shows both an address and its associated data in hexadecimal. The second example shows the same address and data, but the data is now shown in binary.

```
ABCD / 123X;  
  
ABCD /%000100100011XXXX;
```

The following is an example of what an initialization file may look like (range 0-1f):

```
0/ a;  
1-f / 5;  
10 / 1a;  
11-1f / a;
```

You can use an asterisk (*) for an address range. For example, you could rewrite the previous initialization file as:

```
* / a;  
1-f / 5;  
10 / 1a;
```

As you can see, the first line assigns the data value “a” to the full address range, 0-1f. The subsequent lines overwrite the “a” data value with the new data values for the specified addresses.

Pin order is position-dependent. Any order is acceptable as long as the pins match up in position-dependent fashion.

Arguments

- ***modelfile_name***

A string that specifies the pathname to and filename of the modelfile containing the RAM or ROM initialization data a proprietary Siemens EDA modelfile format.

- **-Delete**

A switch that deletes the modelfile initialization assignment you made previously for the specified RAM or ROM gate instance. Use this switch to undo an incorrectly specified modelfile or instance name (due, for example, to a misspelling or a non-existent modelfile or instance).

Note



If you misspelled the RAM/ROM_instance_name when you first assigned the modelfile, you need to use the same spelling when you delete the assignment.

After you remove the incorrect modelfile initialization assignment using the -Delete switch, you can re-issue the Read Modelfile command using the corrected modelfile or instance name.

- ***RAM/ROM_instance_name***

A required string that specifies the instance name of the RAM or ROM gate that you want to initialize, or for which you wish to delete a previous modelfile initialization assignment.

Examples

Example 1

The following example initializes the memory states of a RAM gate, then performs an ATPG run:

```
read modelfile model.ram /p2.ram
set system mode atpg
add faults -all
run
```

If the memory primitive specified in the Read Modelfile command did not exist, then the Set System Mode Atpg command would result in an error message similar to the following:

```
// Error : /p2.ram given in an earlier Read Modelfile command is NOT a RAM
or ROM primitive.
```

Example 2

The next example fixes the error by using the -Delete switch to remove the incorrect entry from the tool's internal list of RAMs and ROMs to be initialized. The example then re-issues the original command using the correct instance name (assumed to be /p1.ram):

```
read modelfile -delete /p2.ram
read modelfile model.ram /p1.ram
```

Related Topics

[Write Modelfile](#)

Read Procfile

Scope: All modes except Setup mode

Reads a test procedure file and merges the timing and named capture procedure data.

Usage

REAd PProcfile *proc_filename*
[-Append_or_timing_update | -Replace]

Description

Reads a test procedure file and merges the timing and named capture procedure data it contains with existing data loaded from previous test procedure files as follows:

- Replaces each existing timeplate with the definition in *proc_filename* for that timeplate.
- Adds any new timeplates defined in *proc_filename*.
- Updates the timing (timeplates) used by existing scan procedures (like test_setup, load_unload and shift) to match the definitions of these procedures in *proc_filename*.
- Merges the named capture procedure (NCP) data in *proc_filename* with NCP data that exists in the tool's database as summarized in the following table.


Table 2-5. Named Capture Procedure Merging Under Different Conditions

Condition	-Append_or_timing_update	-Replace
New NCPs ¹ (no NCP in tool has the same name)	Adds new NCPs	Deletes existing NCPs and adds NCPs from <i>proc_filename</i>
NCPs of same name exist in tool but are not used ²	Updates existing NCP's timing only	Deletes existing NCPs and adds NCPs from <i>proc_filename</i>
NCPs of same name exist in tool and are used ²	Updates existing NCP's timing only	Invalid; results in an error message.
No NCP in <i>proc_filename</i>	Does not alter existing NCPs	Deletes existing NCPs if not used

1. NCP = named capture procedure

2. Used means at least one pattern in the internal or external pattern database uses the NCP.

Tip

 : A good practice is to regularly use the [Write Procfile](#) command to write existing procedures and timing data to a test procedure file when you are experimenting with procedures and timing. This allows you to preserve the most up-to-date procedures and timing data in a form you can use in future runs simply by updating the [Add Scan Groups](#) command in your dofile. It also enables you to recover quickly if existing procedures and timing are altered in an undesirable way.

Arguments

- ***proc_filename***

A required string that specifies the pathname or filename of the test procedure file to read.

- ***-Append_or_timing_update***

An optional switch that does the following using the data in ***proc_filename***:

- Adds any new NCPs in ***proc_filename*** to the tool's existing database of NCPs.

The tool treats as new, any NCP in ***proc_filename*** whose name is not identical to that of an NCP in the tool's existing database. An added NCP is available immediately for ATPG.

- Update the timing (timeplate) of existing NCPs and other procedures as applicable, without altering them in any other way.

This is the default.

Note



An NCP in ***proc_filename*** and its counterpart of the same name in the tool's internal database must have the same event sequence. If their event sequences differ, the command is aborted.

- ***-Replace***

An optional switch that deletes all existing NCPs in the tool's internal database and replaces them with the NCPs if any in ***proc_filename***, provided a current internal or external pattern set does not contain a pattern that uses an existing NCP. If even one existing NCP is used by a pattern, none of them are deleted or replaced. If ***proc_filename*** does not contain any NCPs, the existing NCPs are deleted and not replaced, provided a current pattern does not use any of them.

Examples

Example 1

Test procedure file *orig.proc* defines a simple timeplate as well as a shift, load_unload, and named capture procedure cap1. The following example reads in the procedure file, then lists the current timeplates and procedures stored in the tool's database (for brevity, only the parts of the timeplates and procedures relevant to this example are shown):

```
add scan groups grp1 orig.proc
set system mode atpg
report timeplate

timeplate gen_tpl =
    ...
end;
```

```
report procedure

procedure shift =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    ...
end;

procedure load_unload =
    scan_group grp1 ;
    timeplate gen_tp1 ;
    ...
end;

procedure capture cap1 =
    timeplate gen_tp1 ;
    ...
end;
```

Example 2

Assume a second test procedure file *update.proc* defines a new timeplate in addition to the original one, revises the shift procedure to use the new timeplate, and defines a new named capture procedure cap2. The following example adds the new timeplate, updates the timing in the scan procedures, and adds the new named capture procedure. The timeplate and procedure changes resulting from the Read Procfile command are highlighted in bold in the output of the reporting commands.

read procfile update.proc

```
// The following occurred at line 46 in file update.proc:
// Procedure shift updates timing in same procedure from file orig.proc.
//      (W14)
// Loaded new procedure file successfully.
// 1 new capture procedure is added.
// 2 total capture procedures are in the system.
// 2 capture procedures are active for test generation.
```

report timeplate

```
timeplate gen_tp1 =
    ...
end;

// Timeplate gen_tp1 has an old timeplate
// timeplate gen_tp1 =
//      ...
// end;

timeplate gen_tp2 =
    ...
end;
```

report procedure

```
procedure shift =  
    scan_group grp1 ;  
    timeplate gen_tp2 ;  
    ...  
end;  
  
procedure load_unload =  
    scan_group grp1 ;  
    timeplate gen_tp1 ;  
    ...  
end;  
  
procedure capture cap1 =  
    timeplate gen_tp1 ;  
    ...  
end;  
  
procedure capture cap2 =  
    timeplate gen_tp2 ;  
    ...  
end;
```

If the Read Procfile command of the preceding example had included the -Replace switch, the original NCP would have been replaced with the new one:

read procfile update.proc -replace

```
// The following occurred at line 46 in file update.proc:  
// Procedure shift updates timing in same procedure from file orig.proc.  
// (W14)  
// Loaded new procedure file successfully.  
// 1 new capture procedure is added.  
// 1 total capture procedure is in the system.  
// 1 capture procedure is active for test generation.
```

report procedure

```
procedure shift =  
    scan_group grp1 ;  
    timeplate gen_tp2 ;  
    ...  
end;  
  
procedure load_unload =  
    scan_group grp1 ;  
    timeplate gen_tp1 ;  
    ...  
end;  
  
procedure capture cap2 =  
    timeplate gen_tp2 ;  
    ...  
end;
```

Related Topics

[Add Scan Groups](#)

[Save Patterns](#)
[Report Procedure](#)
[Write Procfile](#)
[Report Timeplate](#)

Report Aborted Faults

Scope: Atpg, Good, and Fault modes

Displays information on testable faults that remain undetected (UD) after the ATPG process.

Usage

For FlexTest

REPort ABorted Faults [*format_type*]

Description

The Report Aborted Faults command can help you determine why the tool aborted faults in the undetected fault list. You can then analyze whether to change the current abort limit to possibly allow ATPG to generate tests for those faults before the tool aborts them. To change the abort limit, use the [Set Abort Limit](#) command.

Arguments

- *format_type*

An optional literal that specifies the type of information that you want the tool to display regarding the aborted faults. The literal choices for the *format_type* argument are as follows:

Summary — A literal that displays a summary of the number of aborted faults for each category. This is the default.

All — A literal that displays all the aborted faults that are currently in the undetected fault list.

Backtrack — A literal that displays all the aborted, undetected faults that exceeded the backtrack limit.

Cycle — A literal that displays all the aborted, undetected faults that exceeded the cycle limit

DEtected — A literal that displays all the faults that the tool aborted and then later detected.

Hypertrophic — A literal that displays all the aborted, undetected faults that later became hypertrophic faults.

INterrupt — A literal that displays all the undetected faults that the tool aborted because you interrupted the ATPG process with a Control-C.

Oscillatory — A literal that displays all the aborted, undetected faults that later became oscillatory faults.

Examples

The following example displays the default summary of all the aborted faults:

report aborted faults

```
10 backtrack
1  clock_restriction
2  time
```

Related Topics

[Report Faults](#)

[Set Atpg Limits](#)

[Set Abort Limit](#)

Report Atpg Constraints

Scope: All modes

Displays all the current ATPG state restrictions and the pins on which they reside.

Usage

REPort ATpg Constraints

Description

The Report Atpg Constraints command displays the pins and their state restrictions defined using the Add Atpg Constraints command. The tool uses the state restrictions (constraints) during the ATPG process.

Arguments

None.

Examples

The following example creates two ATPG pin constraints and then displays the information on those definitions:

```
add atpg functions and_b_in and /i$144/q /i$141/q /i$142/q
add atpg constraints 0 /i$135/q
add atpg constraints 1 and_b_in
report atpg constraints

0 /I$135/Q (23)
1 and_b_in
```

Related Topics

[Add Atpg Constraints](#)

[Delete Atpg Constraints](#)

Report Atpg Functions

Scope: All modes

Displays all the current ATPG function definitions.

Usage

REPort Atpg Functions

Description

The Report Atpg Functions command displays the definitions of the ATPG functions created using the Add Atpg Functions command. You can use an ATPG function as an argument to the Add Atpg Constraints command, which then lets you create state restrictions on pins that the tool uses during the ATPG process.

Arguments

None.

Examples

The following example creates two ATPG functions and then displays their definitions:

```
add atpg functions and_b_in and /i$143/q /i$141/q /i$142/q
add atpg functions select_b_in select /i$144/q /i$142/q
report atpf functions
```

```
USER_AND and_b_in
  Input 0:  /I$143/Q (27)
  Input 1:  /I$141/Q (23)
  Input 2:  /I$142/Q (25)
SELECT selet_b_in
  Input 0:  /I$144/Q (29)
  Input 1:  /I$142/Q (25)
```

Related Topics

[Add Atpg Constraints](#)

[Delete Atpg Functions](#)

[Add Atpg Functions](#)

Report Au Faults

Scope: Atpg and Fault modes

Displays information on ATPG untestable faults.

Usage

REPort AU Faults [Summary | All | TRistate | Tied_constraint | Blocked_constraint |
Uninitialized | Clock | Wire | Others]

Description

The Report Au Faults command helps to determine why faults in the undetected fault list were declared ATPG untestable.

Each of the subcategories in the AU fault class are mutually exclusive. The sequence of classification is as follows:

1. Wire
2. Tri-state
3. Clock
4. Tied_constraint
5. Blocked_constraint
6. Uninitialized
7. Others

Arguments

- Summary
An optional literal that specifies to display a summary of the number of AU faults for each category. This is the default.
- All
An optional literal that specifies to display all AU faults which include AU, UI, PU, HU, and OU faults.
- Tristate
An optional literal that specifies to display all AU faults that have a propagation path to the enable of a tristateable primitive which drives a bus (with no pullup, pulldown, or bus keeper) to establish a known, reliable voltage when all drivers of that bus are disabled. It is not possible to reliably observe the fault effect at the output of such a driver when a fault causes the driver to be off (when it is the only driver of the bus).

- **Tied_constraint**

An optional literal that specifies to display all AU faults whose fault site is held logically constant by a constraint.

For example, a stuck-at-0 fault on the output of an AND gate, which has one or more of its inputs constrained to 0 during test, is placed in the AU fault category. It is also reported in the Tied subcategory because the site of the fault is tied such that a test is impossible. A test for the fault would require a 1 on the line which is constrained to be 0 during test. The fault is in the AU (rather than TIED) category because the constraint may only exist in test mode.

- **Blocked_constraint**


An optional literal that specifies to display all AU faults that have observation paths blocked by constraints (pin constraints or ATPG constraints). These faults are in the Blocked subcategory of the AU fault category. There is also a Blocked fault category which includes faults that are blocked due to circuit connections to Vss, Vdd, and so on. The distinction is important because a stuck-at-0 fault on a line which has a Vss connection cannot cause the system operation to produce an incorrect result. Whereas, an ATPG constraint requiring a line to be 0, might represent a condition which only exists in test mode, and a stuck-a-0 fault on such a line may cause an incorrect result. This latter fault is classified in the AU fault category to indicate that although a test cannot be generated, the fault may still cause improper system operation.

- **Uninitialized**

An optional literal that specifies to display all the AU faults whose fault site is constrained such that fault excitation is not possible.

For example, a stuck-at-0 fault on the output of an AND gate which has one or more of its inputs constrained to unknown during test is placed in the AU fault category, and reported in the Uninitialized subcategory. This is because the site of the fault is constrained to be either 0 or X such that a test is impossible. A test for the fault would require a 1 on the line which is constrained to be 0 or X during test. The fault is in the AU category, because the constraint may only exist in test mode.

Note

 UI faults must be in Uninitialized subcategory, Tied_constraint subcategory or Others subcategory.

- **Clock**

An optional literal that specifies to display all AU faults that are faults which only propagate to the clock input of single-port sequential primitives, such as latch clock inputs and flip flop clock inputs.

- **Wire**

An optional literal that specifies to display all AU faults that propagate only to a wired net that does not have wire-and or wire-or behavior.

- Others

An optional literal that specifies to display any AU fault that does not belong to one of the specific categories (Blocked_constraint, Clock, Tied_constraint, Tristate, or Uninitialized).

Examples

Example 1

```
set system mode atpg
add faults -all
run
report au faults summary
```

Example 2

The following example displays the default summary of all the aborted faults:

```
report au faults

117 tristate
23 clock
4 blocked_constraint
9 uninitialized
3 tied_constraint
2 wire
11 others
```

Related Topics

[Add Faults](#)

[Delete Faults](#)

[Analyze Fault](#)

[Report Faults](#)

Report Black Box

Scope: All modes

Displays information on blackboxes and undefined models.

Usage

```
REPort BLack Box { {-Instance [ins_pathname]} | {-Module [module_name]} |  
  -All | -Undefined } [{> | >>} file_pathname]
```

Description

The Report Black Box command reports on the status of any instance- or module-based blackboxes, or undefined models that are not yet blackboxed. The report follows the following format.

```
MODULE: module_name (default tie value = 0|1|X|Z)  
  SYSTEM: Inout|Output pin pin_name tied to 0|1|X|Z  
  USER: Inout|Output pin pin_name tied to 0|1|X|Z  
INSTANCE: ins_name (default tie value = 0|1|X|Z)  
  SYSTEM: Inout|Output pin pin_name tied to 0|1|X|Z  
  USER: Inout|Output pin pin_name tied to 0|1|X|Z
```

For module-based blackboxes, the tool displays the string MODULE followed by the name of the module and the default tie value (0, 1, X, or Z). The tool then displays a list of module pins. For each pin, the tool displays either SYSTEM or USER followed by the direction type of the pin (Inout or Output), the name of the pin, and its tied value. SYSTEM declares that the pin is tied to the default value by the system while USER declares that you explicitly tied the pin to the specified value.

For instance-based blackboxes, the report replaces the string MODULE with INSTANCE to explicitly declare that it is an instance-based blackbox.

Arguments

- **-Instance** [ins_pathname]

A switch and optional string that specify for the tool to display information on instance-based blackboxes. If you do not supply an *ins_pathname*, the tool displays information on all instance-based blackboxes. If you specify an instance pathname, it reports on that single, instance-based blackbox.

- **-Module** [module_name]

A switch and optional string that specify for the tool to display information on module-based blackboxes. If you do not supply a *module_name*, the tool displays information on all module-based blackboxes. If you specify a *module_name*, it reports on that single, module-based blackbox.

- **-All**
A switch that specifies for the tool to display information on all defined blackboxes, as well as undefined models. This is the command default.
- **-Undefined**
A switch that specifies for the tool to display information on undefined models that have not yet been blackboxed. Use this switch to determine whether your design is complete, or is missing library models. If you intend to blackbox undefined models, this report allows you to verify that only the intended models are undefined.
- **> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- **>> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example defines module- and instance- based blackboxes then reports on them.

```
add black box -module core -pin do1 0 -pin io1 1  
add black box -instance core1 1 -pin do0 0 -pin io0 0  
report black box -all
```

```
MODULE: core (default tie value = X)  
  SYSTEM: Output pin do0 tied to X  
  USER: Output pin do1 tied to 0  
  SYSTEM: Inout pin io0 tied to X  
  USER: Inout pin io1 tied to 1  
INSTANCE: core1 (default tie value = 1)  
  USER: Output pin do0 tied to 0  
  SYSTEM: Output pin do1 tied to 1  
  USER: Inout pin io0 tied to 0  
  SYSTEM: Inout pin io1 tied to 1
```

Related Topics

[Add Black Box](#)

[Report Tied Signals](#)

[Delete Black Box](#)

Report Bus Data

Scope: All modes

Prerequisites: You can use this command only after the tool flattens the design to the simulation model, which happens when you first attempt to exit Setup mode or when you issue the Flatten Model command.

Displays the bus data information for either an individual bus gate or for the buses of a specific type.

Usage

REPort BU_s Data *type*

Description

The Report Bus Data command displays the following bus information:

- Instance name
- Gate identification number
- Contention handling (pass, bidi, fail, or abort)
- Type of bus (strong or weak)
- Number of drivers on the bus
- Any learned behavior of the bus.

The design rule that checks for bus contention mutual exclusivity is rule E10.

Arguments

- *type*

A required literal or integer that specifies the type of bus for which you want the tool to display information. The choices for the *type* argument are as follows:

gate_id# — An integer that specifies the gate identification number whose bus data you want to display.

ALL — A literal that displays the bus data for all buses.

Weak — A literal that displays the bus data for the weak buses.

Strong — A literal that displays the bus data for the strong buses.

Dominant — A literal that displays the bus data for the final bus of every set of cascaded buses.

NONDominant — A literal that displays the bus data for all but the final bus in every set of cascaded buses.

Pass — A literal that displays the bus data for the buses that passed the contention mutual exclusivity checking.

Bidi — A literal that displays the bus data for the bidirectional buses that have possible contention problems. For the tool to place a bus in this category, the bidirectional pin must have only a single tri-state driver.

Fail — A literal that displays the bus data for the buses that failed the contention mutual exclusivity checking.

ABort — A literal that displays the bus data for the buses that aborted contention mutual exclusivity checking.

Examples

The following example displays the information on a specific bus gate—an inverter (INV):

report bus data 31

```
/FA1/XOR1/OUT/ (31)Handling=pass type=strong #Drivers=2 (INV)
  Bus Drivers:  30 (SW) 28 (SW)
```

Related Topics

[Set Learn Report](#)

Report Cell Constraints

Scope: Atpg, Fault, and Good

Displays a list of the constrained cells.

Usage

For FlexTest

REPort CELL Constraints

Description

The Report Cell Constraints command displays a list of all the scan cells (also non-scan cells for DX and SX constraints) that you previously constrained to a constant value using the [Add Cell Constraints](#) command. The display may consist of up to seven columns, as follows (columns that contain no data for all cell constraints are not displayed):

- Constraint value — Specifies the constraint value.
- Chain name — Specifies the scan chain name if the constraint is on a scan cell. If the constraint is on a non-scan cell, then this column is blank.
- Cell position — Specifies the position in the scan chain if the constraint is on a scan cell. If the constraint is on a non-scan cell, then this column is blank.
- Cell instance name — Specifies the cell instance of the constraint. This column is blank unless you originally specified the constraint with a *pin_pathname*.
- Gate instance name — Specifies the gate instance of the constraint. This column is blank unless you originally placed the constraint on a non-scan cell using the -Clock switch.
- Constraint properties — Displays constraint properties, as applicable, from the following list:
 - Internal — Constraint was applied by the tool automatically in response to certain DRC results
 - Whole Chain — Constraint was applied using the -Chain switch
 - Drc C6 — Constraint was applied using the “-Drc C6” argument combination.
 - Clock — Constraint was applied using the -Chain switch
 - Drc D8 Warn — Constraint was applied by the tool automatically to a D8 failing master cell when “set drc handling d8 warning” was in effect for DRC.
 - Dynamic — Constraint was applied in Atpg, Fault or Good mode
 - Static — Constraint was applied in Setup mode

- Pattern scope — Specifies the pattern scope of the constraint: whether the tool uses the constraint only when creating scan patterns, or uses the constraint for both scan and chain pattern creation.

Arguments

None.

Examples

The next example adds DX constraints on all flip-flops clocked by clk2. You can see from the cell constraint report that these flip-flops are not scan cells (no chain or cell position). Notice also that the gate instances are reported for them since each flip-flop is a standalone gate, not part of a scan cell.

add cell constraints -clock /clk2 -type flop dx
report cell constraints

//	const	chain	cell	cell_inst	gate_inst	constraint	pattern scope
//	value	name	pos	name	name	properties	
//	-----	-----	-----	-----	-----	-----	-----
//	DX				/dff6	(Clock) (Dynamic)	Scan Patterns only
//	DX				/dff5	(Clock) (Dynamic)	Scan Patterns only
//	C0	c1	2	sdff1/q		(Static)	Chain + Scan Pattern

Related Topics

[Add Cell Constraints](#)

[Delete Cell Constraints](#)

Report Clocks

Scope: All modes

Displays a list of all clocks specified using the Add Clocks command.

Usage

REPort CLocks

Arguments

None.

Examples

Example 1

The following example adds two clocks to the clock list and then displays a list of the clocks:

```
add clocks 1 clk1
add clocks 0 clk0
report clocks
```

Example 2

The following example adds two clocks to the clock list and then displays a list of the clocks:

```
add clock 1 clk_0 clk_1 clk_2 clk_3 -internal -pin_name group_clk
// Note: Adding primary input group_clk merging 4 nets.

report clocks

list of clocks
/group_clk off-state = 1 (merged internal pin)
```

Related Topics

[Add Clocks](#)

[Delete Clocks](#)

[Analyze Control Signals](#)

Report Cone Blocks

Scope: All modes

Displays the current, user-defined output pin pathnames that the tool uses to calculate the clock and effect cones.

Usage

REPort COne Blocks

Description

The Report Cone Blocks command displays the output pin pathnames you identified as blockage points using the Add Cone Blocks command. The tool uses these blockage points in calculating the clock and effect cones. If you have not specified any blockage points, the tool automatically chooses the output pins that it uses in the calculations.

Arguments

None.

Examples

The following example displays the user-defined clock and effect cones:

```
add cone blocks -clock /ls0/q  
add cone blocks -effect /ls7/q  
report cone blocks
```

```
clock /LS0/Q  
effect /LS7/Q
```

Related Topics

[Add Cone Blocks](#)

[Delete Cone Blocks](#)

Report Contention Free_bus

Scope: All modes

Reports contention free buses.

Usage

REPort COntention Free_bus

Description

The Report Contention Free_bus command reports all the contention free buses you defined in the Add Contention Free_bus command.

Arguments

None.

Examples

The following example reports contention free buses.

```
report contention free_bus  
run
```

Related Topics

[Add Contention Free_bus](#)

[Set Contention Check](#)

[Delete Contention Free_bus](#)

[Set Contention_bus Reporting](#)

[Report Gates](#)

[Set Gate Report](#)

[Set Bus Handling](#)

Report Core Memory

Scope: All modes

Displays the amount of memory FlexTest requires to avoid paging during the ATPG and simulation processes.

Usage

REPort COre Memory

Description

The Report Core Memory command displays the peak memory requirements of FlexTest. However, the peak memory requirement that this command displays is generally much larger than the actual memory requirements during the ATPG and fault simulation processes.

Arguments

None.

Examples

The following example displays the amount of memory FlexTest requires to avoid memory paging during the ATPG and simulation processes:

report core memory

	Peak	Current
Memory for flatten design :	0.127M	0.125M
Memory for fault list :	0.062M	0.062M
Memory for test generation:	0.127M	0.125M
Memory for simulation :	0.004M	0.004M
Memory for ram/rom :	0.000M	0.000M
Total core memory :	0.320M	0.317M

Related Topics

[Report Statistics](#)

[Write Core Memory](#)

Report Drc Rules

Scope: Setup and Drc modes

Displays either a summary of DRC violations (fails) or violation occurrence message(s).

Usage

```
REPort DRc Rules [-Fails_summary | -Summary |  
rule_id... | rule_id-occurrence#... | -All_fails]  
[{> | >>} file_pathname]
```

Description

It can display a report in one of two formats:

- Summary report—Lists for each reported design rule, one line of data per rule, the current number of DRC violations (fails), the violation handling, and a brief description of the rule. When you request a summary of D5 violations, the report lists the number of D5 violations of each type (INIT-0, INIT-1, INIT-X, TIE-0, TIE-1, TIE-X, TLA).
- Occurrence report—Lists one or more violation occurrence messages that give details of specific DRC violations

Table 2-6 provides a summary of the available information displayed and the arguments you use to obtain them. Refer to the Arguments subsection for complete details about the arguments.

Table 2-6. Available Information Displayed and Arguments

Desired Display	Rules/Occurrences Covered	Argument
Summary report	Design rules that resulted in violations (fails) during DRC	-Fails_summary
	All design rules	-Summary
Occurrence report	Specific rule, specific occurrence	rule_id-occurrence#
	Specific rule, all occurrences ¹	rule_id
	All occurrences	-All_fails

1. Additional D5-specific arguments enable you to further customize the D5 information display.

You can use the [Set Drc Handling](#) command to change the handling of many of the A (RAM), C (clock), D (data), E (extra), T (trace), and W (timing) rules.

Arguments

- **-Fails Summary**

A switch that specifies to display the following for each user-controllable rule that resulted in a violation (fail) during DRC:

- Rule identification (ID)
- Number of failures of the rule
- Current handling status of the rule
- Brief description of the rule

This is the default.

Note



This switch does not display anything if there are no rule violations or the tool has not yet performed DRC.

- **-Summary**

A switch that specifies to display a summary report.

- ***rule_id***

A repeatable string that specifies the identification literal (ID) of a particular design rule for which you want to display the violations.

- ***rule_id-occurrence#***

A repeatable string that specifies the identification literal (ID) of a particular design rule and the violation occurrence for which you want to display the occurrence message. This argument must include the specific design rule ID (*rule_id*), the specific occurrence number of the violation, and the hyphen between them. For example, you can analyze the second violation occurrence of the C3 rule by specifying C3-2. Numbers to occurrences of rule violations are assigned as they are encountered; you cannot change the number assigned to a specific occurrence.

- **-All_Fails**

A switch that specifies to display all occurrence messages for all occurrences of rule violations. The displayed information can be lengthy, as it is the same information you would get if you consecutively entered a “report drc rules <*rule_id*>” command for each rule that had a violation. Use this switch to output a report of all violation occurrences (most likely to a log file) for later analysis.

- **> *file_pathname***

An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- `>> file_pathname`

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

Example 1

The following example displays a summary of the rules that resulted in violations during an attempted ATPG run:

report drc rules

```
C3: #fails=2 handling=warning (clock may capture data affected by its
    captured data)
C5: #fails=1 handling=error (clock is connected to multiple ports of same
    latch)
C7: #fails=19 handling=warning (scan cell capture ability check)
C8: #fails=2 handling=warning (PO connected to a clock line)
C9: #fails=2 handling=warning (PO connected to a clock line gated by scan
    cell that uses same clock)
D5: #fails=23 handling=warning (non-scan memory element)
D6: #fails=22 handling=warning (non-transparent non-scan latches)
D7: #fails=22 handling=warning (stable high edge-triggered clock ports)
```

Example 2

The following example displays the occurrence message for the two C3 violations, and the first C7, and first D7 violation reported in the preceding example:

report drc rules c3 c7-1 d7-1

```
// Warning: Clock /sgst failed rule C3 on input 3 of /sp_b_i0.latch
    (1501). (C3-1)
// Source of violation: input 3 of /sp_c_i1.slave (1482).
// Warning: Clock /sgst failed rule C3 on input 3 of /sp_c_i0.latch
    (1527). (C3-2)
// Source of violation: input 3 of /sp_c_i1.slave (1482).
// Warning: Clock input 5 of /c8.master (1495) cannot capture data
    with single clock on. (C7-1)
// Warning: Flipflop /FF1 (103) has clock port set to stable high. (D7-1)
```

Example 3

The following example changes the handling of data rule 7 (D7) from warning to error and also specifies execution of a full test generation analysis when performing the rules checking for the clock (C) rules:

```
set drc handling d7 error atpg_analysis
set system mode atpg
```

```
//-----  
//Begin scan chain identification process, memory elements=8.  
//-----  
// Reading group test procedure file /user/design/tpf.  
// Simulating load/unload procedure in g1 test procedure file.  
// Chain = c1 successfully traced with scan_cells = 8.  
// Error: Flipflop /FF1 (103) has clock port set to stable high.(D7-1)  
// Error: Rules checking unsuccessful, cannot exit SETUP mode.
```

Example 4

The following example displays the occurrence message for the D7 violation:

report drc rules d7-1

```
//Error: Flipflop /FF1 (103) has clock port set to stable high. (D7-1)
```

Related Topics

[Set Drc Handling](#)

Report Environment

Scope: All modes

Displays the current value(s) of many of the most frequently used “Set...” commands.

Usage

REPort ENvironment

Description

When you first invoke the tool, the values displayed are the default settings for these commands.

Arguments

None.

Examples

The output from the Report Environment command may look like the following:

```
abort limit = 30
atpg compression = OFF
atpg limits = none
bist initialization = X
bus simulation method = global
capture clock = none
checkpoint = OFF
clockpo patterns = ON
clock restriction = clock_po
clock_off simulation = OFF
contention check = ON, mode = bus, handling = warning
DRC transient detection = ON -Verbose
fails report = OFF
fault mode = uncollapsed
fault type = stuck
gate level = design
gate report = normal
iddq checks = none, handling = warning
iddq strobe = label
learn reporting = OFF
logfile handling = OFF
net dominance = wired-gate
net resolution = wired-gate
observation point = master
pattern classification = ON
pattern source = internal
possible credit = 50%
pulse generators = ON
RAM initialization = uninitialized
RAM test mode = static_pass_thru
random atpg = ON
random clocks = none
random patterns = 1024
screen display = ON
shadow checking = ON
simulation mode = combinational    depth = 0
skew load = OFF
split capture cycle = OFF
stability check = ON
system mode = setup
TLA loop handling = OFF
trace report = OFF
Z handling = int=X    ext=X
Zhold behavior = ON
```

Related Topics

[Write Environment](#)

Report Faults

Scope: Atpg, Fault, and Good modes

Displays fault information from the current fault list.

Usage

For FlexTest

```
REPort FAults [-Class class_type] [-Stuck_at {01 | 0 | 1}] [-All | object_pathname...]
               [-Hierarchy integer [-Min_count integer]] [-Noeq]
```

Description

The Report Faults command displays faults you added to the fault list using the Add Faults or Load Faults command. You can use the optional arguments to narrow the focus of the report to only specific stuck-at or transition faults that occur on a specific object in a specific class. If you do not specify any arguments, Report Faults displays information on all the known faults.

The Report Faults command displays the following columns of information for each fault:

- fault value - The fault value may be either 0 (for stuck-at-0 or “slow-to-rise” transition faults) or 1 (for stuck-at-1 or “slow-to-fall” transition faults).
- fault code - A code name that indicates the lowest level fault class assigned to the fault.
- fault site - The pin pathname of the fault site.
- cell name (optional) - The name of the cell corresponding to the fault (displayed only if you include the -Cell_name argument).

You can use the -Hierarchy option to display a hierarchical summary of the selected faults. The summary identifies the number of faults in each level of hierarchy whose level does not exceed the specified level number. You can further specify the hierarchical summary by using the -Min_count option which specifies the minimum number of faults that must be in a hierarchical level before they are displayed.

You may select to display either collapsed or uncollapsed faults by using the Set Fault Mode command. Also, some fault data is large and it would be more appropriate to use the Write Faults command, and then read the file contents.

Arguments

- -Class *class_type* [*subclass_name* |*subclass_code*]

An optional switch and literal pair that specifies the class of faults that you want to display. The *class_type* argument can be either a fault class code or a fault class name.

The following table lists the valid fault class codes and their associated fault class names; use either the code or the name when specifying the *class_type* argument:

Table 2-7. Fault Class Codes and Names

Fault Class Codes	Fault Class Names	Fault Class Coverage
FU	Full	TE+UT
TE	TEstable	DT+PD+OS+HY+AU+UD
DT	DETEcted	DS+DI+DR
DS	DET_Simulation	
DI	DET_Implication	
PD	POSDET	PT+PU
PU	POSDET_Untestable	
PT	POSDET_Testable	
OS	OSCillatory	OU+OT
HY	HYPErtrophic	HU+HT
AU	Atpg_untestable	
UD	UNDetected	UC+UO
UC	UNControlled	
UO	UNObserved	
UT	UNTestable	UU+TI+BL+RE
UU	UNUsed	
TI	TIed	
BL	Blocked	
RE	Redundant	

- -Stuck_at 01 | 0 | 1

An optional switch and literal pair that specifies the stuck-at or transition faults you want to display. The choices are as follows:

01 — A literal specifying that for stuck-at faults the tool display both “stuck-at-0” and “stuck-at-1” faults; or for transition faults, the tool display both “slow-to-rise” and “slow-to-fall” faults. This is the default.

0 — A literal specifying to display only the “stuck-at-0” faults (“slow-to-rise” faults for transition faults).

1 — A literal specifying to display only the “stuck-at-1” faults (“slow-to-fall” faults for transition faults).

- **-All**

An optional switch that displays all of the faults, protected as well as unprotected, on all model, netlist primitive, and top module pins. This is the default.

- *object_pathname*

An optional repeatable string that specifies a list of pins, instances, or delay paths whose faults you want to display.

- **-Hierarchy** *integer*

An optional switch and integer pair that specifies the maximum hierarchy level for which you want to display a summary of the faults.

- **-Min_count** *integer*

An optional switch and integer pair that you can use with the -Hierarchy option to specify the minimum number of faults that must be in a hierarchical level to display the hierarchical summary. The default is 1.

- **-Noeq**

An optional switch that turns off the display of “EQ” as the fault class for any equivalent faults; the fault class displayed is then that of the representative fault. When you do not specify this switch, the tool displays an “EQ” as the fault class for any equivalent faults. For more information on representative and equivalent faults, see “[Fault Collapsing](#)” in the *Scan and ATPG User’s Manual*.

Related Topics

[Add Faults](#)

[Set Fault Mode](#)

[Analyze Fault](#)

[Set Fault Sampling](#)

[Delete Faults](#)

[Set Fault Type](#)

[Load Faults](#)

[Write Faults](#)

[Report Testability Data](#)

Report Feedback Paths

Scope: All modes

Prerequisites: You can use this command only after the tool performs the learning process, which happens immediately after flattening a design to the simulation model. Flattening occurs when you first attempt to exit Setup mode or when you issue the Flatten Model command.

Displays a report of currently identified feedback paths.

Usage

REPort FEedback Paths [-All | *loop_id#...*] [-Display *tab...*] [{> | >>} *file_pathname*]

Description

The Report Feedback Paths command displays feedback paths the tool identified during the last circuit learning process. By default, the command displays a textual report of all currently identified feedback paths and their identification numbers. Issuing the command with the identification numbers of specific paths of interest will limit the display to just those paths.

You can use the identification numbers with the [Report Loops -Display](#) command to schematically display specific feedback paths. When you issue this command for specific feedback paths, Tessent Visualizer transcripts the same information as the Report Feedback Paths command, but only for the specified paths.

Note



Feedback paths include, by default, any duplicated gates.

Arguments

- -All
An optional switch that specifies to report all currently identified feedback paths. This is the default.
- *loop_id#*
An optional, repeatable, non-negative integer that specifies the identification number of a particular feedback path to report. The tool assigns the numbers consecutively, starting with 0.
- -Display *tab...*
An optional switch and repeatable literal that display the reported information graphically in the specified Tessent Visualizer tab(s).
- > *file_pathname*
An optional redirection operator and pathname pair for creating or replacing the contents of *file_pathname*.

- `>> file_pathname`

An optional redirection operator and pathname pair for appending to the contents of *file_pathname*.

Examples

The following example enters ATPG mode (which flattens the simulation model and performs the learning process), reports the identification numbers of all learned feedback paths, and then displays one of the reported feedback paths (loop #1):

```
set system mode atpg  
report feedback paths
```

```
Loop#=0, feedback_buffer=26, #gates_in_network=5  
  INV  /I_956__I_582/ (51)  
  PBUS /I_956__I_582/N1/ (96)  
  ZVAL /I_956__I_582/N1/ (101)  
  INV  /I_956__I_582/ (106)  
  TIEX /I_956__I_582/ (26)  
Loop#=1, feedback_buffer=27, #gates_in_network=5  
  INV  /I_962__I_582/ (52)  
  PBUS /I_962__I_582/N1/ (95)  
  ZVAL /I_962__I_582/N1/ (100)  
  INV  /I_962__I_582/ (105)  
  TIEX /I_962__I_582/ (27)
```

```
report feedback paths 1 -display design
```

Related Topics

[Report Loops](#)

[Set Loop Handling](#)

Report Flatten Rules

Scope: All modes

Displays either a summary of all the flattening rule violations or the data for a specific violation.

Usage

REPort FLatten Rules [*rule_id* [{*occurence_id* | -Verbose}]] [{> | >>} *file_pathname*]

Description

The Report Flatten Rules command displays the following information for a specific violation:

- Rule identification number
- Current number of rule failures
- Violation handling

You can use the [Set Flatten Handling](#) command to change the handling of the net, pin, and gate rules.

Arguments

- *rule_id*

A literal that specifies the flattening rule violation for which you want to display information. The flattening rule violations and their identification literals are divided into the following three groups: net, pin, and gate rules violation IDs.

- Net flattening violations are described in sections FN1 through FN9
- Pin flattening violations are described in sections FP1 through FP13
- Gate flattening violations are described in sections FG1 through FG8

- *occurence_id*

A literal that specifies the identification of the exact flattening rule violation (the occurrence) for which you want to display information. For example, you can analyze the second occurrence of the FG4 rule by specifying the *rule_id* and the *occurence_id*, FG4 2. The tool assigns the occurrences of the rules violations as it encounters them; you cannot change either the rule identification number or the ordering of the specific violations.

- -Verbose

A switch that displays the following for each flattening rule:

- Rule identification number
- Number of failures of each rule
- Current handling status of that rule
- Brief description of that rule

- `>file_pathname`
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- `>>file_pathname`
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example shows the summary information of the FG3 rule:

```
report flatten rules fg3
// FG3: fails=2 handling=warning/noverbose
```

Related Topics

[Set Flatten Handling](#)

Report Gates

Scope: All modes

Prerequisites: Although you can use this command in all modes, you can only use it with a flattened netlist. Netlist flattening typically happens when you first attempt to exit Setup mode or when you issue the [Flatten Model](#) command.

Displays the netlist information and simulation results for the specified gates and the simulation results for the specified user-defined ATPG functions.

Usage

All Other Modes

```
REPort Gates {pin_or_net_pathname | gate_id# | instance_name}...  
  [-Type gate_type] [-Depth integer]  
  {[-Endpoints] [-Forward | -Backward] {pin_pathname | gate_id}}...
```

Description

The Report Gates command displays the netlist information for either the design-level or the primitive-level gates you specify. You designate the gate by its gate identification number (ID#), a pathname of a pin connected to a gate, an instance name (design level only), or a gate type. You can specify a design cell by the pathname of a pin on the design cell. If you use a gate ID# or gate type, the command always reports the primitive-level gate. You must flatten the netlist before issuing this command.

The *pin_or_net_pathname* and *instance_name* arguments support regular expressions, which may include any number of '*' or '?' wildcard characters embedded in the pathname string. The '*' character matches any sequence of characters (including none) in a name, and the '?' character matches any single character. If a wildcard name is specified, the command will search for matching instance names from the top library cell level, down to the primitive gates.

The format for the design-level gate report is:

```
instance_name      cell_type  
  input_pin_name   I      (data)  pin_pathname...  
...  
  output_pin_name  0      (data)  pin_pathname...  
...
```

The format for the primitive-level report is:

```
instance_name      (gate_ID#)  gate_type  
  input_pin_name   I      (data)  gate_ID#-pin_pathname...  
...  
  output_pin_name  O      (data)  gate_ID#-pin_pathname...  
...
```

When you provide the name or identification number (ID#) of a user-defined ATPG function, the command reports simulation data of the function. The data reported is that specified by the `Parallel_pattern` option of the `Set Gate Report` command.

The format for the function report is:

```
function_name    (function_ID#)    function_type
input_pin_name   I    (data)    gate_ID#-pin_pathname...
...
output_pin_name  O    (data)    gate_ID#-pin_pathname...
...
```

The list associated with the input and output pin names indicate the pins to which they connect. For the primitive level, this also includes the gate index number of the connecting gate and only includes the pin pathname if one exists at that point. There is a limitation on reporting gates at the design level. If some circuitry inside the design cell is completely isolated from other circuitry, the command only reports the circuitry associated with the pin pathname.

You can also report the fan-in or fan-out cone of a specified gate. The endpoints of a cone are defined as the primary inputs, primary outputs, tied gates, rams, roms, flip-flops, and latches. All gates reported are at the primitive level.

You can change the output of the `Report Gates` command by using the `Set Gate Report` command.

Backslashes (\), Periods (.) and Spaces in Hierarchical Names

When reading hierarchical names into its internal database, the tool hides backslashes (\), converts periods (.) to slashes (/), and ignores spaces. For example, the tool reads the cell name, “FFH.\E/MY_SCAN_FF .SCAN_FF5” in a WGL pattern, as “FFH/E/MY_SCAN_FF/SCAN_FF5”, and stores the latter in its internal database. To report on this cell, you must reference it by the name that is stored internally.

Note



The difference between the internal hierarchical name and the name in the WGL file of the preceding example does not alter the correctness of the tool. That is, the output pattern file remains consistent with the original netlist.

The following example reports on a Verilog gate named “FFH.\E/MY_SCAN_FF .SCAN_FF5”.

```
ATPG> report gates FFH/E/MY_SCAN_FF/SCAN_FF5
```

```
// /FFH/E/MY_SCAN_FF/SCAN_FF5 FD1SF
//      D      I /FFH/E/MY_SCAN_FF/GATE00/Z
//      A      I /C-4/Z
//      SI     I /FFG/SO
//      CP     I /INPBUF1/Z
//      B      I /C-5/Z
//      Q      O /GATE20/C
//      QN     O
//      SO     O /C-7/A
```

Reporting on the First Input of a Gate

Report Gates provides a shortcut to display data on the gate connected to the first input of the previously-reported gate. This lets you quickly and easily trace backward through circuitry. To use Report Gates in this manner, first report on a specific gate and then enter:

SETUP> b

The following example shows how to use Report Gates and B commands to trace backward through the first input of the previously reported gate.

SETUP> report gates 26

```
// /u1/inst__565_ff_d_1__13 (26) BUF
//      "I0"    I 269-
//      "OUT"   O 268- 75-
```

SETUP> b

```
// /u1/inst__565_ff_d_1__13 (269) LA
//      "S"     I 14-
//      "R"     I 145-
//      SCLK    I 4-/clk
//      D       I 265-/u1/_g32/X
//      ACLK    I 2-/scan_mclk
//      SDI     I 20-/u1/inst__565_ff_d_0__dff/Q2
//      "OUT"   O 26- 27-
```

SETUP> b

```
// /u1/inst__565_ff_d_1__13 (14) TIE0
//      "OUT"   O 269- 268-
```

If you include an integer argument with “b”, the trace will be backward through the specified input. For example, “b 3” will trace backward through the third input of the previously specified gate.

For pins that are not at the library cell boundary (pins internal to the model), the pin name is enclosed in double quotes (“”). The following example displays this issue.

ATPG> set gate level primitive

ATPG> rep gate /I_20/I_226/q

```
// /I_20/I_226 dffsr
//   clk      I  (HX)  /I_20/I_225/out
//   d        I  (X)   /I_20/I_222/out
//   pre      I  (H1)  /PRE
//   clr      I  (H1)  /CLR
//   q        O  (X)   /I_16/i0   /I_23/I_221/i0   /I_6/i0
//   qb       O  (X)
```

ATPG> set gate level primitive

```
// Creating schematic for 5 instances (1 was compacted).
```

ATPG> rep gate /I_20/I_226/q

```
// /I_20/I_226 (12) BUF
//   "I0"      I  (0)   39-
//   q         O  (X)   16-/I_16/i0   31-/I_23/I_221/i0
//                               17- /I_6/i0
```

ATPG> b

```
// /I_20/I_226 (39) DFF
//   "S"      I  (LX)  26-
//   "R"      I  (LX)  23-
//   clk      I  (HX)  20-/I_20/I_225/out
//   d        I  (X)   36-/I_20/I_222/out
//   "OUT"    O  (0)   12- 13-
//   MASTER   cell_id=1 chain=c1 group=g1 invert_data=FFFF
```

Reporting on the First Fanout of a Gate

Similar to tracing backward through circuitry, you can also use a shortcut to trace forward through the first fanout of the previously reported gate. To use Report Gates in this manner, first report on a specific gate and then enter:

SETUP> f

The following example shows how to use Report Gate and F commands to trace forward through the first fanout of the previously reported gate.

SETUP> rep ga 269

```
// /u1/inst__565_ff_d_1__13 (269) LA
//   "S"      I  14-
//   "R"      I  145-
//   SCLK     I  4-/clk
//   D        I  265-/u1/_g32/X
//   ACLK     I  2-/scan_mclk
//   SDI      I  20-/u1/inst__565_ff_d_0__dff/Q2
//   "OUT"    O  26- 27-
```

SETUP> f

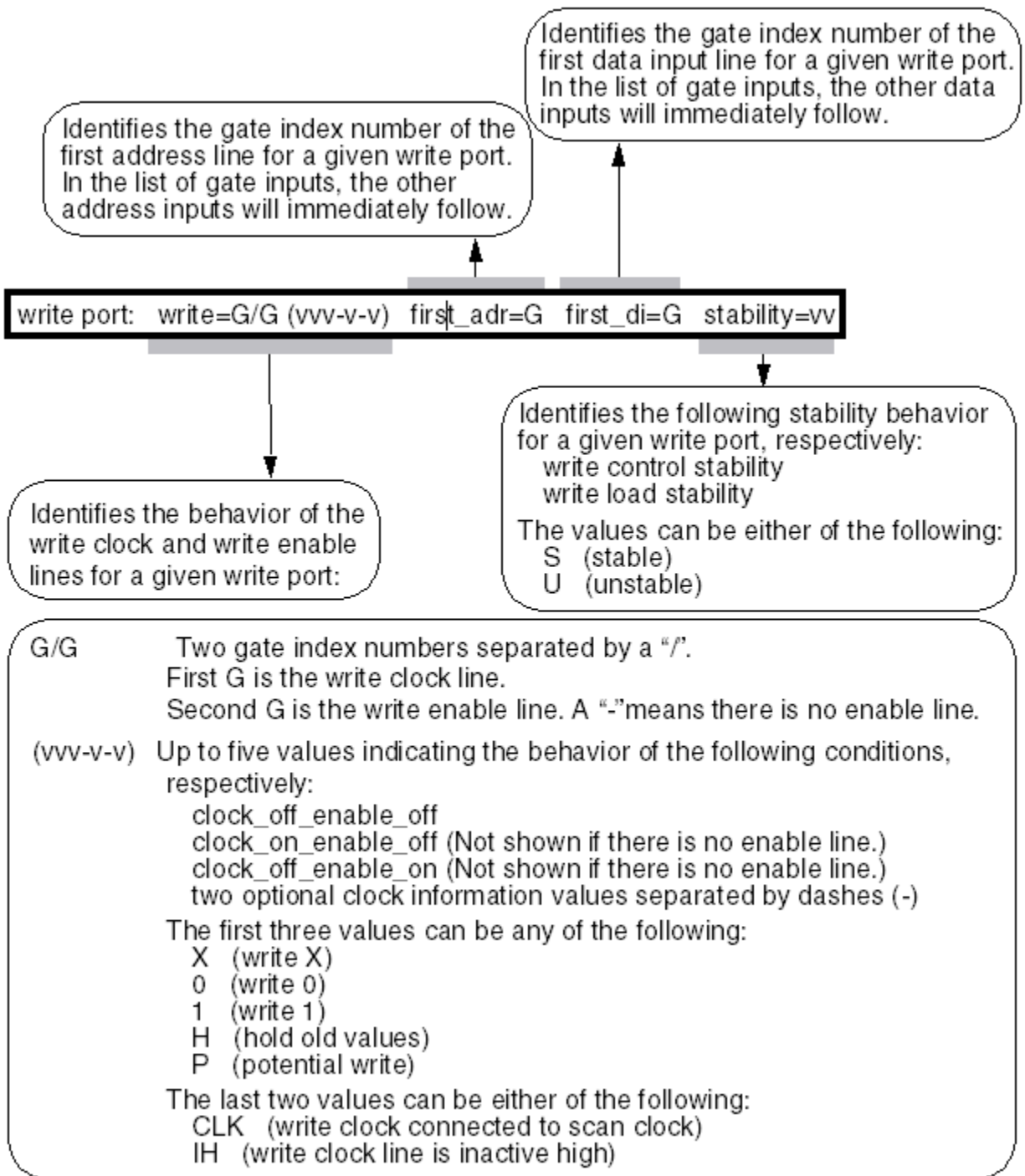
```
// /u1/inst__565_ff_d_1__13 (26) BUF
//      "I0"      I  269-
//      "OUT"     O  268-  75-
```

SETUP> f

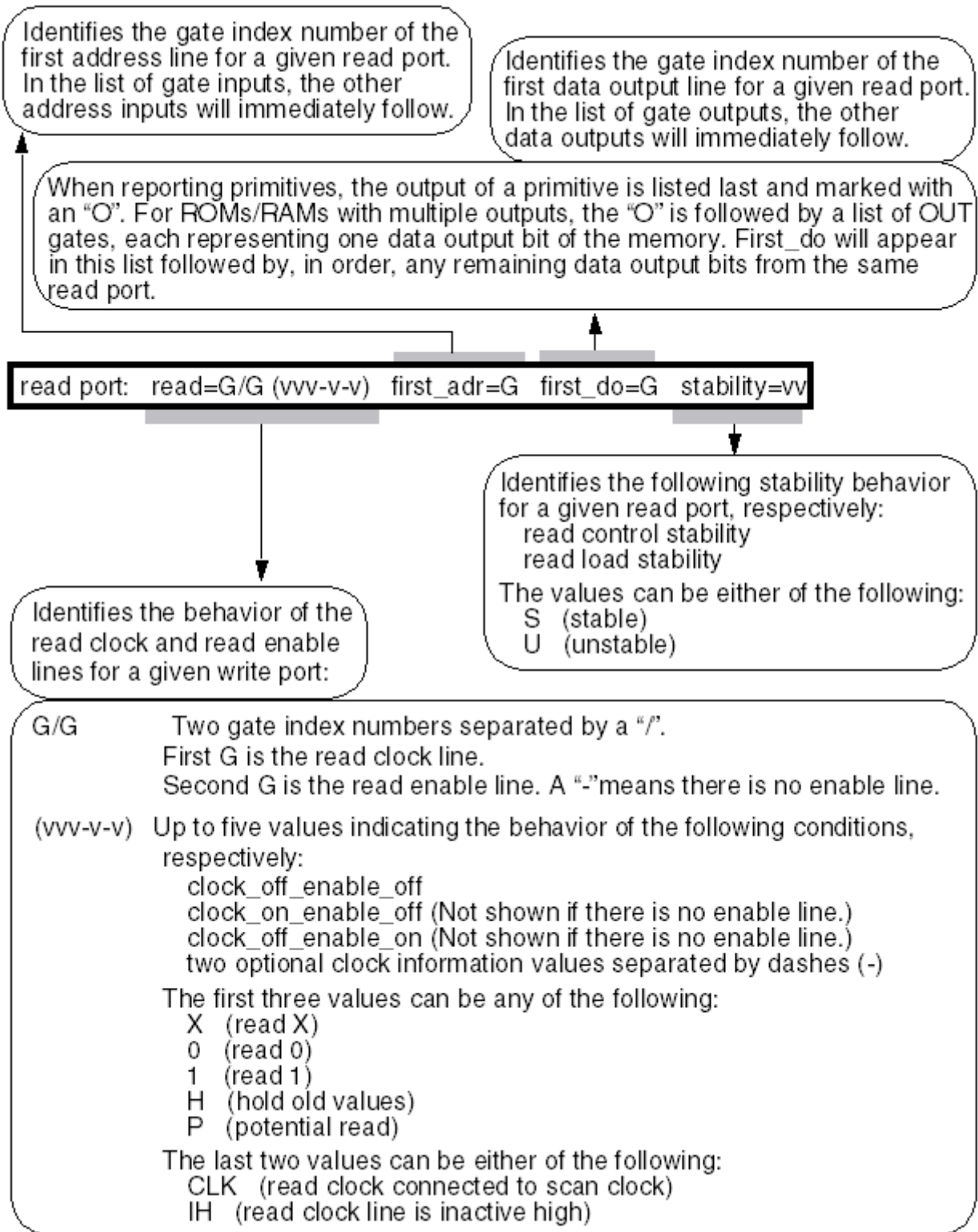
```
// /u1/inst__565_ff_d_1__13 (268) LA
//      "S"       I  14-
//      "R"       I  145-
//      BCLK      I  1-/scan_sclk
//      "D0"      I  26-
//      "OUT"     O  24-  25-
```

If you include an integer argument with “f”, the trace will be forward through the specified fanout of the previously reported gate. For example, “f 2” will trace forward through the second fanout of the previously specified gate.

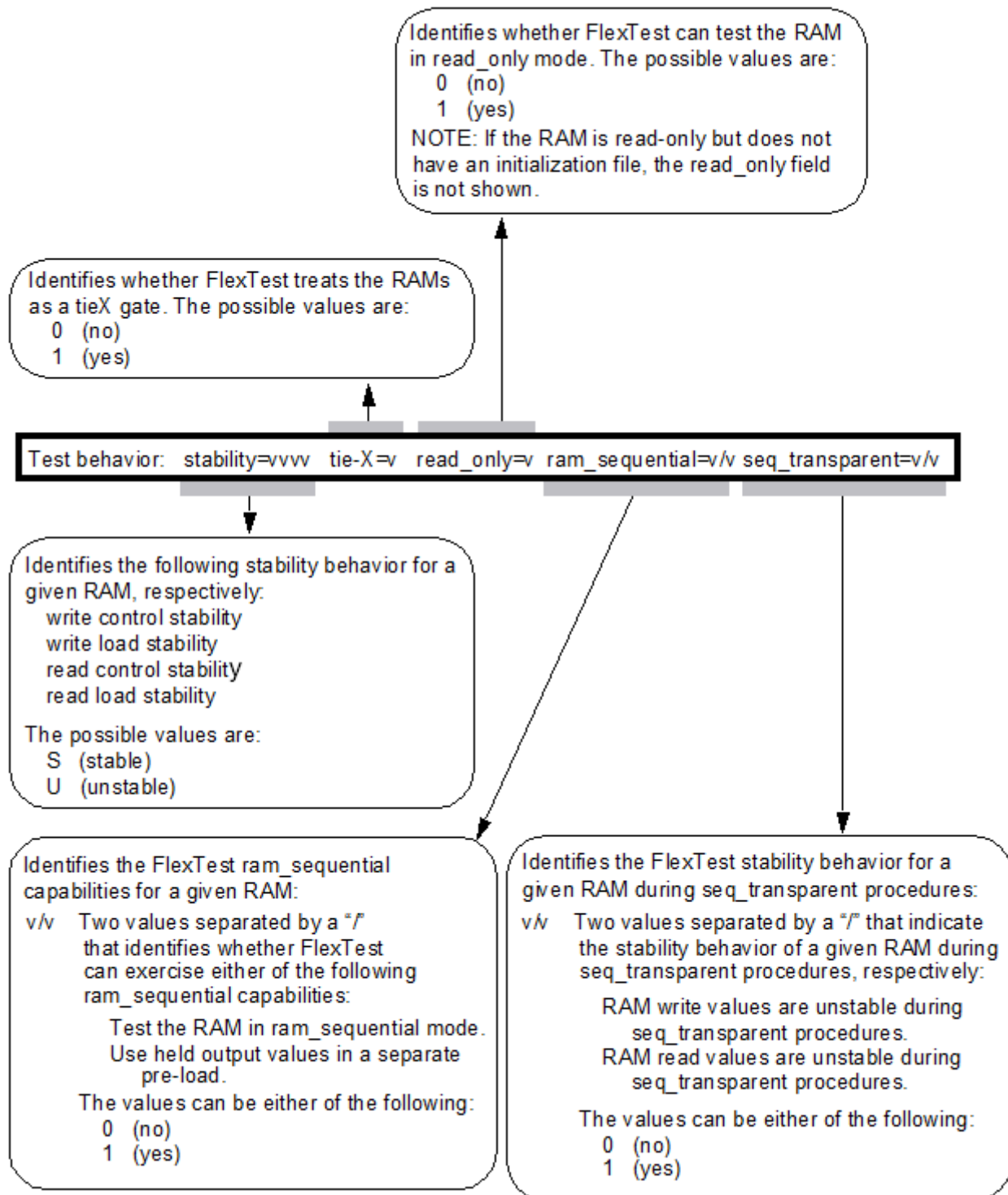
The following describes the fields for the write port message line:



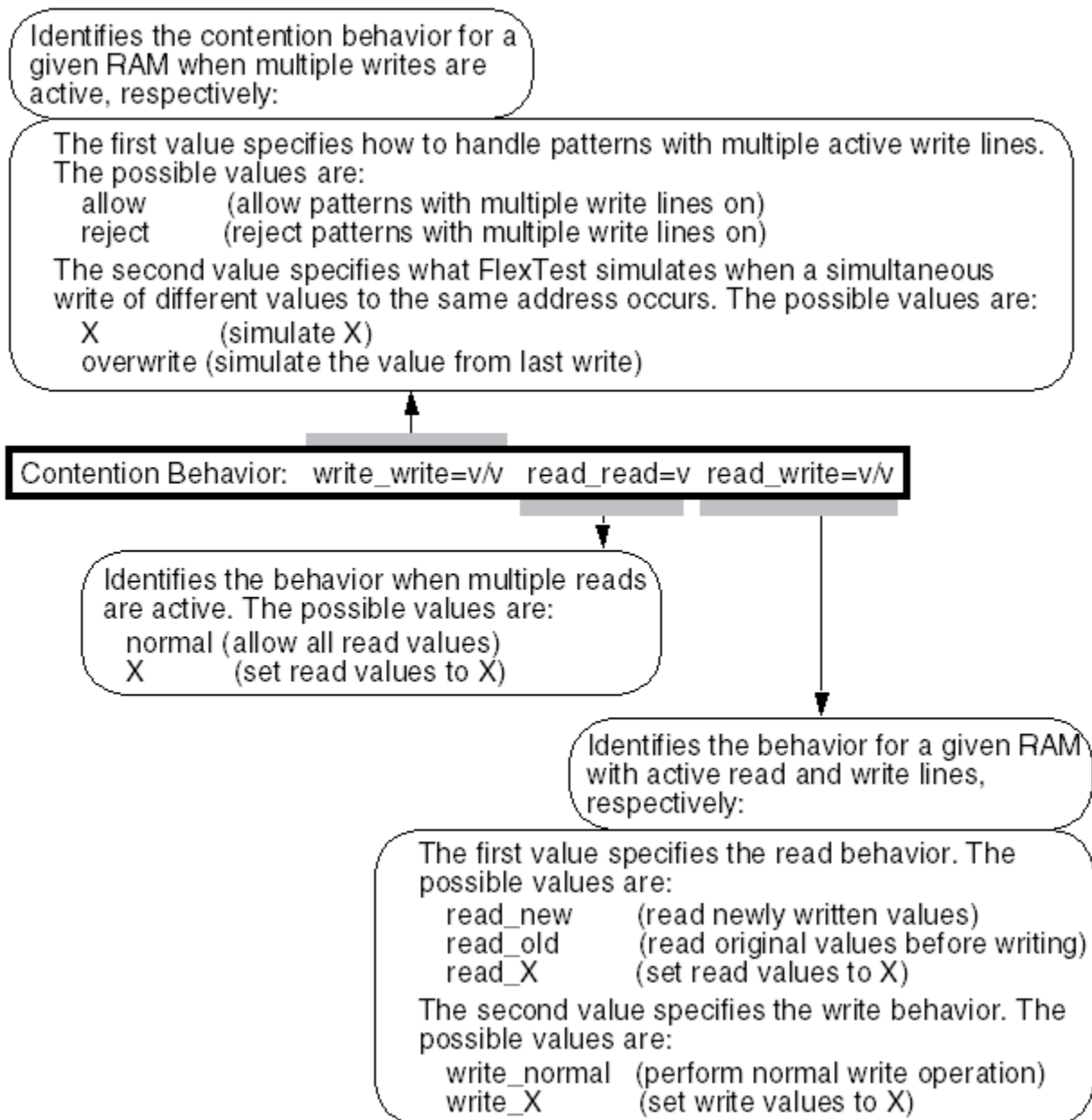
The following describes the fields for the read port message line:



The following describes the fields for the Test behavior message line:



The following describes the fields for the Contention Behavior message line:



Arguments

- *gate_id#*

A repeatable integer that specifies the gate identification numbers of the objects. The value of the *gate_id#* argument is the unique identification number that the tool automatically assigns to every gate within the design during the model flattening process.

- ***pin_or_net_pathname***

A repeatable string that specifies the pathnames of pins or nets in the design netlist. You may use wildcard characters to match multiple pin or net pathnames.

For a hierarchical pathname, the display will include information describing how that pathname maps to the driving design level pin(s) and gate(s) for which data is displayed.

- ***instance_name***

A repeatable string that specifies the hierarchical pathname of an instance of a library cell within the design. If a valid library instance pathname is given when in primitive level, all pins on that library cell are reported. When in primitive level, ***instance_name*** may also be the pathname of a primitive instance.

- -Forward {*pin_pathname* | *gate_id*}...

An optional switch that reports the fan-out cone of the specified gate.

- -Backward {*pin_pathname* | *gate_id*}...

An optional switch that reports the fan-in cone of the specified gate.

- -Endpoints [-Forward | -Backward] {*pin_pathname* | *gate_id*}...

An optional switch that reports only the endpoint of the cone.

Note



Immediately after -Endpoints, you must specify either -Forward or -Backward followed by the specified gate. When using -Endpoints or -CONstraint simultaneously, -Forward or -Backward followed by the specified gate need only be entered once.

- -Depth

An optional switch that extends the cone report to several levels. The next level cones reported will be the cones of the endpoints of the previous level. If there are loops in a circuit, gates may be repeated in several levels. The default is only one level.

- -Type *gate_type*

An optional repeatable switch where *gate_type* specifies the gate types for which you want to display the gate information. [Table 2-8](#) and [Table 2-9](#) list the valid FlexTest learned types.

Table 2-8. Reportable Gate Types

gate_type	Description
BUF	buffer
INV	inverter
AND	and
NAND	inverted and

Table 2-8. Reportable Gate Types (cont.)

gate_type	Description
OR	or
NOR	inverted or
XOR	exclusive-or
NXOR	inverted exclusive-or
DFF	D flip-flop, same as _dff library primitive
DLAT	latch, same as _dlat library primitive
PI	primary input
PO	primary output
TIE0	tied low
TIE1	tied high
TIEX	tied unknown
TIEZ	tied high impedance
TSH	tri-state driver and second input is active high enable line
BUS	tri-state bus
Z2X	Z converter gate, converts Z to X
WIRE	undetermined wired gate
MUX	2-way multiplexor and third line is select line
NMOS	switch gate and second input is active high enable line
SWBUS	pulled bus gate, where the second input is the pulled value
ROUT	memory model gate, created for each read data bit
RAM	random access memory
ROM	read only memory
XDET	X detector, gives 1 when input is X
ZDET	Z detector, gives 1 when input is Z
FB_BUF	internal gate type to break combinational loops in the design

Table 2-9. FlexTest Learned Gate Types

LEARN_BUF	LEARN_XOR	LEARN_TIED_Or
LEARN_INV	LEARN_MUX	FORBId
LEARN_AND	LEARN_TIED_Xor	ZHOLD

Table 2-9. FlexTest Learned Gate Types (cont.)

LEARN_OR	LEARN_TIED_And	
----------	----------------	--

Examples

The following example shows the use of wildcards:

ATPG> report gates /xscan_0_0_cch_scan_32x/ix15*

```
// /xscan_0_0_cch_scan_32x/ix151 NOR2XL
// A I /myop1<0> [1]
// B I /myop2[1]
// Y O /xscan_0_0_cch_scan_32x/sum_add_0_ix27/A0
// /xscan_0_0_cch_scan_32x/ix153 NAND2X1
// A I /myop1<0> [0]
// B I /myop2[0]
// Y O /xscan_0_0_cch_scan_32x/sum_add_0_ix1/A \
      /xscan_0_0_cch_scan_32x/ sum_add_0_ix23/B \
      /xscan_0_0_cch_scan_32x/sum_add_0_ix27/A1
// /xscan_0_0_cch_scan_32x/ix155 NAND2X1
// A I /myop1<0> [1]
// B I /myop2[1]
// Y O /xscan_0_0_cch_scan_32x/ix166/B0 \
      /xscan_0_0_cch_scan_32x/ sum_add_0_ix27/B0
// /xscan_0_0_cch_scan_32x/ix157 NAND2X1
// A I /myop1<0> [3]
// B I /myop2[3]
// Y O /xscan_0_0_cch_scan_32x/ix174/B0 \
      /xscan_0_0_cch_scan_32x/ sum_add_0_ix55/B0
```

ATPG> rep gates /xscan_0_0_cch_scan_32x/ix159

```
// /xscan_0_0_cch_scan_32x/ix159 NAND2X1
// A I /myop1<0> [5]
// B I /myop2[5]
// Y O /xscan_0_0_cch_scan_32x/ix188/B0 \
      /xscan_0_0_cch_scan_32x/ sum_add_0_ix83/B0
```

ATPG> set gate level primitive

ATPG> report gates /xscan_0_0_cch_scan_32x/ix157

```
// /xscan_0_0_cch_scan_32x/ix157 (43) NAND
// A I 7-/myop1<0> [3]
// B I 17-/myop2[3]
// Y O 60-/xscan_0_0_cch_scan_32x/ix174/B0 \
      75-/xscan_0_0_cch_scan_32x/ sum_add_0_ix55/B0
// /xscan_0_0_cch_scan_32x/ix157 (43) NAND
// A I 7-/myop1<0> [3]
// B I 17-/myop2[3]
// Y O 60-/xscan_0_0_cch_scan_32x/ix174/B0 \
      75-/xscan_0_0_cch_scan_32x/ sum_add_0_ix55/B0
// /xscan_0_0_cch_scan_32x/ix157 (43) NAND
// A I 7-/myop1<0> [3]
// B I 17-/myop2[3]
// Y O 60-/xscan_0_0_cch_scan_32x/ix174/B0 \
      75-/xscan_0_0_cch_scan_32x/ sum_add_0_ix55/B0
```

Related Topics

[Set Gate Level](#)

[Set Gate Report](#)

Report Initial States

Scope: All modes

Displays the initial state settings of the specified design instances.

Usage

REPort INitial States [-All | *instance_name...*]

Description

The Report Initial States command displays different information regarding the initialization settings depending on the mode from which you issue the command. If FlexTest is in the Setup mode, the command displays the initialization settings that you created by using the Add Initial States command. If FlexTest is in any other mode, the command displays all the initial state settings (including those in any **test_setup** procedures).

Arguments

- -All
An optional switch that displays the initialization settings for all design hierarchical instances. This is the default.
- *instance_name*
An optional, repeatable string that specifies the names of the design hierarchical instances for which you want to display the initialization settings.

Examples

The following example assumes you are not in Setup mode and displays all the current initial settings:

```
add initial states 0 /amm/g30/ff0
set system mode atpg
report initial states

0 /amm/g30/ff0
```

Report Lists

Scope: Atpg, Fault, and Good modes

The Report Lists command displays pins the tool is currently set up to report on during simulation.

Usage

REPort LIsts

Description

Use the [Add Lists](#) command to add pins to the list and the [Delete Lists](#) command to remove pins from the list.

When you switch to Setup mode, the tool discards all pins from the list.

Arguments

None.

Examples

The following example displays the pins whose values the tool will report during simulation:

```
add lists /i_1006/o /i_1007/o
report lists

2 pins are currently monitored.
i_1006/o
i_1007/o
```

Related Topics

[Add Lists](#)

[Set List File](#)

[Delete Lists](#)

Report Loops

Scope: Atpg, Fault, and Good modes

Prerequisites: This command requires a flattened simulation model created in the current tool session (as opposed to a previously saved flattened model read in at invocation). The tool flattens the design to the simulation model when you first attempt to exit Setup mode or when you issue the [Flatten Model](#) command.

Displays information about circuit loops.

Usage

REPort LOPs [-All | *loop_id#...*] [-Display *tab...*] [{> | >>} *file_pathname*]

Description

The Report Loops command displays information about currently identified loops in the circuit. For each loop, the report indicates whether the loop was broken by duplication. The report shows loops unbroken by duplication to be broken instead by a constant value, which means the loop is either a coupling loop or has a single multiple-fanout gate. The report also includes the pin pathname and gate type of each gate in each loop.

You can write the loops report information to a file by using the command's redirection operators or the [Write Loops](#) command.

Arguments

- -All
An optional switch that specifies to report all the loops in the circuit. This is the default.
- *loop_id#*
An optional, repeatable, positive integer that specifies the identification number of a particular loop to report. The tool assigns loop identification numbers consecutively, starting with 1.
- -Display *tab...*
A switch and literal that displays the reported information graphically in the specified Tessent Visualizer tab(s).
- > *file_pathname*
An optional redirection operator and pathname pair for creating or replacing the contents of *file_pathname*.
- >> *file_pathname*
An optional redirection operator and pathname pair for appending to the contents of *file_pathname*.

Examples

Example 1

The following example displays a list of all the loops in the circuit:

```
set system mode atpg  
report loops  
  
Loop = 1: not_duplicated (coupling loop)  
    my_design/my_minibus (SBUS)  
    my_design/PAD (BUF)  
    my_design/my_minibus (Z2X)  
Loop = 2: not_duplicated (coupling loop)  
...  
Loop = 8: not_duplicated (single multiple fanout)  
    my_design/al/pl/padx (BUF)  
    my_design/al/pl/pad (BUF)  
    my_design/pad (WIRE)
```

Example 2

The following example displays loop 8 only:

```
report loops 8  
  
Loop = 8: not_duplicated (single multiple fanout)  
    my_design/al/pl/padx (BUF)  
    my_design/al/pl/pad (BUF)  
    my_design/pad (WIRE)
```

Example 3

The following example writes the display information for loop 8 to a new file named *my_loop_file*:

```
report loops 8 > my_loop_file  
  
... writing to file my_loop_file
```

Related Topics

[Report Feedback Paths](#)

[Write Loops](#)

Report Nofaults

Scope: All modes

Displays the nofault settings for the specified pin pathnames or pin names of instances.

Usage

```
REPort NOfaults pathname... | -All [-Instance] [-Stuck_at {01 | 0 | 1}] [-Class {Full | User | System}] [{> | >>}] file_pathname
```

Description

The Report Nofaults command displays, for pin pathnames or pin names of instances, the nofault settings that you previously specified with the Add Nofaults command.

Arguments

- *pathname*
A repeatable string that specifies the pin pathnames or the instance pathnames for which you want to display the nofault settings. If you specify an instance pathname, you must also specify the -Instance switch.
- -All
A switch that displays the nofault settings on either all pin pathnames or, if you also specify the -Instance switch, all pin names of instances.
- -Instance
An optional switch that specifies that the *pathname* or -All argument indicates instance pathnames.
- -Stuck_at 01 | 0 | 1
An optional switch and literal pair that specifies the stuck-at or transition nofault settings you want to display. The choices are as follows:
 - 01 — A literal that specifies to display both the “stuck-at-0” and “stuck-at-1” nofault settings for stuck-at faults; or both the “slow-to-rise” and “slow-to-fall” nofault settings for transition faults. This is the default.
 - 0 — A literal that specifies to display only the “stuck-at-0” nofault settings (“slow-to-rise” nofault settings for transition faults).
 - 1 — A literal that specifies to display only the “stuck-at-1” nofault settings (“slow-to-fall” nofault settings for transition faults).
- -Class Full | User | System
An optional switch and literal pair that specifies the source (or class) of the nofault settings which you want to display. The literal choices are as follows:
 - Full — A literal that displays all the nofault settings in the user and system class. This is the default.

User — A literal that displays only the user-entered nofault settings.

System — A literal that displays only the netlist-described nofault settings.

- `>file_pathname`
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- `>>file_pathname`
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example displays all pin names of the instances that have the nofault settings:

```
add nofaults i_1006 i_1007 i_1008 -instance  
report nofaults
```

Related Topics

[Add Faults](#)

[Delete Nofaults](#)

[Add Nofaults](#)

[Report Faults](#)

[Delete Faults](#)

Report Nonscan Cells

Scope: Atpg, Fault, and Good modes

Displays the non-scan cells whose model type you specify.

Usage

REPort NOncan Cells [-All | -INITX | -TIE0 | -TIE1 | -Hold | -Data_capture]

Description

The Report Nonscan Cells command displays the instance name, gate number, and model type of the non-scan cells that you specify. The tool assigns a model type to the non-scan cells during the Design Rules Check (DRC) in order to model non-scan memory behavior more exactly.

During scan loading, scan clocks can affect non-scan memory elements. The worst case is to assume that all non-scan memory elements will have an unknown state right after scan loading, which makes some faults difficult to test. Therefore, the tool assigns an appropriate model type to the non-scan cells, which sets them to known values. The following argument descriptions describe the conditions that the tool uses to make the various model type assignments.

Arguments

- **-All**
An optional switch that displays all non-scan cells. This is the default.
- **-INITX**
An optional switch that displays the non-scan cells which FlexTest initializes to X after each loading.
- **-TIE0**
An optional switch that displays the non-scan cells which are always 0 after each loading and before the next unloading.
Non-scan cells that the tool models as TIE0 indicate that the pin constraints hold the cell's value during non-scan operation.
- **-TIE1**
An optional switch that displays the non-scan cells which are always 1 after each loading and before the next unloading.
Non-scan cells that the tool models as TIE1 indicate that the pin constraints hold the cell's value during non-scan operation.
- **-Hold**
An optional switch that displays the non-scan cells that hold their value during each loading, when all scan capture clocks are off.
Non-scan cells that FlexTest models as Clock Hold indicate that, during the scan loading, all scan capture clocks of the non-scan cell are off.

- **-Data_capture**

An optional switch that displays the non-scan cells that hold their data value during each loading, when all scan capture clocks but one are off.

Non-scan cells that FlexTest models as Data Hold indicate that, during the scan loading, all but one of the scan capture clocks of the non-scan cell are off, that one clock is active at least once, and that its corresponding data input does not change during scan loading.

Examples

The following FlexTest example displays only the non-scan cells that FlexTest initializes to X after each loading.

```
add scan groups g1 proc.g1  
add scan chains c1 g1 scin scout  
add clocks 0 clk  
set system mode atpg  
report nonscan cells -initx  
add faults -all  
run
```


Report Nonscan Handling

Scope: All modes

Displays the overriding learned behavior classification for the specified non-scan elements.

Usage

REPort NOncan Handling [*element_pathname*... | **-All**]

Description

The Report Nonscan Handling command displays the learned behavior classification you created using the Add Nonscan Handling command.

Arguments

- *element_pathname*
A repeatable string that specifies the pathname to the non-scan element for which you want to display the current user-defined learned behavior classification.
- **-All**
A switch that displays the user-defined learned behavior classifications for all non-scan elements. This is the default.

Examples

The following example explicitly defines how you want FlexTest to handle two non-scan elements, and then reports on the current list of learned behavior overrides for the design rules checker:

```
add nonscan handling tie0 i_6_16 i_28_3
report nonscan handling

TIE0   I_6_16
TIE0   I_28_3
```

Related Topics

[Add Nonscan Handling](#)

[Delete Nonscan Handling](#)

Report Output Masks

Scope: All modes

Displays a list of the currently masked primary output pins.

Usage

FREPort OUtput Masks

Description

The Report Output Masks command displays the masked primary output pins using the Add Output Masks command. When you mask a primary output pin, the tool marks that pin as an invalid observation point during the fault detection process. The tool uses all unmasked primary output pins as possible observation points to which the effects of all faults propagate for detection.

Arguments

None.

Related Topics

[Add Output Masks](#)

[Set Output Masks](#)

[Delete Output Masks](#)

Report Pin Constraints

Scope: All modes

Displays the pin constraints added to the primary inputs with the Add Pin Constraint command.

Usage

REPort PIn Constraints [-All | *primary_input_pin...*]

Description

The Report Pin Constraints command, as does the Add Pin Constraint command, performs slightly differently depending on which tool you are using. The following paragraphs describe how the command operates for each tool.

The command displays the cycle behavior constraints of the specified primary inputs. If you do not specify any primary input pins, the command displays the constraints of all the primary inputs. To change the cycle behavior constraints of the primary inputs, use either the Add Pin Constraint or Setup Pin Constraints commands.

Arguments

- -All
An optional switch that displays the current constraints for all primary input pins. This is the default.
- *primary_input_pin*
An optional repeatable string that specifies a list of primary input pins whose constraints you want to display.

Examples

The following FlexTest example displays the cycle behavior constraints of all primary inputs:

```
set test cycle 2
add pin constraint ph1 R1 1 0 1
add pin constraint ph2 R0 1 0 1
report pin constraints -all
```

Related Topics

[Add Pin Constraints](#)

[Setup Pin Constraints](#)

[Delete Pin Constraints](#)

Report Pin Equivalences

Scope: All modes

Displays the pin equivalences of the primary inputs.

Usage

REPort PIn Equivalences

Description

The Report Pin Equivalences command displays a list of primary inputs which you restricted to be at equivalent or complementary values using the Add Pin Equivalences command.

Arguments

None.

Examples

The following example displays all pin equivalences added to the primary inputs:

```
add pin equivalences indata2 indata4
add pin equivalences indata3 -invert indata5
report pin equivalences
```

Related Topics

[Add Pin Equivalences](#)

[Delete Pin Equivalences](#)

Report Pin Strokes

Scope: All modes

Displays the current pin stroke timing for the specified primary output pins.

Usage

REPort PIn Strokes [-All | *primary_output_pin...*]

Description

The Report Pin Strokes command displays the stroke time of each primary output pin that you specify. If you issue the command without any arguments, FlexTest displays all of the pin strokes.

Arguments

- -All
An optional switch that displays the pin stroke values for all of the primary output pins. This is the default.
- *primary_output_pin*
An optional, repeatable string that specifies a list of primary output pins whose pin stroke timing you want to display.

Examples

The following example displays the stroke times of all primary outputs:

```
set test cycle 3
add pin strokes 1 outdata1 outdata3
report pin strokes
```

Related Topics

[Add Pin Strokes](#)

[Setup Pin Strokes](#)

[Delete Pin Strokes](#)

Report Primary Inputs

Scope: All modes

Displays a list of the primary inputs of a circuit.

Usage

```
REPort PRimary Inputs [-All | net_pathname... | primary_input_pin...] [-Class {Full | User |  
System}] [-Verbose] [{> | >>} file_pathname]
```

Description

You can choose to display either the user class, system class, or full classes of primary inputs. Additionally, you can display all of the primary inputs or a specific list of primary inputs. If you issue the command without specifying any arguments, then the tool displays all the primary inputs.

Arguments

- All
An optional switch that displays all the primary inputs. This is the default.
- *net_pathname*
An optional repeatable string that specifies the circuit connections whose user-class primary inputs to display.
- *primary_input_pin*
An optional repeatable string that specifies a list of system-class primary input pins to display.
- -Class Full | User | System
An optional switch and literal pair that specifies the source (or class) of the primary input pins to display. The literal choices are as follows:
 - Full — Displays all the primary input pins in the user and system class. This is the default.
 - User — Displays only the user-entered primary input pins.
 - System — Displays only the netlist-described primary input pins.
- -Verbose
An optional switch that displays all the nets grouped under a primary input.
- > *file_pathname*
An optional redirection operator and pathname pair used at the end of the argument list for creating or replacing the contents of *file_pathname*.

- `>> file_pathname`

An optional redirection operator and pathname pair used at the end of the argument list for appending to the contents of *file_pathname*.

Examples

Example 1

The following example displays the full classes of primary inputs:

```
add primary inputs indata2 indata4
report primary inputs -class full
```

Example 2

The following example displays the user classes of primary inputs:

```
add pri input -internal /dut*/CLK_X -pin_name MYCLKX

//Note: Adding primary input MYCLKX merging 224 nets.

report primary inputs -class user

USER: /MYCLKX (merged internal pin)
Merges 224 nets
```

Related Topics

[Add Primary Inputs](#)

[Report Primary Outputs](#)

[Delete Primary Inputs](#)

[Write Primary Inputs](#)

Report Primary Outputs

Scope: All modes

Displays the specified primary outputs.

Usage

```
REPort PRimary Outputs [-All | net_pathname... | primary_output_pin...]  
[-Class {Full | User | System}] [{> | >>} file_pathname]
```

Description

The Report Primary Outputs command displays a list of primary outputs of a circuit. You can choose to display either the user class, system class, or full classes of primary outputs. Additionally, you can display all of the primary outputs or a specific list of primary outputs. If you issue the command without specifying any arguments, then the tool displays all the primary outputs.

Arguments

- **-All**
An optional switch that displays all primary outputs. This is the default.
- *net_pathname*
An optional, repeatable string that specifies the circuit connections whose user-class primary outputs you want to display.
- *primary_output_pin*
An optional, repeatable string that specifies a list of system-class primary output pins that you want to display.
- **-Class Full | User | System**
An optional switch and literal pair that specifies the source (or class) of the primary output pins that you want to display. The literal choices are as follows:
 - Full** — A literal that displays all of the primary output pins in the user and system class. This is the default.
 - User** — A literal that displays only the user-entered primary output pins.
 - System** — A literal that displays only the netlist-described primary output pins.
- **> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- **>> *file_pathname***
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example displays all primary outputs in the user class:

```
add primary outputs outdata1 outdata3 outdata5  
report primary outputs -class user
```

Related Topics

[Add Primary Outputs](#)

[Report Primary Inputs](#)

[Delete Primary Outputs](#)

[Write Primary Inputs](#)

Report Procedure

Scope: All modes except Setup mode

Displays the specified procedure.

Usage

REPort PProcedure {*procedure_name* [*group_name*] | -**All**} [{> | >>} *file_pathname*]

Description

The Report Procedure command displays all procedures or the specified procedure.

Arguments

- *procedure_name*
A string that specifies which procedure to display.
- *group_name*
An optional string that specifies a particular scan group from which to display the specified procedure.
- -**All**
A switch that specifies for the tool to display all procedures. This is the default.
- > *file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- >> *file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example displays all procedures:

```
report procedure -all
```

Related Topics

[Add Scan Groups](#)

[Save Patterns](#)

[Read Procfile](#)

[Write Procfile](#)

[Report Timeplate](#)

Report Pulse Generators

Scope: All modes

Displays the list of pulse generator sink (PGS) gates.

Usage

REPort PUlse Generators

Description

The Report Pulse Generators command displays the list of PGS gates that the tool identifies during the learning process.

Arguments

None.

Examples

The following example displays the current list of PGS gates:

```
set pulse generators on  
set system mode atpg  
report pulse generators
```

Related Topics

[Set Pulse Generators](#)

Report Read Controls

Scope: All modes

Displays all of the currently defined read control lines.

Usage

REPort REad Controls

Description

The Report Read Controls command displays all the read control lines specified using the Add Read Controls command. The display also includes the corresponding off-state with each read control line.

Arguments

None.

Examples

The following example displays a list of the current read control lines:

```
add read controls 0 r1 r3
add read controls 1 r2 r4
report read controls
```

Related Topics

[Add Read Controls](#)

[Delete Read Controls](#)

[Analyze Control Signals](#)

Report Scan Cells

Scope: Atpg, Fault, and Good modes

Displays a report on the scan cells that reside in the specified scan chains.

Usage

REPort SCan CELls [-All | {*chain_name* [*cell_id...*]}...] [-VErilog]

Description

The Report Scan Cells command provides a report on the scan cells within specific scan chains. The report provides the following information for each scan cell:

- Chain cell index number (cell ID), where 0 is the scan cell closest to the scan-out pin.
- Scan chain in which the scan cell resides.
- Scan group in which the scan cell resides. (The scan group data will be hidden if the design has only scan group.)
- Sequential element type, primitive type, and latch triggering type during shifting. The latch triggering type can be one of the following: LE, TE, AH, AL.
- Inversion data for each sequential element.
- Gate index number for each sequential element.
- Library cell name of each sequential element.
- Instance name of each sequential element.
- Library instance name, if one is defined.
- Cell input and output pins.

If you issue the command without specifying any arguments, then the tool displays a report on the scan cells for all scan chains.

Tip



: If you are saving a log of the tool session, reporting all scan cells will fill the logfile with information for every scan cell—a lot of information if the design is large! To reduce the impact on the logfile in this case, consider reporting on only one scan chain.

Note



Although scan cells are listed in the order of nearest-to-output first, the sequential elements inside one cell are not listed in that order (the master is always listed first).

Be aware that when reading hierarchical names into its internal database, the tool hides backslashes (\), converts periods (.) to slashes (/), and ignores spaces. For example, the tool

reads the cell name, “FFH.\E/MY_SCAN_FF .SCAN_FF5” in a WGL pattern, as “FFH/E/MY_SCAN_FF/SCAN_FF5”, storing the latter in its internal database. To report on this cell, you must reference it by the name that is stored internally.

Note



The difference between the internal hierarchical name and the name in the WGL file of the preceding example does not alter the correctness of the tool. That is, the output pattern file remains consistent with the original netlist.

Arguments

- **-All**
An optional switch that displays information for all the scan cells in all the scan chains. This is the default.
- *chain_name*
An optional, repeatable string that specifies a scan chain whose scan cell data you want to display. When you use this argument without accompanying cell IDs, the tool displays information for all the scan cells in the scan chain. If you follow the *chain_name* argument with the -Range switch (and beginning and ending cell IDs for the range), the tool displays information for every cell in the range. If you provide one or more cell IDs without the -Range switch, the tool displays information for just those cells in that chain.
- *cell_id*
An optional repeatable integer that specifies the index number (cell ID) of a scan cell and, together with the *chain_name* argument, identifies a specific scan cell whose data you want to display.
- **-Verilog**
An optional switch that outputs the cell instance and pin pathnames in Verilog syntax, rather than using the default netlist independent format.

Related Topics

[Add Scan Chains](#)

[Add Scan Groups](#)

Report Scan Chains

Scope: All modes

Displays specific information in a report for each scan chain.

Usage

REPort SCan CHains

Description

Displays the following information in a report for each scan chain:

- Name of the scan chain
- Name of the scan chain group
- Scan chain input and output pins
- Length of the scan chain
- Name of the scan clock(s)

If the DRC E8 handling is set to anything other than Ignore when leaving Setup system mode, the command additionally lists the clocks specified to be pulsed in the shift procedure in the test procedure file. This enhanced report is only available in non-Setup modes. The clocks reported are likely a superset of the clocks that are actually used to clock the scan cells. For a precise report of the shift clocks, use the [Report Scan Cells](#) command.

Arguments

None.

Examples

The following example displays a report of all the scan chains.

```
add scan groups group1 scanfile
add scan chains chain1 group1 indata2 outdata4
add scan chains chain2 group1 indata3 outdata5
report scan chains

chain = chain1 group = group1 input = /indata2 output =
/outdata4 length = 14 master_clock(s) = /CLK0_7
chain = chain2 group = group1 input = /indata3 output =
/outdata4 length = 15 master_clock(s) = /CLK8_15,/CLK0_7
```

Related Topics

[Add Scan Chains](#)

[Report Scan Cells](#)

[Delete Scan Chains](#)

Report Scan Groups

Scope: All modes

Displays a report on all the current scan chain groups.

Usage

REPort SCan Groups [{> | >>} *file_pathname*]

Description

The Report Scan Groups command provides the following information in a report for each scan chain group:

- Name of the scan chain group
- Number of scan chains within the scan chain group
- Number of shifts
- Name of the test procedure file, which contains the information for controlling the scan chains in the reported scan chain group

Arguments

- > *file_pathname*
An optional redirection operator and pathname pair for creating or replacing the contents of *file_pathname*.
- >> *file_pathname*
An optional redirection operator and pathname pair for appending to the contents of *file_pathname*.

Examples

The following example displays a report of all the scan groups:

```
add scan groups group1 scanfile
add scan groups group2 scanfile1
report scan groups
```

Related Topics

[Add Scan Groups](#)

[Delete Scan Groups](#)

Report Scan Instances

Scope: All modes

Displays the currently defined sequential scan instances.

Usage

REPort SCan Instances [-Class {Full | User | System}]

Description

The Report Scan Instances command displays the sequential scan instances which you added by using the Add Scan Instances command.

Arguments

- -Class Full | User | System

A switch and literal pair that specifies the source (or class) of the sequential scan instances which you want to display. The literal choices are as follows:

Full — A literal that displays all the scan sequential instances in the user and system class. This is the default.

User — A literal that displays only user-entered scan sequential instances.

System — A literal that displays only netlist-described scan sequential instances.

Examples

The following example displays all sequential scan instances from the scan instance list:

```
set system mode setup
add scan instances i_1006 i_1007 i_1008
report scan instances -class user
```

Related Topics

[Add Scan Instances](#)

[Delete Scan Instances](#)

Report Scan Models

Scope: All modes

Displays the sequential scan models currently in the scan model list.

Usage

REPort SCan Models

Description

The Report Scan Models command displays sequential models which you previously added to the scan model list by using the Add Scan Models command.

Arguments

None.

Examples

The following example displays all the sequential scan models from the scan model list:

```
set system mode setup
add scan models d_flip_flop1 d_flip_flop2
report scan models
```

Related Topics

[Add Scan Models](#)

[Delete Scan Models](#)

Report Statistics

Scope: All modes

The Report Statistics command displays a detailed statistics report to the screen.

Usage

```
REPort STAtistics [-Instance instance_pathname | -MODEL {ALL | model_name}] [{> | >>}  
  file_pathname]
```

Description

The statistics report information depends on which tool you use. The following paragraphs describe the contents of the statistics report for each tool.

The tool also reports CPU time that is the cumulative CPU time of the master process since you invoked the run.

The FlexTest statistics report lists the following four groups of information:

- Circuit Statistics which consists of total numbers for the following:
 - primary inputs
primary outputs
library model instances
netlist primitive instances
combinational gates
sequential elements
simulation primitives
scan cells
scan sequential elements
 - sequential instances
defined nonscan instances
nonscan instances identified by the DRC
defined scan instances
scan instances identified by the DRC
identified scan instances

- Fault List Statistics that consist of the following:
 - The number of collapsed and total faults that are currently in each class. The report does not display fault classes with no members.
 - The percentage of test coverage, fault coverage, and ATPG effectiveness for both collapsed and total faults.
- Test Patterns Statistics that list the total numbers for the following:
 - total patterns currently in the test pattern set
 - total number of patterns simulated in the preceding simulation process
- Runtime Statistics that lists the following:
 - Machine and user names
 - total user cpu time
 - total system cpu time
 - total memory usage

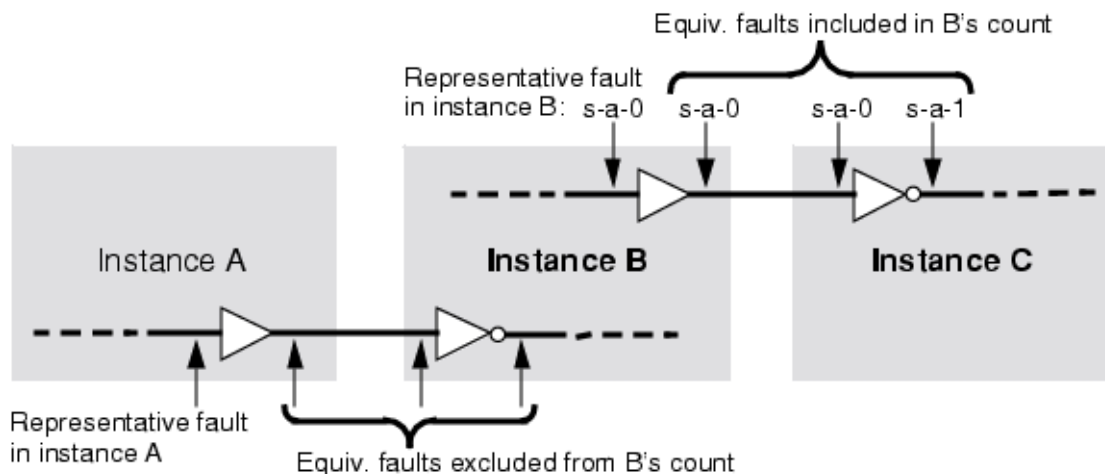
Arguments

- -Instance *instance_pathname*

An optional switch and string pair that reports fault statistics for a specific instance. The *instance_pathname* is the name of a circuit block whose statistics you want to report. Only fault statistics are affected by this option; pattern count statistics apply to the entire design.

As illustrated in [Figure 2-1](#), this switch uses the representative fault to determine whether to include a fault in the count for the specified block. For more information about representative faults, refer to [Fault Collapsing](#) and [Fault Reporting](#) in the *Scan and ATPG User's Manual*.

Figure 2-1. How the -Instance Switch Determines Fault Counts



- -Model ALL | *model_name* (**Model Mode only**)

An optional switch and string pair that specifies reporting statistics for all models or a single named model in an ATPG library. When specifying a single model, *model_name* must match an ATPG model in the ATPG library. If you use this switch in other than model mode, the tool issues the following error:

```
Error: -model reporting only available for -model tool
invocations.
```

- > *file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- >> *file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example shows a FlexTest statistics report for the Report Statistics command:

```
Total number of sequential instances    = 2
*****Circuit Statistics*****
# of primary inputs = 12
# of primary outputs = 6
# of library model instances = 14
# of combinational gates = 12
# of sequential elements = 2
# of simulation primitives = 62
# of scan cells = 2
# of scan sequential elements = 2
*****Fault List Statistics*****
Fault Class          Uncollapsed   Collapsed
Full (FU)             120           56
Det_simulation (DS)    72           28
Det_implication (DI)   48           28
Fault coverage         100.00%      100.00%
Test coverage          100.00%      100.00%
Atpg effectiveness     100.00%      100.00%
*****Test Patterns Statistics*****
Total Test Cycles Generated = 26
Total Scan Operations Generated = 13
Total Test Cycles Simulated = 26
Total Scan Operations Simulated = 13
***** Runtime Statistics *****
Machine Name      : checklogic
User Name         : Steve
User CPU Time     : 1.9 seconds
System CPU Time   : .6 seconds
Memory Used       : 2.137M
```

Related Topics

[Write Statistics](#)

Report Test Stimulus

Scope: Atpg, Fault, and Good modes

Displays the stimulus necessary to satisfy the specified set, write, or read conditions.

Usage

REPort TEST Stimulus

```
-Set { {id# | pin_pathname} {0 | 1 | Z} }...  
| -Write { {id# | RAM_instance_name} address_values data_values }...  
| -Read { {id# | RAM_instance_name} address_values }...  
[-Port port_number] [-Verbose] [-Noverbose] [-PRevious] [-STore] [-Frame frame#]
```

Description

It identifies how to sensitize scan chain blockage points. For example, if you first delete all scan groups and then go to the ATPG system mode, you can issue the Report Test Stimulus command for possible conditions to satisfy sensitization. That is, if the blockage was at an AND gate, you can try to set an input of the gate to 1.

If test generation is successful, the stimulus necessary to satisfy the specified conditions is displayed. The stimulus for scan cells is identified by the gate index number, instance name, and cell ID number of the scan chain.

Arguments

- **-Set** *id#* | *pin_path_name* **0** | **1** | **Z**

A switch with a repeatable argument and literal pair that specifies the pin and its value for which you want to generate the appropriate stimulus. You may specify multiple argument pairs with a single -Set switch.

id# — An integer that specifies the identification number of the gate for which you want to set the output pin. The gate cannot be a RAM or ROM gate.

pin_path_name — A string that specifies the pathname of the pin you want to set.

0 — A literal that sets the *id#* or *pin_path_name* to 0.

1 — A literal that sets the *id#* or *pin_path_name* to 1.

Z — A literal that sets the *id#* or *pin_path_name* to Z.

- **-Write** *id#* | *instance_name* *address_values* *data_values*

A switch with a repeatable argument that specifies the RAM to which you want to write and, optionally, its address and data values. You may specify multiple argument triplets with a single -Write switch. The switch arguments are:

id# — An integer that specifies the identification number of the RAM gate to which you want to write.

instance_name — A string that specifies the pathname of the RAM instance to which you want to write.

address_values — A required character string consisting of 0's and 1's that specifies the values you want to place on the RAM address lines. The least significant value must be the last character in the string. The number of characters in the string must not exceed the number of RAM address lines available.

data_values — An optional character string consisting of 0's, 1's, and X's that specifies the values you want to place on the RAM data lines. The least significant value must be the last character in the string. The number of characters in the string must not exceed the number of RAM data lines available.

If you do not specify the -Port switch, the command assumes the first port (port 0).

- **-Read *id#* | *instance_name address_values***

A switch with a repeatable argument that specifies the RAM from which you want to read and, optionally, its address value. You may specify multiple argument pairs with a single -Read switch.

id# — An integer that specifies the identification number of the RAM gate from which you want to read.

instance_name — A string that specifies the pathname of the RAM instance from which you want to read.

address_values — A required character string consisting of 0's and 1's that specifies the values you want to place on the RAM address lines. The least significant value must be the last character in the string. The number of characters in the string must not exceed the number of RAM address lines available.

If you do not specify the -Port switch, the command assumes the first port (port 0).

- **-Port *port_number***

An optional switch and integer pair that specifies the identification number of the port to use for reading or writing RAM. The port identification number is zero-based; that is, the first port is "port 0." The default is 0.

- **-Verbose**

An optional switch that displays the pattern that the command creates to satisfy the specified settings. This is the default.

- **-Noverbose**

An optional switch that specifies to not display the pattern that the command creates to satisfy the specified settings.

- **-PPrevious**

An optional switch that retains the settings from the previous Report Test Stimulus command and adds them to the current settings. When this switch is used, the command displays all of the retained settings. The default is to not retain the settings.

- **-STore**

An optional switch that places the command-created pattern in the internal test pattern area. You can then write this pattern to a file, in any format, by using the Save Patterns command. The default is to not place the pattern in the internal test pattern area.

- **-Frame *frame#***

An optional switch that specifies at which frame in a test cycle the specified set, write, or read conditions need to be satisfied. If this switch is not specified, every time frame is checked in the test cycle to find a time frame that can generate the stimulus successfully. The successful stimulus time frame is stored, so it can be used the next time the Report Test Stimulus command and -Previous switch combination is executed. However, if the test stimulus is not generated, you can specify the frame to use current conditions rather than previous conditions.

Examples

The following example command line is used for determining what stimulus is needed to write 11011 to address location 01011 for a RAM gate (gate ID number is 67):

```
set system mode atpg  
report test stimulus -write 67 01011 11011
```

The following is an example of the display from the previous command line:

```
// Time = 0  
// Force 1 /W1 (1)  
// Force 0 /A1[4] (2)  
// Force 1 /A1[3] (3)  
// Force 0 /A1[2] (4)  
// Force 1 /A1[1] (5)  
// Force 1 /A1[0] (6)  
// Force 0 /OE (7)  
// Force 1 /D1[0] (14)  
// Force 1 /D1[1] (15)  
// Force 0 /D1[2] (16)  
// Force 1 /D1[3] (17)  
// Force 1 /D1[4] (18)
```

Related Topics

[Save Patterns](#)

Report Testability Data

Atpg and Fault modes

Analyzes collapsed faults for the specified fault class and displays the analysis.

Usage

REPort TEstability Data **-Class** *class_type* [*filename*] [-Replace]

Description

The Report Testability Data command identifies and displays any circuitry connections that may cause test coverage problems for the specified fault classes.

The display may include any of the following connection types:

- Tied or blocked by constraints
- Connected with clock lines
- Tie-x gates
- Tri-state-driver enable lines
- Non-scan latches
- Non-observable scan latches
- RAM gates
- Unresolved wired-gates
- Primary outputs that connect to clocks
- Tied latches
- ROM gates
- No-strobed POs
- Uninitialized latches
- Internally tied gates (identified by learning)

If you specify a *filename* argument, the command writes to the file the list of faults with their connection information, and displays to the screen the summary of the results.

Arguments

- **-Class** *class_type*

A switch and literal pair that specifies the class of faults whose collapsed faults you want to analyze for test coverage problems. The *class_type* literal can be either a fault class code or a fault class name.

[Table 2-7](#) on page 270 lists the valid fault class codes and their associated fault class names. Use either the code or the name when specifying the *class_type* literal.

- *filename*
An optional string that specifies the name of the file to which you want to write the fault connection information. The command still displays a summary of the results to the screen.
- -Replace
An optional switch that specifies to replace the contents of the *filename* or filename.protected file, if they already exist.

Examples

The following example analyzes the redundant (RE) faults to identify connections causing test coverage problems and displays a summary of the results:

```
set system mode atpg
add faults -all
report testability data -class re

// fault analysis summary of 206 faults
// number faults connected to RAM = 196
// number faults unclassified = 10
```

Related Topics

[Add Faults](#)

[Load Faults](#)

[Analyze Fault](#)

[Report Faults](#)

[Delete Faults](#)

[Write Faults](#)

Report Tied Signals

Scope: All modes

Displays a list of the tied floating signals and pins.

Usage

REPort Tied Signals [-Class {Full | User | System}] [{> | >>} *file_pathname*]

Description

The Report Tied Signals command displays either the user class, system class, or full classes of tied floating signals and pins. If you do not specify a class, the command displays all the tied floating signals and pins.

Arguments

- -Class Full | User | System

An optional switch and literal pair that specifies the source (or class) of the tied floating signals or pins which you want to display. The literal choices are as follows:

Full — A literal that displays all the tied floating signals or pins in the user and system class. This is the default.

User — A literal that displays only the tied floating signals or pins created using the Add Tied Signals command. This includes all instance-based blackbox tied signals.

System — A literal that displays only the netlist-described tied floating signals or pins.

- > *file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.

- >> *file_pathname*

An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example displays the tied floating signals from the user class:

```
add tied signals 1 vcc vdd
report tied signals -class user
```

Related Topics

[Add Tied Signals](#)

[Report Black Box](#)

[Delete Tied Signals](#)

[Setup Tied Signals](#)

Report Timeplate

Scope: All modes except Setup mode

Displays the specified timeplate.

Usage

REPort TImeplate {*timeplate_name* | -All} [{> | >>} *file_pathname*]

Description

The Report Timeplate command displays all timeplates or the specified timeplate.

Arguments

- *timeplate_name*
A string that specifies which timeplate to display.
- -All
A switch that specifies for the tool to display all timeplates. This is the default.
- > *file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for creating or replacing the contents of *file_pathname*.
- >> *file_pathname*
An optional redirection operator and pathname pair, used at the end of the argument list, for appending to the contents of *file_pathname*.

Examples

The following example displays the timeplate named *tp1*.

```
report timeplate tp1
```

Related Topics

[Add Scan Groups](#)

[Save Patterns](#)

[Read Procfile](#)

[Write Procfile](#)

[Report Procedure](#)

Report Version Data

Scope: All modes

Displays the current software version information.

Usage

REPort VErSion Data [{> | >>} *file_pathname*]

Description

The Report Version Data command displays information relating to the software title, version, and date.

Arguments

- > *file_pathname*
An optional redirection operator and pathname pair for creating or replacing the contents of *file_pathname*.
- >> *file_pathname*
An optional redirection operator and pathname pair for appending to the contents of *file_pathname*.

Examples

The following is an example of the Report Version Data display information:

report version data

```
Version data: FlexTest v8.2004_1.10 Mon Feb 9 22:43:45 PST 2004
```

Report Write Controls

Scope: All modes

Displays the currently-defined write control lines and their off-states.

Usage

REPort WRite Controls

Description

The Report Write Controls command displays the write control lines, with corresponding off-states, added using the Add Write Controls command.

Arguments

None.

Examples

The following example adds four write control lines and then displays a list of the control line definitions:

```
add write controls 0 w1 w3
add write controls 1 w2 w4
report write controls
```

Related Topics

[Add Write Controls](#)

[Delete Write Controls](#)

[Analyze Control Signals](#)

Reset Au Faults

Scope: Atpg and Fault modes

Reclassifies the faults in certain untestable categories.

Usage

RESet AU Faults

Description

Reclassifies the faults in certain untestable categories as shown in [Table 2-10](#).

Table 2-10. Untestable Faults that are Reclassified by Reset Au Faults

Untestable Fault	Fault Reclassification
Oscillatory untestable (OU)	Oscillatory testable (OT)
Hypertrophic untestable (HU)	Hypertrophic testable (HT)
Possibly detected untestable (PU)	Possibly detected testable (PT)
ATPG untestable (AU)	Uncontrolled (UC)
Un-initialized (UI)	Uncontrolled (UC)

Deterministic fault simulators classify some untestable faults differently depending on the algorithm. You can use the Reset Au Faults command to align those (potentially) misclassified faults.

When the command changes the fault classification, the tool then has the ability to analyze and further reclassify each previously-untestable fault. Allowing the tool the ability to analyze and reclassify those particular untestable faults increases the tool's efficiency.

Arguments

None.

Examples

The following example sets up the tool to run the simulation with an external pattern file and resets the ATPG untestable faults so that the tool can determine their appropriate fault category:

```
set pattern source external testpatterns
load faults /user/design/fault_file -restore
reset au faults
run
```

Related Topics

[Load Faults](#)

Reset State

Scope: Atpg, Fault, and Good modes

Resets the circuit status.

Usage

RESet SState

Description

The Reset State command resets the circuit status differently depending on the mode from which you issue the command. The following describes what the command does for each system mode:

- ATPG system mode — Resets all faults, except AU and RE, to be undetected in order to run a new simulation; deletes the internal patterns; and resets the circuit status.
- Fault system mode — Resets the faults to be undetected in order to run a new simulation and resets the circuit status.
- Good system mode — Resets the circuit status.

When FlexTest resets the circuit status, it re-reads the RAM initialization files and resets all sequential elements to their initial states.

Arguments

None.

Resume Interrupted Process

Scope: All modes

Prerequisites: The Set Interrupt Handling must be on and you must interrupt a FlexTest command with a Control-C.

Continues a command that you placed in a suspended state by entering a Control-C interrupt.

Usage

RESume INterrupted Process

Description

The Resume Interrupted Process command resumes (continues) a FlexTest command interrupted by pressing the Control-C keys. This removes the interrupted command from the suspend-state and allows the command process to complete.

For a list of commands that you can issue while an interrupted command is in the suspend-state, refer to the [Set Interrupt Handling](#) command description.

Arguments

None.

Examples

The following example enables the suspend-state interrupt handling, begins an ATPG run, and (sometime before the run completes) interrupts the run:

```
set interrupt handling on
set system mode atpg
add faults -all
run
<Ctrl-C>
```

Now with the Run suspended the example continues by writing the existing untestable faults to a file for review and then resuming the Run:

```
write faults faultlist -class ut
resume interrupted process
```

Related Topics

[Abort Interrupted Process](#)

[Set Interrupt Handling](#)

Run

Scope: Atpg, Fault, and Good modes

The Run command executes a simulation or ATPG process.

Usage

RUN [-Begin *begin_number*] [-End *end_number*] [-Record *cycles*] [-REtain_abort]
[-NOAnalyze] [-Message *integer*]

Description

The Run command can be used multiple times to further increase test coverage for an ATPG process. The [Update Implication Detections](#) command is automatically issued after the Run command executes a fault simulation when using external patterns or when using internal patterns in non-Atpg mode.

The CPU time the tool reports is the time local to the command. If the command has multiple phases, then the reported CPU time is with respect to the beginning of that phase.

The Run command performs a fault or “good” simulation, or an ATPG process, using the current test pattern source (only in fault and good simulation modes). To suspend or terminate the simulation, use the Set Interrupt Handling command and the Control-C keys.

During a random and deterministic ATPG run, the Run command displays statistics. The statistics consist of the number of test cycles, the number of detected faults and other faults placed into a fault class, the test coverage, and the ATPG effectiveness.

By default, FlexTest outputs information for any change to the status of the fault list, which increases the storage requirement for every ATPG run. To reduce the information displayed or written out to a logfile, you can specify the periodicity of the information reported after issuing the Run command by using the -Message switch.

If a run analysis fails, you can either 1) use the Analyze ATPG Constraints command to learn which ATPG constraints have caused the problem, or 2) issue the Run command again using the -Noanalyze switch to skip the analysis and proceed with normal test generation, or 3) increase the abort limit and reissue the Run command again to see if the run succeeds.

Arguments

- -Begin *begin_number*

An optional switch and integer pair that specifies the number of the FlexTest cycle at which you want the command to begin running a simulation or ATPG process. The default cycle number is 0.

- **-End *end_number***

An optional switch and integer pair that specifies the number of the FlexTest cycle at which you want the command to stop running the process. The default cycle number is the last cycle of the test pattern set.

- **-Record *cycles***

An optional switch and integer pair that specifies the number of last test cycles for which you want to save (record) internal values. You can display the internal values by using the Report Gates command. You can use this argument for backward tracing internal values to a problem source.

- **-Message *integer***

An optional switch and integer pair that specifies the period, in minutes, of displaying the transcript or writing a logfile. A logfile is defined by using the Set Logfile Handling command. Information is reported at the given period, *integer*, only if there was a status change during the period. The final status before the completion of the run or just before the run is interrupted is reported.

- **-REtain_abort**

An optional switch that specifies to not target any faults that were targeted and aborted in a previous run that was interrupted before it finished.

Normally, a fault targeted and aborted by ATPG in a given run can be retargeted in a subsequent run. However, retargeting is not always desirable. For example, if you interrupted the previous run using Control-C, or a command like “set atpg limits -cpu” stopped the run, you may want to rerun with the same settings (abort limit, and so on). This switch prevents the tool from retargeting the previously targeted, aborted faults on the rerun.

You can interrupt what would have been one run any number of times without retargeting any fault aborted in any of the interrupted runs; just include this switch with the Run command following each interrupted run.

- **-NOAnalyze**

An optional switch that specifies for the tool to skip the analysis of test generation problems.

Note



Because this switch prevents the tool from performing its normal analysis, it can easily cause the tool to target all faults and expend much ATPG effort, yet fail to generate any patterns. The extra effort may also give the appearance the tool is stuck. Use this switch only in special cases where you are willing to accept these risks.

Examples

The following example runs an ATPG process after adding all faults to the circuit:

```
set system mode atpg
add faults -all
run
```

Related Topics

[Compress Patterns](#)

[Set Logfile Handling](#)

[Report Gates](#)

[Set Pattern Source](#)

[Set Atpg Limits](#)

[Set System Mode](#)

[Set Interrupt Handling](#)

[Write Faults](#)

Save History

Scope: All modes

Saves the command line history file to the specified file.

Usage

SAVe Hlstory *filename* [-Replace]

Description

The Save History command saves the list of previously executed commands in the file that you specify. You can then execute the file using the Dofile command.

Arguments

- *filename*
A required string that specifies the name of the file in which the tool saves the command line history list.
- -Replace
An optional switch that specifies for the tool to overwrite the contents of *filename*, if a file by that name already exists.

Examples

The following example displays the current history list, then saves it in a file called *my_history*, which already exists.

history -nonumbers

```
dof instructor/fault.do
set system mode atpg
set fault type stuck
add faults -all
run
report statistics
history -nonumbers
```

save history my_history -replace

Related Topics

[Dofile](#)

[History](#)

Save Patterns

Scope: ATPG mode

Saves the current test pattern set to a file in a specified format.

Usage

SAVe PATterns *pattern_filename* [*format_switch*] [-Replace]
[-PROcfile *procedure_filename*] [-PARALlel] [-Serial] [-NOInitialization]
[-BEGin *pattern_number*] [-END *pattern_number*]
[-CELL_placement {Bottom | Top | None}]
[-ALL_test | -CHain_test | -CYcle_test] [-One_setup]
[-NOPadding | -PAD0 | -PAD1] [-PAttern_size *integer*] [-Noz]

Description

By default, the module name created for Verilog test benches consist of the specified design name followed by an underscore, followed by the name of the specified test procedure file followed by `_ctl`. All periods are converted to underscores. For example:

design_name_test_proc_file_name_ctl

Note



Type Ctrl-C on the command line to cancel the Save Pattern operation with out creating any files.

Arguments

- *pattern_filename*

A required string that specifies a name for the saved test pattern file. If the filename ends with “.gz”, the file is automatically compressed with the GNU gzip utility. If the filename ends with “.Z”, the file is automatically compressed with the UNIX compress utility. For more information, see the [Set File Compression](#) command.

- *format_switch*

An optional switch that specifies a format for the saved test patterns. You should always save simulation patterns in the same format as the design netlist.

ASIC vendor test data formats usually support only a single timing file. You can specify the test timing that each ASIC vendor requires by using different timing definition files for each format.

Any time you save the pattern set in an ASIC vendor data format, you should also save the patterns in ASCII, binary, STIL or WGL format as backup. This is in case you want to read in the patterns from an external file using the [Set Pattern Source](#) External command. Only ASCII, binary, STIL or WGL formatted test patterns can be read by the ATPG tools.

Options include:

-Ascii — A switch that writes test patterns in the standard ASCII format. The ASCII format includes the statistics report, environment settings, scan structure definition, scan chain functional test, scan test patterns, and the scan cell information. This is the default.

Excess load and unload values are padded with Xs. You can override this behavior with the **-Nopadding** switch.

If you use the **-External** or the **-Begin** and **-End** switches, test coverage and fault information is omitted from the ASCII pattern set.

-Test_setup — An optional switch and string that saves non-scan test patterns as a test_setup procedure in a test procedure file.

-STil — A switch that writes the patterns in STIL format that conforms to IEEE Std. 1450.0. You can customize the output format with a parameter file.

-Verilog — A switch that writes the patterns in Verilog format. The Verilog format contains pattern information and timing information from the timing file as a sequence of events. You can use the Verilog patterns to interface to a Verilog simulator such as ModelSim. You can customize the output format with a parameter file.

-Wgl — A switch that writes the patterns in the WGL format. The WGL format contains the waveform pattern information and any timing information from the timing file. You can use the WGL format patterns to generate test patterns in a variety of tester and simulator formats.

Note

The WGL format does not support patterns for designs with differential input pins.

-Fjtdl — A switch that writes the test patterns in the Fujitsu ASIC format: FTDL-E.

-MITdl — A switch that writes the test patterns in the Mitsubishi ASCII format: MITDL.

For MITDL format there is restriction that multiple pulse timing must be a cyclical repetition of the first pulse. Consequently, multi-pulse and double-pulse timing in the procedure file only works in the MITDL output without an error if the timing fits the restrictions of the MITDL syntax.

-Mode Lsi — A switch that changes the functionality of the Verilog and WGL output formats so that the saved pattern files meet LSI Logic requirements. This switch is valid only with the **-Verilog** and **-Wgl** switches.

-Tltdl — A switch that writes the test patterns in the Texas Instruments ASIC format: TDL91. You can customize the output format with the parameter file.

Note

If the design uses a single timeplate the TDL version 4.3 format is used. This removes the **TIMING**, **END_TIMING**, and **SET_TIMING** statements from the test pattern file. Also, the **VERSION** statement changes from 6.0 to 4.3. However, if the design uses multiple timeplates, then the TDL version output is 6.0.

-TSTl2 — A switch that writes the patterns in Toshiba ASIC format: TSTL2.

- -Replace

An optional switch that replaces the contents of an existing *pattern_filename*.

- -PROfile *procedure_filename*

An optional switch and string that specifies the name of the test procedure file to retrieve non-scan event timing information from. This option saves test patterns using the enhanced pattern output that includes timing information from the test procedure file.

You can specify the procedure filename with this argument or specify it before using this command by issuing the Read Procfile command. You cannot use this argument with the -Ascii or -Binary formats.

- -PARALlel

An optional switch that saves all scan cells in parallel. This is the default.

In designs with scan cells, only scan pins are active during the scan shift cycles. If the tool tries to represent the state of each pin during each shift cycle, it may produce very large pattern files. Simulating the shift operations of these patterns might require a considerable amount of time if you use a different simulator. You can avoid these problems by saving all scan cells in parallel. The -Parallel switch is compatible with all formats.

- -Serial

An optional switch that saves all scan cells in series. You can only use this switch with the -STil, -STIL2005, -Tltdl (enhanced pattern output only), -Wgl, and -Verilog format type switches.

When you use this switch together with the -Verilog switch and the -Parameter switch in the same command, the tool checks the specified parameter file for the SIM_SHIFT_DEBUG keyword. If this keyword exists and is set to 1, every scan chain in the Verilog patterns contain a parallel observe of the entire scan chain for every shift, while serially shifting the test patterns. If this keyword is set to 2 or more, the parallel observe occurs for that many shifts, not all. This format is particularly useful for debugging scan chain problems because it allows you to locate the exact scan element causing a problem in a scan chain.

Tip



: Use SIM_SHIFT_DEBUG serial Verilog patterns only for debugging scan chain problems. They typically take more time to create than other kinds of patterns and contain large amounts of data (so file sizes tend to be large). Also, because they simulate serially, the patterns take more time to simulate. Consider using the -Chain, -Begin and -End switches to limit the size of the pattern file.

- -NOInitialization


A switch that writes patterns without creating the initialization cycle in the pattern file. The -Noinitialization switch is valid with the following output formats:

- STil

- Verilog
- WG1 (Ascii)

The -Noinitialization switch is valid with all enhanced pattern output formats.

Note

 If DRC was run with initialization enabled ([Set Drc Handling](#) -Initialization_cycle On), which is the default, using the -Noinitialization switch when saving patterns may result in simulation mismatches when you verify the patterns in a timing-based simulator.

- -B~~E~~gin *begin_number*

An optional switch and integer pair that specifies the number of the FlexTest cycle begin storing patterns. The default cycle number is 0.

If you save only a portion of the internal patterns in the -Ascii format, the tool does not include test coverage and fault information.

- -END *end_number*

An optional switch and integer pair that specifies the number of the FlexTest cycle at which you want the command to stop storing patterns. This argument is inclusive; therefore, the tool stores the pattern identified by the *end_number* you specify. The cycle number is the cycle of the pattern set.

If you save only a portion of the internal patterns in the -Ascii format, the tool does not include test coverage and fault information.

- -CELL_placement Bottom | Top | None

An optional switch and literal pair that controls the placement of the scan cell data in the ASCII pattern file. The literal choices are as follows.

Bottom — A literal that places the scan data after the patterns, at the end of the file. This is the default.

Top — A literal that places the scan data before the patterns, at the top of the file.

None — A literal that excludes the scan data from the file.

- -ALL test

An optional switch that specifies for the tool to save the chain test and the cycle test.

- -CHain_test

An optional switch that specifies for the tool to save only the chain test patterns in the pattern file.

- -CYcle_test

An optional switch that specifies for FlexTest to save only the cycle test in the pattern file.

- **-One_setup**

An optional switch that specifies to apply only one test setup procedure when both chain and scan test patterns are saved in one pattern file. The single test setup procedure is applied before the chain test patterns. By default, one test setup procedure is applied before the chain test patterns and another before the scan test patterns. When saving patterns in Verilog, STIL, and WGL, a single test_setup procedure is saved in the first file generated when the -maxloads switch is used with the -One_setup switch.

- **-NOPadding**

An optional switch that removes X values from the input data in saved ASCII test patterns. By default, X values are used for unknown or don't care bits in the input data. Use this switch to eliminate X values in the input data due to short scan chains. This switch can only be used for ASCII test patterns.

To load unpadding test patterns into a tool, you must use the Set Pattern Source command with the -Nopadding switch.

- **-PAD0 | -PAD1**

Optional switches that replace X values with either 1 or 0 value in the saved test patterns. By default, X values are used for unknown or don't care bits in the input and output data.

Options include:

-PAD0 — Replaces X values with 0 values.

-PAD1 — Replaces X values with 1 values.

This switch is only valid for test patterns that support scan chain padding.

- **-Noz**

An optional switch that changes all Z output values to the value specified by the last Set Z Handling command.

- **-PATtern_size *integer***

A optional switch and integer pair which specifies the size of the memory buffer and pattern file in which to save. *Integer* is given in kilobytes.

Note

The -Pattern_size switch is valid only when using -Verilog output.

The default pattern size is 128MB. However, any size specified for a pattern size will be adjusted to hold a multiple of the largest pattern.

Examples

Example 1

The following example performs an ATPG run, and then saves only the first 15 test patterns to a file in the Verilog format, including the timing information contained in the timing file:

```
set system mode atpg  
add faults -all  
run  
save patterns file1 -verilog timefile -end 14
```

Example 2

The following example illustrates the module name created in the enhanced Verilog output when using the -Procfile switch. If the design name is “MAIN”, and you issue the following Save Patterns command:

```
save patterns pattern1.pat -procfile -verilog -replace
```

the module name in the testbench will be “MAIN_pattern1_pat_ctl”.

Related Topics

[Add Scan Groups](#)

[Report Timeplate](#)

[Read Procfile](#)

[Set Pattern Source](#)

[Report Procedure](#)

[Write Procfile](#)

Select Iddq Patterns

Scope: Atpg mode

Prerequisites: You must have set the fault type to IDDQ by using the Set Fault Type -Iddq command. Also, you must use either the internal or external pattern source; you cannot use the random pattern source. You can set the pattern source to internal or external by using the Set Pattern Source command with either the Internal or External argument.

Selects the patterns that most effectively detect IDDQ faults.

Usage

```
SElect Iddq Patterns [-TEst_coverage [integer]] [-Max_measures number]  
[-Threshold number] [-Percentage number] [-Window number]  
[-Eliminate | -Noeliminate] [-EXhaustive | -Incremental]
```

Description

The Select Iddq Patterns command selects the patterns (cycles) that most effectively detect IDDQ faults, given an external or internal pattern set. If you set the pattern source to external, the tool places the patterns in the internal pattern set and performs its modifications and selections on that internal set. The tool uses the following three-step selection process to select the most effective patterns:

1. The tool fault simulates the patterns in the current pattern source, without dropping faults, and stores the fault simulation results for all patterns, ignoring any possibly-detected faults.

You can modify the results of this process by preceding the Select Iddq Patterns command with the Set Iddq Strobe and Set Iddq Checks commands. If the pattern set already contains patterns with IDDQ measurements and those are the only patterns of interest, use the Set Iddq Strobe -Label command to simulate only those patterns that contain an IDDQ measure statement. This option is the default upon tool invocation. If you want to simulate all the patterns in the set, with the assumption that there is an IDDQ measurement at the end of each test cycle, use the Set Iddq Strobe command with the -All option.

Use the Set Iddq Checks command to restrict IDDQ measurements to those that satisfy the restrictions you specify.

2. The tool identifies all the IDDQ measurements that fall within the boundaries that you specify by performing these steps:
 - a. The command identifies the IDDQ measurement that detects the most faults from the simulation results. The command selects an IDDQ measurement if it passes two tests: 1) it must detect the minimum number of faults that you specify with the optional -Threshold switch, and 2) the total number of selected IDDQ measurements cannot exceed the number that you specify with the -Max_measures switch.

- b. The tool then removes from the active fault list the faults that the fault simulation detected for that measurement and places them in the detected-by-simulation fault class.
- c. The tool displays the normal fault simulation message for that measurement.
- d. The tool repeats the selection process until it either reaches the maximum number of allowed IDDQ measurements, or the remaining measurements fail to detect the minimum number of faults.

For FlexTest, the selection process will stop if the test_coverage reaches the specified number.

During the fault simulation process, the tool does not give credit for any possibly-detected faults.

3. The tool performs a final fault grade for the internal patterns, giving detection credit for only those patterns that contain the IDDQ measure statement. The tool uses this simulation to calculate the final test coverage and also to give credit for the possibly-detected faults.

After the tool finishes this IDDQ pattern selection process, you can save the selected patterns to an external file with the Save Patterns command.

FlexTest may run out of memory during Step 1 of the selection process, if you are working with a large design with the default -Exhaustive switch active. This is because the -Exhaustive switch causes FlexTest to simulate all the IDDQ measurements, without fault dropping, before beginning the selection process. To do this, FlexTest creates a table to keep track of which faults it detects in each measurement. In some cases, the table size can be too large for the amount of available memory.

To circumvent the memory problem, use the -Window switch in combination with the -Exhaustive switch to define how many measurements FlexTest is to allow in the table. If you allow fewer IDDQ measurements in the table than the total number of IDDQ measurements in the simulation, FlexTest makes multiple passes until it simulates all measurements.

When you specify the -Window switch, FlexTest still keeps track of the simulation results and enters them into the table. However, when the table is full, FlexTest pauses the simulation and begins the selection process outlined in Step 2a. When the selection process is complete on that window's worth of measurements, FlexTest keeps only the qualified IDDQ measurements in the table. FlexTest then begins simulating the next set of measures, using the remaining space in the detection table. It repeats this simulation, followed by the selection process, until it simulates the entire pattern set. For this method to work, the window size must be at least twice the -Max_measures number (unless the window size is large enough to hold all of the IDDQ measurements.)

The -Test_coverage switch lets you stop the selection process, if the selected IDDQ measurements can reach the specified test coverage.

Arguments

- **-Max_measures *number***
An optional switch and integer pair that specifies the maximum number of patterns (cycles) with an IDDQ measure statement that the tool allows in the final set. Once the command identifies the maximum number of IDDQ measurements, the command terminates. The default is all patterns with an IDDQ measure statement.
- **-Threshold *number***
An optional switch and integer pair that specifies the minimum number of IDDQ faults an IDDQ measurement must detect to pass the selection process. The default is 1.
- **-TEst_coverage [*number*]**
An optional switch and number pair which sets the IDDQ selection process to stop when the accumulated test coverage reaches number percent. *number* can be an integer from 0 to 100. The default is 100%.
- **-Percentage *number***
An optional switch and integer pair that specifies the minimum percentage of remaining undetected IDDQ faults an IDDQ measurement must detect to pass the selection process. The default is 0.
- **-Window *number***
An optional switch and integer pair that specifies the size of the data detection table by setting the table's maximum number of allowed IDDQ measurements. Use this switch in conjunction with the -Exhaustive switch if you encounter a memory size problem. The default is 1000.
- **-Eliminate**
An optional switch that removes only the cycles that follow the last cycle having an IDDQ measure statement. This is the default.
- **-Noeliminate**
An optional switch that retains all patterns in the pattern set. You can look for the IDDQ measure statement to identify the patterns that the tool selected to perform an IDDQ measurement.
- **-EXhaustive**
An optional switch that specifies for FlexTest to first simulate all test cycles for all faults, and then perform the selection process to pick the ones that detect the largest number of IDDQ faults. This is the default.
- **-Incremental**
An optional switch that specifies for FlexTest to simulate test cycles one at a time, checking each time that the IDDQ measurement satisfies the -Threshold and -Percentage requirements. If FlexTest qualifies the maximum number of cycles containing an IDDQ

measure statement, it stops the simulation process at that point without simulating the remaining cycles.

Examples

The following example fault simulates an external IDDQ pattern file and selects the ten patterns with IDDQ measure statements that most effectively detect IDDQ faults:

```
set system mode atpg
set pattern source external pat_file
set fault type iddq
add faults -all
set iddq strobe -label
select iddq patterns -max_measures 10 -noeliminate
```

Related Topics

[Set Fault Type](#)

[Set Iddq Strobe](#)

[Set Iddq Checks](#)

[Set Pattern Source](#)

Set Abort Limit

Scope: All modes

Specifies the abort limit for the test pattern generator.

Usage

SET ABort Limit [-Backtrack *integer*] [-Cycle *integer*] [-Time *integer*]

Description

The Set Abort Limit command performs slightly differently depending on which tool you are using. Use this command when there are some remaining undetected faults and the test coverage is still too low. By increasing the abort limit, you can allow the tool to detect those remaining undetected faults, and thereby raise the coverage. The following paragraphs describe how the command operates for each tool.

The Set Abort Limit command specifies three ways for the test pattern generator to abort a target fault. One way is to set the maximum number of conflicts that the test pattern generator allows before aborting a target fault. The second way is to set the maximum number of test cycles that the generator allows before aborting a target fault. The third way is to set the maximum CPU time (in seconds) that the test pattern generator can run before aborting a target fault.

If any of these limits are too low, the test pattern generator may abort too many faults and fault coverage could be too low. However, if the limits are too high, it may take too much time to finish the test generation of a circuit. The invocation defaults are 30 conflicts, 300 test cycles, and 300 seconds. If you enter the command without any options, then the test pattern generator uses the default -Backtrack value.

Arguments

- -Backtrack *integer*

An optional switch and positive, nonzero integer pair that specifies the number of conflicts that the test pattern generator allows before aborting the target fault. If you enter the command without specifying any option, the command uses the current -Backtrack value. The invocation default is 30 conflicts.

- -Cycle *integer*

An optional switch and greater-than-0 integer pair that specifies the number of test cycles that the test pattern generator allows before aborting the target fault. The invocation default is 300 test cycles.

- -Time *integer*

An optional switch and greater-than-0 integer pair that specifies the number of CPU seconds that the test pattern generator can run before aborting the target fault. The invocation default is 300 seconds.

Examples

The following FlexTest example performs an ATPG run, then continues the run with a higher abort limit for the maximum number of allowed conflicts:

```
set system mode atpg
add faults -all
run
set abort limit -backtrack 100
run
```

Related Topics

[Report Aborted Faults](#)

[Report Faults](#)

Set Atpg Limits

Scope: All modes

Specifies the ATPG process limits at which the tool terminates the ATPG process.

Usage

```
SET ATPg Limits [-CPu_seconds {OFF | integer}] [-Test_coverage {OFF | real}]  
[-CYcle_count {OFF | integer}]
```

Description

The Set Atpg Limits command determines the limitations under which the ATPG process operates. Upon invocation of the tool, all the command option limitations are off. If you set any of the limitations, and during an ATPG run the tool reaches one of those limits, the tool terminates the ATPG process. You can use any combination of the three arguments.

You can check the current settings of the Set Atpg Limits command by using the [Report Environment](#) command.

Arguments

- -Cpu_seconds OFF | *integer*

An optional switch and argument pair that specifies the maximum number of CPU seconds that any future ATPG process can consume before the tool terminates the process. The argument choices are as follows:

OFF — A literal specifying that there is no limit to the amount of CPU time the ATPG process consumes during an ATPG process. This is the invocation default.

integer — A positive integer that specifies the maximum number of CPU seconds that the tool can consume during an ATPG process. When the tool reaches the maximum, it terminates the ATPG process.

- -Test_coverage OFF | *real*

An optional switch and argument pair that specifies the maximum percentage of test coverage that any future ATPG process need reach before the tool terminates the process. The argument choices are as follows:

OFF — A literal specifying the 100 percent test coverage limit during an ATPG process. The tool terminates the ATPG process when either 100 percent coverage is attained or when the ATPG process has completed. This is the invocation default.

real — A positive real number that specifies the maximum percentage of test coverage that the tool should achieve during an ATPG process. When the tool reaches the maximum, it terminates the ATPG process.

- **-Cycle_count** Off | *integer*

An optional switch and argument pair that specifies the maximum number of cycles that any future ATPG process can use before FlexTest terminates the process. The argument choices are as follows:

Off — A literal specifying that there is no limit to the number of test cycles the ATPG process uses during an ATPG process. This is the invocation default.

integer — A positive integer that is greater than, or equal to, the current number of internal patterns and that specifies the maximum number of test cycles that FlexTest can use during an ATPG process. FlexTest counts the test cycles in both the scan operations as well as in the fault simulation to determine the number of test cycles it uses. When FlexTest reaches the maximum, it terminates the ATPG process.

Examples

The following FlexTest example sets two of the three limits on the ATPG process and then displays the relevant setting data using the Report Environment command:

```
set atpg limits -cpu_sec 500 -test_coverage 99.5 -cycle_count 50000  
report environment
```

```
...  
atpg limits =      500.0 sec 95.50% coverage 50000 cycles  
...
```

If the ATPG process reaches either of these two limits, the process terminates. Notice that the information from the Report Environment command only shows the settings that are different than the invocation defaults of Off.

Related Topics

[Report Environment](#)

[Write Environment](#)

Set Atpg Window

Scope: Atpg mode

Lets you specify the size of the FlexTest simulation window.

Usage

SET ATpg Window *integer*

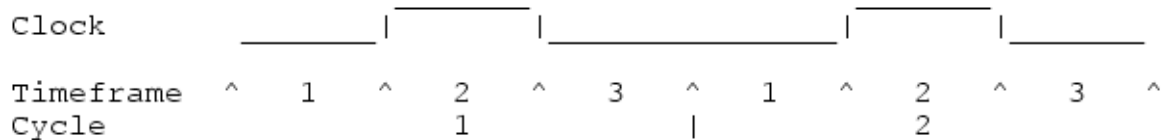
Description

The Set Test Cycle command defines the number of timeframes per test cycle. With the Set Atpg Window command, you can specify the number of cycles the deterministic ATPG will use to generate the test patterns for the fault under consideration. That is, for the given objective to be satisfied, ATPG will try to find a solution in this window by assuming full controllability at the present state lines at the left most time frame of the window. If there are any state requirements at the present state line of the left most time frame, the tool stores the information in the current window. Then, the tool resets the current window and uses it to find a pattern to justify the state required by the previous window.

Since no knowledge of the invalid states can be obtained in advance, increasing the size of the ATPG window may reduce the chances of generating invalid states between each cycle.

For example, [Figure 2-2](#) illustrates how an ATPG window can be set.

Figure 2-2. Set Atpg Window Example



If you set the ATPG window size to 1 (which is the default value) and the test cycle to include 3 timeframes, the first cycle in the above diagram will be considered one window. To find a pattern to detect a fault, the tool may assign state AB=11 at the present state lines of the leftmost time frame. This means the tool must have this specific value on AB at the beginning of a test cycle.

To justify this requirement, the tool may need an extra window if it cannot directly control the state of AB from primary inputs or scan cells. If state AB=11 is identified as an invalid state, meaning it cannot be satisfied, in the current window, the tool needs to backtrack to the previous window.

If you set the ATPG window size to 2 with the same test cycle, then for each fault the tool tries to detect, it now has two cycles available. This helps to avoid generating invalid state AB between timeframes 1 and 2.

In general, using a larger ATPG window may help increase the resolution of the ATPG process, but it requires more memory. On the other hand, a large ATPG window does not always produce better performance than a small ATPG window. For example, if a fault can be detected in a window with 3 timeframes, and you set the ATPG window to include 9 timeframes, the tool devotes additional resources maintaining the large ATPG window.

Note

A design with high sequential depth should use a large ATPG window to increase the ATPG resolution.

Arguments

- *integer*

A required integer value that specifies the number of basic windows used by FlexTest test generation. The number of cycles contained in a basic window is determined by the pin constraints. The default window size includes one basic window.

Examples

Example 1

The following example sets the ATPG window to 4.

```
set system mode atpg
add faults -all
set atpg window 4
run
```

Example 2

The following example causes FlexTest to use 9 timeframes in a window.

```
set test cycle 3
set system mode atpg
set atpg window 9
```

Related Topics

[Add Faults](#)

[Set Test Cycle](#)

[Set System Mode](#)

[Run](#)

Set Bus Handling

Scope: Atpg, Good, and Fault Modes

Specifies the bus contention results that you desire for the identified buses.

Usage

SET BUs Handling {**Pass** | **Fail** | **Abort**} {-**All** | -**Pass** | -**Fail** | -**Bidi** | -**ABort**} [-Instance *instance_name...*] | *bus_gate_id#...* | *net_pathname...*}

Description

The Set Bus Handling command preassigns the contention check handling result that you desire during simulation for the buses that you specify. Upon invocation, the tool automatically calculates the bus contention handling as documented under the [Set Contention Check](#) command description. The tool rejects (from the internal test pattern set) ATPG-generated patterns that can cause bus contention. The Set Bus Handling command lets you override the automatic contention calculations, thereby changing whether or not the tool performs a simulator-based check using such a pattern.

The tool resets the bus contention handling back to the automatically calculated value whenever you make a change to the modeling that requires the tool to perform a complete re-analysis of the contention mutual exclusivity (such as changing the net resolution or the pin constraints).

Caution



Overriding the automatically calculated contention check handling results can cause trouble downstream if there is a problem with the design that required modification due to the bus contention.

Arguments

- **Pass**

A literal that specifies for the tool to not perform bus contention evaluations on the buses that you identify and to treat them as if they had passed. This allows the tool to retain patterns that it would otherwise reject due to a contention check failure.

- **Fail**

A literal that specifies for the tool to treat the buses that you identify as if they had failed the bus contention evaluations. This causes the tool to reject patterns that it would otherwise retain due to passing the contention check.

- **Abort**

A literal that specifies the bus aborted the bus contention evaluations before determining whether the bus passed or failed. This can be used with the Analyze Bus -Drc_check command to verify ATPG constraints which you have added to correct bus failures.

- **-All**
A switch that specifies for the tool to change all buses to a specified state reset.
- **-Pass**
A switch that specifies for the tool to change to specified state for the busses that have previously passed the E10 DRC rule.
- **-Fail**
A switch that specifies for the tool to change to specified state for the busses that have previously failed the E10 DRC rule.
- **-Bidi**
A switch that specifies for the tool to change to specified state for bidi busses identified by the E10 DRC rule.
- **-ABort**
A switch that specifies for the tool to change to specified state for the busses that have previously aborted the E10 DRC rule.
- **-Instance *instance_name***
An optional switch and repeatable string that specifies the instances where the included busses are to be changed to the specified state.
- ***bus_gate_id#***
A repeatable integer that specifies the gate identification numbers of the buses whose contention handling you want to override. If a bus is cascading, you must specify the dominant bus.
- ***net_pathname***
A repeatable string that specifies a net name of the buses to be changed to a specified state.

Examples

The following example turns the bus contention checking off, allowing the bus to pass the evaluations. However, this action can cause trouble in the future if there is a problem with the design that required modification due to the bus contention.

report bus data 321

```
/FA1/ha1/XOR1/OUT/ (321) handling=fail type=strong #Drivers=4  
Learn Data:poss_X=yes, poss_Z=yes, poss_mult_drivers_on=yes  
BUS Drivers: 156(SW) 252(SW) 307(SW) 308(SW)
```

set bus handling pass 321 report bus data 321

```
/FA1/Ha1/Xor1/OUT/ (321) handling=pass type=strong #Drivers=4  
Learn Data:poss_X=yes, poss_Z=yes, poss_mult_drivers_on=yes  
BUS Drivers: 156(SW) 252SW) 307(SW) 308(SW)
```

Related Topics

[Report Bus Data](#)

[Set Contention Check](#)

Set Capture Clock

Scope: All modes

Specifies the capture clock for random patterns or, optionally, for all ATPG patterns.

Usage

SET Capture Clock [*primary_input_pin*] [-Atpg] | ANY

Description

The Set Capture Clock command performs slightly differently depending on which tool you are using. In any case, you can use the Report Environment command to list the capture clock and the Report Clocks command to identify the current list of clocks.

The Set Capture Clock command allows the design rules checker to support the internal scan circuitry within certain boundary scan designs.

For certain boundary scan designs to support their internal scan designs, they only allow one cycle for each capture. For that one capture cycle, FlexTest must use the boundary scan clock and all other clocks must be at their off states.

In FlexTest, the capture limit must be set at one, the capture clock must have an R0 or R1 pin constraint, and all other clocks must have a C0, C1, CR0, or CR1 pin constraint. The period of all pin constraints must be 1.

If you define a capture clock to pass the design rules checker, FlexTest will not place the chain test (which does not use any capture clock) in any test pattern set. If you request to save both the cycle and chain test by using the Save Patterns command, FlexTest will save only the cycle test, and displays a message indicating that FlexTest cannot save the chain test. If you specify to save only the chain test, FlexTest generates an error message.

If you want a chain test, when it is necessary for FlexTest to force a capture clock for a successful scan test, you can remove the capture clock from the load_unload procedure in the test procedure file. You must remove the forced capture clock and then, after the design successfully passes the rules checking, you can store the chain test in a separate file.

Arguments

- *primary_input_pin*
An optional string that specifies the name of the primary input pin you want to assign as the capture clock.
- -Atpg
An optional switch that directs the tool to use the specified capture clock as the only capture clock for all patterns created during the ATPG process, not just random patterns. If specified in Setup mode, this switch turns on some additional design rule checks (DRCs).

- ANY

An optional string that specifies any clock can be a capture clock. This is the default if *primary_input_pin* is not specified.

Examples

The following example specifies a capture clock:

```
add clocks 1 clock1  
set capture clock clock1  
set system mode fault
```

Related Topics

[Add Clocks](#)

[Report Environment](#)

[Delete Clocks](#)

[Set Pattern Source](#)

[Report Clocks](#)

Set Capture Limit

Scope: All modes

Specifies the number of test cycles between two consecutive scan operations.

Usage

SET Capture Limit **OFF** | {*test_cycle_limit* [-Maximum | -Exact]}

Description

The Set Capture Limit command lets you limit the number of test cycles that FlexTest captures between two consecutive scan operations for internal ATPG patterns; the command does not affect external patterns. You may need to use this command with hardware testers that limit the number of test cycles between two consecutive scan operations. FlexTest classifies any undetected faults due to the capture limit as atpg_untestable (AU) faults.

You can use the Report Environment command to display the current capture limit.

Arguments

- **OFF**
A literal specifying that there is no limit to the number of test cycles that FlexTest allows between two consecutive scan operations. This is the default behavior of FlexTest upon invocation.
- *test_cycle_limit* -Maximum | -Exact
A positive integer and optional switch pair that specifies either the maximum or the exact number of test cycles that FlexTest allows between two consecutive scan operations. The switch choices are as follows:
 - Maximum — An optional switch that specifies for FlexTest to interpret the *test_cycle_limit* argument value as the maximum number of capture test cycles. FlexTest will not allow any more than the specified number of capture test cycles between two consecutive scan operations. This is the *test_cycle_limit* argument default.
 - Exact — An optional switch that specifies for FlexTest to interpret the *test_cycle_limit* argument value as the exact number of capture test cycles. FlexTest must always use the specified number of capture test cycles between two consecutive scan operations.

Examples

The following example specifies the maximum number of capture test cycles that FlexTest can allow between two consecutive scan operations:

```
set capture limit 3 -maximum
```

Related Topics

[Report Environment](#)

[Write Environment](#)

Set Checkpoint

Scope: All modes

Prerequisites: You must use the Setup Checkpoint command prior to this command.

Specifies whether the tool uses the checkpoint functionality.

Usage

SET CHECKpoint **OFF** | **ON**

Description

The Set Checkpoint command determines whether the tool uses the checkpoint functionality that you specified with the Setup Checkpoint command. At tool invocation, the checkpoint functionality is off, which means that during ATPG, the tool does not save patterns into the file that you specified with the Setup Checkpoint command. The tool only retains the internal patterns at the end of the ATPG run.

If you set the checkpoint functionality on, then during ATPG, the tool saves the patterns into the checkpoint file at the end of each time period specified by the Setup Checkpoint command.

For additional information on checkpointing, refer to “[Checkpointing Setup](#)” the Scan and ATPG User’s Manual.

Arguments

- **OFF**
A literal that specifies for the tool not to use the checkpoint functionality during test pattern generation. Patterns are not written to any file. This is the invocation default.
- **ON**
A literal that specifies for the tool to use the checkpoint functionality. The tool writes test patterns that it generates to the file that you specified with the Setup Checkpoint command.

Examples

The following example turns on the checkpoint functionality after setting up the checkpoint file and time period:

```
set system mode atpg
setup checkpoint my_checkpoint_file 5 -sequence
set checkpoint on
```

Related Topics

[Setup Checkpoint](#)

Set Clock Restriction

Scope: All modes

Specifies whether ATPG can create patterns with more than one active capture clock.

Usage

SET CLock Restriction **ON** | **OFF**

Description

The Set Clock Restriction command changes the default behavior of ATPG regarding the creation of test patterns that have more than one active clock line. The invocation default behavior is different depending on which tool you are using.

Note



Free-running clocks are excluded from clock restriction checking when ON is selected. For example, if ON is selected and two free-running clocks are defined, then only one clock is allowed in addition to the free-running clocks.

Arguments

- **ON**

A literal that specifies for the ATPG to create only patterns with, at most, a single active clock. Prevent race conditions, due to multiple active clocks, by specifying this behavior. This is the default behavior upon invocation of FlexTest.

- **OFF**

A literal that specifies for the ATPG process to create patterns with as many clocks on as it requires to detect faults. Using this option may cause race conditions due to multiple active clocks. Prevent these race conditions by specifying the On argument.

Set Contention Check

Scope: All modes

Specifies whether or not contention checking is on and the conditions under which checks are performed. Contention checking is set to On when the tool is invoked.

Usage

SET COntention CHeck **OFF** | { **ON** [-Warning | -Error] [-Soft | -Hard] [-Bus | -Port | -All] [-ATpg] [-Start *frame#*] }

Description

When a bused output of a tri-state driver (or switch) that is driving an X caused by an X on its enable is encountered, the contention is not reported if the data input to the driver is at the same level as other drivers on the bus. Also, the simulated value of the bus gate is resolved to a binary value if it can do so without tracing through additional bus gates.

Arguments

- **OFF**
A literal that specifies to not perform contention checking during simulation.
- **ON**
A literal that performs contention checking during simulation. FlexTest performs contention checking for every timeframe so captured data effects are propagated. This is the default.
- **-Warning**
An optional switch that displays a warning message, but continues simulation if bus contention occurs during simulation. This is the default.
- **-Error**
An optional switch that displays an error message and stops the simulation if bus contention occurs.

You can debug contention errors by using the -Error switch to stop simulation at the point of the first contention error.

Using this option, you can then view the simulated values of all gates in the first bus contention pattern by using the Report Gates command.
- **-Soft**
A switch that checks only soft bus contention.

Soft bus contention is divided into two categories, driver restriction turned on and driver restriction turned off.

When driver restriction is turned on, the tool reports soft contention when it detects no more than one bus gate input at a binary value and other bus gate inputs at unknown values.

When driver restriction is turned off, the tool reports soft contention when it detects no bus gate inputs at opposite binary values and other bus gates at unknown values.

- **-Hard**

A switch that checks only hard bus contention.

Hard bus contention is divided into two categories, driver restriction turned on and driver restriction turned off.

When driver restriction is turned on, the tool reports hard contention when it detects any two bus gate inputs at binary values.

When driver restriction is turned off, the tool reports hard contention when it detects any two bus gate inputs at opposite binary values.

- **-Bus**

An optional switch that performs contention checking of tri-state driver buses. This is the default.

Tri-state logic allows several bus drivers to time-share a bus. However, if the circuit enables two bus drivers of opposite logic to drive the bus, physical damage can occur. This switch allows the tool to identify these conditions and notify you of their existence.

The tool identifies buses which have circuitry that prevents bus contention and does not check for bus contention problems. This eliminates false bus contention reporting when multiple inputs to a bus are at X. Bus contention that occurs on weak buses does not result in an E4 rules checking violation or pattern rejection during simulation. The tool continues to simulate them as an X state.

- **-Port**

An optional switch that performs contention checking for multiple-port flip-flops and latches. The tool identifies and rejects patterns during which any multiple-port latch or flip-flop has more than one clock, set, or reset input active (or at X).

- **-All**

An optional switch that performs contention checking for both tri-state driver buses and multiple-port flip-flops and latches.

- **-ATpg**

An optional switch that forces all buses to a non-contention state, which ensures that test patterns are not created causing bus contention.

- **-Start *frame#***

An optional switch and integer that specifies the number of timeframes after initialization, or after each scan loading, when ATPG begins the contention check. The default is timeframe 0.

Due to sequential initialization, the initial states on a bus may be unknown and possible contention may be unavoidable. Thus, this switch lets you begin the contention checking after design initialization.

Examples

The following example performs contention checking on multiple-port flip-flops and latches, stops the simulation if any bus contention occurs, and displays an error message. The message indicates the number of patterns rejected and the bus gate on which the bus contention occurred:

```
set contention check on -port -error
set system mode atpg
add faults -all
run
```

Related Topics

[Add Contention Free_bus](#)

[Set Contention_bus Reporting](#)

[Report Gates](#)

[Set Gate Report](#)

[Set Bus Handling](#)

Set Contention_bus Reporting

Scope: All modes

Specifies the conditions of contention bus reporting.

Usage

SET COntention_bus Reporting **OFF** | {ON [-All] [-Soft | -Hard] [-Verbose] [-ATpg]}

Description

The Set Contention_bus Reporting command lets you specify whether all bus contentions are reported in ATPG mode, reports either soft or hard contentions, and displays the input values of the bus gates that have bus contention.

Arguments

- **OFF**
A literal that specifies for the tool to report the last detected bus contention. This is the default behavior upon invocation of the tool.
- **ON**
A literal that specifies for the tool to report all bus contentions immediately.
- **-All**
A switch that specifies for the tool to report all bus contentions associated with a pattern in ATPG mode. It can be used in conjunction with the -Soft or -Hard switch to report all soft or hard contentions.
- **-Soft**
A switch that specifies for the tool to report only one soft bus contention for each pattern used. However, if the -Soft switch is used in conjunction with the -All switch, then all soft bus contentions are reported.
- **-Hard**
A switch that specifies for the tool to report only one hard bus contention for each pattern used. However, if the -Hard switch is used in conjunction with the -All switch, then all hard bus contentions are reported.
- **-Verbose**
A switch that specifies for the tool to display the input values of the gate in contention.
- **-ATpg**
A switch that specifies for the tool to report all bus contention sites for every pattern test generator generated.

Examples

The following example performs contention bus reporting on all soft bus contentions and displays the input values of the gate in contention.

```
set contention_bus reporting on -ALI -Soft -verbose  
set system mode atpg  
add faults -all  
run
```

Related Topics

[Add Contention Free_bus](#)

[Set Contention Check](#)

[Delete Contention Free_bus](#)

[Set Gate Report](#)

[Report Contention Free_bus](#)

[Report Gates](#)

[Set Bus Handling](#)

Set Display

Scope: All modes

Sets the DISPLAY environment variable from the tool's command line.

Usage

SET DISPlay *display_name*

Description

The Set Display command sets the DISPLAY environment variable to *display_name* without exiting the currently running application. If you invoke the tool in command line mode (the default), then the DISPLAY variable is not required in order to use most commands successfully. If the variable is not set, however, and you issue a command that requires a GUI, the tool will not be able to show the applicable window. This is because the display connection provided by the variable is absent. Set Display enables you to set the variable, if needed for the display, without the inconvenience of exiting to the shell in order to set it.

Note



This command affects the DISPLAY setting within the currently running application only. When you exit the tool, the setting in the invocation shell will be what it was when you invoked the tool.

Arguments

- *display_name*

A required string that specifies a valid display setting for the machine on which the tool is running.

Examples

The following example sets the DISPLAY variable. The example also uses the System command to pass a UNIX “echo \$DISPLAY” command to the shell in order to check the variable's setting.

```
system echo $DISPLAY
```

```
set display my_workstation:0.0  
system echo $DISPLAY
```

```
my_workstation:0.0
```

Related Topics

[System](#)

Set Dofile Abort

Scope: All modes

Specifies whether the tool aborts or continues dofile execution if it detects an error condition.

Usage

SET DOfile Abort **ON** | **OFF** | **Exit**

Description

By default, if an error occurs during the execution of a dofile, processing stops, and the line number causing the error in the dofile is reported. The Set Dofile Abort command lets you to turn this functionality off so that the tool continues to process all commands in the dofile.

Arguments

- **ON**
A literal that halts the execution of a dofile upon the detection of an error. This is the default upon invocation of the tool.
- **OFF**
A literal that forces dofile processing to complete all commands in a dofile regardless of error detection.
- **Exit**
A literal that directs the tool to exit the session if it detects an error while executing a dofile. Whereas “on” leaves the tool running, “exit” ends the tool session and returns you to the invocation shell prompt. Use a “set dofile abort exit” as the first command in a dofile to prevent a batch job from hanging at the command prompt if an error occurs in the dofile.

Examples

The following example sets the Set Dofile Abort command off to ensure that all commands in test1.dofile are executed.

```
set system mode atpg
set dofile abort off
dofile test1.dofile
```

Related Topics

[Dofile](#)

Set Drc Handling

Scope: All modes

Specifies how the tool handles design rule violations.

Usage

```
SET Drc Handling {drc_id [Error | Warning | NOTe | Ignore]
  [NOVerbose | Verbose]
  [NOAtpg_analysis | Atpg_analysis]
  [ATPGC]
  [-CONSERvative {ON | OFF}]
  [-Mode A clk_name]
  [-Interval number]
  [-Mode {Sequential | Combinational}]
  [-SKip_procedure {OFF | TEst_setup}] |
  -INItialization_cycle {ON | OFF} |
  -SCan_chain_tracing {Aggressive | Conservative}
```

Description

The Set Drc Handling command specifies the handling of the messages for the scan cell RAM rules checking, Clock rules checking, Data rules checking, Extra rules checking, Trace rules checking, and certain Timing rules checking. You can specify that the violation messages for these checks be either error, warning, note, or ignore. If you do not specify error, warning, note, or ignore, then the tool uses either the handling from the last Set Drc Handling command or, if you did not change the handling, the Design Rules Checker's invocation default.

Note




The Set DRC Handling command does not support any (F) rules. Use the [Set Flatten Handling](#) command to specify how design rule violations are handled for (F) rules.

Each rules violation has an associated occurrence message and summary message. The tool only displays the occurrence message for either error conditions or if you specify the Verbose option for that rule. The tool also displays the rule identification number in all rules violation messages.

The Atpg_analysis option provides test generation analysis when performing rules checking for some clock (C) rules, for some data (D) rules, and for some extra (E) rules. For example, if Atpg_analysis is enabled for clock rule C1, and the tool simulates a clock input as X, the rule violation occurs when it is possible for the test generator to create a test pattern while that clock input is on, all defined clocks are off and all constrained pins are at their constrained state.

Note

 When Atpg_analysis is enabled, the tool requires some additional CPU time and memory to perform the test generation analysis. (The Atpg_analysis option is enabled by default for rules C1, E4, E10, E11 and E13; you can disable it for these rules by specifying the Noatpg_analysis option.)

Arguments

- *drc_id*

A required non-repeatable literal that specifies the identification of the exact design rule violation whose message handling you want to change.

The design rule violations and their identification literals are divided into the following seven groups: A (RAM), C (clock), D (data), E (extra), T (trace), and W (timing).

- Error

An optional literal that specifies for the tool to both display the error occurrence message and immediately terminate the rules checking.

- Warning

An optional literal that specifies for the tool to display the warning summary message indicating the number of violations for that rule. If you also specify the Verbose option, the tool displays the occurrence message for each occurrence of the rules violation.

- NOTe

An optional literal that specifies for the tool to display the summary message indicating the number of violations for that rule. If you also specify the Verbose option, the tool also displays the occurrence message for each occurrence of the rules violation

- Ignore

An optional literal that specifies for the tool not to display any message for the rule's violations. The tool must still check some rules and they must pass to allow later performance of certain functions.

- NOVerbose

An optional literal that, for a DRC whose handling is set to Warning or Note, specifies to display a summary message indicating the total number of occurrences of the rule violation. This is the default.

If DRC handling is set to Error or Ignore, the tool will behave as follows regardless of whether you specify the Noverbose (or Verbose) option:

- Error—The tool will stop at each error and display detailed instance-specific violation information to help you debug it.
- Ignore—The tool will not perform the check and therefore, the consequences of the check on ATPG, whether good or bad, will not occur. Also, the tool will not display or store instance-specific violation information.

- **Verbose**

An optional literal that, for a DRC whose handling is set to Warning or Note, specifies to display the occurrence message for each occurrence of the rule violation. All instance-specific violations (maybe hundreds or thousands) will be listed, one message per instance violating the rule. To obtain a summary message instead, use the Noverbose option.

If the DRC's handling is set to Error or Ignore, the tool will behave as follows regardless of whether you specify the Verbose (or Noverbose) option:

- **Error**—The tool will stop at each error and display detailed instance-specific violation information to help you debug it.
- **Ignore**—The tool will not perform the check and therefore, the consequences of the check on ATPG, whether good or bad, will not occur. Also, the tool will not display or store instance-specific violation information.

- **NOatpg_analysis**

An optional literal that specifies for the tool not to use test generation analysis when performing rules checking. This is the default except for rules C1, E4, E10, E11 and E13.

- **Atpg_analysis**

An optional literal that specifies for the tool to use test generation analysis when performing rules checking for clock rules (such as C1, C3, C4, C5 and C6), some D rules (such as D9) and some E rules (such as E4, E5, E8, E10, E11 and E12). This is the default for rules C1, E4, E10, E11 and E13.

For clock rules C3 and C4, the Atpg_analysis option generates a check of the clocks of the source and sink to see if they are gated off. To see if a path exists from the Q output of the source to the sink, use the Set Sensitization Checking command with checking turned on. It is recommended that you use the Atpg_analysis option with the Set Sensitization Check On analysis to remove the maximum number of false C3 and/or C4 violations.

- **-Mode A *clk_name***

A switch, a literal (A), and a string triplet that specifies the name of a clock on which you want the tool to perform further analysis to screen out false C3 and C4 clock rules violations. The tool performs this extra analysis on one clock only.

- **-Interval *number***

An optional switch and integer pair that you can only use with C3 and C4 clock violations to specify how often you want the tool to display a message during the ATPG analysis of those violations. The *number* argument indicates multiples of violation occurrences that cause the tool to display a message. The default is 0.

The message includes the number of sequential elements that the tool checked, the number of sequential elements remaining to check, the current number of ATPG passes during the C3 or C4 clock rules checking, and the current CPU time used by the tool for clock rules checking.

The value of the **number** parameter must be either zero or a positive integer. You can only specify one **number** value that the tool uses for both the C3 and C4 violations. If you issue multiple Set Drc Handling commands (one for C3 and one for C4) that specify different values for the **number** argument, the tool uses the last interval value you specified.

- ATPGC

An optional literal that specifies for the design rules checker to use all the current ATPG constraints when analyzing E10 rule violations. You can also use the [Add Atpg Constraints -Static](#) command to do the same thing.

- -CONSERvative {ON | Off}

An optional switch and literal for C6 DRC analysis. The default is OFF.

The default C6 checking may miss some C6 violations when [Set Clock Restriction](#) OFF is used. A more conservative analysis can be enabled by setting the -CONSERvative to ON and will identify those C6 violations. However, it is generally recommended not to turn off clock restriction such that the -CONSERvative switch does not usually need to be used.

- -Mode {Combinational | Sequential}

An optional switch and literal for the tool to use with the E10 rule. The Combinational option is the default. It performs bus contention, mutual-exclusivity checking and is limited by the combinational logic boundary.

The Sequential option considers the inputs to a single level of sequential cells behaving as “staging” latches in the enable lines of tri-state drivers. All of the latches found in a back trace must share the same clock. There must also be only a single clocked data port on each cell, and both set and reset inputs must be tied (not pin constrained) to the inactive state. This check ensures that there is no connectivity from the cells in the input cone of the sequential cells and enables of the tri-state devices except through the sequential cells.

- -SKip_procedure {Off | TEst_setup}

An optional switch and literal pair that affects the tool’s handling of the E4 rule. The default option, off, specifies for the tool to perform E4 rule checks for the test_setup procedure during DRC; the Test_setup option specifies to skip these checks.

It is very common during test_setup, when a design is being initialized, to have valid but momentary bus contention before the design reaches its initialized state. The E4 rule often detects these momentary occurrences of bus contention. As a result, the tool may issue many E4 violation messages for the test_setup procedure. If you do not wish to be reminded of this type of bus contention, use the Test_setup option with this switch to force the tool to skip the E4 rule checks for test_setup; this will eliminate the messages.

Note


This switch has no effect if E4 handling is set to Ignore.

- -INItialization_cycle {ON | Off}

A optional switch and literal pair that controls whether the tool forces all clocks to their off states and all constrained pins to their constrained values prior to simulation of the

test_setup procedure during DRC. By default, the tool performs these forces, so you do not have to explicitly specify the forces, which matches the default initialization cycle the tool includes in saved patterns. If you have occasion to save patterns without this initialization cycle ([Save Patterns](#) -Noinitialization), issue a Set Drc Handling -Initialization Off command and rerun DRC before saving patterns.

Tip

 : If DRC is run with initialization enabled, turning off initialization when saving patterns (Save Patterns -Noinitialization) may result in simulation mismatches during pattern verification in a timing based simulator. The converse (DRC run with initialization disabled and patterns saved with initialization enabled) is typically not a problem.

The argument choices are as follows:

ON — A literal that specifies to initialize the clocks and constrained pins to their off states and constrained values, respectively, prior to DRC simulation of the test_setup procedure. This is the invocation default.

Off — A literal that specifies not to initialize the clocks and constrained pins prior to DRC simulation of the test_setup procedure.

- **-Scan_chain_tracing{[Aggressive](#) | [Conservative](#)}**

An optional switch and literal pair that specify whether or not scan chain tracing passes with a T25 violation. The choices are as follows:

[Aggressive](#) — Scan chain tracing passes with a T25 violation. This is the default.

Conservative — Scan chain tracing does not pass with a T25 violation. In this case, scan chain tracing fails, design rule checking stops, and the tool issues a T3 violation.

Examples

The following example specifies rule checking E4 to be an error:

```
add scan groups group1 scanfile
add scan chains chain1 group1 indata2 outdata4
add clocks 1 clock1
add clocks 0 clock2
set drc handling e4 error
set system mode atpg
```

Related Topics

[Report Drc Rules](#)

[Set Sensitization Checking](#)

Set Driver Restriction

Scope: Setup mode

Prerequisites: You can only use this command before the tool begins generating test patterns.

Also, the Set Contention Checking command must be issued to turn contention checking on.

Specifies whether the tool allows multiple drivers on buses and multiple active ports on gates.

Usage

SET DRIver Restriction **Off** | **ON** | **Tg**

Description

The Set Driver Restriction command lets you specify for the tool to report a contention problem whenever there are multiple nets driving values onto a bus, or when multiple ports are active on an individual gate. The default upon tool invocation is to allow multiple driving nets on a bus or multiple active ports on a gate as long as the signals are driving the same value. If multiple signals are on and are not driving the same values, then the tool flags it as contention.

However some design processes only allow a single driver to be on at a time regardless of whether the signals are driving the same values. The Set Driver Restriction command lets you place this same restriction on the tool.

Arguments

- **Off**
A literal that specifies to allow multiple drivers to be on for a bus and multiple ports to be active for a gate as long as the driving signals are of the same value, and therefore there is no contention. This is the default behavior when you invoke the tool.
- **ON**
A literal that restricts buses to only have one driver on at a time and gates (dff and latches with multiple clocks) to only have one active port; the tool flags multiple active drivers or ports as contention problems.
- **Tg**
A literal that restricts FlexTest to generating only test patterns that do not allow multiple drivers. However, FlexTest does allow multiple drivers to be driving the same values during the fault simulation process. This option improves the test generation performance, but can cause FlexTest to incorrectly classify a detectable fault as an ATPG untestable fault.

Examples

The following example creates a strict contention checking environment. The first command specifies for the tool to check for contention on both multiple port gates and buses. If there is a contention problem where signals are driving different values, the tool reports an error and stops the simulation. The second command further restricts the contention checking environment by not allowing multiple drivers to even drive the same values.

set contention check on -error -all
set driver restriction on

Related Topics

[Set Contention Check](#)

Set Fails Report

Scope: All modes

Specifies whether the design rules checker displays clock rule failures.

Usage

SET Fails Report **Off** | **ON**

Description

The Set Fails Report command displays all clock rule failures of the design rules checker. The default mode upon invocation of the tool is Off.

Arguments

- **Off**
A literal that specifies for the design rules checker to not display clock failures. This is the default upon invocation of the tool.
- **ON**
A literal that specifies for the design rules checker to display clock failures.

Examples

The following example displays clock failures from the design rules checking process:

```
add scan groups group1 scanfile
add scan chains chain1 group1 indata2 outdata4
add clocks 1 clock1
add clocks 0 clock2
set fails report on
set system mode atpg
```

Related Topics

[Add Clocks](#)

[Report Clocks](#)

[Delete Clocks](#)

Set Fault Dropping

Scope: Fault mode

Specifies whether FlexTest drops DS faults in Fault mode.

Usage

SET Fault Dropping **ON** | {**OFF** [-Dictionary *filename*] [-Oscillation]
[-Hypertrophic] [-Replace]}

Description

The Set Fault Dropping command enables or disables the dropping of DS faults when FlexTest is in Fault mode. During the fault simulation process, the tool drops faults after they are detected by a sub-test sequence and ignores the dropped faults afterwards. This speeds up the fault simulation process.

To facilitate fault diagnosis, you can set the tool to carry out fault simulation without dropping faults. Use the Set Fault Dropping command with the Off switch to do this. The default behavior of the tool is to set fault dropping on.

When you set fault dropping off, you have the option of writing the detected faults to a specified file. This file consists of two parts:

- Head

This section contains information on each fault detected and applies a unique identification number (ident_num) to each fault, which is used by the fault dictionary section. When you issue the Run command while in Fault mode, this section displays an additional item, DS_NOW, to show the number of faults detected in the current cycle as fault dropping is turned off. The format of the Head is as follows:

```
HEAD =  
    DS_NOW = X;  
    <ident_num> <fault_type> <pin_pathname>;  
    ...  
END HEAD;
```

- Fault Dictionary

This section specifies at which cycle each fault was detected. The format of the Fault Dictionary is as follows:

```
DICTIONARY =  
  CYCLE = 1;  
  DS = X;  
  FAULTS = {<ident_num> <ident_num> ... };  
  CYCLE = 2;  
  DS = Y;  
  FAULTS = {<ident_num> <ident_num> ... };  
  ...  
END DICTIONARY;
```

A line in the file starting with “//” indicate comment lines. To end the dictionary file, you can either turn fault dropping back on or move out of Fault mode.

When fault dropping is turned off, you cannot add faults to or delete faults from the current fault list. You also cannot issue the Reset State command.

Note



The Set Fault Dropping command with the Off switch is valid only with stuck-at, toggle, and transition.

Caution



Setting fault dropping off will dramatically increase fault simulation time. Plus, an “Out of Memory” error may occur since the tool must keep each fault during the fault simulation. The tool will quit when an “Out of Memory” error occurs.

Arguments

- **ON**

A literal that specifies for the tool to enable fault dropping during the fault simulation. This is the default.

- **OFF**

A literal that specifies for the tool to disable fault dropping during the fault simulation.

- **-Dictionary *filename***

An optional switch and string pair that specifies for the tool to write the faults detected in each test cycle to a file with name *filename*. This switch is only effective when fault dropping is turned off.

- **-Oscillation**

An optional switch that specifies for the tool not to drop oscillation faults. By default, if a fault is marked as an oscillation fault and fault dropping is turned off, the tool will drop it from the fault list.

Note



Simulation of oscillation faults may consume a great deal of CPU time.

- **-Hypertrophic**

An optional switch that specifies for the tool not to drop hypertrophic faults. By default, if a fault is marked as a hypertrophic fault and fault dropping is turned off, the tool will drop it from the fault list.

Note



Simulation of hypertrophic faults may consume a great deal of CPU time.

- **-Replace**

An optional switch which specifies for the tool to replace the contents of the *filename* specified with the -Dictionary switch, if it already exists.

Examples

The following example sets fault dropping off and specifies for FlexTest to write all faults to the file *faultdictionary*.

```
set system mode fault
set fault dropping off -dictionary faultdictionary
```

Set Fault Mode

Scope: All modes

Specifies whether the fault mode is collapsed or uncollapsed.

Usage

SET FAULT Mode **Uncollapsed** | **Collapsed**

Description

The Set Fault Mode command specifies whether the tool uses collapsed or uncollapsed fault lists for fault counts, test coverages, and fault reports. The default fault mode upon invocation of the tool is Uncollapsed. When you display a report on uncollapsed faults, the tool lists the representative fault first followed by its equivalent faults.

Tessent Visualizer displays fault information based on this setting.

Arguments

- **Uncollapsed**
A literal specifying that the tool include equivalent faults in the fault lists. This is the default mode upon invocation of the tool.
- **Collapsed**
A literal specifying that the tool not include equivalent faults in the fault lists.

Examples

Example 1

The following example sets the fault mode to collapsed and then displays only the collapsed faults:

```
set system mode atpg
add faults -all
set fault mode collapsed
report faults -all
```

Example 2

The following shows an example when reporting uncollapsed tied faults as compared to reporting collapsed tied faults:

Uncollapsed:	Collapsed:
0 TI /I_140/I	0 TI /I_140/I
1 TI /II_140/O	1 TI /II_140/O
1 EQ /II_140/I	

Related Topics

[Add Faults](#)

[Report Testability Data](#)

[Delete Faults](#)
[Set Fault Sampling](#)
[Load Faults](#)
[Set Fault Type](#)
[Report Faults](#)
[Write Faults](#)

Set Fault Sampling

Scope: All modes

Specifies the fault sampling percentage used for circuit evaluation.

Usage

SET Fault Sampling *percentage* [-Seed *integer*]

Description

Fault sampling enables you to process a fraction of the total faults to decrease processing time for evaluating large circuits. Once you specify a percentage, the tool randomly picks the fault samples to process.

Arguments

- *percentage*
A required, positive number that specifies the fault sampling percentage used for circuit evaluation. Any number greater than 0 and less than or equal to 100 can be specified. The default is 100.
- -Seed *integer*
A switch and hex number pair that specifies a seed value to use in the selection of fault samples. Specifying unique seed values for different runs can provide more accurate fault sampling results. The number value must be a lower case, 32-bit hex representation. 0 is not a valid seed value. The default is 0XCCCCCCCC.

Related Topics

[Add Faults](#)

[Set Fault Mode](#)

[Load Faults](#)

[Set Fault Type](#)

[Report Faults](#)

[Write Faults](#)

Set Fault Type

Scope: All modes

Specifies the fault model for which the tool develops or selects ATPG patterns.

Usage

SET FAULT Type **Stuck** | **Iddq** | **TOGGLE** | **TRANSITION**

Description

The Set Fault Type command specifies the fault model type for which you want the tool to develop ATPG patterns. The default upon invocation of the tool is Stuck.

The fault sites of all models are the input and output pins of the design cells in addition to external pins. The tool uses the values 0 and 1 for all fault models to indicate the type of fault at the fault site. Each fault model has its own separate fault collapsing according to the model's rules of equivalence.

When you change the fault type, the tool deletes both the current fault list and the internal pattern set.

For more information on the different fault models, refer to [Scan and ATPG User's Manual](#).

Arguments

- **Stuck**
A literal that specifies for the tool to develop or select ATPG patterns for the single stuck-at fault model. This is the default upon invocation of the tool.
- **Iddq**
A literal that specifies for the tool to develop or select ATPG patterns for the IDDQ fault model.
- **TOGGLE**
A literal that specifies for the tool to develop or select ATPG patterns for the toggle fault model.
- **TRANSITION**
A literal that specifies for the tool to develop or select ATPG patterns for the transition fault model.

Related Topics

[Add Faults](#)

[Set Fault Mode](#)

[Delete Faults](#)

[Write Faults](#)

[Report Faults](#)

Set File Compression

Scope: All modes

Controls whether the tools read and write files with .Z or .gz extensions as compressed files (the default).

Usage

SET File Compression [ON | Off]

Description

Files that contain large pattern sets consume a very large amount of disk space. Fault lists and the design data itself also take up a lot of disk space. To conserve this space, the tools normally store files in one of two compressed formats when you provide a filename with the appropriate extension, as follows:

- “.Z” specifies to compress the file using the UNIX compress command.

Linux



Some versions of Linux do not support the compress command. In these cases, the tool will issue a warning, use the GNU gzip command to compress the file(s), and change the extension of the file being written to .gz. The tool can still read .Z files using zcat; however, Siemens EDA recommends that you use the .gz extension when writing compressed files.

- “.gz” specifies to compress the file using the GNU gzip command. You can control the type of GNU compression with the [Set Gzip Options](#) command.

When compressed file handling is enabled and you provide a filename with either of the above extensions, the tool will automatically decompress (for reading) or compress (for writing) the specified file.

The Set File Compression command allows you to turn off the tool’s normal compressed file handling functionality. This is useful in rare cases where files have either of the compressed file extensions, but should not be saved or read as compressed files.

Arguments

- ON
An optional literal that enables compressed file handling. This is the default.
- Off
An optional literal that disables compressed file handling. When set to off, the tools process .Z or .gz files without using compression.

Examples

Example 1

Suppose the file testpat.ascii.gz is not a compressed file. The following example disables compressed file handling so the tool will read testpat.ascii.gz as a normal file rather than as a compressed file:

```
set file compression off
```

Example 2

The following example re-enables compressed file handling, then saves the file fault.pat in GNU format:

```
set file compression on  
save patterns fault.pat.gz
```

Related Topics

[Save Patterns](#)

[Set Gzip Options](#)

Set Flatten Handling

Scope: All modes

Specifies how the tool globally handles flattening violations.

Usage

SET FLatten Handling *rule_id* [**Error** | **Warning** | **NOTe** | **Ignore**] [**Verbose** | **Noverbose**]

Description

The Set Flatten Handling command specifies the handling of the messages for net checking, pin checking, and gate checking. You can specify that the violation messages for these checks be either error, warning, note, or ignore. If you do not specify error, warning, note, or ignore, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the initial invocation setting as specified in the following list of rules.

Each rules violation has an associated occurrence message and summary message. The tool displays the occurrence message only for either error conditions or if you specify the Verbose option for that rule. The tool displays the rule identification number in all rules violation messages.

Arguments

- *rule_id*

A required, non-repeatable literal that specifies the identification of the exact flattening rule violations whose message handling you want to change.

The flattening rule violations and their identification literals are divided into the following three groups: net, pin, and gate rules violation IDs.

- Net flattening violations are described in sections FN1 through FN9
- Pin flattening violations are described in sections FP1 through FP13
- Gate flattening violations are described in sections FG1 through FG8

- Error

An optional literal that specifies for the tool to display both the error occurrence message and immediately terminate the rules checking.

If you do not specify the Error, Warning, Note, or Ignore option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

- Warning

An optional literal that specifies for the tool to display the warning summary message indicating the number of times the rule was violated. If you also specify the Verbose option, the tool displays the occurrence message for each occurrence of the rules violation.

If you do not specify the Error, Warning, Note, or Ignore option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

- **NOTE**

An optional literal that specifies for the tool to display the summary message indicating the number of violations for that rule. If you also specify the Verbose option, the tool also displays the occurrence message for each occurrence of the rules violation.

If you do not specify the Error, Warning, Note, or Ignore option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

- **Ignore**

An optional literal that specifies for the tool to not display any message for the rule's violations. The tool must still enforce some rules and they must pass to allow certain functions to be performed later.

If you do not specify the Error, Warning, Note, or Ignore option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

- **NOVerbose**

An optional literal that specifies for the tool to display the occurrence message only once for the rules violation and give a summary of the number of violations.

If you do not specify the Noverbose or Verbose option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

- **Verbose**

An optional literal that specifies for the tool to display the occurrence message for each occurrence of the rules violation.

If you do not specify the Noverbose or Verbose option, then the tool uses either the handling from the last Set Flatten Handling command or, if you have not changed the handling, the tool uses the initial invocation setting.

Examples

The following example changes the handling of the FG7 flattening rule to warning and specifies that each occurrence should be listed:

```
set flatten handling fg7 warning -verbose
```

Related Topics

[Report Flatten Rules](#)

[Set Drc Handling](#)

Set Gate Level

Scope: All modes

Specifies the hierarchical level of gate reporting and displaying.

Usage

SET GATE Level **Design** | **Primitive**

Description

The Set Gate Level command specifies the hierarchical gate level at which the tool operates. This includes the reporting and schematic display of gate information. Once you set the gate level, the tool processes all subsequent commands using the new gate level.

Whenever you issue a command which invalidates the flattened model, the tool also invalidates the hierarchical gate display structure. You can rebuild the hierarchical gate structure by creating a new flattened model. To do so, either enter and exit the Setup mode, or use the [Flatten Model](#) command.

Arguments

- **Design**
A literal that specifies to display gate information at the design library hierarchical gate level. These are the top-level cells of the design library which are instantiated in your design. This is the default upon invocation of the tool.
- **Primitive**
A literal that specifies to display gate information at the built-in primitive gate level.

Examples

The following example sets the gate report level so that reporting and display show the simulated values of the gate and its inputs (assuming a rules checking error occurred when exiting the Setup system mode):

```
set system mode atpg
set gate level primitive
set gate report error_pattern
report gates i_1006/o
```

Related Topics

[Flatten Model](#)

[Set Gate Report](#)

[Report Gates](#)

Set Gate Report

Scope: All modes

Specifies the information displayed by the Report Gates command.

Usage

For FlexTest

```
SET GATe REport {Normal | Race | Trace | Error_pattern | Tle_value | CONstrain_value |  
  Clock_cone pin_name | Analysis [Control | Observe] | {Drc_pattern { {Test_setup [{ {-  
  Cycle | -Time} n1 } [n2 | End]] } | Load_unload | SHift | SKew_load | SHADOW_Control  
  | Master_observe | SHADOW_Observe | STate_stability } [-All | time] } | REcord  
  {cycle_number | -All } | SIimulation | CONTRol}
```

Description

The Set Gate Report command controls the type of additional information that the Report Gates command displays. When you exit the Setup system mode, the trace and any rules-checking error pattern results will not be available with this command. For information on the format output by the different options in this command, refer to the [Report Gates](#) reference page.

Arguments

- **Normal**

A literal that specifies for the Report Gates command to display only its standard information. This is the default mode upon invocation of the tool.

- **Race**

A literal that specifies for the Report Gates command to display the simulated values of the gates for race conditions. To make the race conditions reportable, you must first use the Analyze Race command to check for race conditions between clock and data signals.

The Report Gates command displays any one of five possible simulated values. These values are: S, N, 0, 1, or Z, where S (same) indicates an unknown value that remains unchanged since the previous timeframe, and where N (new) indicates an unknown value that changes after the previous timeframe.

- **Trace**

A literal that specifies for the Report Gates command to display the simulated values of the gates during the scan chain tracing. The format of these values depends on the contents of the shift procedure. If the shift procedure contains additional frames, the additional frames will also be displayed in the gate report data. The trace data relates to the simulation performed during the scan chain tracing. Use the Trace option to determine why a scan chain was not properly sensitized during the shift procedure.

See for an example of the information displayed.

- **Error_pattern**

A literal that specifies for the Report Gates command to display the simulated value of the gates and its inputs for the pattern at which an audit error occurred.

- **Tie_value**

A literal that specifies for the Report Gates command to display the simulated values that result from all natural, tied gates and learned constant value non-scan cells.

Tie_value is available at any time the flattened model exists. The value displayed on the TIE node is set by simulation. All internal nodes are initially set to X for simulation.

Subsequently, ATPG or various commands may put values on the nodes. Occasionally, a set of circumstances may occur where an X shows up on a TIE node.

During operation, ATPG typically marks for simulation only those gates which are of interest for sensitization and propagation of a particular fault. If a gate is not marked, it will not be simulated, and any nodes associated with it may remain at an “X” value, even though they are connected to a TIE gate.

While tie_value reports the status of node values once it comes out of the DRC checks, the constrain_value tells you what the gate is suppose to act like for future consideration.

- **Constrain_value**

A literal that specifies for the Report Gates command to display the simulated values that result from all natural tied gates, learned constant value non-scan cells, constrained pins, and constrained cells.

The Report Gates command displays three values which are separated by a slash (/). These values are the gate constrained value (0, 1, X, or Z), the gate forbidden values (-, 0, 1, Z, or any combination of 01Z), and the fault blockage status (- or B, where B indicates all fault effects of this gate are blocked).

The displayed values are only available at the end of DRC once the tool successfully leaves Setup Mode. While tie_value reports the status of node values once it comes out of the DRC checks, the constrain_value tells you what the gate is suppose to act like for future consideration.

- **Clock_cone *pin_name***

A literal and string pair that specifies the clock pin for which the Report Gates command displays the clock cone data.

The clock cone data from the Report Gates command is the same data that is available as error data for clock rules violations. You can only use this option after flattening the simulation model using the [Flatten Model](#) command.

The *pin_name* must be a valid clock pin or an error condition occurs. The tool considers the pin equivalents when calculating the clock cones. State elements that the tool identifies as capturing on the clock's trailing edge will not propagate the clock effect cone. During the Setup system mode, this information is not available and the tool assumes all state elements capture with the leading edge of the selected clock.

- **Analysis** [Control | Observe]

A literal that sets gate reporting to display control or observe data learned from the fault analysis done with the Analyze Fault command

Control — A literal that specifies the Report Gate command to display the gate values needed to excite the fault site.

Observe — A literal that specifies the Report Gate command to display the gate values needed to detect the fault.

- **Drc_pattern_procedure_name** [-All | time]

Two literals and an optional, time triplet that specifies the name of the procedure and the time in the test procedure file that the Report Gates command uses to display a gates simulated value.

The valid literal choices for the *procedure_name* option (for use with Drc_pattern) are as follows:

Drc_F_rule — A literal that specifies an F rule between F1-F16.

Test_setup — A literal that specifies the use of the test_setup procedure. In the test procedure file, this procedure sets non-scan elements to the state you desire for the **load_unload** procedure. The tool uses the entire test_setup procedure unless you restrict the report to a certain portion using the -Cycle or -Time switch. In order to conserve screen width, time values are listed vertically in Test_setup gate reports.

-Cycle — A switch that indicates *n1* is a cycle number and specifies to start the report at cycle *n1*.

-Time — A switch that indicates *n1* is a time and specifies to start the report at time *n1*. The time units are based on the timescale defined in the test procedure file, which by default is 1 nanosecond.

n1 — An integer that specifies a cycle or time at which to start reporting. When used with the -Time switch, *n1* specifies a time. When used with the -Cycle switch, *n1* specifies a cycle. The tool numbers cycles beginning with 0; so, for example, to specify the second cycle, you would use “-cycle 1”.

n2 — An optional integer that specifies to report from cycle (or time) *n1* to cycle (or time) *n2* and stop reporting.

End — An optional literal that specifies to report from cycle (or time) *n1* to the end of the test_setup procedure.

Note

When using the -Cycle or -Time switch, if you do not include either the *n2* or End argument, gate reports will show data for only the *n1* cycle (or time).

Load_unload — A literal that specifies the use of the load_unload procedure. The test procedure file must contain this procedure, which describes how to load and unload data in the scan chains.

SHift — A literal that specifies the use of the shift procedure. The test procedure file must contain this procedure, which describes how to shift data one position down the scan chain.

SKew_load — A literal that specifies the use of the skew_load procedure. In the test procedure file, this procedure describes how to propagate the output value of the preceding scan cell into the master memory element of the current cell (without changing the slave), for all scan cells.


SHADOW_Control — A literal that specifies the use of the shadow_control procedure. In the test procedure file, this procedure describes how to load the contents of a scan cell into the associated shadow.

Master_observe — A literal that specifies the use of the master_observe procedure. In the test procedure file, this procedure describes how to place the contents of a master into the output of its scan cell.

SHADOW_Observe — A literal that specifies the use of the shadow_observe procedure. In the test procedure file, this procedure describes how to place the contents of a shadow into the output of its scan cell.

STate_stability — A literal that specifies the display of the simulation values for the **load_unload** procedure and the capture clock cycle that the tool used to determine the constant value state elements at the initial load time. The report separates the **shift** procedure values, **load_unload** procedure values, and the capture clock cycle with parentheses. This format provides information that is helpful when you are trying to debug boundary scan.

Note

 Only the simulation values for the first **load_unload** procedure that follows the **test_setup** procedure are displayed. Use the load_unload literal to display the final stable values in the **load_unload** procedure.

The state_stability option is not a procedure.

For detailed information about state stability reporting, refer to “[Display of State Stability Data](#)” in the *Scan and ATPG User’s Manual*.

TEST_End — A literal that specifies the use of the test_end procedure. In the test procedure file, test_end is used to add a sequence of events to the end of a test pattern set. The tool uses the entire test_end procedure. In order to conserve screen width, time values are listed vertically in test_end gate reports.

-All — An optional switch that specifies use of all times in the test procedure file. This is the default.

time — An optional positive integer, greater than 0, that specifies a time in the test procedure file.

- **REcord** *cycle_number* | **-All**

A literal and argument pair that specifies the recorded test cycles from the previous simulation run that you want the Report Gates command to display. The argument choices for the Record option are as follows:

cycle_number — A positive integer greater than 0 that specifies the recorded test cycle for which you want the Report Gates command to display internal values. A 1 indicates the last test cycle recorded, a 2 indicates the second from last, a 3 indicates the third from last, and so on. The total number of recorded test cycles is determined by the Run -Record command option.

-All — A switch that specifies for the Report Gates command to display all the recorded internal values as determined by the Run -Record command option.

Caution



The number of recorded test cycles multiplied by the number of timeframes per cycle must be less than 1024, to prevent exceeding the maximum string length of the Report Gates command.

- **Simulation**

A literal that specifies for the Report Gates command to display the current simulation value of the gate.

- **CONTRol**

A literal that specifies for the Report Gates command to display the controllability value of the gate.

Examples

The following example sets the gate report so that reporting and display show the simulated values of the gate and its inputs (assuming a rules checking error occurred when exiting the setup system mode):

```
set system mode atpg
set gate report error_pattern
report gates i_1006/o
```

Related Topics

[Report Gates](#)

[Set Gate Level](#)

Set Gzip Options

Scope: All modes

Specifies GNU gzip options to use with the GNU gzip command.

Usage

SET GZip Options [-Path *gzip_path*] [-Fast | -Best | -digit]

Description

The Set Gzip Options command specifies GNU gzip options the tool will use when compressing or decompressing files using the GNU gzip command. When file compression handling is enabled (as described under the [Set File Compression](#) command), the tool uses GNU gzip when processing files having a .gz extension.

The default speed and amount of compression corresponds to the gzip -4 option.

Arguments

- -Path *gzip_path*

An optional literal that specifies the full path *gzip_path* to a gzip executable file. You need to provide this pathname only if gzip is not in your normal UNIX search path. If you have specified a pathname with this option, you can restore the default behavior of using your UNIX path to find gzip by issuing the command with the -Path gzip option.

The following switches all control the speed of compression:

- -Fast

An optional switch that specifies to use the fastest compression method, which yields the least compression. This corresponds to the gzip -1 option.

- -Best

An optional switch that specifies to use the slowest compression method, which yields the most compression. This corresponds to the gzip -9 option.

- -digit

An optional switch that specifies an integer from 1 to 9 that the tool passes to gzip to control the rate of compression. You obtain the least amount of compression with -1, the greatest amount of compression with -9.

Examples

The following example ensures file compression is enabled, sets gzip compression to the fastest method, then saves a file using the .gz file naming extension in order to activate gzip file compression handling:

```
set file compression on
set gzip options -fast
save patterns fault.pat.gz
```

Related Topics

[Save Patterns](#)

[Set File Compression](#)

Set Hypertrophic Limit

Scope: All modes

Specifies the percentage of the original design's sequential primitives that can differ from the good machine, before the tool classifies them as hypertrophic faults.

Usage

SET Hypertrophic Limit **Off** | **Default** | **To *percentage***

Description

The Set Hypertrophic Limit command specifies the maximum percentage of original design to good machine difference that the tool allows, before classifying the fault as hypertrophic, and dropping it from the active fault list.

The term hypertrophic fault refers to a fault whose effects spread extensively throughout the design, meaning that the tool finds many internal value differences between the faulty machine and the referenced good machine. In fault simulation, hypertrophic faults require large amounts of memory and CPU time to process, and can significantly affect the performance of fault simulation. To improve fault simulation performance, FlexTest can drop these faults with little consequence to the accuracy of fault coverage.

Arguments

- **Off**
A literal that specifies for FlexTest to not define any hypertrophic faults.
- **Default**
A literal that resets the hypertrophic limit to the default value of 30 percent.
- **To *percentage***
A literal and positive integer pair that specifies the maximum percentage of differing sequential primitive output values that FlexTest allows before defining a fault as hypertrophic. The integer must be a value from 1 to 100.

Examples

The following example sets the hypertrophic fault limit at 10% of the total sequential primitives for the ATPG run:

```
set system mode atpg
add faults -all
set hypertrophic limit to 10
run
```

Set Iddq Checks

Scope: All modes

Specifies the restrictions and conditions to use when creating or selecting patterns for detecting IDDQ faults.

Usage

```
SET Iddq Checks [-NONE | {[-Int_float] [-Bus] [-WEAKBus] [-Pull] [-WIrre] [-WEAKHigh]  
[-WEAKLow] [-VOLTGain] [-VOLTLoss] [-Clock] [-WRite] [-Read]}] [-Warning |  
-Error]
```

Description

These restrictions only apply during the actual time of the IDDQ measurement; they are ignored at other times during a pattern.

To ensure no input is floating due to an unconnected module or instance input, before the design is flattened, you should issue the following commands at the beginning of Setup:

```
set flatten handling fn1 error  
set flatten handling fn3 error
```

If you are using the tool to create Iddq patterns, a violation of any of the Iddq Check options you select causes the tool to reject an offending pattern. If you select from existing patterns using the Select Iddq Patterns command, you can change the default behavior of rejecting offending patterns (-Eliminate option) by issuing the -Noeliminate option, allowing offending patterns to be retained.

If you are using FlexTest, it does not allow an IDDQ measurement when a violation of these restrictions occurs.

During simulation, whenever violations of the restrictions occur, a message displays identifying the gate associated with the violation and the number of patterns in which the violations occurred. The handling of the violation can be either a warning or error.

If you select -Error, simulation terminates at the first occurrence of a violation. You can use the Report Gates command to inspect the simulation values of all gates for patterns that violate the restrictions by first using the Set Gate Report command with the Error_pattern option.

For more information on IDDQ, see “[IDDQ Test](#)” in the *Scan and ATPG User’s Manual*.

Arguments


- **-NONE**

An optional switch that specifies to not perform any checks. This is the default.

- **-Int_float** — An optional switch that specifies to not allow a floating (Z) value on any ZVAL primitive gate that drives any Boolean logic. A ZVAL gate is placed between a source (Z producer) and a sink (nonZ consumer).

A Z producer is any primitive that has a Z output. Z producers include top-level inputs, bidi/inouts, tristate drivers, and FETs/switches. A nonZ consumer is any gate that treats Z the same as X for all inputs. NonZ consumers include any Boolean gate. You can see the primitive truth tables for all DFT-supported primitives, in section “[Supported Primitives](#)” of the *Tessent Cell Library Manual*.


Note

 A ZVAL gate is not placed between a Z producer and a PO or the PO half of a bidi because both of these have the ability to output a Z value. However, a ZVAL gate is placed between the PI half of the bidi and any internal _buf as typifies the input path of a bidi/inout IO pad. For those pads, if the output stage is disabled (Z out), and the bidi/inout PAD is driven externally by Z from the tester, and the IO pad has no modeled pullup/down, the ZVAL in front of the _buf input path can have a Z on its inputs; it fails the -int_float check. These “floating” checks only occur at buses. To ensure no input is floating due to an unconnected module or instance input, before the design is flattened, you should issue the following commands at the beginning of Setup:

```
set flatten handling fn1 error
set flatten handling fn3 error
```

- **-Bus**
An optional switch that specifies to not allow contention conditions on bus gates.
- **-WEAKBus**
An optional switch that specifies to not allow contention conditions on weak-bus gates.
- **-Pull**
An optional switch that specifies to not allow contention conditions on pull gates.
- **-WIrre**
An optional switch that specifies that all inputs of a wire gate must be set to the same value. There should be no contention on wires during IDDQ measurements because contention can raise the IDDQ current.
- **-WEAKHigh**
An optional switch that specifies that a bus gate must not be at a high state controlled by a weak value at its input. For designs with enhancement mode pullups, if a bus gate does not have a zhold or zhold1 bus_keeper, this switch causes the tool to fail the IDDQ measurement.

Note

 This switch should only be used when the design has an enhancement mode pullup to check for zhold or zhold1 bus_keeper. This switch is rarely ever needed.

- -WEAKLow

An optional switch that specifies that a bus gate must not be at a low state controlled by a weak value at its input. For designs with enhancement mode pulldowns, if a bus gate does not have a zhold or zhold0 bus_keeper, this switch causes the tool to fail the IDDQ measurement.

Note



This switch should only be used when your design has an enhancement mode pulldown to check for zhold or zhold0 bus_keeper. This switch is rarely ever needed.

- -VOLTGain

An optional switch that specifies that a PMOS transistor must not be at a logic zero unless a zhold0 or zhold bus_keeper DFT library attribute is between the PMOS transistor and the BUS root. A compromised (Vss+) logic zero is pulled down to the Vss rail (zero volts).

- -VOLTLoss

An optional switch that specifies that a NMOS transistor must not pass a logic one unless a zhold1 or zhold bus_keeper DFT library attribute is between the NMOS transistor and the BUS root. A compromised (Vdd-) logic one is pulled up to the Vdd rail (Vdd volts).

- -Clock

An optional switch that specifies to not allow clock pins to be on during the IDDQ measure. When specifying this switch, the [Select Iddq Patterns](#) command filters out the clock_po patterns and includes patterns that pulse clocks in for the selection.

Note



The advantage of the -Clock switch, during pattern generation, is that it prevents creation of Clock PO style patterns, one of the causes of multiple timeplates. The switch also ensures that when a clock is turned on, it is pulsed at the normal time.

For FlexTest, the pin constraints of all clock inputs must not be at an on state at the last timeframe of each test cycle. Otherwise, the tool cannot perform an IDDQ measurement.

- -Write

An optional switch that specifies to not allow write control pins to be on during the IDDQ measure.

For FlexTest, the pin constraints of all write control inputs cannot be at an on state at the last timeframe of each test cycle. Otherwise, the tool cannot perform an IDDQ measurement.

- -Read

An optional switch that specifies to not allow read control pins to be on during the IDDQ measure.

For FlexTest, the pin constraints of all read control inputs cannot be at an on state at the last timeframe of each test cycle. Otherwise, the tool cannot perform an IDDQ measurement.

- [-WArning](#)
An optional switch that treats violations of IDDQ checks as warnings. This is the default.
- [-ERror](#)
An optional switch that treats violations of IDDQ checks as errors and stops the simulation process immediately.

Related Topics

[Report Gates](#)

[Set Gate Report](#)

[Select Iddq Patterns](#)

[Set Iddq Strobe](#)

[Set Fault Type](#)

Set Iddq Strobe

Scope: All modes

Prerequisites: You must be fault simulating, or selecting from, an external pattern source.

Specifies on which patterns (cycles) the tool will simulate IDDQ measurements.

Usage

SET IDdq Strobe **-Label** | **-All**

Description

The Set Iddq Strobe command affects stand-alone fault simulation as well as pattern selection when you use the Select IDDQ Patterns command. Set IDDQ Strobe determines whether the tool simulates only existing IDDQ measure statements within an external pattern source, or if it should assume that every pattern (cycle) in the set has an IDDQ measure statement. The command performs slightly differently depending on which tool you are using. In either case, use the Report Environment command to list the current setting. The following paragraphs describe how the Set Iddq Strobe command operates for each tool.

The Set Iddq Strobe command specifies which test cycles will perform IDDQ measures during simulation.

Arguments

- **-Label**
A switch that restricts IDDQ measures to those test cycles which have the IDDQ measure statement. This is the default behavior upon invocation of the tool.
- **-All**
A switch that allows FlexTest to use all test cycles for IDDQ fault detection.

Examples

The following example fault grades an external IDDQ pattern file and restricts IDDQ detection/measures to those patterns/test cycles which have the IDDQ measure statement:

```
set system mode fault
set pattern source external pat_file
set fault type iddq
set iddq strobe -label
add faults -all
run
```

Related Topics

[Select Iddq Patterns](#)

[Set Iddq Checks](#)

[Set Fault Type](#)

[Set Pattern Source](#)

Set Instruction Atpg

Scope: Atpg mode

Specifies whether FlexTest generates instruction-based test vectors using the random ATPG process.

Usage

SET INstruction Atpg **Off** | {**ON** *filename*}

Description

The Set Instruction Atpg command specifies that during ATPG FlexTest either generates functional test vectors using the instruction set that you specify in a file, or generates common test vectors using the standard sequential-based ATPG.

Typically, instruction-based test vectors are useful for high-end, non-scan designs. Such high-end designs usually contain a large block of logic, like a microprocessor, that lends itself to instruction-based test vectors.

When you provide an ASCII file containing the information on the instruction set of a design, FlexTest can randomly combine these instructions to produce a high- coverage functional pattern set. FlexTest first chooses an instruction, and then tries to detect as many faults as possible with that instruction.

Arguments

- **Off**
A literal that disables FlexTest from performing instruction-based test generation. This is the default upon invocation of FlexTest.
- **ON**
A literal that enables FlexTest to perform instruction-based test generation using the information you provide in *filename*.
- *filename*
A string specifying the name of the ASCII file that describes all the input pins and the instruction set that you want the instruction-based test generation to use.

Examples

The following example enables instruction-based test generation:

```
set instruction atpg on /user/design_one/instruction_file
```

Set Internal Fault

Scope: Setup mode

Specifies whether the tool allows faults within or only on the boundary of library models.

Usage

SET Internal Fault **OFF** | **ON**

Description

The Set Internal Fault command specifies whether the tool allows faults on internal nodes of library models (internal faulting) or only on the library model boundary. The default upon invocation of the tool is to allow faults only on the library model boundary.

Arguments

- **OFF**
A literal that allows faults only on the boundary of the library models. This is the default upon invocation of the tool.
- **ON**
A literal that allows faults on the internal nodes of library models.

Set Internal Name

Scope: Setup mode

Specifies whether to delete or keep pin names of library internal pins containing no-fault attributes.

Usage

SET INternal Name **OFF** | **ON**

Description

The Set Internal Name command specifies whether to keep internal library pins with no-fault attributes. Normally, you should delete these names for memory and performance reasons. The default operation (OFF) upon invocation is to delete these names.

Arguments

- **OFF**
A literal that deletes the lowest level pin names if they have the nofault attribute. This is the default.
- **ON**
A literal that keeps the lowest level pin names, even if they have the nofault attribute.

Examples

The following example deletes pin names with the nofault attribute.

```
set internal name off
```

Set Interrupt Handling

Scope: All modes

Specifies how FlexTest interprets a Control-C interrupt.

Usage

SET INTERRUPT HANDLING [OFF | ON]

Description

The Set Interrupt Handling command controls the tool's ability to place a command in a suspended state.

By default, if you enter a Control-C during the execution of a command, FlexTest aborts the command process and there is no way for you resume; if you want to complete the interrupted command, you must start the command over from the beginning.

Once you enable suspend-state interrupt handling, a Control-C no longer abruptly aborts a command process. Rather, FlexTest places the command in a suspended state so that you can check the status or make minor adjustments to the suspended command and then either abort or resume the suspended command. The following lists the commands that you can issue while a command is in suspend-state:

- Help
- All Report commands
- Set Abort Limit
- Set Atpg Limits
- Set Checkpoint
- Set Fault Mode
- Set Gate Level
- Set Gate Report
- Set Logfile Handling
- Save Patterns
- All Write commands

If you turn interrupt handling on, you can either abort the process using the Abort Interrupted Process command, or continue the process using the Resume Interrupted Process command.

Arguments

- OFF
An optional literal that disables suspend-state interrupt handling. This is the default.
- ON
An optional literal that enables suspend-state interrupt handling.

Examples

The following example enables suspend-state interrupt handling, begins an ATPG run, and (sometime before the run completes) interrupts the run:

```
set interrupt handling on
set system mode atpg
add faults -all
run
<control-c>
```

Now, with the Run command suspended, the example continues by writing all the untestable faults to a file for review, and then resumes the Run:

```
write faults faultlist -class ut
resume interrupted process
```

Related Topics

[Abort Interrupted Process](#)

[Resume Interrupted Process](#)

Set Learn Report

Scope: All modes

Specifies whether the Report Gates command can display the learned behavior for a specific gate.

Usage

SET LEarn Report OFF | ON

Description

The Set Learn Report command specifies whether the Report Gates command should include the information that the tool collects during the static learning process. The application automatically performs the static learning process immediately after it flattens the simulation model, which happens when you leave the Setup mode or issue the Flatten Model command. The static learning process provides general information on the design that the tool can then use in speeding up the ATPG process (such as values that are impossible on other gates if the selected gate is at a specific value.)

Once you enable access to the static learned information with the Set Learn Report command, you can specify for the tool to display the learned information on a selected gate by using the Report Gates command.

While you can also access the learned information with the Report Gates command by using the -Type option, this method displays the information for all the gates of the specified gate type. When you enable access with the Set Learn Report command, the tool automatically displays the learned information with all command options. Therefore, you can restrict the report to the learned information on an individual object.

Arguments

- OFF
An optional literal that disables access to the learned behavior information. This is the default.
- ON
An optional literal that enables access to the learned behavior information.

Examples

The following example enables access to the learned behavior and then accesses that information:

```
set learn report on  
report gates 28
```



```

/MX3/OR1 (28)  OR
  IO      I   20-/MX3/AN2/OUT
  I1      I   24-/MX3/AN1/OUT
  OUT     O   37-/OUT0
Learned behavior:  MUX(9,13,17)

```

Related Topics

[Report Gates](#)

Set List File

Scope: All modes

Specifies the name of the list file into which the tool places the pins' logic values during simulation.

Usage

SET List File **-Default** | *{filename [-Replace]}*

Description

The Set List File command specifies the file in which the tool places the simulation values for the pins which you previously identified with the Add Lists command. The default behavior is for the tool to display the simulation values for the pins on standard output.

You can display the list of reported pins by using the Report Lists command.

Arguments

- **-Default**
A switch that specifies for the tool to display the logic values of pins to the standard output. This is the default behavior upon the invocation of FlexTest.
- *filename*
A string that specifies the name of the file in which the tool places the logic values of pins during simulation.
- **-Replace**
An optional switch that replaces the contents of the file if the *filename* already exists.

Related Topics

[Add Lists](#)

[Report Lists](#)

[Delete Lists](#)

Set Logfile Handling

Scope: All modes

Specifies for the tool to direct the transcript information to a file.

Usage

SET LOGfile Handling [-RESume | {*filename* [-Replace | -Append]]

Description

The Set Logfile Handling command causes the tool to write the transcript information, which includes the commands and the corresponding output (if any), into the file you specify. You can execute the Set Logfile Handling command at any time, as many times as you need.

In the logfile, the command keyword precedes all commands that the tool executes. You can easily search for the executed commands, generate a separate dofile containing those commands, and then execute the dofile, thereby rerunning those commands within the tool.

When you set the logfile handling, the tool still writes the same information to the session transcript window in addition to the logfile. However, you can disable the writing of the information to the transcript window with the Set Screen Display command.

If you want to stop writing to a logfile, issue the Set Logfile Handling command with no options, which closes the appropriate file.

Arguments

- **-Resume**
An optional switch that specifies for the tool to resume writing transcript output to the logfile specified at invocation with the -Logfile invocation switch. You do not need to include the name of the invocation logfile with this switch; the tool will remember the name.
- *filename*
A string that specifies the name of the file to which you want the tool to write the transcript output. This string can be a full pathname or a leafname. If you only specify a leafname, the tool creates the file in the directory from which you invoked the tool.
If you do not specify a *filename*, the tool discontinues writing to the logfile and closes it.
- **-Replace**
An optional switch that forces the tool to overwrite the file if a file by that name already exists.
- **-Append**
An optional switch that causes the tool to begin writing the transcript at the end of the specified file.

Examples

The following example writes a logfile and disables the display of the transcript information in the transcript window:

```
set logfile handling /user/designs/setup_logfile  
set screen display off  
add clocks 0 clk  
add clocks 1 pre clr  
report clocks
```

The following information shows what the logfile contains after running the preceding set of commands:

```
// command: set screen display off  
// command: add clocks 0 clk  
// command: add clocks 1 pre clr  
// command: report clocks  
PRE, off_state 1  
CLR, off_state 1  
CLK, off_state 0
```

Related Topics

[Report Environment](#)

[Set Screen Display](#)

Set Loop Handling

Scope: Setup mode

Specifies how the tool handles feedback networks.

Usage

SET LOOP Handling **Tiex** | **Delay** | **Simulation**

Description

The Set Loop Handling command lets you perform DRC simulation of circuits containing combinational feedback networks.

The Set Loop Handling command specifies FlexTest loop handling behavior by either 1) using a TIE-X gate to break the loop, or 2) inserting a delay element to break the loop, or 3) using a simulation process to identify the loop behavior.

For another look at combinational feedback loops, refer to “[Feedback Loops](#)” in the *Scan and ATPG User’s Manual*.

Arguments

- **Tiex**
A literal that specifies that TIE-X gates are used to break combinational loops.
- **Simulation**
A literal that specifies for the tool to use a simulation process to stabilize values in the loop. This option gives more accurate simulation results than other options. This is the default.
- **Delay**
A literal that inserts a delay element to break a loop.

Set Net Dominance

Scope: Setup mode

Specifies the fault effect of bus contention on tri-state nets.

Usage

SET NEt Dominance **Wire** | **And** | **Or**

Description

The Set Net Dominance command specifies the fault effect of bus contention on tri-state nets. This provides the capability to detect some faults on tri-state driver enable lines when those drivers connect to a tri-state bus. These faults would be ATPG-untestable unless the tool can use the Z-state for detection.

The truth tables for each type of bus contention fault effect are given in Tables [2-11](#), [2-12](#), and [2-13](#). This command does not affect Good machine behavior.

Table 2-11. WIRE Bus Contention Truth Table

	X	0	1	Z
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

Table 2-12. AND Bus Contention Truth Table

	X	0	1	Z
X	X	0	X	X
0	0	0	0	0
1	X	0	1	1
Z	X	0	1	Z

Table 2-13. OR Bus Contention Truth Table

	X	0	1	Z
X	X	X	1	X
0	X	0	1	0
1	1	1	1	1
Z	X	0	1	Z

Arguments

- **Wire**

A literal that specifies for the tool to use unknown behavior for the fault effect of bus contention on tri-state nets. This is the default behavior upon invocation of the tool.

- **And**

A literal that specifies for the tool to use wired-AND behavior for the fault effect of bus contention on tri-state nets.

- **Or**

A literal that specifies for the tool to use wired-OR behavior for the fault effect of bus contention on tri-state nets.

Set Net Resolution

Scope: Setup mode

Specifies the behavior of multi-driver nets.

Usage

SET NEt Resolution **Wire** | **And** | **Or**

Description

The Set Net Resolution command specifies the behavior of non-tri-state, multi-driver nets. The default upon invocation of the tool is Wire, which requires all inputs be at the same value to achieve a value. If you can model your nets using the And or Or option, you can improve your test coverage results.

Arguments

- **Wire**
A literal that specifies for the tool to use unknown behavior for non-tri-state, multi-driver nets. This requires all inputs to be at the same value to achieve a value other than X. This is the default upon invocation of the tool.
- **And**
A literal that specifies for the tool to use wired-AND behavior.
- **Or**
A literal that specifies for the tool to use wired-OR behavior.

Set Nonscan Model

Scope: Setup mode

Specifies how FlexTest classifies the behavior of nonscan cells with the HOLD and INITX functionality during the operation of the scan chain.

Usage

SET NOscan Model **DRC** | **HOLD** | **INITX**

Description

By default, the Design Rules Checker (DRC) classifies the behavior of each non-scan cell during the operation of the scan chain. However, the DRC sometimes classifies nonscan cells with the HOLD capability as having the INITX functionality, which has the disadvantage of decreasing the test coverage.

Despite this decreased test coverage, there is an advantage to FlexTest classifying nonscan cells as INITX rather than as HOLD. The method that FlexTest uses for pattern compaction operates better with INITX gates rather than HOLD gates.

Thus, the Set Nonscan Model command lets you choose between either better pattern compaction with the disadvantage of having a decreased test coverage, or not as good pattern compaction with the advantage of having a higher test coverage.

This command has no effect on nonscan cells that you identified with the Add Scan Model or Add Scan Instance commands.

Arguments

- **DRC**

A literal that specifies for FlexTest to allow the Design Rules Checker to classify each nonscan cell based on the criteria listed in the following table.

Table 2-14. DRC Non-scan Cell Classifications

Classification	Criteria
HOLD	The state value of the nonscan cell is unknown (X), but it remains the same as immediately before a scan operation.
INITX	Behaves the same as the HOLD gate, but its state can change values during a scan operation.
INIT0	The state of the nonscan cell remains low (0) immediately after a scan operation.
TIE0	Behaves the same as the INIT0 gate, but also remains at a low state during all nonscan operations.
INIT1	The state of the nonscan cell remains high (1) immediately after a scan operation.

Table 2-14. DRC Non-scan Cell Classifications (cont.)

Classification	Criteria
TIE1	Behaves the same as the INIT1 gate, but also remains at a low state during all nonscan operations.

This is the default upon invocation of FlexTest.

- **HOLD**

A literal that specifies for FlexTest to classify as HOLD those nonscan cells that have the criteria to be INITX. This increases the test coverage.

- **INITX**

A literal that specifies for FlexTest to classify as INITX those nonscan cells that have the criteria to be HOLD. This increases pattern compactability.

Examples

The following example specifies for FlexTest to classify as HOLD all nonscan cells that qualify as an INITX to increase the test coverage:

```
set nonscan model hold
```

Related Topics

[Report Environment](#)

[Write Environment](#)

Set Output Comparison

Scope: All modes

Specifies whether FlexTest performs a good circuit simulation comparison.

Usage

SET OUtput Comparison **Off** | {**ON** [-X_ignore [None | Reference | Simulated | Both]]} [-Io_ignore]

Description

The Set Output Comparison command lets you specify for FlexTest to compare good circuit simulation results to an external test pattern set. The purpose is to verify the correctness of the simulation model.

The -X_ignore options will allow you to control whether x values in either simulated results or reference output should be ignored when output comparison capability is used.

Arguments

- **Off**
A literal that prevents FlexTest from performing a comparison of the outputs. This is the default upon invocation of FlexTest.
- **ON**
A literal that specifies for FlexTest to compare the good circuit simulation results.
- **-X_ignore None**
A switch that specifies FlexTest to compare x values between the simulated results and the reference output.
- **-X_ignore Reference**
A switch that specifies FlexTest to ignore the comparison between x values in the reference output. This is the default.
- **-X_ignore Simulated**
A switch that specifies FlexTest to ignore the comparison between x values in the simulated output.
- **-X_ignore Both**
A switch that specifies FlexTest to ignore the comparison of x values in both the simulated output and the reference output.
- **-Io_ignore**
An optional switch that specifies FlexTest to ignore bidirectional pins when they are in input mode.

Examples

The following example specifies for FlexTest to do good circuit simulation comparison on an external test pattern set during the run:

```
set output comparison on  
set system mode good  
set pattern source external pattern.refs  
run
```

If the reference value is 0 or 1, and the simulated value is different, the command reports the following:

For primary output:

```
DIFF, PO pol: expected=0 actual=1 at cycle=5 time=2
```

For scan unload:

```
DIFF, CHAIN chain1 POSITION 5: expected=0 actual=1 at cycle=5
```

Set Output Masks

Scope: Setup mode

Ignores any fault effects that propagate to the primary output pins you select.

Usage

SET OUtput Mask **OFF** | **ON**

Description

The tool uses primary output pins as the observe points during the fault detection process. When you mask a primary output pin, you inform the tool to mark that pin as an invalid observation point during the fault detection process. This command allows you the ability to flag all or a portion of the primary output pins that do not have strobe capability (are not observable). The tool classifies the faults whose effects only propagate to that observation point as Atpg_Untestable (AU).

You can use the [Report Output Masks](#) command to display the current setting of the output masks.

Arguments

- **OFF**
A required literal that specifies for the tool to use primary output pins as the observe points during the fault detection process. In addition, the tool will give possible credit for fault effects that reach a pin that the pattern set specifies as being at an unknown value. This is the default behavior upon invocation.
- **ON**
A required literal that specifies for the tool to mask the simulated primary output pin values with Xs for all patterns. As a result, the tool does not give any credit for fault effects that reach those masked primary output.

Related Topics

[Add Output Masks](#)

[Report Environment](#)

[Delete Output Masks](#)

[Report Output Masks](#)

Set Pattern Source

Scope: Atpg, Fault, and Good modes

Specifies the source of the test patterns for subsequent Run commands.

Usage

SET PAttern Source **Internal** | { {**External *filename*** } [-**Ascii**] -Table | { -Vcd
{-Control *control_filename*} } | -STil | -WGl] [-NOPadding]}

Arguments

- **Internal**

A literal that specifies the internal set of patterns. This is the default upon invocation of the tool.

In ATPG system mode, this option directs the Run command to perform the basic ATPG process. In Fault simulation or Good circuit simulation system mode, this option directs the Run command to perform simulation for the internal set of patterns that Run generated during the previous ATPG system mode.

- **External *filename***

A literal and string pair that specifies the set of patterns contained in *filename* when performing a simulation run. For simulation, the specified patterns are loaded into an external pattern set database.

In ATPG system mode, the tool adds effective patterns from a simulation run to the internal pattern set, if there is at least one undetected fault in the fault list. When you use the -All_patterns switch, the tool adds all simulated patterns from the external pattern set to the internal pattern set, not just the effective patterns.

In Fault and Good modes, a simulation run does not alter the internal pattern set.

For FlexTest, the external patterns can be in ASCII test pattern format, table format, VCD (Value Change Dump), STIL, or WGL pattern file format. If *filename* contains external patterns in table format, you must also use the -Table option. If *filename* is a VCD pattern file, you must use the -Vcd option AND specify a control_filename by using the -Control option.

Note



You may notice a drop in test coverage when using an external pattern set as compared to using generated patterns. This is an artificial drop.

There is an issue regarding artificial drops in test coverage due to redundant faults. When you run ATPG, the tool classifies some faults as redundant (RE). The tool knows these faults are untestable when calculating test coverage. If you restart the tool, or go back to Setup mode before reading the patterns back in, the faults that were previously classified as RE will now be classified as UC or UO. Now, the tool does not know these faults are untestable when calculating test coverage because faults cannot be proven redundant by

fault simulation. The tool will only fault simulate the external patterns. It will not target undetected faults and try to prove them redundant as it does when the pattern source is internal. Therefore, you may see a slight drop in test coverage. This drop is an artificial drop and coverage still remains the same.

- **-AScii**

An optional switch that specifies the external test pattern set is in ASCII format. This is the default for FlexTest.

You cannot use this option with the Internal pattern source.

- **-WGL**

An optional switch specifying that the external test pattern set is in WGL format. You must use this option if you specify external patterns that are in WGL format.

Restrictions:

- FlexTest allows only one timeplate in the WGL file.
- You can load either parallel or serial scan patterns. The patterns must be WGL patterns originally created in these tools.
- If you re-simulate scan patterns, only serial patterns are allowed.

FlexTest requires that you set a test cycle and pin strobing before reading patterns. For instance:

```
set test cycle 2
setup pin strobes 1
```

Example dofile:

```
add clocks 0 CLOCK
set test cycle 2
setup pin strobes 1
set system mode fault
set pattern source external results/testpat.wgl -wgl
add faults -all
run
```

- **-STil**

An optional switch specifying that the external test pattern set is in STIL format. You must use this option if you are loading STIL functional patterns into the tool. You can load either parallel or serial scan patterns, but you must have originally created the STIL patterns in FlexTest.

Analysis of the STIL patterns conforms to the IEEE 1450.0 specification. The following features from the IEEE 1450.0 specification are not supported:

- Multiple data elements per test cycle (See section 5.7 of IEEE 1450.0)
- Non-cyclized test data (See section 5.9 of IEEE 1450.0)

- Goto statement referring to a previous pattern label (See section 22.8 of IEEE 1450.0)
- MatchLoop statement (See section 22.7 of IEEE 1450.0)

Analysis of the STIL patterns only supports waveform descriptions in the WaveformTable that Siemens EDA tools currently support. Therefore, a waveform description should not have more than two event edges, unless it is describing a double pulse. All other waveforms that have more than two event edges are not allowed. Also, timing expressions are limited to using only the +, -, *, /, (, and) operators on constants or variables that have time-based units.

FlexTest requires that you set a test cycle and pin strobing before reading patterns. For instance:

```
set test cycle 2
setup pin strobes 1
```

Example dofile:

```
add clocks 0 CLOCK
set test cycle 2
setup pin strobes 1
set system mode fault
set pattern source external testpat.stil -stil
add faults -all
run
```

- **-Table**

An optional switch specifying that the External test pattern set is in table format. You must use this option if you specify External patterns that are in table format.

You cannot use this option with the Internal pattern source.

- **-Vcd {-Control control_filename}**

A control file name that contains the waveform information of each primary input and output pins. These two switches must be used together or you will receive an error message.

- **-NOPadding**

An optional switch specifying that the source test pattern set contains ASCII patterns that are not padded for the scan load and unload data. For example, the source pattern set may be one that you wrote with the Save Patterns command using its -NOPadding switch.

You cannot use this option with the Internal pattern source.

Examples

Example 1

The following example performs fault simulation on an external pattern file:

```
set system mode fault  
set pattern source external file1  
add faults -all  
run  
report statistics
```

Example 2

The following example shows how to combine patterns from several different files into one pattern set, which you can save in any supported format. First, the example loads patterns from three separate pattern files into the tool's external pattern set. The patterns are in the same order in which they are loaded. The example then writes the contents of the external pattern set into a new file in the WGL format. Notice the patterns loaded are in mixed formats (ASCII and binary); also, that the tool automatically knows to use the GNU gzip utility when processing the input file with the ".gz" extension.

```
set pattern source external my_patterns1.ascii  
set pattern source external my_patterns2.binary -append  
set pattern source external my_patterns3.ascii.gz -append  
save patterns my_patterns_all.wgl -wgl -external
```

Related Topics

[Save Patterns](#)

[Set Abort Limit](#)

[Set Random Atpg](#)

[Set Capture Clock](#)

[Setup Checkpoint](#)

Set Possible Credit

Scope: All modes

Specifies the percentage of credit that the tool assigns possible-detected faults.

Usage

SET POSSible Credit *percentage*

Description

The Set Possible Credit command specifies the percentage of possible-detected faults that the tool considers as detected when calculating the test coverage, fault coverage, and ATPG effectiveness. For the equations that the tool uses in these calculations, refer to “[Testability Calculations](#)” in the *Scan and ATPG User’s Manual*.

When you invoke the tool, the default credit value for possible-detected faults is 50 percent.

Arguments

- *percentage*

A required integer, from 0 to 100, that specifies the percentage of possible-detected faults that you want the tool to consider as detected when calculating the test coverage, fault coverage, and ATPG effectiveness. The default value upon invocation of the tool is 50 percent.

Examples

The following example sets the credit for possible detected faults as 25 percent for determining the fault coverage, test coverage, and ATPG effectiveness:

```
set system mode atpg
add faults -all
set possible credit 25
run
report statistics
```

Set Procedure Cycle_checking

Scope: Setup mode

Enables test procedure cycle timing checking to be done immediately following scan chain tracing during design rules checking.

Usage

SET PRocedure Cycle_checking **ON** | **OFF**

Description

This command helps detect timing problems in test procedures earlier on in the ATPG process. By default, the test procedure cycle timing checking is performed after scan chain tracing. If an error condition is detected, the tool remains in the Setup mode. You can then modify the test procedures and reissue the “set system mode” command.

Arguments

- **ON**
A literal that specifies for the tool to set procedure cycle_checking ON. This is the default behavior upon invocation of the tool.
- **OFF**
A literal that specifies for the tool to set procedure cycle_checking OFF. In order to turn procedure cycle_checking off, you must be in Setup mode.

Examples

```
set system mode setup
set procedure cycle_checking OFF
```

Related Topics

[Set System Mode](#)

Set Pulse Generators

Scope: Setup mode

Specifies whether the tool identifies pulse generator sink (PGS) gates.

Usage

SET PULse Generators **ON** | **OFF**

Description

The Set Pulse Generators command specifies the identification control of PGS gates. When on, the tool identifies a certain structure of the reconvergent PGS gates during the learning process. It then displays a summary message showing the number of identified PGS gates. The default setting is off.

The pulse generator learning is restricted to the 2-input AND/OR type of reconvergence structure. For more comprehensive pulse generator support, use the `_pulse_generator` primitive—see “[Pulse Generators With User-Defined Timing](#)” in the *Tessent Cell Library Manual* for complete information.

Arguments

- **ON**
A literal that specifies for the tool to identify the PGS gates during the learning process.
- **OFF**
A literal that specifies for the tool not to identify the PGS gates. This is the default behavior upon invocation of the tool.

Examples

The following example enables the identification of PGS gates during the learning process:

```
set pulse generators on
set system mode atpg
```

Related Topics

[Report Pulse Generators](#)

Set Race Data

Scope: Setup mode

Specifies how FlexTest handles the output states of a flip-flop when the data input pin changes at the same time as the clock triggers.

Usage

SET RAcE Data **Old** | **New** | **X**

Description

You can display the current setting of the race data with the Report Environment command.

Arguments

- **Old**
A literal specifying that a flip-flop retain the data on its output pins from the previous clock trigger. This is the default behavior upon invocation of FlexTest.
- **New**
A literal specifying that a flip-flop capture the new state that is on the data input.
- **X**
A literal specifying that a flip-flop output an unknown (X) state on its output pins.

Examples

The following FlexTest example captures the new state on the data input pin of all flip flops within the design:

```
set race data new
```

Related Topics

[Report Environment](#)

[Write Environment](#)

Set Rail Strength

Scope: All modes

Specifies for FlexTest to set the strongest strength of a fault site to a bus driver.

Usage

SET RAIl Strength ON | **Off**

Description

The Set Rail Strength command is useful in cases where the fault effect needs to propagate to the output of a bus. You can display the current setting of the rail strength with the Report Environment command.

Arguments

- **ON**
A literal specifying that the fault site has the strongest strength to a bus.
- **Off**
A literal that turns off rail strength properties. This is the default.

Examples

The following example enables rail strength properties.

```
set rail strength on
```

Set Random Atpg

Scope: All modes

Specifies whether the tool uses random patterns during ATPG.

Usage

SET RANdom Atpg **ON** | **OFF**

Description

The Set Random Atpg command controls whether the tool uses random test generation techniques to create patterns during the ATPG process.

Arguments

- **ON**
A literal that specifies for the tool to use random patterns to create test patterns. This is the default behavior upon invocation of the tool.
- **OFF**
A literal that specifies for the tool not to use random patterns to create test patterns. When you use this option, the tool only performs deterministic ATPG.

Examples

The following example turns off the random ATPG process, so only the deterministic ATPG is performed:

```
set system mode atpg
add faults -all
set random atpg off
run
```

Set Redundancy Identification

Scope: All modes

Specifies whether FlexTest performs the checks for redundant logic when leaving the Setup mode.

Usage

SET REdundancy Identification **ON** | **OFF**

Description

Use the Report Environment command to display the redundancy logic setting.

Arguments

- **ON**
A literal that specifies for FlexTest to perform checks for redundant logic when leaving the Setup mode. This is the default behavior upon invocation of FlexTest.
- **OFF**
A literal that prevents FlexTest from checking for redundant logic when leaving the Setup mode.

Examples

The following example disables the logic redundancy checks:

```
set redundancy identification off
```

Related Topics

[Report Environment](#)

Set Screen Display

Scope: All modes

Specifies whether the tool writes the transcript to the session window.

Usage

SET SScreen Display **ON** | **OFF**

Description

If you create a logfile with the Set Logfile Handling command, you may want to disable the tool from writing the same information to the session transcript window.

Arguments

- **ON**
A literal that enables the tool to write the session information to the transcript window. This is the default behavior upon invocation of the tool.
- **OFF**
A literal that disables the tool from writing any of the session information to the transcript window, including error messages.

Examples

The following example shows how to use the logfile functionality to capture the transcript in a file and then disable the tool from writing the transcript to the display.

```
set logfile handling /user/design/setup_file  
set screen display off
```

Related Topics

[Report Environment](#)

[Set Logfile Handling](#)

Set Self Initialization

Scope: Setup, ATPG, and Fault modes

Specifies whether FlexTest turns on/off self-initializing sequence behavior.

Usage

SET SELF Initialization **ON** | **OFF**

Description

In order to enable/disable generation or simulation of self-initializing sequences, the Set Self Initialization command must be issued prior to an ATPG run. The self-initialization setting is then valid until the command is re-issued.

You can save self-initializing pattern boundary information by writing patterns in ASCII format. Each self-initializing boundary starts at the PATTERN=nnn keyword and ends at the next occurrence of the PATTERN statement. Each pattern may have one or more cycles, where the cycle number is reset to zero at the beginning of the pattern.

When reading patterns, FlexTest reads in self-initializing information if it is present in the pattern file (and assuming Set Self Initialization On). In this case, the pattern file has additional statistics regarding the total number of self-initializing test patterns. The Report Statistics command displays the total number of test patterns in addition to the total cycle count. This report will include the following information:

```
...
Total Test Patterns = nnn
...
Total Test Patterns Generated = nnn
Total Test Patterns Simulated = nnn
...
```

Arguments

- **ON**
A literal that turns on self-initializing sequence behavior.
- **OFF**
A literal that turns off self-initializing sequence behavior. This is the default upon invocation of FlexTest.

Examples

```
set system mode atpg
set self initialization on
add faults -all
run
save patterns filename -ascii
```

The following example shows a pattern file with self-initializing information:

```
PATTERN=0 ;  
    CYCLE=0 ;  
    ...  
    ...  
    CYCLE=1 ;  
    ...  
    ...  
PATTERN=1 ;  
    CYCLE=0 ;  
    ...  
    ...  
    CYCLE=1 ;  
    ...  
    ...
```

Related Topics

[Report Statistics](#)

Set Sensitization Checking

Scope: All modes

Specifies whether DRC checking attempts to verify a suspected C3 or C4 rules violation.

Usage

SET SEnsitization Checking **OFF** | **ON**

Description

The Set Sensitization Checking command specifies whether the DRC verifies that the path from the source and sink of a suspected C3 or C4 violation exists when the source and sink clocks are on and all other clocks are off. If sensitization checking is on and the paths associated with the violation meet these conditions, the DRC reports the violation.

Arguments

- **OFF**
A literal that disables the C3 or C4 DRC sensitization check. This is the default behavior upon invocation of the tool.
- **ON**
A literal that enables the C3 or C4 DRC sensitization check.

Examples

The following example verifies suspected C3 or C4 rules violations.

```
set sensitization checking on
```

Related Topics

[Set Drc Handling](#)

Set Sequential Learning

Scope: All modes

Specifies whether the tool performs the learning analysis of sequential elements to make the ATPG process more efficient.

Usage

SET SEquential Learning **OFF** | **ON**

Description

The Set Sequential Learning command controls whether the tool performs the learning analysis immediately after design flattening. FlexTest uses the learned behavior for intelligent decision making in later processes, such as ATPG and DRC.

By enabling sequential learning, you prevent FlexTest from unnecessarily remaking many decisions, thereby improving the ATPG performance.

For more information about the learning analysis, refer to “[Learning Analysis](#)” in the *Scan and ATPG User’s Manual*.

Arguments

- **OFF**
A literal that disables the tool to perform additional, static learning analysis.
- **ON**
A literal that enables the tool to perform additional, static learning analysis. This is the default for FlexTest.

Examples

```
set sequential learning off
set system mode atpg
add faults -all
run
```

Related Topics

[Set Static Learning](#)

Set Shadow Check

Scope: Setup mode.

Specifies whether the tool will identify sequential elements as a “shadow” element during scan chain tracing.

Usage

SET SHadow Check **ON** | **OFF** | **REstricted**

Description

You can use the Set Shadow Check command to disable the checking and avoid corresponding error messages. This will prevent identification of any non-scan sequential element as a shadow element.

Arguments

- **ON**
A literal that enables shadow checking. This is the initial state upon invocation of the tool.
- **OFF**
A literal that disables shadow checking.
- **REstricted**
A literal that disables the tool’s identification of shadow cells that require independent shifts in certain pattern formats (for example, parallel Verilog format).

Note



An independent shift results from a one-shift apply shift statement in the load_unload procedure.

Examples

The following example disables shadow checking.

```
set shadow check off
```

Related Topics

[Set Drc Handling](#)

[Report Environment](#)

[Save Patterns](#)

Set Static Learning

Scope: Setup mode

Prerequisites: This command is only useful before the tool flattens the design to the simulation model, which happens when you first attempt to exit Setup mode or when you issue the Flatten Model command.

Specifies whether the tool performs the learning analysis to make the ATPG process more efficient.

Usage

SET SStatic Learning {**ON** [-Limit *integer*]} | **OFF**

Description

The Set Static Learning command controls whether the tool performs the learning analysis immediately after design flattening. The tools use the learned behavior for intelligent decision making in later processes, such as ATPG and DRC.

If you use the Set Static Learning ON command, the additional learned behavior focuses primarily on bus gates. This command also allows the test pattern generation process to immediately recognize conflicts and restricted decisions on ATPG constraints that result from the gate assignments. By enabling static learning, you prevent the tools from unnecessarily remaking many decisions, thereby improving the ATPG performance.

For more information about the learning analysis, refer to “[Learning Analysis](#)” in the *Scan and ATPG User’s Manual*.

Arguments


- **ON** -Limit *integer*

A literal and an optional switch and integer pair that performs additional static learning analysis. This is the default behavior upon invocation of the tools.

The optional switch and integer pair description is as follows:

-Limit *integer* — A switch and integer pair that specifies a single gate simulation activity threshold. When the tool reaches that threshold, it discontinues learning on gates in that design region. You specify the -Limit switch for performance reasons. The default value for the *integer* option is 1000.

Caution

 While changing the learning limit increases performance for some designs, it significantly decreases performance for a vast majority of designs. It is very unusual to need to change the learning limit and is therefore not recommended.

- **Off**

A literal that disables the tool from performing any learning analysis. You may want to do this to save time if you are not going to be running ATPG.

Examples

The following example first enables access to the learned information and then enables FlexTest to perform additional learning analysis:

```
set learn report on
set static learning on -limit 500
set system mode atpg
```

Related Topics

[Set Learn Report](#)

Set Stg Extraction

Scope: All modes

Specifies whether FlexTest performs state transition graph extraction.

Usage

SET STg Extraction **ON** | **OFF**

Description

The Set Stg Extraction command controls whether FlexTest automatically performs state transition graph extraction during the pre-processing of the non-scan circuit. State transition graph extraction can reduce the effort of state justification during ATPG. However, it can also lead to an increased test set size. Thus, if you are primarily concerned with the size of the test set, you should turn state transition graph extraction off.

Arguments

- **ON**
A literal that specifies for FlexTest to automatically perform state transition graph extraction for non-scan circuits. This is the default behavior upon invocation of FlexTest.
- **OFF**
A literal that specifies for FlexTest to not perform state transition graph extraction.

Examples

The following example turns off the state transition graph extraction:

```
set stg extraction off
```

Set System Mode

Scope: All modes

Specifies the system mode you want the tool to enter.

Usage

SET SYstem Mode {**Setup** | { **Atpg** | **Fault** | **Good** | **Drc** } [-Force]}

Description

The Set System Mode command directs the tool to a specific system mode. The system mode you specify may be any of the modes your tool supports. The default mode upon invocation of the tool is Setup.

When you switch from the Setup mode to any other mode, the tool builds a flat, gate-level simulation model.

Caution



When switching from any other mode to the Setup mode, the tool discards all the results generated in that mode. Before returning to Setup mode, use the [Save Patterns](#) and/or [Write Faults](#) commands to save any patterns or fault information you want to keep.

Arguments

- **Setup**

A literal that specifies for the tool to enter the Setup system mode.

Within this mode, you set up the design and simulation environments. If you used DFTAdvisor for scan insertion, the design settings are available in a dofile. The dofile usually contains, among other items, the clock, scan group, and scan chain definitions.

- **Atpg**

A literal that specifies for the tool to enter the Test Pattern Generation system mode.

In this mode, the Run command performs the test pattern generation process using the patterns indicated by the selected pattern source. The tool performs fault simulation to determine test coverage and places all effective patterns into the internal test pattern set.

- **Fault**

A literal that specifies for the tool to enter the Fault Simulation system mode.

In this mode, the Run command performs fault simulation on the selected pattern source. The tool calculates the test coverage, but does not store into the internal test pattern set the patterns that it used to achieve the test coverage.

- **Good**

A literal that specifies for the tool to enter the Good Simulation system mode.

In this mode, the Run command performs good machine simulation on the selected pattern source. You would normally use this mode for debugging.

- **Drc**

A literal that specifies for FlexTest to enter the Design Rule Checker mode, which you can enter to troubleshoot rule violations.

. The Drc mode retains the flattened design model that FlexTest used during the design rules checking process. When you exit Drc mode into either the Atpg, Fault, or Good system modes, FlexTest uses any information it learned while in the Drc mode.

- **-Force**

An optional switch that forces an exit of the Setup mode in the presence of non-fatal rules checking errors. This option has no effect except when exiting the Setup system mode.

Examples

The following example changes the system mode so you can perform an ATPG run:

```
add scan groups group1 scanfile
add scan chains chain1 indata2 outdata4
set system mode atpg
add faults -all
run
```

Related Topics

[Save Patterns](#)

[Write Faults](#)

Set Test Cycle

Scope: Setup mode

Specifies the number of timeframes per test cycle.

Usage

SET TEST Cycle *integer*

Description

The Set Test Cycle command specifies the number of timeframes per test cycle. Specifying a greater cycle width gives better resolution when using the Add Pin Constraints command. On the other hand, a greater width produces a larger performance overhead.

Arguments

- *integer*

A required integer that specifies the number of timeframes that you want for each test cycle. The default value upon invocation of the tool is 1.

If the number that you specify is less than the cycle width required for the Add Pin Constraints or Add Pin Strokes command, then the Set Test Cycle command displays a warning message. The message states that the conflicting pin constraints or pin strokes will be reset to the current default value.

Examples

The following example sets the test cycle width to allow the addition of pin constraints:

```
set test cycle 2
add pin constraints ph1 r1 1 0 1
add pin constraints ph2 r0 1 0 1
```

Related Topics

[Add Pin Constraints](#)

[Setup Pin Constraints](#)

[Add Pin Strokes](#)

[Setup Pin Strokes](#)

Set Trace Report

Scope: All modes

Specifies whether the tool displays gates in the scan chain trace.

Usage

SET TRace Report **Off** | **ON**

Description

The Set Trace Report command controls whether the tool displays all of the gates in the scan chain trace during rules checking.

Arguments

- **Off**
A literal that specifies for the tool not to display gates in the scan chain trace. This is the default behavior upon invocation of the tool.
- **ON**
A literal that specifies for the tool to display gates in the scan chain trace during rules checking.

Examples

The following example displays the gates in the scan chain trace during rules checking:

```
add scan groups group1 scanfile
add scan chains chain1 group1 indata2 outdata4
set trace report on
set system mode atpg
```

Related Topics

[Add Scan Chains](#)

[Report Scan Chains](#)

Set Transient Detection

Scope: Setup mode

Specifies whether the tool detects all zero width events on the clock lines of state elements.

Usage

SET TRansient Detection **Off** | {**ON** [-Verbose | -NOVerbose]}

Description

The Set Transient Detection command sets the simulator to detect all zero width events on the clock lines of state elements. If the zero width event causes a change of state in the state element, the tool sets that state element to X.

If Off is specified, DRC simulation treats all events on state elements as valid. Because the simulator is a zero-delay simulator, it is possible for DRC to simulate zero width, monostable circuits with ideal behavior which is rarely matched in silicon. The resulting zero width output pulse from the monostable circuit is also treated as a valid clocking event for other state elements.

Arguments

- **Off**

A literal that specifies for the tool to set transient detection off.

Caution



This argument is only for special cases where you understand the potential hazards from transients on the clock lines and know how to handle them. Turning transient detection off without taking care of the hazards can result in bad patterns that mismatch when verified in a timing based simulator. Be sure you understand the consequences and are willing to accept the risks before turning transient detection off.

- **ON**

A literal that specifies for the tool to set transient detection on. This is the invocation default.

- -Verbose

An optional switch that specifies for the tool to display a message identifying each time a state element is set to X due to transient detection. This is the invocation default and is valid only when transient detection is enabled.

- -NOVerbose

An optional switch which specifies for the tool not to display a message identifying each time a state element is set to X due to transient detection. This switch is valid only when transient detection is enabled.

Examples

Transient detection is enabled in verbose mode by default. Unless you use the Set Transient Detection command to modify the default, you will see a message similar to the following for each transition that is caused by a transient, or zero-width, pulse:

```
// Warning: 0 width pulse detected on pin 2 of  
/my_design/my_inst/my_latch/ (463228) in procedure my_procedure at time 0.
```

You can eliminate these messages, while retaining the advantages of transient detection, by issuing this command:

set transient detection on -noverbose

Or you can disable transient detection, causing DRC simulation to treat all events on state elements as valid.

set transient detection off

Set Unused Net

Scope: Setup mode

Specifies whether FlexTest removes unused bus and wire nets in the design.

Usage

SET UNUsed Net { **-Bus** { **ON** | **OFF** } | **-Wire** { **OFF** | **ON** } }...

Description

To properly handle bus and wire contention, FlexTest should not remove those unused nets. You can display the current settings of unused nets with the Report Environment command.

Arguments

- **-Bus ON | OFF**

A switch and literal pair that specifies whether FlexTest removes all unused bus nets in the design. The literal choices for the **-Bus** switch are as follows:

ON — A literal that specifies for FlexTest to keep all the unused bus nets in the design. This is the default behavior upon invocation of FlexTest.

OFF — A literal that specifies for FlexTest to remove all the unused bus nets in the design.

- **-Wire OFF | ON**

A switch and literal pair that specifies whether FlexTest removes all unused wire nets in the design. The literal choices for the **-Wire** switch are as follows:

OFF — A literal that specifies for FlexTest to remove all the unused wire nets in the design. This is the default behavior upon invocation of FlexTest.

ON — A literal that specifies for FlexTest to keep all the unused wire nets in the design.

Examples

The following example specifies for FlexTest to change the default for unused wire nets, but to retain the invocation default for buses (on):

```
set unused net -wire on
```

Related Topics

[Report Environment](#)

[Write Environment](#)

Set Z Handling

Scope: Setup mode

Specifies the simulation handling for high impedance signals on internal and external tri-state nets.

Usage

SET Z Handling {**Internal state**} | {**External state**}

Description

The Set Z Handling command specifies how to handle the high impedance state for internal and external tri-state nets during pattern simulation. If the high impedance value can be made to behave as a binary value, certain faults may become detectable. If you do not use this command to set the Z handling, the default value is X.

This command does not impact expected values on tri-state and bidirectional pins in the final saved patterns. The command changes the behavior of how Z values are simulated on internal nets. By default, Z values are treated as X values when fed into another internal gate, such as an AND gate. However, a Z value on a tri-state output pin is not converted to an X by default.

Note



This command applies to simulation only; it does not replace Z values in generated patterns. To alter the Z values in patterns to match the Z handling specified with this command, use the -Noz switch with the [Save Patterns](#) command when you save the patterns.

Arguments

- **Internal state**

A literal pair that specifies how the tool simulates high impedance values for internal tri-state nets. The literal choices for the *state* option are as follows:

X — A literal that specifies to treat high impedance states as an unknown state. This is the default behavior upon invocation of the tool.

0 — A literal that specifies to treat high impedance states as a 0 state.

1 — A literal that specifies to treat high impedance states as a 1 state.

Hold — A literal that specifies to hold the state previous to the high impedance.

- **External state**

A literal pair that specifies how the tool simulates high impedance values for external tri-state nets. The literal choices for the *state* option are as follows:

X — A literal that specifies not to measure high impedance states; they cannot be distinguished from a 0 or 1 state. This is the default behavior upon invocation of the tool.

- 0** — A literal that specifies to treat high impedance states as a 0 state that can be distinguished from a 1 state.
- 1** — A literal that specifies to treat high impedance states as a 1 state that can be distinguished from a 0 state.
- Z** — A literal that specifies to uniquely measure the high impedance states; they can be distinguished from both a 0 or 1 state.
- Hold** — A literal that specifies to hold the state previous to the high impedance.

Examples

The following example specifies to treat high impedance values as 1 states when they feed into logic gates, and as 0 states at the output of the circuit, then performs a pattern simulation run:

```
set z handling internal 1
set z handling external 0
set system mode atpg
add faults -all
run
```

Related Topics

[Save Patterns](#)

Setenv

Scope: All modes

Sets a shell environment variable within the tool environment.

Usage

SETENV *variable_name* [*value*]

Description

The Setenv command sets or changes the value of a shell environment variable within the current tool environment, enabling the tool to access the variable for the rest of the session just as if the variable had been set from the shell. This command is useful if you ever find that the tool requires an environment variable that was not set prior to invocation.

Note



A shell variable you set with this command is set only with respect to the current tool session. The command does not alter the value of the variable in the invocation shell.

To unset the variable before you exit, use the [Unsetenv](#) command. To view the value of the variable, use the [System](#) command to pass an appropriate shell command (echo or env for example) to the operating system for execution.

Arguments

- *variable_name*
A required string that specifies the name of the variable.
- *value*
An optional string that specifies the value of the variable.

Examples

The following example assigns a value to the environment variable SSH_AGENT_PID, then displays its value:

```
setenv SSH_AGENT_PID 66216
system echo $SSH_AGENT_PID
```

```
66216
```

Related Topics

[System](#)

[Unsetenv](#)

Setup Checkpoint

Scope: All modes

Specifies the checkpoint file to which the tool writes test patterns or fault lists during ATPG.

Usage

```
SETUp CHeckpoint {filename | -Nopattern} [period] [-Replace] [-Overwrite | -Sequence]  
[-Faultlist fault_file]
```

Description

The Setup Checkpoint command specifies the filename and time period in which the tool writes test patterns during test pattern generation. If you use the -Overwrite option and the tool does not create any new test patterns, the file is not updated. The -Faultlist *fault_file* option enables you to save a fault list.

In order to save only the fault list, use both the -Nopattern and the -Faultlist switches together on the command line.

Note



Although you can issue the Setup Checkpoint command in any mode, the tool will only write out a checkpoint file in ATPG mode with the [Set Checkpoint](#) command turned on.

For additional information on checkpointing, refer to “[Checkpointing Setup](#)” in the Scan and ATPG User’s Manual.

Arguments

- ***filename***
A string that specifies the name of the file into which you want to write the test patterns during test pattern generation.
- **-Nopattern**
A switch that specifies the tool should not save the test set. This option is provided for cases where you only want to save the fault list and not the test pattern set (use this option in conjunction with the -Faultlist *fault_file* option).
- ***period***
An optional integer that specifies the number of minutes between each write of the test patterns. The default is 100 minutes.
- **-Replace**
An optional switch that forces the tool to overwrite the file if a file by that name already exists.

- **-Overwrite**
An optional switch that specifies to overwrite the test patterns each time there are any differences. This is the default.
- **-Sequence**
An optional switch that writes the new test patterns to a new file each time a test pattern differs. The first file that the tool writes to is *filename*; each subsequent file is named “*filenameN*”, where N is an integer that starts at 1 and increases by one for each additional file.
- **-Faultlist *fault_file***
An optional switch and string pair that allows the fault list to be saved.

Related Topics

[Save Patterns](#)

[Set Pattern Source](#)

[Set Checkpoint](#)

[Write Faults](#)

Setup Pin Constraints

Scope: Setup mode

Prerequisites: You must execute the Set Test Cycle command before adding pin constraints.

Changes the default cycle behavior for non-constrained primary inputs.

Usage

SETUP PIn Constraints *constraint_format*

Description

The Setup Pin Constraints command changes the default cycle behavior for all primary inputs not specified with the Add Pin Constraint command. You must first specify the test cycle width with the Set Test Cycle command.

Arguments

- *constraint_format*

A required argument that specifies the new *constraint_format* default for all primary inputs not specified with the Add Pin Constraint command. The *constraint_format* argument choices are as follows:

NR period offset — A literal and two-integer triplet that specifies application of the non-return waveform value to the primary input pins. The test pattern set you provide determines the actual value FlexTest assigns to the pins.

C0 — A literal that specifies application of the constant 0 to the chosen primary input pin. If the value of the pin changes during the scan operation, the tool uses the non-return waveform.

C1 — A literal that specifies application of the constant 1 to the chosen primary input pins. If the value of the pin changes during the scan operation, the tool uses the non-return waveform.

CZ — A literal that specifies application of the constant Z (high impedance) to the chosen primary input pins. If the value of the pin changes during the scan operation, FlexTest uses the non-return waveform.

CX — A literal that specifies application of the constant X (unknown) to the chosen primary input pins. If the value of the pin changes during the scan operation, the tool uses the non-return waveform.

R0 period offset width — A literal and three integer quadruplet that specifies application of one positive pulse per period.

SR0 period offset width — A literal and three integer quadruplet that specifies application of one suppressible positive pulse during non-scan operation.

CR0 period offset width — A literal and three integer quadruplet that specifies no positive pulse during non-scan operation.

R1 period offset width — A literal and three integer quadruplet that specifies application of one negative pulse per specified period during non-scan operation.

SR1 *period offset width* — A literal and three integer quadruplet that specifies application of one suppressible negative pulse.

CR1 *period offset width* — A literal and three integer quadruplet that specifies no negative pulse during non-scan operation.

Where:

period — An integer that specifies the total number of test cycles. The Set Test Cycle command defines the number of timeframes per test cycle.

offset — An integer that specifies the timeframe in which values start to change in each period.

width — An integer that specifies the pulse width of the pulse type waveform.

Examples

The following example sets one primary input to behave as a clock and the rest of the primary inputs to behave as a constant high signal:

```
set test cycle 2
add pin constraint ph1 r1 1 0 1
setup pin constraints c1
```

Related Topics

[Delete Pin Constraints](#)

[Set Test Cycle](#)

[Report Pin Constraints](#)

Setup Pin Strobes

Scope: Setup mode

Changes the default strobe time for primary outputs without specified strobe times.

Usage

SETup PIn Strobes *integer* [-Period *integer*]

Description

The Setup Pin Strobes command changes the default strobe time of each test cycle for all primary outputs not specified with the Add Pin Strobes command. For scan circuits, FlexTest gives the last timeframe of each test cycle as the strobe time. For nonscan circuits, FlexTest gives time 1 of each test cycle as the strobe time.

Arguments

- *integer*
An integer which specifies the strobe time of each test cycle for all primary outputs without a specified strobe time. This number should not be greater than the period set with the Set Test Cycle command.
- -Period *integer*
Specifies the number of cycles for the period of each strobe. The default is 1.

Examples

The following example sets the strobe time to 2 for two primary outputs and changes the default strobe time to 3 for the rest:

```
set test cycle 4
add pin strobes 2 outdata1 outdata3
setup pin strobes 3
```

Related Topics

[Add Pin Strobes](#)

[Report Pin Strobes](#)

[Delete Pin Strobes](#)

Setup Tied Signals

Scope: Setup mode

Changes the default value for floating pins and floating nets which do not have assigned values.

Usage

SETup Tied Signals **X** | **1** | **0** | **Z**

Description

The Setup Tied Signals command specifies the default value that the tool ties to all floating nets and floating pins that you do not specify with the Add Tied Signals command. Upon invocation of the tool, if you do not assign a specific value, the tool assumes the default value is unknown (X).

If the model is already flattened and then you use this command, you must delete and recreate the flattened model.

Arguments

- **X**
A literal that ties the floating nets or pins to unknown. This is the default upon invocation of the tool.
- **0**
A literal that ties the floating nets or pins to logic 0 (low to ground).
- **1**
A literal that ties the floating nets or pins to logic 1 (high to voltage source).
- **Z**
A literal that ties the floating nets or pins to high-impedance.

Examples

The following example ties floating net vcc to logic 1, ties the remaining unspecified floating nets and pins to logic 0, then performs an ATPG run:

```
setup tied signals 0
add tied signals 1 vcc
set system mode atpg
add faults -all
run
```

Related Topics

[Add Tied Signals](#)

[Report Tied Signals](#)

[Delete Tied Signals](#)

Step

Scope: Setup mode.

Single-steps through several cycles of a test set.

Usage

STEp [*integer*] [-Record [*integer*]]

Description

The Reset State, Set Pattern Source, and Set System Mode commands will reset the cycle count such that the next Step command will start from the beginning of the external test set.

Arguments

- *integer*
Specifies the number of cycles to be simulated. This number indicates a window of cycles to be simulated. The first cycle to be simulated is the cycle after the one last simulated.
The default for integer is one global cycle.
- -Record
An optional switch used to record several cycles of simulation data which can be displayed later with the REPort GAtE command.

Examples

The following example specifies for FlexTest to simulate 3 cycles.

step 3

Related Topics

[Reset State](#)

[Set Pattern Source](#)

[Set System Mode](#)

System

Scope: All modes

Passes the specified command to the operating system for execution.

Usage

SYStem *os_command*

Description

The System command executes one operating system command without exiting the currently running application.

Arguments

- *os_command*
A required string that specifies any legal operating system command.

Examples

The following example performs an ATPG run, then displays the current working directory without exiting the tool:

```
set system mode atpg  
add faults -all  
run  
system pwd
```

Unsetenv

Scope: All modes

Unsets a shell environment variable within the tool environment.

Usage

UNSETEnv *variable_name*

Description

The Unsetenv command unsets a shell environment variable within the current tool environment. When you only need a variable unset within the tool, this command saves you the time and trouble of exiting, issuing the UNIX unsetenv command in the shell, and re-invoking.

Note



A shell variable you unset with this command is unset only with respect to the current tool session. The command does not alter the value of the variable in the invocation shell.

To set or change the value of a variable, use the [Setenv](#) command. To view the value of the variable, use the [System](#) command to pass an appropriate shell command (echo or env for example) to the operating system for execution.

Arguments

- *variable_name*

A required string that specifies the name of the variable.

Examples

The following example unsets the SSH_AGENT_PID environment variable within the current tool environment, then confirms that it is unset:

```
ATPG> system echo $SSH_AGENT_PID
```

```
66216
```

```
ATPG> unsetenv SSH_AGENT_PID
```

```
ATPG> system echo $SSH_AGENT_PID
```

```
ATPG>
```

Related Topics

[Setenv](#)

[System](#)

Update Implication Detections

Scope: Atpg and Fault modes

Prerequisites: You can use this command when there is an active fault list and you are using the stuck-at fault model.

Performs an analysis on the undetected and possibly-detected faults to determine if any of those faults can be classified as detected-by-implication.

Usage

UPDate IMplication Detections

Description

The Update Implication Detections command is automatically issued after the Run command executes a fault simulation when using external patterns or when using internal patterns in non-Atpg.

By invocation default, only scan-path-associated faults for the detected-by-implication classification are analyzed. The following faults are classified as detected-by-implication when the Update Implication Detections command is issued:

- A stuck-at-1 fault on the set input line of a transparent latch, scan latch, scan D flip-flop, shadow, copy, or sequential cell when the tool detects the stuck-at-1 fault on the output.
- A stuck-at-1 fault on the reset input line of a transparent latch, scan latch, scan D flip-flop, shadow, copy, or sequential cell when the tool detects the stuck-at-0 fault on the output.
- A stuck-at-0 fault on a clock input line of a transparent latch, scan latch, scan flip flop, shadow, copy, or sequential cell when the tool detects both the stuck-at-0 and stuck-at-1 faults for the associated data line.
- A stuck-at-0 fault on a data input line of a transparent latch, scan latch, scan D flip-flop, shadow, copy or sequential cell when the tool detects the stuck-at-0 fault and no other ports can capture a 1.
- A stuck-at-1 fault on a data input line of a transparent latch, scan latch, scan D flip-flop, shadow, copy, or sequential cell when the tool detects the stuck-at-1 fault on the data output and no other port can capture a 0.

Arguments

None.

Related Topics

[Report Faults](#)

Write Core Memory

Scope: All modes

Writes a file specifying the amount of memory required by FlexTest in order to avoid paging during the ATPG and simulation processes.

Usage

WRItE COre Memory *filename* [-Replace]

Description

You can use the Report Core Memory command to report the same information to the transcript. The peak memory requirement is generally much larger than the real memory required during the ATPG and fault simulation processes.

Arguments

- *filename*
A required string that specifies the name of the file where FlexTest is to write the current memory usage statistics.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.

Examples

The following example writes to the specified file the amount of memory required to avoid memory paging during the ATPG and simulation processes:

```
write core memory /user/design1/core_memory.file
```

The following file listing shows an example output of the Write Core Memory command:

	Peak	Current
Memory for flatten design	: 0.127M	0.125M
Memory for fault list	: 0.062M	0.062M
Memory for test generation	: 0.127M	0.125M
Memory for simulation	: 0.004M	0.004M
Memory for ram/rom	: 0.000M	0.000M
Total core memory	: 0.320M	0.317M

Related Topics

[Report Core Memory](#)

[Write Statistics](#)

Write Environment

Scope: All modes

Writes the current environment settings to the file that you specify.

Usage

WRItE ENvIRONMENT *filename* [-Replace]

Description

The Write Environment command outputs the same information as the Report Environment command except that FlexTest writes it to the file that you specify rather than to the session transcript.

Arguments

- *filename*
A required string that specifies the name of the file where FlexTest is to write the current environment settings.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.

Examples

The following example writes the current environment settings to the specified file:

```
write environment /user/designs/settings_file
```

The following listing shows the contents of an example Write Environment command:

```
Abort Limit = (backtrack=30, cycle=300, time=300 seconds)
ATPG Limit = (cpu_seconds=off, cycle_count=off, test_coverage=off)
Capture Clock (ATPG) = none
Capture Limit = off
Checkpoint = off
Checkpoint File = NULL
Clock Restriction = on
Contention Check = (bus=(0,noatpg,warning), port=off)
Dofile Abort = on
Fault Mode = uncollapsed
Fault Sampling = 100%
Fault Type = stuck_at
Gate Level = design
Gate Report = normal
Hypertrophic Limit = 30%
Iddq Checks = none, noatpg, warning
Iddq Strobe = label
Identification Model = (clock=original, disturb=on)
Internal Faults = on
Internal Name = off
List File = list1
Logfile Handling =
Loop Handling = hold
Net Dominance = wire
Net Resolution = wire
Nonscan Model = drc
Output Comparison = off
Output Mask = off
Pattern Source = internal
Pin Constraints (default) = type NR, period 1, offset 0
Pin Strokes (default) = 0
Possible Credit = 50%
Pulse Generators = on
Race Value = old
Random ATPG = on
Rundundancy Identification = on
Scan Identification = automatic Onternal Full backtrack=30 cycle=16
time=100
Screen Display = on
State Learning = on
System Mode = setup
Test Cycle Width = 1
Tied Signal = x
Trace Report = off
Unused Net = (bus=on wire=off)
Z Handling = (external=x, internal=x)
```

Related Topics

[Report Environment](#)

Write Faults

Scope: Atpg, Fault, and Good modes

Writes fault information from the current fault list to a file.

Usage

For FlexTest

WRite FAults *filename* [-Replace] [-Class *class_type*] [-Stuck_at {01 | 0 | 1}] [-All | *object_pathname...*] [-Hierarchy *integer*] [-Min_count *integer*] [-Noeq]

Description

The Write Faults command is identical to the Report Faults command, except that the data is written into a file. You can review or modify the file and later load the information into the fault list with the Load Faults command. You can use the optional arguments to narrow the focus of the report to only specific stuck-at or transition faults that occur on a specific object in a specific class. If you do not specify any of the optional arguments, Write Faults writes information on all the known faults to the file.

The file contains the following columns of information for each fault:

- fault value - The fault value may be either 0 (for stuck-at-0 or “slow-to-rise” transition faults) or 1 (for stuck-at-1 or “slow-to-fall” transition faults).
- fault code - A code name that indicates the lowest level fault class assigned to the fault.
- fault site - The pin pathname of the fault site.
- cell name (optional) - The name of the cell corresponding to the fault (included only if you used the -Cell_name argument).

You can use the -Hierarchy option to write a hierarchical summary of the selected faults. The summary identifies the number of faults in each level of hierarchy whose level does not exceed the specified level number. You can further specify the hierarchical summary by using the -Min_count option which specifies the minimum number of faults that must be in a hierarchical level before writing.

You may select to display either collapsed or uncollapsed faults by using the Set Fault Mode command.

Arguments

- *filename*
A required string that specifies the name of the file where the tool is to write the fault information.
- -Replace
An optional switch that replaces the contents of *filename* if the file already exists.

- **-Class** *class_type* [*subclass_name* | *subclass_code*]

An optional switch and literal pair that specifies the class of faults that you want to write. The *class_type* argument can be either a fault class code or a fault class name. If you do not specify a *class_type*, the default is to write all fault classes.

- **-Stuck_at** 01 | 0 | 1

An optional switch and literal pair that specifies the stuck-at or transition faults you want to write. The choices are as follows:

01 — A literal that specifies for the tool to write both “stuck-at-0” and “stuck-at-1” faults for stuck-at faults; or to write both “slow-to-rise” and “slow-to-fall” faults for transition faults. This is the default.

0 — A literal that specifies to write only the “stuck-at-0” faults (“slow-to-rise” faults for transition faults).

1 — A literal that specifies to write only the “stuck-at-1” faults (“slow-to-fall” faults for transition faults).

- **-All**

An optional switch that writes all faults, protected as well as unprotected, on all model, netlist primitive, and top module pins to the file. This is the default.

- *object_pathname*

An optional repeatable string that specifies the list of pins, instances, or delay paths whose faults you want to write.

- **-Hierarchy** *integer*

An optional switch and integer pair that specifies the maximum fault class hierarchy level for which you want to write a hierarchical summary of the faults.

- **-Min_count** *integer*

An optional switch and integer pair that you can use with the **-Hierarchy** option and that specifies the minimum number of faults that must be in a hierarchical level to write a hierarchical summary of the faults. The default is 1.

- **-Noeq**

An optional switch that specifies for the tool to write the fault class of equivalent faults. When you do not specify this switch, the tool writes an “EQ” as the fault class for any equivalent faults. This switch is meaningful only when the Set Fault Mode command is set to Uncollapsed.

- **-Keep_aborted**

An optional switch that specifies for the tool to append an “a” to aborted fault classes when it writes the current fault list. For example, aborted UC, UO, and PT faults would be listed in the file as “uca”, “uoa”, and “pta”. When you later load these faults into the tool, it will treat them as aborted faults.

Examples

Example 1

The following example performs an ATPG run, then writes all the untestable faults to a file for review:

```
set fault type stuck
set system mode atpg
add faults -all
run
write faults faultlist -class ut
```

The content of the *faultlist* file will look similar to the following:

```
0      UU      /ip_scan_clk
1      UU      /ip_scan_clk
0      UU      /ipt_scan_tf_en
1      UU      /ipt_scan_tf_en
0      UU      /ff0/qb
1      UU      /ff0/qb
...
```

Example 2

The following example rewrites all the untestable faults to the file, but includes the name of the cell associated with each fault in the written information:

```
write faults faultlist -class ut -cell_name -replace
```

The file now contains an additional column with the cell information:

```
0      UU      /ip_scan_clk  primary_input
1      UU      /ip_scan_clk  primary_input
0      UU      /ipt_scan_tf_en primary_input
1      UU      /ipt_scan_tf_en primary_input
0      UU      /ff0/qb      dsffd
1      UU      /ff0/qb      dsffd
```

Related Topics

[Add Faults](#)

[Report Faults](#)

[Delete Faults](#)

[Set Fault Mode](#)

[Load Faults](#)

[Set Fault Sampling](#)

[Setup Checkpoint](#)

[Set Fault Type](#)

Write Initial States

Scope: All modes

Writes the initial state settings of design instances into the file that you specify.

Usage

WRItE INItial States *filename* [-Replace] [-All | *instance_name*...]

Description

The Write Initial States command writes different information regarding the initialization settings depending on the mode from which you issue the command. If FlexTest is in the Setup mode, the command writes the initialization settings that you created by using the Add Initial States command. If FlexTest is in any other mode, the command writes all the initial state settings (including those in any **test_setup** procedures).

Arguments

- *filename*
A required string that specifies the file where you want FlexTest to write the initialization settings.
- -Replace
An optional switch that replaces the contents of the file if the filename already exists.
- -All
An optional switch that writes the initialization settings for all design hierarchical instances. This is the default.
- *instance_name*
An optional repeatable string that specifies the name of a design hierarchical instance for which you want to write the initialization setting.

Examples

The following example assumes you are not in Setup mode and writes all the current initial settings:

```
add initial states 0 /amm/g30/ff0
set system mode atpg
write initial states /user/design/initialstate_file -all
```

Related Topics

[Add Initial States](#)

[Report Initial States](#)

[Delete Initial States](#)

Write Loops

Scope: Atpg, Fault, and Good modes

Writes a list of all the current loops to a file.

Usage

WRItE LOPs *filename* [-Replace]

Description

The Write Loops command writes all feedback loops in the circuit to a file. For each loop, the file contents show whether the loop was broken by duplication. The file shows loops unbroken by duplication as being broken by a constant value, which means the loop is either a coupling loop or has a single, multiple-fanout gate. The report also includes the pin pathname and gate type of each gate in each loop.

Arguments

- *filename*
A required string that specifies the name of the file to which you want to write the loops.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.

Examples

The following example writes a list of all the loops in the circuit to a file:

```
set system mode atpg
write loops loop.info -replace
```

Related Topics

[Report Loops](#)

Write Modelfile

Scope: Atpg, Fault, and Good modes

Writes all internal states for a RAM or ROM gate into the file that you specify.

Usage

WRItE MOdelfile *filename* *RAM/ROM_instance_name* [-Replace]

Description

The Write Modelfile command writes, in the proprietary Siemens EDA modelfile format, all the internal states for a RAM or ROM gate into a file.

The ROM internal states are identical to the initial values.

Arguments

- *filename*
A required string that specifies the name of the file to which you want to write the internal states for the RAM or ROM gate. The information is written in the proprietary Siemens EDA modelfile format.
- *RAM/ROM_instance_name*
A required string that specifies the instance name of the RAM or ROM gate whose internal states you want to write. The command reports an error condition if the instance contains multiple RAM/ROM gates.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.

Examples

The following example writes all the internal states of a RAM gate into a file for review:

```
add write controls 0 w1
set system mode atpg
add faults -all
run
write modelfile model.ram /p1.ram
```

Related Topics

[Read Modelfile](#)

Write Netlist

Scope: All modes

Writes the current design in the Verilog netlist format to the specified file.

Usage

WRItE NETlist *filename* [-Replace]
[-User_setup] [-POsitional | -MODEL_SOURCE]
[-Block *module_name...* {-FULL | -CHildren} -File *block_filename*]

Description

It writes the netlist that was read into the system when you invoked the tool. Use the -Block switch to write specific modules to a separate file.

Arguments

- *filename*
A required string that specifies the name of the file to which the tool writes the netlist.
- -Replace
An optional switch that replaces the contents of the file if the file already exists.
- -User_setup
An optional switch that specifies to write the netlist for the design based on the current state of the design in the tool, with respect to the Add Black Box and Delete Black Box commands.
- -POsitional
An optional switch that specifies to make connections between signals in module instantiations and ports in module definitions based on position. If neither the -Positional nor -Model_source switch is specified, these connections are made by name.

Note



This switch is only valid if you have invoked the tool using the -Model invocation switch. For more information, see the [flectest](#) shell command.

- -MODEL_SOURCE
An optional switch that is only valid if you have invoked the tool using the -Model invocation switch; it specifies to use the following **model_source** statements in the defining ATPG library models to determine whether pinnames or positional connections are used to write an instance portlist of a model:

```
model_source = verilog_module;  
model_source = verilog_udp;  
model_source = verilog_unnamed_port_module  
model_source = verilog_parameter_override;
```

Modules are instantiated using pinnames; UDPs are instantiated positionally as required to produce legal Verilog for a UDP instance. If an ATPG model has no **model_source** statement, the model is instantiated positionally.

Note



This switch is only valid if you have invoked the tool using the -Model invocation switch. For more information, see the [flextest](#) shell command.

For more information, see [Set Model Source](#) in the *Tessent Cell Library Manual*.

- -Block *module_name...* {-FULL | -CHildren} -File *block_filename*

An optional switch set that writes specified modules to a separate file. Module definitions redirected to the separate file are not written to the file specified by the *filename* argument.

All of the following arguments are required for the -Block switch.

module_name...

A repeatable string that specifies the names of the modules you want written to the separate file (*block_filename*).

-FULL

A switch that writes the full netlist for the specified *module_names* to the separate file, including any children modules.

-CHildren

A switch that writes only the children of the specified *module_names* to the separate file.

-File *block_filename*

A switch and string pair that specifies the file to which the separate netlists are written.

Examples

The following example writes the invoked design to a Verilog netlist:

```
write netlist design_out.v
```

Related Topics

[Add Black Box](#)

[Delete Black Box](#)

Write Primary Inputs

Scope: All modes

Writes the primary inputs to the specified file.

Usage

WRItE PRImary Inputs *filename* [-Replace] [-All | *net_pathname...* | *primary_input_pin...*] [-Class {Full | User | System}]

Description

The Write Primary Inputs command writes a list of the primary inputs into a file where you can review it. You can choose to write either the user class, system class, or full classes of primary inputs. Additionally, you can write all the primary inputs or a specific list of primary inputs. If you issue the command without specifying any arguments other than *filename*, then the tool writes all the primary inputs.

This command is identical to the Report Primary Inputs command except the data is written into a file.

Arguments

- *filename*
A required string that specifies the name of the file to which you want to write the list of primary inputs.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.
- -All
An optional switch that writes all the primary inputs. This is the default.
- *net_pathname*
An optional repeatable string that specifies the circuit connections whose user class of primary inputs you want to write.
- *primary_input_pin*
An optional repeatable string that specifies a list of system class primary input pins that you want to write.
- -Class Full | User | System
An optional switch and literal pair that specifies the source (or class) of the primary input pins that you want to write. The literal choices are as follows:
 - Full — A literal that writes all the primary input pins in the user and system classes. This is the default.
 - User — A literal that writes only the user-entered primary input pins.

System — A literal that writes only the netlist-described primary input pins.

Examples

The following example writes all primary inputs in both the user and system classes to a file:

```
add primary inputs net_100 net_200  
write primary inputs inputfile -class full
```

Related Topics

[Add Primary Inputs](#)

[Report Primary Inputs](#)

[Delete Primary Inputs](#)

Write Primary Outputs

Scope: All modes

Writes the primary outputs to the specified file.

Usage

WRItE PRImary Outputs *filename* [-Replace] [-All | *net_pathname...* | *primary_output_pin...*]
[-Class {Full | User | System}]

Description

The Write Primary Outputs command writes a list of the primary outputs into a file where you can review it. You can choose to write either the user class, system class, or full classes of primary outputs. Additionally, you can write all the primary outputs or a specific list of primary outputs. If you issue the command without specifying any arguments other than *filename*, the tool writes all the primary outputs.

This command is identical to the Report Primary Outputs command except the data is written into a file.

Arguments

- *filename*
A required string that specifies the name of the file to which you want to write the list of primary outputs.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists. By default, existing data is not overwritten.
- -All
An optional switch that writes all the primary outputs. This is the default.
- *net_pathname*
An optional repeatable string that specifies the circuit connections whose user class of primary outputs you want to write.
- *primary_input_pin*
An optional repeatable string that specifies a list of system class primary output pins that you want to write.
- -Class Full | User | System
An optional switch and literal pair that specifies the source (class) of the primary output pins which you want to write. The literal choices are as follows:
 - Full — A literal that writes all the primary output pins in the user and system classes. This is the default.
 - User — A literal that writes only the user-entered primary output pins.

System — A literal that writes only the netlist-described primary output pins.

Examples

The following example writes all primary outputs in both the user and system classes to a file:

```
add primary outputs net_300 net_400 net_500  
write primary outputs outputfile -class full
```

Related Topics

[Add Primary Outputs](#)

[Report Primary Outputs](#)

[Delete Primary Outputs](#)

Write Procfile

Scope: All modes except Setup mode

Writes existing procedure and timing data to the named test procedure file.

Usage

WRItE PRocfile *proc_filename* [-Replace] [-Full [-EXternal]]

Description

The Write Procfile command writes out existing procedure and timing data to the named test procedure file.

Arguments

- *proc_filename*
A required string that specifies the name of the file to which you want to write existing procedure and timing data.
- -Replace
An optional switch that replaces the contents of the file if the *proc_filename* already exists.
- -Full
An optional switch that causes the tool to parse the ATPG pattern list (if any) and create all needed non-scan procedures before writing the procedure file data. This option will always create a sequential procedure.

Note



The -Full option can cause the Write Procfile command to take more time if there are a large number of ATPG-generated patterns in the internal pattern list.

- -EXternal
An optional switch that causes the -Full switch to use the external patterns specified by the Set Pattern Source External command instead of internally generated patterns. This switch behavior is similar to the -External switch behavior for the Save Patterns command. The -External switch is only valid when using the -Full switch.

Examples

The following example writes the existing procedure and timing data to the specified file:

```
write procfile myprocfile.proc
```

Related Topics

[Add Scan Groups](#)

[Report Timeplate](#)

[Read Procfile](#)

[Save Patterns](#)

[Report Procedure](#)

Write Statistics

Scope: All modes

Writes the current simulation statistics to the specified file.

Usage

WRItE STatistics *filename* [-Replace]

Description

The Write Statistics command writes a detailed statistics report to the file that you specify. The statistics report lists the following four groups of information:

- Circuit Statistics which consists of total numbers for the following:
 - primary inputs
 - primary outputs
 - library model instances
 - netlist primitive instances
 - combinational gates
 - sequential elements
 - simulation primitives
 - scan cells
 - scan sequential elements
 - sequential instances
 - defined nonscan instances
 - nonscan instances identified by the DRC
 - defined scan instances
 - scan instances identified by the DRC
 - identified scan instances
- Fault List Statistics which consists of:
 - The number of collapsed and total faults that are currently in each class. FlexTest does not write fault classes with no members.
 - The percentage of test coverage, fault coverage, and ATPG effectiveness for both collapsed and total faults

- Test Patterns Statistics which lists the total numbers for the following:
 - total patterns currently in the test pattern set
 - total number of patterns simulated in the preceding simulation process
- Runtime Statistics which lists the following:
 - Machine and user names
 - total user CPU time
 - total system CPU time
 - total memory usage

Arguments

- *filename*
A required string that specifies the name of the file to which you want to write the statistics report.
- -Replace
An optional switch that replaces the contents of the file if the *filename* already exists.

Examples

The following example writes the current statistics data to the specified file:

write statistics /user/designs/statfile

The following listing shows the contents of the example file:

```
Total number of sequential instances    = 2

*****Circuit Statistics*****
# of primary inputs = 12
# of primary outputs = 6
# of library model instances = 14
# of combinational gates = 12
# of sequential elements = 2
# of simulation primitives = 62

# of scan cells = 2
# of scan sequential elements = 2
*****Fault List Statistics*****
Fault Class          Uncollapsed   Collapsed
Full (FU)             120           56
Det_simulation (DS)    72           28
Det_implication (DI)   48           28
Fault coverage         100.00%       100.00%
Test coverage          100.00%       100.00%
Atpg effectiveness     100.00%       100.00%
*****Test Patterns Statistics*****
Total Test Cycles Generated = 26
Total Scan Operations Generated = 13
Total Test Cycles Simulated = 26
Total Scan Operations Simulated = 13
***** Runtime Statistics *****
Machine Name      : machine1
User Name         : user1
User CPU Time     : 1.9 seconds
System CPU Time   : .6 seconds
Memory Used       : 2.137M
```

Related Topics

[Report Statistics](#)

Shell Command Description

The notational conventions used to describe the shell commands are the same as those used in other parts of the manual. Do not enter any of the special notational characters (such as, {}, [], or |) when typing the command.

For a complete description of the notational conventions used in this manual, refer to [“Command Line Syntax Conventions”](#) on page 106.

The following table contains a summary of the shell commands described in this chapter.

Table 2-15. Shell Command Summary

Command	Description
flextest	Invokes FlexTest on a specified design ready to set up and generate test patterns.

flextest

Prerequisites: A gate-level Verilog netlist.

Invokes FlexTest on a specified design ready to set up and generate test patterns.

Usage

```
flextest {{ {design_name -VERILOG } | {-MODEL {cell_name | ALL}} } {-LIBrary
  {library_name...} } [-INCDIR include_directory...]
[-INSENSitive | -SENSitive] [-LOGfile filename] [-REPlace] [-FaultSIM]
[-TOP module_name] [-DOFile dofile_name [-HIStory]]
[-LICense_wait {minutes | NONE | UNLimited}] [-32 | -64]
[-LOAD_warnings]} | {[-HELP | -USAGE | -MANUAL | -VERSION]}
[-LIBRARY_ARRAY_DELIMITER {square | angle}]
```

Arguments

- *design_name* -VERILOG

A required repeatable string and optional switch that specifies the pathname and format of the design netlist.

-VERILOG

An optional switch that specifies that *design_name* is a netlist in Verilog format. This is the default.

- -MODEL {*cell_name* | ALL}

An optional switch and string or literal pair that invokes FlexTest on a cell model or library of cell models. This is useful for verifying a library.

- -LIBrary {*library_name*...}

A required switch and repeatable string pair that specifies the names of the files containing the ATPG library descriptions for all cell models in *design_name*. You can use wildcards to specify multiple library files.

- -INCDIR *include_directory*

An optional switch and repeatable string that specifies directories to search for files included in the Verilog design with the 'include compiler directive. This enables you to use simple filenames in 'include directives. Each *include_directory* must be relative to the current tool invocation directory or an absolute directory pathname.

Include files are searched for in the following order of precedence:

- Absolute pathnames specified by 'include directives in the Verilog design.
- Directories specified with the -Incdir invocation switch.

For relative 'include file targets, directories are searched in the order they are listed, left to right, in the command. If identically named files exist in multiple directories, the first file found is used and the others are ignored.

A search path may contain the asterisk (*) and other wildcard characters supported by the invocation shell. For example, if you specify “-incdir sp*”, every directory within the current directory that starts with “sp” is used as a search path.

- c. Directory where the Verilog design file is located
- d. Directory FlexTest is invoked from (current directory)

- **-INSENSitive | -SENSitive**

Optional switch that determines whether FlexTest treats pin, instance, and net pathnames within the design netlist as case-insensitive or case-sensitive. The default case-sensitive for Verilog and case-insensitive for flat netlists. Commands are always case insensitive.

- **-LOGfile *filename***

An optional switch and string pair that specifies the name of the file to write all session transcript information to. By default, session information displays on the screen only. The file begins with a version banner that includes: the tool name and version, the date, and the tool platform.

- **-REPlace**

An optional switch that overwrites the -Logfile *filename* if one by the same name already exists.

- **-FaultSIM**

An optional switch that invokes the FlexTest fault simulator and restricts you from entering the ATPG system mode.

- **-TOP *module_name***

An optional switch and string pair that specifies the name of the top-level module in the netlist. This is a required switch when multiple top modules exist in the design. If you don't use this switch and the tool detects multiple top modules, the tool issues an error message that lists the top modules..

- **-DOFile *dofile_name***

An optional switch and string pair that specifies the name of a dofile to execute upon invocation.

- **-HIStory**

An optional switch that adds the commands from a specified dofile to the command line history list. This switch is ignored if no dofile is specified with the -Dofile switch.

- **-LICense_wait *minutes* | NONE | UNLimited**

An optional switch and integer, or switch and literal, that specifies the tool's response if the license is unavailable.

Choose one of the following options:

minutes — An positive integer that specifies the number of minutes to wait for the license.

NONE — A literal that directs the Tessent tool to exit immediately if no license is available.

UNlimited — A literal that directs the Tessent tool to wait with no time limit for a license. This is the default.

- **-32 | -64**

An optional switch that invokes the 32-bit or 64-bit version of the software. The default is 64-bit. If the platform does not support the specified version, the tool gives a warning message and ignores the switch.

- **-LOAD_warnings**

An optional switch that reports the warnings and notes discovered while loading the netlist and library. By default, a summary message for some warnings and notes is reported.

- **-HELP**

An optional switch that displays each FlexTest invocation switch with a brief description.

- **-USAGE**

An optional switch that lists the FlexTest invocation switches.

- **-MANUAL**

An optional switch that opens the Tessent DFT Documentation System.

- **-VERSION**

An optional switch that displays the FlexTest software version.

- **-LIBRARY_ARRAY_DELIMITER {*square* | *angle*}**

An optional switch that sets the default array delimiter for library parsing to either square brackets “[]” or angle brackets “< >”. The default is “[]”.

Examples

Example 1

The following example invokes FlexTest in command line mode on a Verilog netlist named *design1.v* and two library files called *lib10* and *lib20*. FlexTest keeps a session log in a file called *design1_atpg.log*, replacing the contents of the file if it already exists:

```
shell> <mgcdft tree>/bin/flextest design1.v -lib mitsu_lib10 \
      mitsu_lib20 -log design1_atpg.log -replace
```

Example 2

The following example is like the preceding example except that multiple libraries (all libraries ending in “.lib” in the *libs* directory) are specified using the asterisk (*) wildcard character:

```
shell> <mgcdft tree>/bin/flextest design1.v -lib libs/*.lib \
      -log design1_atpg.log -replace
```


Appendix A

Test Pattern File Formats

FlexTest can read in two types of pattern formats: ASCII and table. This section describes the contents of both formats.

ASCII Pattern Format.....	488
Table Pattern Format	495
VCD Support Using VCD Plus.....	502

ASCII Pattern Format

The ASCII file that describes the test patterns is divided into four sections, which are named `setup_data`, `functional_chain_test`, `test_data`, and `scan_cell`. Each section begins with a section name statement and finishes with an end statement. The format of the data contained in each section is described as follows. Also in this file, any line starting with a double slash (//) is a comment line.

Setup_Data	488
Functional_Chain_Test	491
Test_Data	492
Scan_Cell	493
Example Circuit	495

Setup_Data

The `setup_data` section contains the definition of the test cycle width, the primary input bus, and the primary output bus that will be referenced in the description of the test patterns. This section will also contain any scan chain information, if there are any scan chains defined in the circuit.

The data will be in the following format:

```
SETUP =  
    <setup information>  
END;
```

The setup information may include the following:

```
TEST_CYCLE_WIDTH = <integer>;
```

This defines the width of the test cycle that specifies the number of time units in each test cycle for forcing and/or measuring values at specific time units.

```
DECLARE INPUT BUS "ibus_name" = <ordered list of primary inputs>;
```

This optional statement groups several primary inputs into one bus name. Each primary input will be enclosed in double quotes and be separated by commas. For bidirectional pins, they will be placed in both the input and output bus.

```
DECLARE OUTPUT BUS "obus_name" = <ordered list of primary outputs>;
```

This optional statement groups several primary outputs into one bus name. Each primary output will be enclosed in double quotes and be separated by commas.

If the circuit has scan operation defined, the scan related information will also be described here. The type of information includes clock information, test_setup information, and scan group information.

The clock information is as follows:

```
CLOCK "clock_name1" =  
    OFF_STATE = <off_state_value>;  
END;  
CLOCK "clock_name2" =  
    OFF_STATE = <off_state_value>;  
END;  
....  
....
```

This defines the list of clocks that are contained in the circuit, and used with the scan operation. The clock data will include the clock name enclosed in double quotes, and the off-state value. For edge-triggered scan cells, the off-state is the value that places the initial state of the capturing transition at the clock input of the scan cell. The clock information must be consistent with the Add Clocks command used in the Setup system mode.

The test_setup information is as follows:

```
PROCEDURE TEST_SETUP "test_setup" =  
    FORCE "primary_input_name1" <value> <time>;  
    FORCE "primary_input_name2" <value> <time>;  
    ....  
    ....  
END;
```

This procedure must be identical to the test_setup procedure in the Test Procedure file. This procedure is used to set non-scan memory elements to a constant state for both ATPG and the load/unload process. It is applied once at the beginning of the test pattern set. This procedure may only include force commands.

The scan group information is as follows:

```
SCAN_GROUP "scan_group_name1" =  
    <scan_group_information>  
END;  
SCAN_GROUP "scan_group_name2" =  
    <scan_group_information>  
END;
```

This defines each scan group that is contained in the circuit. A scan chain group is a set of scan chains that are loaded and unloaded in parallel. The scan group name will be enclosed in double quotes and each scan group will have its own independent scan group section. Within a scan group section, there is information associated with that scan group, such as scan chain

definitions and procedures. Any scan groups listed in the test pattern file must be defined with Add Scan Groups command.

```
SCAN_CHAIN "scan_chain_name1" =  
    SCAN_IN = <scan_in_pin>;  
    SCAN_OUT = <scan_out_pin>;  
    LENGTH = <length_of_scan_chain>;  
END;  
SCAN_CHAIN "scan_chain_name2" =  
    SCAN_IN = <scan_in_pin>;  
    SCAN_OUT = <scan_out_pin>;  
    LENGTH = <length_of_scan_chain>;  
END;
```

The scan chain definition defines the data associated with a scan chain in the circuit. If there are multiple scan chains within one scan group, each scan chain will have its own independent scan chain definition. The scan chain name will be enclosed in double quotes. The scan-in pin will be the name of the primary input scan-in pin enclosed in double quotes. The scan-out pin will be the name of the primary output scan-out pin enclosed in double quotes. The length of the scan chain will be the number of scan cells in the scan chain. Any scan chains listed in the test pattern file must be defined with the Add Scan Chains command.

```
PROCEDURE <procedure_type> "scan_group_procedure_name" = <list of events>  
END;
```

The type of procedures in each scan group may include shift procedure, load and unload procedure, shadow-control procedure, master-observe procedure, and shadow-observe procedure. These procedures should be exactly the same as the test procedure file. The list of events of each procedure may be any combination of the following commands:

```
FORCE "primary_input_pin" <value> <time>;
```

This command is used to force a value (0,1, X, or Z) on a selected primary input pin at a given time. The time values must not be lower than previous time values for this command. The primary input pin will be enclosed in double quotes.

```
APPLY "scan_group_procedure_name" <#times> <time>;
```

This command indicates the selected procedure name is to be applied the selected number of times beginning at the selected time. The scan group procedure name will be enclosed in double quotes. This command may only be used with the load and unload procedures.

```
FORCE_SCI "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be placed on the scan chain inputs. The scan chain name will be enclosed in double quotes.

```
MEASURE_SCO "scan_chain_name" <time>;
```

This command indicates the time in the shift procedure that values are to be measured on the scan chain outputs. The scan chain name will be enclosed in double quotes.

Functional_Chain_Test

If the circuit has scan operation defined for the test pattern set generated by FlexTest, a functional chain test pattern will be included. However, if the test patterns are created by the user, this section is optional. The purpose of the test is to verify the operation of the scan circuitry before it is used to test the other circuitry. For each scan chain group, the functional chain test will simply load a series of zeros and ones into the scan chains and then unload them to verify the operation of the scan circuitry.

The format is as follows:

```
CHAIN_TEST =  
  APPLY "test_setup" <value> <time>;  
  <scan cycles>  
END;
```

The optional “test_setup” line is applied at the beginning of the functional chain test pattern if there is a test_setup procedure in the Setup_Data section.

The scan cycles will include multiple cycles of the following:

```
SCAN = <number>;  
APPLY "scan_group_unload_name1" <time> =  
  CHAIN "scan_chain_name1" = "values...";  
  CHAIN "scan_chain_name2" = "values...";  
  ....  
END;  
APPLY "scan_group_load_name1" <time> =  
  CHAIN "scan_chain_name1" = "values...";  
  CHAIN "scan_chain_name2" = "values...";  
  ....  
END;  
APPLY "scan_group_unload_name2" <time> =  
  CHAIN "scan_chain_name1" = "values...";  
  CHAIN "scan_chain_name2" = "values...";  
  ....  
END;  
APPLY "scan_group_load_name2" <time> =  
  CHAIN "scan_chain_name1" = "values...";  
  CHAIN "scan_chain_name2" = "values...";  
  ....  
END;
```

The number for the scan is the sequence where a functional scan chain test for all scan chains in the circuit is to be tested. The scan group load and unload name and the scan chain name will be enclosed in double quotes. Since loading and unloading of a scan chain happens at the same time for each scan group, its loading and unloading will have the same time value in each scan cycle. The order of the values to load and unload the scan chain will be in the order the values are shifted through the scan chain, and will be enclosed in double quotes.

Test_Data

The test_data section contains all the test patterns for the target faults.

The format of the test_data section is as follows:

```
CYCLE_TEST =  
  APPLY "test_setup" <value> <time>;  
  <cycles>  
END;
```

The optional “test_setup” line is applied at the beginning, if there is a test_setup procedure in the Setup_Data section.

All test patterns are grouped into cycles. There are two kinds of cycles. One is the normal test cycle. The other is the scan cycle, if any scan operation is used in the circuit. A scan cycle specifies all the values that are unloaded and loaded onto all the defined scan chains. The format of a scan cycle is the same as that used in the functional_chain_test. All cycle patterns have to have correct timing order.

A normal test cycle specifies the values that should be applied at the primary inputs and be expected at the primary outputs. A normal test cycle will include the following:

```
CYCLE = <number>;  
  <list of events>;  
END;
```

An event in a normal test cycle can be a force event, or measure event. All events have to have correct timing order, as defined by the Add Pin Constraint and Add Pin Strobes commands.

The format of a force event is as follows.

```
FORCE "ibus_name" "primary_input_values" <time>;
```

A force event is used to force values on the selected primary input pins at the given time unit within the specified test cycle. The name is either a primary input name or a input bus name defined in Setup_Data part, and is enclosed in double quotes. The values will also be enclosed in double quotes. If a bus name is used, the values will be in a one-to-one correspondence with the order of the specified primary inputs in Setup_Data part.

The format of a measure event is as follows.

```
MEASURE "obus_name" "primary_output_values" <time>;
```

A measure event is used to measure the value of the selected primary output pins at the given time unit within the specified test cycle. The name is either a primary output name or an output bus name defined in Setup_Data part, and is enclosed in double quotes. The values will also be enclosed in double quotes. If a bus name is used, the values will be in a one-to-one correspondence with the order of the specified primary outputs in Setup_Data part.

If the test set contains patterns for IDDQ testing, an additional measure event will be listed for IDDQ patterns. The format of the IDDQ measure event is as follows.

```
MEASURE IDDQ ALL <time>:
```

Scan_Cell

The scan_cell section contains the definition of the scan cells used in the circuit.

The scan cell data will be in the following format:

```
SCAN_CELLS =
  SCAN_GROUP "group_name1" =
    SCAN_CHAIN "chain_name1" =
      SCAN_CELL = <cellid> <type> <sciinv> <scoinv> <relsciinv>
        <relscoinv> <instance_name> <model_name>
        <input_pin> <output_pin>;
      ....
    END;
  SCAN_CHAIN "chain_name2" =
    SCAN_CELL = <cellid> <type> <sciinv> <scoinv> <relsciinv>
      <relscoinv> <instance_name> <model_name>
      <input_pin> <output_pin>;
    ....
  END;
  ....
END;
SCAN_GROUP "group_name2" =
  SCAN_CHAIN "chain_name1" =
    SCAN_CELL = <cellid> <type> <sciinv> <scoinv> <relsciinv>
      <relscoinv> <instance_name> <model_name>
      <input_pin> <output_pin>;
    ....
  END;
  SCAN_CHAIN "chain_name2" =
    SCAN_CELL = <cellid> <type> <sciinv> <scoinv> <relsciinv>
      <relscoinv> <instance_name> <model_name>
      <input_pin> <output_pin>;
    ....
  END;
  ....
END;
```

The fields for the scan cell memory elements are the following:

cellid - A number that identifies the position of the scan cell in the scan chain. The number 0 indicates the scan cell closest to the scan-out pin.

type - The type of scan memory element. The type may be MASTER, SLAVE, SHADOW, OBS_SHADOW, COPY, or EXTRA.

sciinv - Inversion of the library input pin of the scan cell relative to the scan chain input pin. The value may be T (inversion) or F (no inversion).

scoinv - Inversion of the library output pin of the scan cell relative to the scan chain output pin. The value may be T (inversion) or F (no inversion).

relsciinv - Inversion of the memory element relative to the library input pin of the scan cell. The value may be T (inversion) or F (no inversion).

relscoinv - Inversion of the memory element relative to the library output pin of the scan cell. The value may be T (inversion) or F (no inversion).

instance_name - The top level boundary instance name of the memory element in the scan cell.

model_name - The internal instance pathname of the memory element in the scan cell (if used - blank otherwise).

input_pin - The library input pin of the scan cell (if it exists, blank otherwise).

output_pin - The library output pin of the scan cell (if it exists, blank otherwise).

Example Circuit

See the following figure for an example that illustrates the scan cell elements in a typical scan circuit.

Figure A-1. Example Scan Circuit

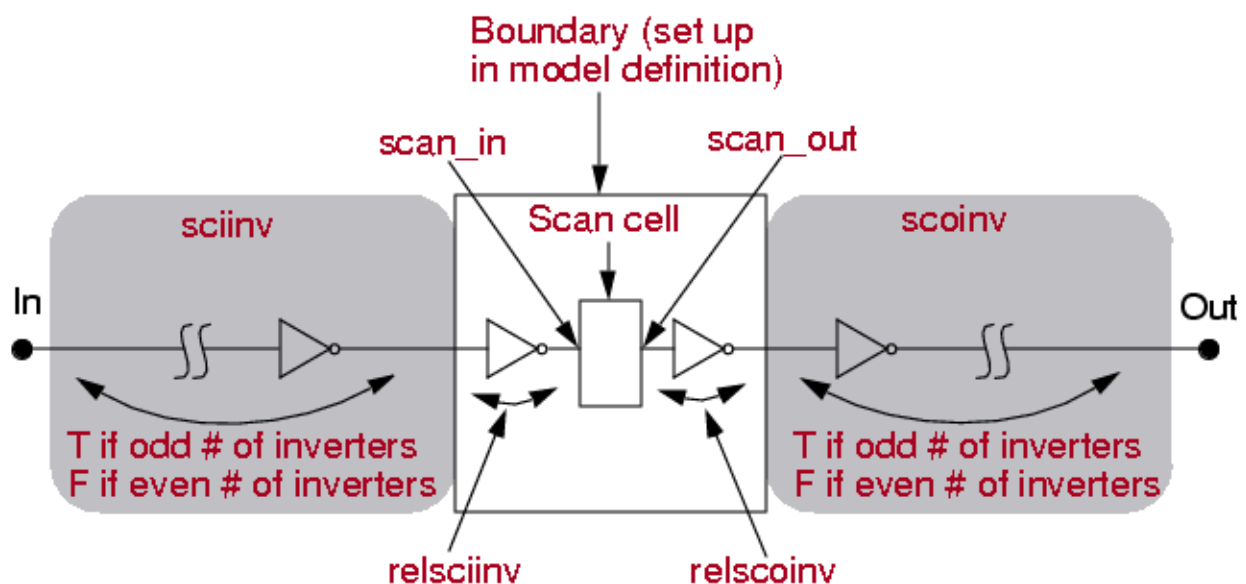


Table Pattern Format

The -Table option from the Set Pattern Source command specifies that the input test vectors are in a table pattern format. This format currently cannot accept scan patterns. The table pattern file contains two sections: control section and data section. The control section defines the pin order. The data section contains the test vectors, where each line corresponds to one test cycle.

Here is an example of the test patterns in the table format:

```
// TABLE FORMAT PATTERNS
PI CLOCK
PI G3
PI G2
PI G1
PI G0
PO G17

P01111
P00101
P00101
P10011
P10010
P10000
P00100
P01111
P10001
P10010
P00011
P00101
```

If any lines start with a double slash (//), it will be treated as a comment and ignored. In order to distinguish between the control and data sections, a blank line must separate the two sections.

Data Section

This section contains the test patterns, where each line corresponds to one test cycle. Each column corresponds to each pin name and the order is defined in the control section. The total number of columns must equal the total number of lines in the control section. If a line is too long, “\” is used to break the line. The following values are the only ones that can be used in the data section and are case insensitive:

P, N, 0, 1, H, L, Z, X

The value H is treated as 1, and the value L is treated as 0. For SR0 and R0 pin constraints, P and 1 represent a positive pulse present. For SR1 and R1 pin constraints, N and 0 represent a negative pulse present. The values P and N cannot be used for pins that have the NR constraint format.

Control Section

This section defines the pin order, where each line defines one primary pin. The format for each line is as follows:

```
<type> <pin_name>  
or  
<type> <pin_name> <control_name>  
(for styles 1a and 1b only)
```

The types and pin names are case insensitive. The pin name refers to the primary input, output, or bidirectional name. The type can be PI for primary input, PO for primary output, or UU for unused. For bidirectional pins, some styles require two lines, and there are four styles that can be used:

Style #1a: (two columns required)

The types are IO_DA and IO_C1. If IO_C1 is 0, then IO_DA is the input value (force value). If IO_C1 is 1, then IO_DA is the output value (measure value). IO_C1 must be either a 0 or 1. Multiple IO_DA can share one IO_C1. The pin name of IO_DA must be the bidirectional pin name.

If the format IO_DA <pin_name> is used, then its IO_C1 should use the same pin name as IO_DA.

If the format IO_DA <pin_name> <control_name> is used, then its IO_C1 should use the same control name as IO_DA.

Style #1b: (two columns required)

The types are IO_DA and IO_C0. If IO_C0 is 1, then IO_DA is the input value (force value). If IO_C0 is 0, then IO_DA is the output value (measure value). IO_C0 must be either a 0 or 1. Multiple IO_DA can share one IO_C0. The pin name of IO_DA must be the bidirectional pin name.

If the format IO_DA <pin_name> is used, then its IO_C1 should use the same pin name as IO_DA.

If the format IO_DA <pin_name> <control_name> is used, then its IO_C1 should use the same control name as IO_DA.

Style #2: (two columns required)

The types are IO_PI and IO_PO. IO_PI is the input value and IO_PO is the output value. Both the pin names must be the same as the bidirectional pin name.

Style #3: (only one column required)

The type is IO_HL. If the value is H, L, or Z, it is the output value. Otherwise, it is the input value. The following describes the behavior of each symbol:

- 0 = driving 0, measure X.
- 1 = driving 1, measure X.
- X = driving X, measure X.
- Z = driving Z, measure X.
- H = driving Z, measure 1.
- L = driving Z, measure 0.

Style #4: (only one column required)

The type is IO_10. IO_10 is used for a primary bidirectional pin. If the value is 0, 1, or Z, it is the output value. If the value is H, L, or X, it is the input value. The IO_10 format is opposite to the IO_HL format.

- L = driving 0, measure X.
- H = driving 1, measure X.
- X = driving X, measure X.
- Z = driving Z, measure X.
- 1 = driving Z, measure 1.
- 0 = driving Z, measure 0.

The waveform shape of each input pin is defined by the Add Pin Constraint command. If the waveform type for a input pin is specified as C0, C1, CX, CZ, RO, or R1, then it is optional to list that input pin and type.

To create a table format for non-scan test patterns that were generated in the ASCII format, we do the following:

Here is the ASCII test patterns for a non-scan circuit, with a test cycle width set to 2, and a primary input CLOCK constrained to R0 with period 1, offset 0, and width 1:

```

SETUP =
    TEST_CYCLE_WIDTH = 2;
    DECLARE INPUT BUS "ibus" = "/clock", "/G3", "/G2", "/G1",
        "/G0";
    DECLARE OUTPUTBUS "obus_1" = "/G17";
END;
CYCLE_TEST =
    CYCLE = 0;
    FORCE "ibus" "10111" 0;
    FORCE "ibus" "00111" 1;
    MEASURE "obus_1" "1" 2;
    CYCLE = 1;
    FORCE "ibus" "10010" 0;
    FORCE "ibus" "00010" 1;
    MEASURE "obus_1" "1" 2;
    CYCLE = 2;
    FORCE "ibus" "10010" 0;
    FORCE "ibus" "00010" 1;
    MEASURE "obus_1" "1" 2;
    CYCLE = 3;
    FORCE "ibus" "11001" 0;
    FORCE "ibus" "01001" 1;
    MEASURE "obus_1" "1" 2;
    ....
END;

```

To create the control section, all the primary inputs and outputs should be listed in the same order as the ASCII patterns as follows:

```

PI CLOCK
PI G3
PI G2
PI G1
PI G0
PO G17

```

To create the data section, each line will correspond to one test cycle. For cycle = 0, we have:

```

FORCE "ibus" "10111" 0;
FORCE "ibus" "00111" 1;
MEASURE "obus_1" "1" 2;

```

Since CLOCK changes from 1 to 0 during the force of the input values, we assign P to correspond to a positive pulse present during the test cycle. For the rest of the input pins the

values do not change, it remains the same. We use the same measure value for the output pin. Thus, we have the first line in the data section:

```
P01111
```

Looking at cycle=1 and the rest of the test cycles, we see that only CLOCK changes values during the force of the input values and the rest remains the same. This is because we specified CLOCK to have a pin constraint of R0. Thus, for the rest of the lines in the data section, we have the following:

```
P00101  
P00101  
P10011
```

Since, we have a pin constraint for CLOCK (R0), we could have left out CLOCK from the control section and the first column (P) in the data section. This is because the system already knows that CLOCK will have a positive pulse every test cycle specified with the Add Pin Constraint command.

To create a table format that contains bidirectional pins, we do the following:

Here is an ASCII patterns for a non-scan circuit that contains two, tri-state devices (The outputs of the tri-state are bidirectional pins and both enable lines are connected together):

```
SETUP =  
  DECLARE INPUT BUS "ibus" = "/IA", "/IB", "/E", "/IOA",  
  "/IOB";  
  DECLARE OUTPUT BUS "obus_1" = "/IOA", "IOB";  
END;  
CYCLE_TEST =  
  CYCLE = 0;  
  FORCE "ibus" "100ZZ" 0;  
  MEASURE "obus_1" "ZZ" 1;  
  CYCLE = 1;  
  FORCE "ibus" "001ZZ" 0;  
  MEASURE "obus_1" "00" 1;  
  CYCLE = 2;  
  FORCE "ibus" "111ZZ" 0;  
  MEASURE "obus_1" "11" 1;  
END;
```

To use style 1a, the user needs to know if the control line is a 1, then the data will be the output value. To use style 1b, the user need to know if the control line is 0, then the data will be the output value. Let's assume it is style 1a.

To create the control section, all the primary inputs and primary outputs, as well as the bidirectional pins with its control pin, should be listed:

```
PI  IA
PI  IB
PI  E
IO_C1 G1
IO_DA IOA G1
IO_DA IOB G1
```

G1 is the control name given to IO_C1. Since the enable line of the tri-states are connected together, IOA and IOB share the same control name. To create the data section, each line will correspond to one test cycle. Thus, we get the following:

```
1000ZZ
001100
111111
```

When IO_C1 is 1, then IOA and IOB will be the output values. Conversely, when IO_C1 is 0, then IOA and IOB will be the input values.

VCD Support Using VCD Plus

FlexTest accepts existing Verilog functional patterns through its VCD (Value Change Dump) Plus files which can be generated during simulation. This functionality is useful because FlexTest can use existing functional patterns, tracing only external pins, to get some initial fault coverage. FlexTest can then perform ATPG on the remaining faults. This can result in smaller test pattern sets and shorter run times. Also, due to that fact that the FlexTest fault simulation engine doesn't consider timing, FlexTest fault simulation should be faster than other fault simulators.

This feature involves FlexTest completing two steps:

- Parsing a VCD Plus file.
- Converting the event based patterns in the VCD file to cycle-based vectors stored in FlexTest internal pattern data structure, which can be used by the cycle-based FlexTest fault simulation engine to perform fault simulation.

The VCD Plus format supported is LSI Logic's extended VCD format. Unlike the standard VCD format, the extended VCD format provides sufficient simulation information on bidirectional signals, namely driving direction, driving strength and collision detection. For detailed information on LSI Logic's extended VCD format and the methods of generating it from various simulators, contact LSI Logic and the respective EDA vendor.

To create a VCD Plus file for a Verilog design from ModelSim EE/Plus (using version 5.1e or later), use the following ModelSim commands:

```
vcd dumpports -file filename.vcd <pathname-to-module>/*
```

The stimulus in a VCD file must be periodic to be used in the VCD Reader. You must define all pin waveforms using the [Setup Pin Constraints](#) or [Add Pin Constraints](#) commands in FlexTest's setup system mode before invoking the VCD Reader. You will also need to define waveforms for all primary input pins and "data_sample_time" times for all primary input and output pins in a separate control file. The data_sample_time should not be matched to the exact edge you expect the pin to change, but rather to a time when the state on the pin is stable.

For example, if your clock rises at an offset of 200 and has a width of 200, the data_sample_time should not be set to 200, but rather to 300, to make sure the proper state for the pin is captured. Missing the pulse of a clock will have a major effect on your fault grade. Likewise, on an output pin, the time should be specified when you know the pin has changed and the state is stable. This information is used to cycle events. The information provided in the control file must be consistent with the pin waveforms defined in FlexTest.

The VCD Reader can perform timing checks on the patterns in the VCD file, against the pin waveform specified in the control file. Any value change on a pin at a time which is not consistent with its offset or pulse width specified in the control file is dumped into a message file. You can use this information to modify the original test vector to make it periodic. By

default, the VCD Reader doesn't perform the timing checking. You can turn this checking on by using the Set Timing Checking command in the control file.

The converted cycle vectors can be saved in all vector formats that are supported in FlexTest by using the save pattern command with external option.

The [Set Pattern Source](#) command supports the VCD Reader.

SET PAttern Source **Internal** | { {**External filename**} [-Ascii | -Table| -Vcd] [-Control *control_filename*] [-NOPadding]}

The above usage is for FlexTest only.

VCD Reader Control File Commands	503
Required Versions For Use of VCD Patterns	505
Example of Using VCD Reader	505

VCD Reader Control File Commands

The following commands are supported in the control file of the VCD Reader.

1. Add Timeplate *timeplate_name period data_sample_time offset [pulse_width]*
 - o This command adds a timeplate. Timeplates are used to define the waveforms for primary input pins. All time values in this command must be based upon the timescale that appears in the VCD file. Therefore the period, data_sample_time, offset, and optional pulse_width must be scaled using the timescale. Otherwise the conversion from VCD format to FlexTest's cycle representation may be inaccurate.


For example, assume that the timescale = 10 ps but that simulation was performed using simulation data that used a 1 ns period. All literals for the time must be stated in terms of 10 ps. Thus, the period would be 100 since 100 * 10 ps equals 1 ns (or 1000 ps).
 - o *period* defines the period of the timeplate.
 - o *data_sample_time* defines the time that VCD Reader uses to strobe the values of the pins in each cycle. This value determines when the VCD data stream is sampled during the conversion of VCD format to FlexTest's cycle format. It refers to the "data sampling" time for inputs and outputs. Stable state times should be selected for all pins, not transition times. *data_sample_time* must be greater than or equal to the *offset* and less than or equal to (*offset*+*pulse_width*).
 - o *offset* defines the offset of the pins.
 - o *pulse_width* defines the pulse width for the pins with return timing waveform.

- The smallest period defined in the add timeplate commands is used as the test cycle length.
 - Periods defined in add timeplate commands must be equal to multiples of the test cycle length.
2. Setup Input Waveform *timeplate_name*
This command sets the default timeplate for all primary input pins.
 3. Add Input Waveform *timeplate_name pin_list*
This command defines a timeplate for the input pins which are listed in the *pin_list*. Pin names in the *pin_list* must be separated by space(s).
 4. Setup Output Strobe *strobe_time*
This command sets the default strobe time for all primary output pins.
 5. Add Output Strobe *strobe_time pin_list*
This command defines a strobe time for the output pins which are listed in the *pin_list*. Pin names in the *pin_list* must be separated by space(s).
 6. Set Time Check *filename*
This command turns the timing checking on. By default, it is off. The results from the timing checking are put in the named file.
 7. Set Collision Check <off>
By default, when there are collisions on bidirectional pins, the VCD Reader aborts. This command turns this feature off.
 8. Set VCD_module Name *module_name*
This command sets the module name where the primary input and output pins are defined in the VCD file.
 9. Set Dump Character Check Off
This command turns off the strict dump character checking feature. By default, this checking is on. When an unknown direction dump character is used for a unidirectional pin in a VCD file, the VCD Reader will issue a warning and continue to process the VCD file.

The handling of unknown direction dump characters on unidirectional pins includes the following:
 - Unknown direction dump character '0' will be used as an input '0' on an input pin or a measure '0' on an output pin.
 - Unknown direction dump character '1' will be used as an input '1' on an input pin or a measure '1' on an output pin.

- All other unknown direction dump characters, '?', 'F', 'A', 'a', 'B', 'b', 'C', 'c', and 'f', will be used as an input 'Z' on an input pin and a measure 'X' on an output pin.

Note


 To Debug questionable results from VCD reader, you may wish to dump an ASCII file of the patterns and check to see that the pins seem to be in the proper states. This would especially be helpful in checking to see you have captured pulsing clocks.

In the VCD file, only primary pins are allowed. "\$var ports" and "\$var wires" are used for internal pins and will cause an error.

Required Versions For Use of VCD Patterns

The VCD Plus format supported is the LSI Logic's extended VCD format. A VCD file can be created using ModelSim EE/Plus or Verilog-XL.

Note

 Note that version 8.5_5.1d or later of ModelSim must be used. This version provides support for the LSI extensions to the VCD file format. These extensions are required by FlexTest. Verilog-XL supports this format in version 2.3 or later.

To use Flextest with a VCD file, you must use version 8.5_5.1d or later of qhSim. This version provides support for the LSI extensions to the VCD file format. These extensions are required by Flextest.

In the FlexTest Dofile, the VCD Pattern file and the VCD Control file are referenced when using the Set Pattern Source command.

Example of Using VCD Reader

Following is an example of using VCD Reader from FlexTest.

Design netlist in Verilog

```
/*
 *   DESC: Generated by DFTAdvisor at Tue Mar 11 17:24:02 1997
 */
module test_vcd (rb , in2 , cnt1 , clk , buf_in , out_ff , out1 , buf_out
, ix0 );
input  rb , in2 , cnt1 , clk , buf_in ;
output out_ff , out1 , buf_out ;
inout  ix0 ;
wire \N$10 ;

MZTH \I$1 (.IO ( ix0 ), .OUT( out1 ), .C ( cnt1 ), .IN ( in2 ));
MD20E \I$2 (.NQ ( \N$10 ), .Q ( out_ff ), .CK ( clk ), .D ( \N$10 ),
.R( rb ));
MOPH \I$3 (.OUT ( buf_out ), .IN ( buf_in ));
endmodule
```

Verilog Test Bench Which Generates LSI Extension of VCD File

```
// Verilog format test patterns produced by FlexTest
// Filename      : PAT/pat1_verilog
// Timefile      : DEFAULT
// Scan operation : PARALLEL
// Fault         : STUCK
// Coverage      : 77.27(TC) 73.91(FC)
// Date          : Fri Jun 6 15:01:50 1997
//
`timescale 1ns / 1ns
module test_vcd_ctl;
integer    _compare_fail;
integer    _bit_count;
integer    _pattern_count;
reg[5:0]    _ibus;
reg[3:0]    _exp_obus, _msk_obus;
wire[3:0]    _sim_obus;
wire rb, in2, cnt1, clk, buf_in, ixo, out_ff, out1, buf_out;
assign rb = _ibus[5];
assign in2 = _ibus[4];
assign cnt1 = _ibus[3];
assign clk = _ibus[2];
assign buf_in = _ibus[1];
assign ixo = _ibus[0];
assign _sim_obus[3] = out_ff;
assign _sim_obus[2] = out1;
assign _sim_obus[1] = buf_out;
assign _sim_obus[0] = ixo;

reg [55:0] _nam_obus[3:0];
initial $readmemh("pat1_verilog.po.name",_nam_obus,3,0);

event      compare_exp_sim_obus;
always @(compare_exp_sim_obus) begin
    if ((_exp_obus&_msk_obus) != (_sim_obus&_msk_obus)) begin
        $write($time, ": Simulated response %b pattern %d\
n",_sim_obus,_pattern_count);
        $write($time, ": Expected  response %b pattern %d\
n",_exp_obus,_pattern_count);
        for(_bit_count = 0; _bit_count < 4 ; _bit_count =_bit_count +1)
            begin
                if((_exp_obus[_bit_count]&_msk_obus[_bit_count]) !=
(_sim_obus[_bit_count]&_msk_obus[_bit_count])) begin
                    $write($time, ": Mismatch at pin %d name %s, Simulated %b, Expected %b\
n",_bit_count,_nam_obus[_bit_count],
_sim_obus[_bit_count],_exp_obus[_bit_count]);
                end
            end
        _compare_fail = _compare_fail + 1;
    end
end

test_vcd test_vcd_inst (.rb(rb), .in2(in2), .cnt1(cnt1), .clk(clk),
.buf_in(buf_in),
.ixo(ixo), .out_ff(out_ff), .out1(out1), .buf_out(buf_out));
initial begin
```

```
// This is the command used for generating LSI extension of VCD file
$dumpsports(test_vcd_inst,"lixin_dump");

_compare_fail = 0;
_pattern_count = 0;
/* The beginning of output pattern section */

/* Cycle test block */

/* Pattern 0 */
_pattern_count = 0;
#0; /* 4000 */
_ibus=6'b01101Z;
#2000; /* 6000 */
_ibus=6'b01111Z;
#1000; /* 7000 */
_exp_obus=4'b0Z1Z;
_msk_obus=4'b1111;
-> compare_exp_sim_obus;

/* Pattern 1 */
_pattern_count = 1;
#1000; /* 8000 */
_ibus=6'b10101Z;
#2000; /* 10000 */
_ibus=6'b10111Z;
#1000; /* 11000 */
_exp_obus=4'b1Z1Z;
_msk_obus=4'b1111;
-> compare_exp_sim_obus;

/* Pattern 2 */
_pattern_count = 2;
#1000; /* 12000 */
_ibus=6'b01001Z;
#2000; /* 14000 */
_ibus=6'b01001Z;
#1000; /* 15000 */
_exp_obus=4'b0111;
_msk_obus=4'b1111;
-> compare_exp_sim_obus;

/* Pattern 3 */
_pattern_count = 3;
#1000; /* 16000 */
_ibus=6'b00000Z;
#2000; /* 18000 */
_ibus=6'b00010Z;
#1000; /* 19000 */
_exp_obus=4'b0000;
_msk_obus=4'b1111;
-> compare_exp_sim_obus;

/* Pattern 4 */
_pattern_count = 4;
#1000; /* 20000 */
_ibus=6'b001001;
#2000; /* 22000 */
```

```
_ibus=6'b001101;  
#1000; /* 23000 */  
_exp_obus=4'b0101;  
_msk_obus=4'b1111;  
-> compare_exp_sim_obus;  
  
/* Pattern 5 */  
_pattern_count = 5;  
#1000; /* 24000 */  
_ibus=6'b001000;  
#2000; /* 26000 */  
_ibus=6'b001000;#1000; /* 27000 */  
_exp_obus=4'b0000;  
_msk_obus=4'b1111;  
-> compare_exp_sim_obus;  
/* Total time: 28 */  
#1;  
if (_compare_fail == 0) begin  
    $display("No error between simulated and expected patterns\n");  
end  
#1;  
$finish;  
end  
endmodule
```

LSI Extension of VCD File

```

$date
    Fri Jun  6 15:12:09 1997
$end
$version
    dumpports $Revision: 1.11.4.6 $
$end
$timescale
    1ns
$end

$scope module test_vcd_ctl.test_vcd_inst $end
$var port      1 <0          rb $end
$var port      1 <1          in2 $end
$var port      1 <2          cnt1 $end
$var port      1 <3          clk $end
$var port      1 <4          buf_in $end
$var port      1 <5          out_ff $end
$var port      1 <6          out1 $end
$var port      1 <7          buf_out $end
$var port      1 <8          ix0 $end
$upscope $end

$enddefinitions $end

#0
pD  6  0  <0
pU  0  6  <1
pU  0  6  <2
pD  6  0  <3
pU  0  6  <4
pX  6  6  <5
pX  6  6  <6
pX  6  6  <7
pX  6  6  <8

#65
pL  6  0  <5pH  0  6  <7

#2000
pU  0  6  <3

#4000
pU  0  6  <0
pD  6  0  <1
pD  6  0  <3

#6000
pU  0  6  <3

#6149
pH  0  6  <5

#8000
pD  6  0  <0
pU  0  6  <1
pD  6  0  <2

```

```

pD  6  0  <3

#8065
pL  6  0  <5

#8305
pH  0  6  <8

#8488
pH  0  6  <6

#12000
pD  6  0  <1
pD  6  0  <4

#12401
pL  6  0  <7

#12490
pL  6  0  <8

#12616


#148
pf  0  0  <8

#413
pL  6  0  <6

#14000
pU  0  6  <3

#16000
pU  0  6  <2
pD  6  0  <3
pB  6  6  <8

#16096
pU  0  6  <8

#16279
pH  0  6  <6

#18000
pU  0  6  <3

#20000
pD  6  0  <3
pD  6  0  <8

#20126
pL  6  0  <6

```

FlexTest Dofile

```
set hypertrophic limit off
set test cycle 3
add clock 0 CLK
add pin con clk r0 1 1 1
add pin con clk rb 1 1 1
set sys m g
set pattern source external lsivcd_dump -vcd -c control
run
save pat results/pattern1.ascii.f -re -ext
save pat results/pattern2.ts.f -wgl -serial -re -ext
save pat results/pattern3.vs.f -verilog -serial -re -ext
```

VCD Reader Control File Example

```
set collision check off
add timeplate tp 4000 1900 0
add timeplate tp_clk 4000 3000 2000 2000
setup input waveform tp
add input waveform tp_clk clk
setup output strobe 3900
set time check results/time_check
```


Appendix B

Using the FlexTest Tcl Interface

FlexTest uses a Tcl-based command interface, which provides basic Tcl capabilities within the tool session, either at the command line or in dofiles. You can embed Tcl constructs in tool commands. You can also embed tool commands within Tcl constructs the same as any Tcl command.

If Tcl procedures are available in separate files, you can source these files from within the tool or dofiles. You can also place Tcl procedures in a *.tool_startup* file so that they are available for use. Tcl file input/output is also supported.

The following sections explain using the Tcl interface:

Using Tcl Within FlexTest	513
Modifying Existing Dofiles for Use with Tcl	516
Dollar Sign	516
Quotation Marks	516
Set Command	517
Optional Single Quotes	517
Environment Variables	517
Special Tcl Characters	518
Tcl Comments Can Be Tricky	519
The Dofile Command and Tcl Source Command Are Different	520
Tcl Resources	520

Using Tcl Within FlexTest

You can use Tcl variables interchangeably with legacy Tessent tool variables and environment variables.

Siemens EDA recommends using Tcl syntax for setting and referencing variables, including using `$env(ENVARNAME)` for accessing environment variables. For example, if the value of environment variable 'foo' equals 'model' (`$foo = model`), you can compare this value to a Tcl variable value using the following syntax:

```
set bar model2
if {$env(foo) == $bar} {puts Match} else {puts "No match"}
```

In addition, the tool returns all output of tool commands, which you can assign to a variable. If a command fails, then the command returns the string `DFT_ERROR`.

Note



You should use Tcl namespaces to avoid any possibility of creating procedures that conflict with existing tool or Tcl commands.

When embedding comments within a dofile, the tool processes as a comment any entry beginning with a double slash (//) until the end of line. In Tcl files, the pound sign (#) specifies a comment that goes to the end of line. When using dofiles containing Tcl constructs, the tool recognizes both ‘//’ and ‘#’ as the start of a comment that extends to the end of the line.

Tcl Example 1

In this example, the Add Scan Chains command is used to define every scan chain in the design. This command is sometimes used hundreds of times and makes the dofile very long. Using Tcl, you can shorten this action substantially by using a loop construct as shown in the following example:

```
for {set xx 1} {$xx < 257} {incr xx} {  
    add scan chains int chain$xx group1 /top/edt_si$xx /top/edt_so$xx  
}
```

For the values of xx from 1 up to 256, the tool executes a separate Add Scan Chains command. The transcript and logfile will contain all 256 Add Scan Chains commands.

Tcl Example 2

The following example controls the flow of creating test patterns and uses a variable to execute different commands based on the variable’s value:

```
if {$mode == stuck} {  
    set fault type stuck  
    . . .  
} elseif {$mode == transition} {  
    set fault type transition no_shift_launch  
    . . .  
}
```

Only the executed commands appear in the tool’s transcript and log file.

Tcl Example 3

This example places the output from a report command in a variable for subsequent processing:

```
SETUP> set chain_report [report scan chains]  
  
chain = chain1 group = grp1 input = /scan_in1 output = /scan_out1 length = unknown  
...  
  
chain = chain256 group = grp1 input = /scan_in256 output = /scan_out256 length = unknown
```

SETUP> puts \$chain_report

chain = chain1 group = grp1 input = /scan_in1 output = /scan_out1 length = unknown

...

chain = chain256 group = grp1 input = /scan_in256 output = /scan_out256 length = unknown

Modifying Existing Dofiles for Use with Tcl

When using an existing dofile with the Tcl interface, you should evaluate the dofile for issues that could cause incorrect evaluation by the Tcl interpreter. The most common issue you can run across with an existing dofile is accounting for Tcl special characters.

See [Table B-1](#), which provides a list of typically-used Tcl special characters.

The following sections provide guidance for correcting the most common issues you could run across:

Dollar Sign	516
Quotation Marks	516
Set Command	517
Optional Single Quotes	517
Environment Variables	517

Dollar Sign

A dollar sign (\$) in Tcl specifies variable substitution. In some netlists, path names (for example, like *foo/pin\$p7*) can contain the dollar sign. When using path names with dollar signs, enclose the path name with braces ({ }) to prevent the tool from substituting the value as shown in the following example:

```
report gates {foo/pin$p7}
```

Quotation Marks

In Tcl, quotation marks (" ") instruct the Tcl interpreter to treat the enclosed words as a single argument.

For example:

```
puts " Hello World "
```

If embedded quotes are required, then you must use backslashes (\) to escape the embedded double quotes. For example:

```
puts " Hello \"World\" "
```

Otherwise, the tool issues an error message.

Set Command

The Tcl “set” command assigns a value to a variable, for example “set color blue.” Many FlexTest commands use “Set ...” as part of the command, which can conflict with this syntax.

Also note that the following command “set display foobar,” sets the DISPLAY environment variable rather than a local Tcl variable named “display,” since “set display” is a valid FlexTest command.

Optional Single Quotes

Optional single quotes (‘ ’) are no longer valid for tool commands.

For example, the following produces an error:

```
add clocks 1 ‘clock1’
```

Correct this by using no quotes, double quotes, or braces:

```
add clocks 1 clock1
```

```
add clocks 1 “clock1”
```

```
add clocks 1 {clock1}
```

Environment Variables

To preserve backward compatibility, all environment variables are set as Tcl variables at invocation so the variables may be referenced as \$var or \$env(var). Subsequently, if you set a Tcl variable with the same name as an existing environment variable, the tool returns the Tcl variable’s new value when you reference \$var.

You must use \$env(var) to return the original environment variable.

Special Tcl Characters

In Tcl scripts, you often see characters used for special purposes. For a complete list of special characters, you should consult the a Tcl resource.

Table B-1 lists and describes the more-common special characters you can encounter when reading the examples in this manual.

Table B-1. Common Tcl Characters

Character	
;	The semicolon terminates the previous command, allowing you to place more than one command on the same line.
\	Used at the end of a line, the backslash continues a command on the following line.
\\$	The backslash with other special characters, like a dollar sign, instructs the Tcl interpreter to treat the character literally.
\n	The backslash with the letter “n” instructs the Tcl interpreter to create a new line.
\$	The dollar sign in front of a variable name instructs the Tcl interpreter to access the value stored in the variable.
[]	Square brackets group a command and its arguments, instructing the Tcl interpreter to treat everything within the brackets as a single syntactical object. You use square brackets to write nested commands. Example: set chain_report [report scan chain]
{ }	Curly braces instruct the Tcl interpreter to treat the enclosed words as a single string. The Tcl interpreter accepts the string as is, without performing any variable substitution or evaluation. Example: Create a string that contains special characters, such as \$ or \: set my_string {This book costs \$25.98.}
" "	Quotes instruct the Tcl interpreter to treat the enclosed words as a single string. However, when the Tcl interpreter encounters variables or commands within string in quotes, it evaluates the variables and commands to generate a string. Example: Create a string that displays a final cost calculated by adding two numbers: set my_string "This book costs \\${expr \$price + \$tax}"

Tcl Comments Can Be Tricky	519
The Dofile Command and Tcl Source Command Are Different.....	520
Tcl Resources	520

Tcl Comments Can Be Tricky

In Tcl, you create the equivalent of comments with the pound sign (#). When using the pound sign, you must use it where a command starts, and at the beginning of a command, not within a command. The pound sign directs the Tcl compiler to not evaluate the rest of the line.

Tricky Point 1: Evaluate does not equal parse. Despite the pound sign, the comment below gives an error because Tcl detects an open lexical clause.

```
# if (some condition) {  
if { new text condition } {  
...  
}
```

Tricky Point 2: The apparent beginning of a line is not always the beginning of a command:

```
# This is a comment
```

In the following code snippet, the line beginning with “# -type” is not a comment, because the line right above it has a line continuation character (\). In fact, the “#” confuses the Tcl interpreter, resulting in an error when it attempts to create the tk_messageBox.

```
tk_messageBox -message "The diagnosis report was successfully written."\  
# -type ok  
  
# and this is also a comment. This one will span \  
multiple lines, even without the # at the beginning \  
of the second and third lines.
```

In general, it is good practice to begin all comment lines with a #.

Tricky Point 3: The beginning of a command is not always at the beginning of a line:

Usually, you begin new commands at the beginning of a line. That is, the first character that is not a space will be the first character of the command name. However, you can combine multiple commands into one line using the semicolon “;” to signal the end of the previous command:

```
set myname "John Doe" ;set this_string "next command"  
  
set yourname "Ted Smith" ; #this is a comment
```

See also “[Tcl Example 3.](#)”

The Dofile Command and Tcl Source Command Are Different

The tool's Dofile command is normally used to execute a file of tool commands. Tcl also has a “source” command that executes a file of commands, but you should only use this command to load Tcl procs or set some Tcl variables or other strictly Tcl commands.

If the command file contains tool commands, then you should use only the Dofile command because the “source” command is not affected by the `set_dofile_abort` command, and only the Dofile command transcripts the commands to the shell and logfile. The “source” command always stops execution if any command in the file encounters an error.

Tcl Resources

The following website a place to start in your search for the reference material that works best for you.

It is not an endorsement of any book or website.

<http://www.tcl.tk/>

— A —

Abort Interrupted Process, [123](#)

Aborted faults

 changing the limits, [74](#)

ATPG

 constraints, [72](#)

 function, [72](#)

 with FlexTest, [57](#)

Automatic test equipment, [58](#)

— B —

BACK algorithm, [57](#)

Bus

 dominant, [40](#)

 float, [52](#)

Bus contention, [52](#)

— C —

Chain test, [95](#)

Combinational loop, [50](#)

Commands

 Abort Interrupted Process, [123](#)

Constraints

 ATPG, [72](#)

 pin, [65](#)

Cycle count, [97](#)

Cycle test, [95](#)

— D —

Design flattening, [42](#)

Differential scan input pins, [95](#)

Dominant bus, [40](#)

— F —

Fault

 aborted, [75](#)

 classes, [28](#)

 collapsing, [20](#)

 representative, [20](#), [28](#)

 undetected, [75](#)

Fault models

 psuedo stuck-at, [20](#)

 toggle, [20](#)

 transition, [20](#)

Flattening, design, [42](#)

FlexTest

 ATPG method, [57](#)

 introduced, [17](#)

FlexTest commands

 abort interrupted process, [62](#)

 add lists, [68](#)

 add pin constraints, [65](#)

 add pin strobes, [65](#)

 compress patterns, [73](#)

 delete faults, [69](#)

 delete pin constraints, [65](#)

 delete pin strobes, [66](#)

 load faults, [70](#)

 report faults, [70](#)

 report pin constraints, [65](#)

 report pin strobes, [66](#)

 reset state, [68](#)

 resume interrupted process, [62](#)

 run, [68](#)

 set abort limit, [74](#)

 set fault type, [69](#)

 set hypertrophic limit, [72](#)

 set interrupt handling, [62](#)

 set list file, [68](#)

 set output comparison, [68](#)

 set possible credit, [72](#)

 set test cycle, [65](#)

 setup pin constraints, [65](#)

 setup pin strobes, [65](#)

 write faults, [70](#)

— H —

Hierarchical instance, definition, [36](#)

— I —

IDDQ testing
 psuedo stuck-at fault model, 20
 test pattern formats, 94
Instance, definition, 36

— L —

Learning analysis, 45
Line holds, 23
Loop count, 97

— M —

Module, definition, 37

— O —

Offset, 59
Off-state, 35

— P —

Path sensitization, 22
Period, 59
Pin constraints, 64, 65
Possible-detect credit, 24
Possible-detected faults, 24
Primary inputs
 constraints, 64, 65
 cycle-based requirements, 60
Primary outputs
 strobe requirements, 60
Primitives, simulation, 38
Pulse width, 59

— S —

Scan chains
 serial loading, 94
Scan related events, 86
Scan sub-chain, 92
Scan test, 95
Sequential loop, 50, 52
Serial scan chain loading, 94
Simulation data formats, 94
Simulation formats, 91
Simulation primitives
 TLA, 39
Simulation primitives, 38 to 42
 AND, 39
 BUF, 38

BUS, 40
DFF, 39
INV, 39
LA, 39
MUX, 39
NAND, 39
NMOS, 40
NOR, 39
OR, 39
OUT, 42
PBUS, 40
PI, 38
PO, 38
RAM, 41
ROM, 41
STFF, 39
STLA, 39
SW, 40
SWBUS, 40
TIE gates, 40
TSD, 40
TSH, 40
WIRE, 40
XNOR, 39
XOR, 39
ZHOLD, 40
ZVAL, 38

Structural loop, 50
 combinational, 50
 sequential, 50

— T —

Table Pattern Format
 Control Section, 497
 Data Section, 496
Test cycle
 defined, 59
Test Pattern File Format
 ASCII Pattern Format, 488
Test patterns
 chain test block, 95
 cycle test block, 95
Test Procedure File, definition of, 106
Time frame, 59, 64
Timeplate statements
 bidi_force_pi, 89

bidi_measure_po, [89](#)
force, [89](#)
force_pi, [89](#)
measure, [89](#)
measure_po, [89](#)
offstate, [89](#)
period, [90](#)
pulse, [90](#)

— U —

Undetected faults, [75](#)

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.

