SpyGlass® DFT Submethodology (for GuideWare 2017.12)

N-2017.12-SP2, June 2018

SYNOPSYS®

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Report an Error

The SpyGlass Technical Publications team welcomes your feedback and suggestions on this publication. Please provide specific feedback and, if possible, attach a snapshot. Send your feedback to *spyglass_support@synopsys.com*.

Contents

Preface **9**
About This Book **9**
Contents of This Book **10**
Typographical Conventions **11**

The Need for DFT-Optimized Design **13**
The Need for DFT-Optimized Design **13**
Introduction 14
Objective 14
Tool Versions 14
References 14
Document Overview **15**
GuideWare Reference Methodology **16**
Prerequisites 16
Designing for Test **17**
Scan Flip-flops and Chains 17
Latch Transparency 18
Capture 19
Performing DFT Analysis using SpyGlass DFT **21**
DFT Setup 22
DFT Setup Manager 22
Create Constraints for Bidirectional Ports 26
Steps to maximize fault coverage 27
Achieve Scannability 27
Make Latches Transparent 29
Compliance to DFT Best Practices 30
Preparing design for BIST 31
Adding Testpoints 31
Verifying Scan Chains 34
Verifying Test Signal Connections in Full-chip Designs 35
Create test Constraints for Full Chip 36
Connection Verification Procedure 38
GuideWare Methodology for DFT 40
Step-by-step Solution **43**

Setup for DFT (goal name = dft_setup) 43
Create necessary *clock* Constraints....46
Create necessary *test_mode* Constraints....47
Achieve Scannability (goal name = dft_scan_ready) 47
Clock_11 Debug 48
Diagnose_testclock 49
Info_testclock 50
Async_07 Debug....50
Async_08 debug 51
Diagnose_testmode 51
Info_testmode....52
Info_scanwrap debug 52
Info_noscan debug 52
Info_inferredNoscan debug 52
Viewing the estimate of fault coverage of the design....53
Ensure Compliance to DFT Best Practices (goal name = dft_best_practice) ..57
Review the stuck_at_coverage_audit report....57
Make flip-flops scannable 59
Make Latches Transparent....59
Scan-wrap black boxes 59
Combinational Loops Made Transparent 60
Testmode/Tied pins made controllable 60
Hanging nets made controllable 61
Tristate nets made observable 61
The *force_ta* nets and *test_point* constraint pins made testable....61
The *no-scan* flip-flops made scannable....62
Using the Coverage_Audit report to ensure test operation 62
Async_02 violations 62
Async_11 violations 63
Clock_04 violations 63
Clock_08 violations 64
Clock_16 violations 64
Clock_17 violations 65
Clock_21 violations 65
Clock_27 violations 65
Clock_28 violations 66
Scan_07 violations....66
Scan_22 violations....66
Topology_03 violations 67
Topology_05 violations 67
Topology_13 violations 67

Tristate_06 violations 67
Achieve BIST Readiness (goal name = dft_bist_ready) 68
Adding Testpoints (goal name = dft_test_points) 69
TA_09 debug 69
Validating Scan Chains (goal name = dft_scan_chain) 70
Scan_22 70
Scan_24 70
Scan_25 70
Scan_26 70
Info_scanchain 71
Verifying Test Signal Connections in Full-chip Designs (goal name = dft_block_check) 71
Create test SGDC for Full-chip 72
Subblock Check (goal name = dft_block_check) 72
Creating and Validating an Abstract Model for a Block (goal names = dft_abstract, dft_abstract_validate) 73
Using Autofix/Selective Autofix 74

Appendix A 75

Preface

About This Book

The SpyGlass® DFT methodology guide describes the flow for using the DFT methodology.

Contents of This Book

The SpyGlass DFT methodology guide has the following sections:

Section	Description
The Need for DFT-Optimized Design	The need for DFT-optimized design

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	OUT <= IN;
Reworked example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

The Need for DFT-Optimized Design

The Need for DFT-Optimized Design

Manufacturing test is performed by patterns automatically generated by ATPG (Automatic Test Pattern Generation) tools. To operate effectively, these tools require that the circuits be correctly designed for testing. This document and the corresponding SpyGlass DFT software directly address this issue.

The Need for DFT-Optimized Design

Introduction

Objective

The objective of this document is to describe a method to run SpyGlass DFT that focuses on goals that are well matched with the requirements for full scan testability.

Tool Versions

- SPYGLASS VERSION: N-2017.12-SP2
- DFT VERSION: N-2017.12-SP2
- GuideWare: 2017.12

References

- SpyGlass DFT User Guide
- Design Read Methodology Document

Document Overview

This document is organized as follows:

- The *GuideWare Reference Methodology* section describes the Guideware method as well as prerequisites for applying GuideWare to DFT
- The *Designing for Test* section describes full scan design
- The *Performing DFT Analysis using SpyGlass DFT* section describes how the rules built into SpyGlass addresses the requirements of full scan design
- The *Step-by-step Solution* section describes a step by step description for using GuideWare to meet the specific goals for achieving high fault coverage.

GuideWare Reference Methodology

GuideWare Reference Methodology groups SpyGlass rules selected from various SpyGlass product areas (including Lint, CDC, DFT, Power and Constraints) into goals aligned with chip development process and validating them for high impact. The GuideWare Reference Methodology provides a jumpstart for design groups with SpyGlass goals readily usable out-of-the-box at various phases of IC design flow (RTL, IP and Chip Integration design phases). The GuideWare Reference Methodology can be configured to map to customer specific design style and handoff requirements. For more details of GuideWare Reference Methodology, please refer to the documentation as part of this release installation.

Prerequisites

You are expected to have basic knowledge of SpyGlass operations.

SpyGlass DFT requires a clean design read. Please refer to the *SpyGlass Design Read-In Methodology guide* for details.

The RTL also must be “lint” clean before analyzing for DFT. This can best be done by using the GuideWare methodology. Please refer to the *GuideWare User Guide* for details. However, if the user customizes these into different local goals then they should ensure that all goals taken together include those goals.

Designing for Test

In order to understand the Guideware goals for test, it is necessary to understand the basic design for test strategy followed by the most electronic companies.

The ATPG tools in common used today are much more efficient for processing combinational circuits than for sequential circuits. As a result, the primary DFT approach commonly used is to implement full scan on the design. Key objectives of this design method are:

- to allow any internal state necessary for testing to be achieved by forming shift registers called scan chains
- to force latch enables to be active so that the latch may be treated as a buffer
- to allow easy control of clocks so test results at internal nodes can be captured

Scan Flip-flops and Chains

A typical scan flip-flop is shown in *FIGURE 1. Scan Flip-flop*. Scan flip-flops are connected (see *FIGURE 2. Scan Chain Segment*) so that shift registers are formed when the “SE” pin is set to 1.

A critical aspect of this shifting action is that shift clocks must reach the scan flip-flops and the sets and resets must remain inactive regardless of circuit state. A major part of SpyGlass DFT processing will be to achieve these requirements.

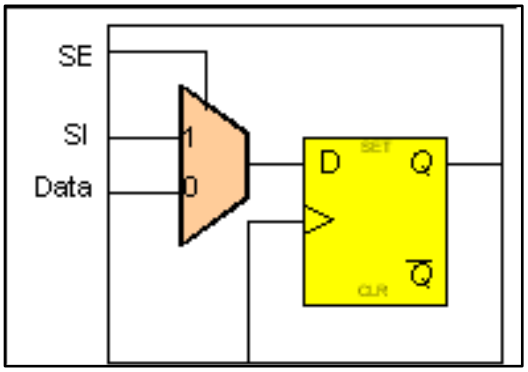


FIGURE 1. Scan Flip-flop

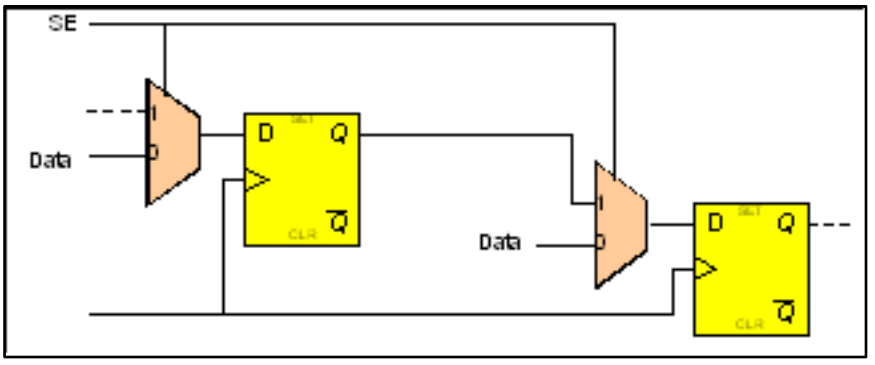


FIGURE 2. Scan Chain Segment

Latch Transparency

Latches are state elements that can defeat combinational ATPG tools. Tests for circuits such as shown in *FIGURE 3. Latch Transparency*, may require signal propagation through latches. The appropriate DFT method is to

design the latch enables so that the latch is forced active in capture mode. In this way, the latch may be treated as a simple buffer by ATPG tools.

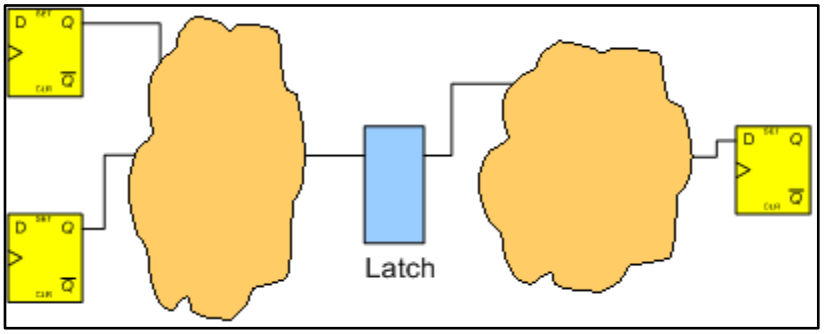


FIGURE 3. Latch Transparency

Capture

The waveforms in *FIGURE 4. Waveforms for One ATPG Vector*, are typically used to shift tests into a circuit and to capture test data using these chains. To perform the tests, the scan multiplexers, see *FIGURE 1. Scan Flip-flop*, must be switched back to 0 so that the test results can be captured for scan-out. The capture clock must be operated in the system mode and therefore the circuit must be designed to guarantee that the capture pulses reach the scan flip-flops regardless of the scan-in state.

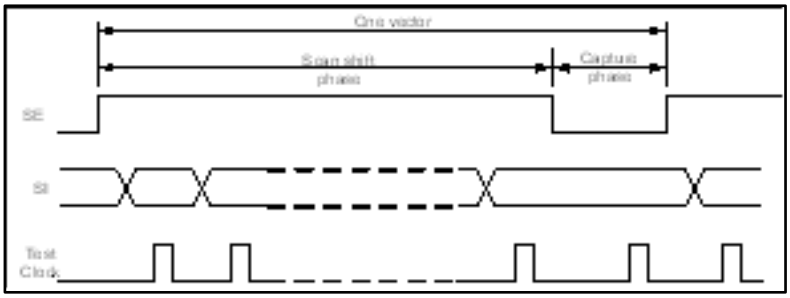


FIGURE 4. Waveforms for One ATPG Vector

The remaining sections of this document will describe how to use SpyGlass DFT for these functions.

Performing DFT Analysis using SpyGlass DFT

This section of the document describes the use of SpyGlass DFT for:

- Block-level designs (see the *Creating and Validating an Abstract Model for a Block (goal names = dft_abstract, dft_abstract_validate)* section)
- Verifying scan chains in gate-level netlist (see *Verifying Scan Chains* section)
- Verifying test signal connections in full-chip designs (see *Verifying Test Signal Connections in Full-chip Designs* section)

The preferred flow is illustrated in *FIGURE 5. Block-level Flow*. Processing a design in the order shown in the illustration helps to reduce the time to complete the job, the number of test points (if any) and unnecessary interactions with the tool.

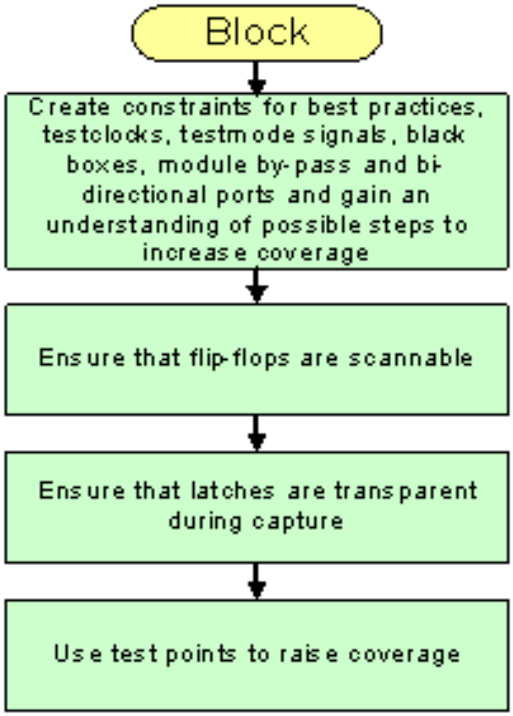


FIGURE 5. Block-level Flow

DFT Setup

After completing the design read, use the reports generated by SpyGlass to create a test setup environment.

This section explains the following topics:

- *DFT Setup Manager*
- *Create Constraints for Bidirectional Ports*

DFT Setup Manager

The DFT Setup Manager can be used to generate the necessary DFT constraints for a design. The Setup Manager is accessible from the Console's GUI interface via the `dft_setup` goal. The setup manager guides you through the various steps that help identify the following:

- Black box resolution
- Test clocks
- Asynchronous set/reset signals
- Test modes
- PLL and clock shapers
- Clock gating cells
- Setting of no_scan, scan_wrap, and so on

For general information on Console Setup Managers and the Common Design Setup, please refer to the *SpyGlass Explorer User Guide*.

The following sections detail the different test related constraints that are relevant to run DFT specific goals:

- *Define Initial Testclocks*
- *Define Initial Testmodes*
- *Use bypass Constraint for Modules with Internal bypass Logic*
- *Black Boxes— scan_wrap*

Define Initial Testclocks

The information generated by DFT setup includes root-level pins that may

be system clocks. It often includes pins that are clock enables, so generally the list cannot be used directly. If some of the pins in this list are recognized as testclocks, they can be used as testclock constraints for SpyGlass DFT.

For example, if sysClockA shown in the simple example in *FIGURE 6. System Clock Example* is also a test clock, then the constraint for use by DFT is as follows:

```
clock –name sysClockA -testclock
```

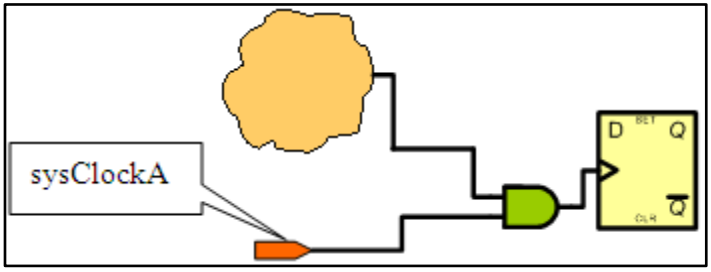


FIGURE 6. System Clock Example

DFT Setup manager automatically identifies system clocks, and allows user to mark them as test clocks as applicable.

Define Initial Testmodes

The design setup also includes the pins that may be system set or reset pins. Since controlling asynchronous pins is a key aspect of scan design, this list may be used as a starting point in creating test_mode constraints.

Scan design requires that flip-flop sets and resets must be disabled during scan shifting. For example, in *FIGURE 7. System Reset Example*, the signal resetDisable would be disabled in test mode by the following constraint:

```
test_mode –name resetDisable –value 1
```

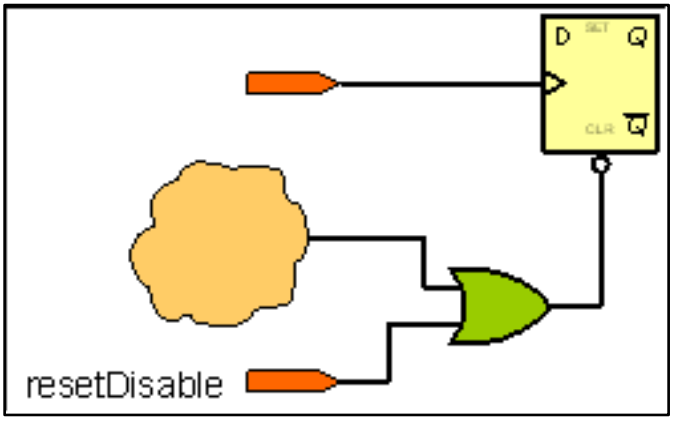


FIGURE 7. System Reset Example

DFT Setup Manager can automatically identify all set/reset signals, and derive test_mode constraints based on those signals – to hold them off during scan-shift. Through the GUI File editor, user can add any additional test constraints, or modify the automatically generated test constraints.

Use bypass Constraint for Modules with Internal bypass Logic

Some blocks, such as memories or complex IP blocks with BIST circuitry, may be designed with "bypass" logic to provide a connection path from block inputs directly to block outputs. Often RTL models for such blocks are not available. The module_bypass constraint can be used to properly handle these blocks within a larger design while running SpyGlass DFT.

For example, in the design shown in *FIGURE 8. Module with bypass*, when the signal “By-pass” = 1 then the Data-in port is directly connected to the “Data-out” port. From a DFT point of view, the Data-out port can be controlled by Data-in and the Data-in port can be observed by the Data-out if a logical one is applied to By-pass. The module_bypass constraint can be used as follows:

```
module_bypass –name modBP –bpin By-pass –value 1 –iport Data-in –oport Data-out
```

Refer to the DFT User Guide section *SpyGlass DFT Constraints Currently Defined*” for details on using this constraint.

Using the module_bypass constraint will provide a more accurate black box model and more accurate SpyGlass DFT results.

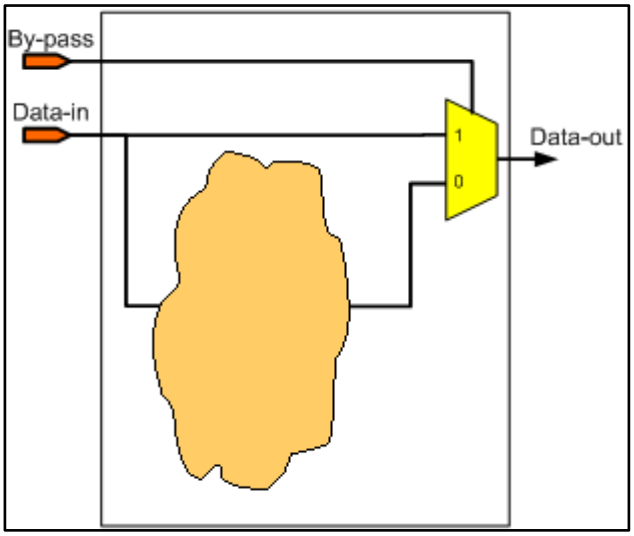


FIGURE 8. Module with bypass

Black Boxes— scan_wrap

DFT Setup Manager reports all the black box modules in the design. If any black box module in the design is known to be designed with internal boundary scan chains, then the Setup Manager allows the user to identify and create scan_wrap constraints for such modules. For example:

```
scan_wrap –name IntRingBox
```

(See DFT User Guide section *"SpyGlass DFT Constraints Currently Defined"* for details on using the scan_wrap constraint.)

NOTE: *DFT Setup Manager currently only allows setting of scan_wrap constraints on modules, and not on individual instances. To set this constraint on individual instances, you can use the GUI file editor to add or modify these constraints in the Setup Manager.*

Create Constraints for Bidirectional Ports

If the design has bidirectional IO ports that are shared with test control signals, then constraints must be defined to configure these ports so that signals at the root level do propagate into the design. Use the Diagnose_testclock and Diagnose_testmode rules to verify that the testclocks and the test_mode constraints penetrate into the design without bus contention. Test_mode constraints used for this purpose should be declared for both scanshift and capture mode.

For example in *FIGURE 9. Simple Bidirectional Connection*, in order to use ctrl as a testclock, the enaOut must have a value that tri-states the buffer. This means that the test_mode constraint(s) must be written to ensure that enaOut is 0 (assuming an active high tristate buffer) for both scanshift and capture.

```
test_mode –name enaOut –value 0
```

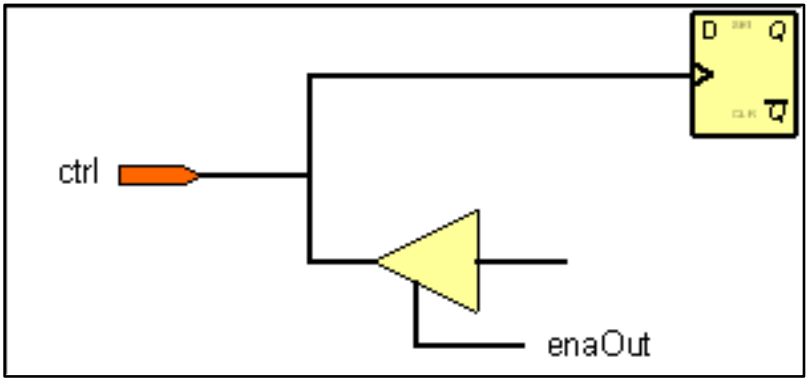


FIGURE 9. Simple Bidirectional Connection

Steps to maximize fault coverage

Once the design constraints are in place it is useful to audit the design for testability of the design. At this step one may find out the current test status and possible steps that can be taken to improve the fault coverage.

This section explains the following topics:

- *Achieve Scannability*
- *Make Latches Transparent*
- *Compliance to DFT Best Practices*
- *Preparing design for BIST*
- *Adding Testpoints*
- *Verifying Scan Chains*
- *Verifying Test Signal Connections in Full-chip Designs*

Achieve Scannability

The objective of this step is to refine the list (see Section *Define Initial Testclocks* and *Define Initial Testmodes*) of clocks and testmode signals. This is an important consideration, since successful SpyGlass DFT analysis requires use of both logic simulation and testability analysis. Both of these calculations require device representations at the generic gate level. These analyses critically depend on having testclocks and test_mode constraints. (See SpyGlass DFT User Guide section “Scannability and Testability.”)

Testclocks for scan

Test logic using scan design techniques requires operating the circuit in a mode different from functional mode. System clocks that are internally generated must be controlled in a well-defined way when test vectors are applied. *FIGURE 10. Derived Clock with bypass* shows a case where sysClk should be declared as a testclock and Clk_bypass should be declared as a testmode signal with a value that selects sysClk.

```
clock –name sysClk –testclock
test_mode –name Clk_bypass –value 1
```

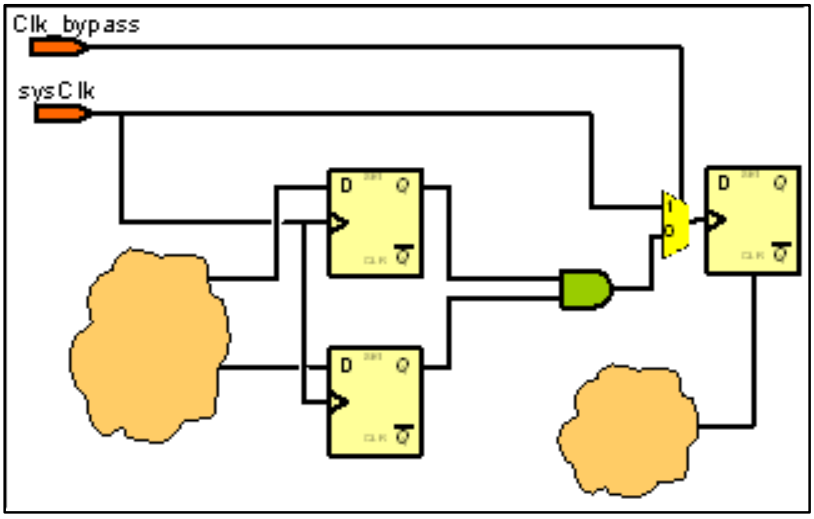


FIGURE 10. Derived Clock with bypass

FIGURE 11. PLL bypass shows a PLL that has been designed with a by-pass. In such cases, the pllBypass should be declared as a test_mode with value 1 and sysClk should be declared as a testclock.

```
clock –name sysClk –testclock
test_mode –name pllBypass –value 1
```

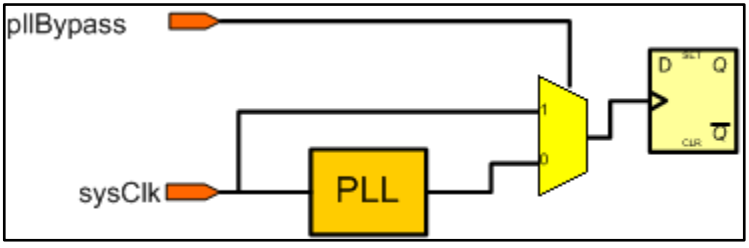


FIGURE 11. PLL bypass

The dft_scan_ready goal is designed to detect and diagnose testclock

issues.

Asynchronous Sets and Resets for Scan

The asynchronous sets and resets must also be controlled by well-defined root-level procedures. *FIGURE 12. Reset bypass example* shows an example of derived asynchronous reset logic that feeds an active low reset pin that has been bypassed with a resetDisable signal to force the reset to the inactive state during scan shifting. In this case, resetDisable should be declared in a test_mode constraint:

```
test_mode –name resetDisable –value 1 –scanshift
```

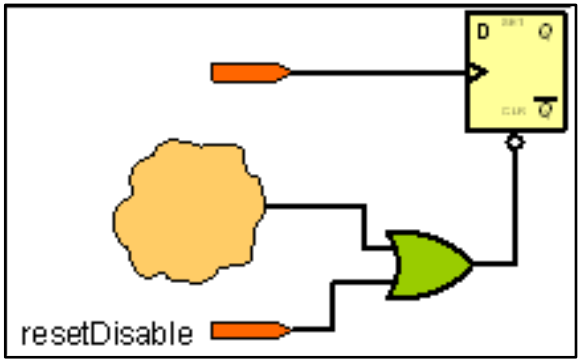


FIGURE 12. Reset bypass example

The `dft_scan_ready` goal is designed to detect and diagnose flip-flop set and reset issues.

Make Latches Transparent

As mentioned earlier, ATPG tools are usually run in combinational mode so that the flip-flop outputs can be easily controlled and the flip-flop inputs can be easily observed. As a result, from an ATPG point of view, the scan flip-flop outputs are treated as if they were primary inputs and the scan flip-flop inputs are treated as if they were primary outputs.

Latches are also sequential devices and therefore pose a problem for combinational ATPG. Instead of making latches scannable, latch enables

are designed to be forced active during the capture phase so that ATPG tools may treat latches as simple buffers. If this condition is not satisfied, any logic that requires such a latch or any logic that only feeds such a latch will not be testable and coverage will be compromised.

FIGURE 13. DFT Logic to Force Latch Transparency shows an example of a latch with an OR gate added to allow forcing of latch transparency. In this case, the following constraint could be used:

```
test_mode –name forceTrans –value 1 –capture
```

The test_mode option “-capture” is used because the latch is only required to be transparent for ATPG. Latches are transparent for ATPG when their enable pin is active in the “off” state of the clock. Latch enables can be “don’t care” during scan shifting.

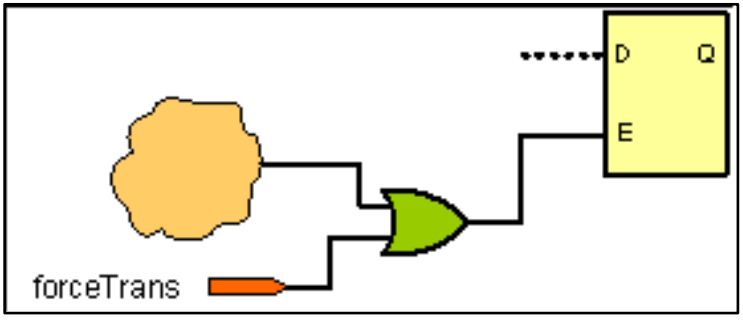


FIGURE 13. DFT Logic to Force Latch Transparency

The `dft_best_practice` goal contains several rules designed to detect and diagnose latch enable issues.

Compliance to DFT Best Practices

The following is a list of various structures that block ATPG tools from achieving high coverage:

- Test clock also used as data
- Asynchronous combinatorial loop that is not broken during testmode
- Test clock also drives the set/reset pin of any flip-flop

- Tri-state bus enables are not ‘one-hot’ encoded
- Flip-flops have large fan-in cones

Designers can avoid these pitfalls by discovering such structural issues at the RTL coding stage. They may check their designs for best DFT practices even without testmode setup knowledge.

Preparing design for BIST

A design is BIST ready when no unknown values (“X”) are captured in the scannable flip-flops of the design. This is because in BIST (Built-In-Self-Test) designs, the scan output is compressed into a signature register, and capturing unknown values will corrupt the signature. This can also be the case when ATPG Compression is used, so you should consider running this goal also for designs using such Compression techniques. This can be achieved by removing all sources of X-propagation.

Adding Testpoints

If, after satisfying all prior steps, the coverage is not satisfactory, consider adding test points to the design. A test for a specific fault requires that the fault be both controllable and observable. Therefore, test coverage can be potentially improved by *Improving Observability* or by *Improving Controllability*. This can be achieved by adding observe or control testpoints.

The DFT rule TA_09 produces a coverage report that can be used as a guide for test point selection. The final coverage listed in that report will be achieved if all the testpoints are used. The coverage reported at an intermediate point will be achieved if all testpoints listed up to that position in the report are used. If a non-consecutive subset of the testpoints is selected from the TA_09 report, then the resulting coverage may be established by adding testpoint constraints for just the selected locations to the sgdc file and rerunning.

TA_09 also produces an .sgdc with testpoint constraints for all nodes listed in the coverage report. The coverage for any subset of testpoints can be obtained by using testpoint constraints for any subset of testpoint locations and rerunning SpyGlass DFT.

Improving Observability

FIGURE 14. Unobservable Net illustrates a block of logic whose output net is unobservable even though it fans out to multiple places. The lack of observability may be due to black boxes, IPs designed without scan, blocked paths or a variety of other causes.

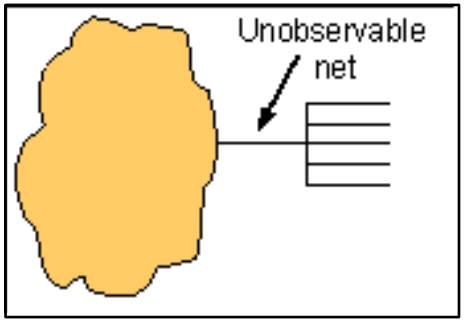


FIGURE 14. Unobservable Net

FIGURE 15. Observation Testpoint shows a scannable flip-flop added to provide observability to an existing unobservable net.

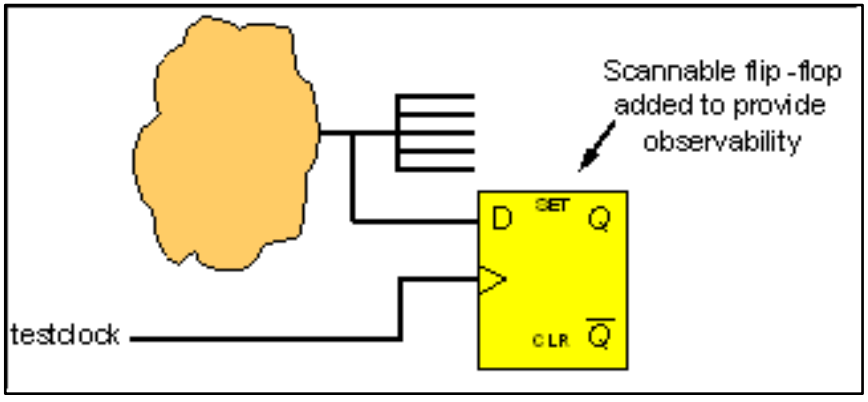


FIGURE 15. Observation Testpoint

Improving Controllability

An example of a bad controllability situation is illustrated in *FIGURE 16. Uncontrollable Net,* where the net driven by “A” is uncontrollable. As a result, the downstream logic is untestable.

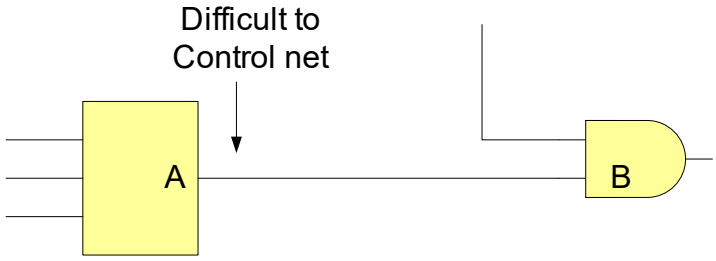


FIGURE 16. Uncontrollable Net

A fix for this case is shown in *FIGURE 17. Flip-flop added for Controllability*. The added flip-flop is scannable so it is fully controllable. The signal “testmode” will be held to “1” during capture so the point “B” is also fully controllable.

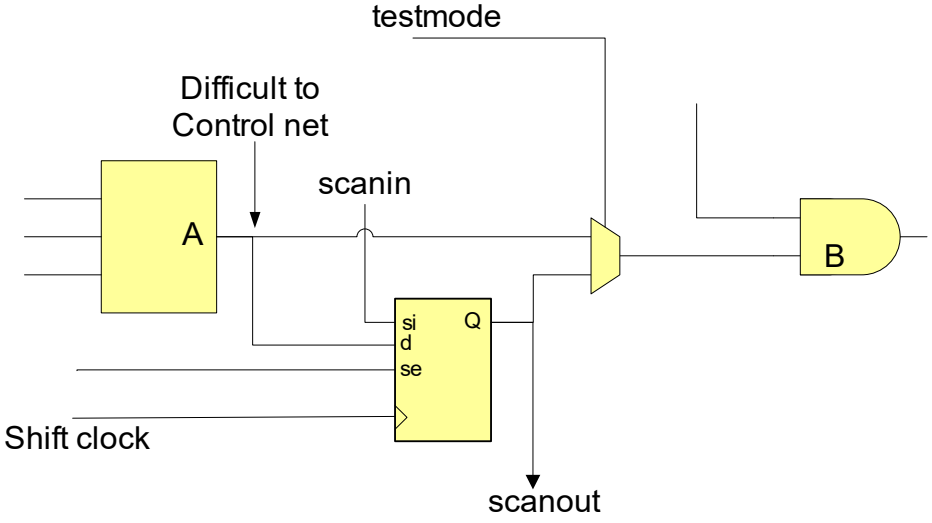


FIGURE 17. Flip-flop added for Controllability

Verifying Scan Chains

Scan chains are usually inserted into the design either during or after synthesis. Using constraints to specify the scan pins and the scan enable conditions, SpyGlass DFT can check the integrity of these chains post-synthesis. Since SpyGlass DFT is designed for both RTL input as well as netlist input, the integrity of these chains can be checked. This requires using constraints to specify the scan pins and the scan enable condition.

Assume that *FIGURE 18. Scan chain* is a portion of a scan chain with scan-in pin sin1, scan-out sout1 and scan enable se1. The constraints for this chain are:

```
scan_chain –scanin sin1 –scanout sout1-scanenable chain1
define_tag –tag chain1 –name se1 –value 1
```

A define_tag constraint (see SpyGlass DFT User Guide section “SpyGlass Constraints Currently Defined” for details) is used to define the conditions to enable a chain. There can be as many such constraints as there are distinct scan enable conditions.

A scan_chain constraint is used to define the scan-in port, the scan-out port and the define_tag for this chain.

If different chains in a design have different scan enable conditions, then multiple define_tag constraints are necessary. Scan_chain constraints for a design with 4 chains are illustrated below.

```
scan_chain –scanin sin1 –scanout sout1-scanenable chain1
define_tag –tag chain1 –name se1 –value 1
scan_chain –scanin sin2 –scanout sout2-scanenable chain2
define_tag –tag chain2 –name se1 –value 1
scan_chain –scanin sin3 –scanout sout3-scanenable chain3
define_tag –tag chain3 –name se1 –value 1
scan_chain –scanin sin4 –scanout sout4-scanenable chain4
define_tag –tag chain4 –name se1 –value 1
```

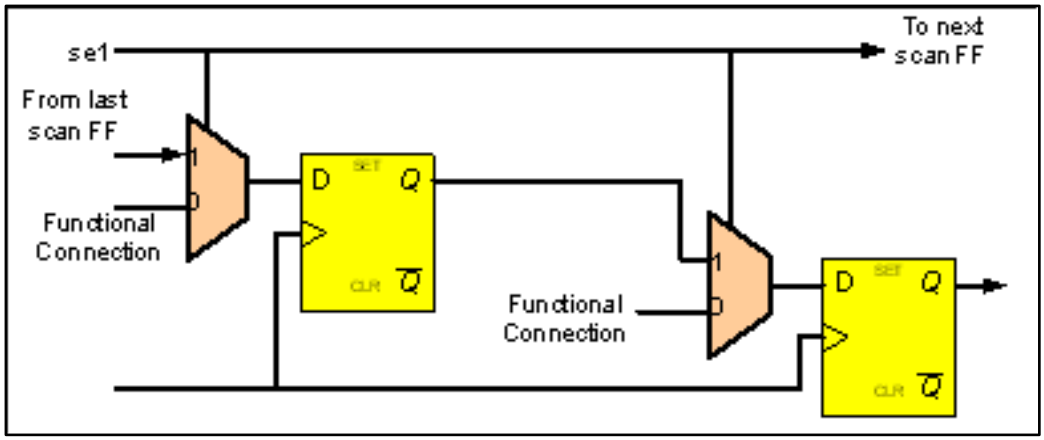


FIGURE 18. Scan chain

Verifying Test Signal Connections in Full-chip Designs

The objective is to ensure that the constraints written for the full chip satisfy the constraints already developed for the various blocks. The flow to accomplish this verification is illustrated in *FIGURE 19. Connection Verification Flow*.

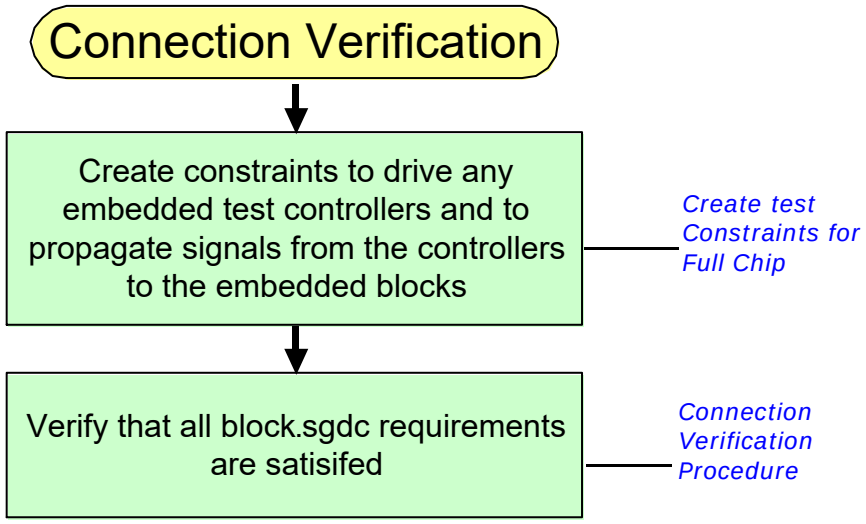


FIGURE 19. Connection Verification Flow

Create test Constraints for Full Chip

From a test point of view, sub-blocks are lower-level blocks that will be processed as a single object by ATPG tools. Often the root-level setup requirements will be unique for each sub-block. *FIGURE 20. Test Connections Checking* shows an example of a chip with three sub-blocks and a common test controller. In such cases, check test signal connections with the following strategy:

- Verify that the test controller outputs can be driven to a state necessary for a selected sub-block
- Verify that logical connections exist between the test controller and the sub-block
- Verify that the sub-block gets the correct signals from the test controller

Repeat this process for each sub-block.

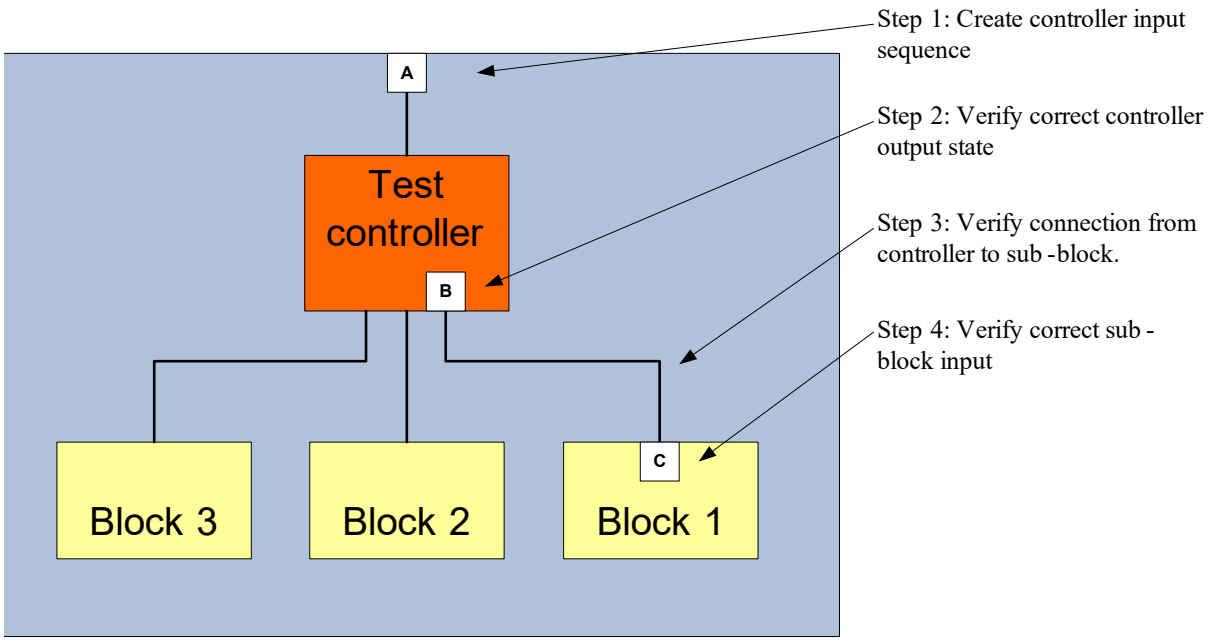


FIGURE 20. Test Connections Checking

Step 1: Drive Controller for a Block

Use define_tag constraints to specify a set of sequences on the test controller inputs that drive the test controller to a state necessary to test a sub-block. For information on using the define_tag constraint, refer to the "SpyGlass DFT Design Constraints" section in the DFT User Guide.

Example:

```
define_tag –tag block1 -name top.A -value <sequence of values for this port>
```

Step 2: Specify Controller Output

Use require_value constraints to specify values that the test controller output should have for the input sequence created in Step 1. For information on using the require_value constraint, refer to the “SpyGlass

DFT Design Constraints” section in the DFT User Guide.

Example:

```
require_value -tag block1 –name top.controller.B -value
<final value expected on these pins for the input conditions
specified in a define_tag statement.
```

Step 3: Define Controller/Block Connection Paths

Use require_path constraints to specify the controller output port and the sub-block input port that should be connected. For information on using the require_path constraint, refer to the “SpyGlass DFT Design Constraints” section in the DFT User Guide.

Example:

```
require_path -tag block1 -from top.controller.B –to
top.Block_1.C
```

Step 4: Specify Block Test Ports and Values

Use require_value constraints to specify the values that should be achieved on the sub-block input port when the controller has the state created in Step 1. For information on using the require_value constraint, refer to the “SpyGlass DFT Design Constraints” section in the *DFT User Guide*.

Example:

```
require_value -tag block1 –name top.Block_1.C -value <final
value expected on these pins for the input conditions
specified in the define_tag statement in Step 1>
```

Connection Verification Procedure

The constraint file described in the *Create test Constraints for Full Chip* section can be used to verify that the block-level requirements are satisfied. Note that if the test controller logic allows only one sub-block at a time to be tested, then a separate constraint file for each sub-block is required and multiple runs must be used.

The flow shown in *FIGURE 21. Test Connection Verification Procedure* is a step-by-step process:

- Verify that the proper controller outputs for a given sub-block are achieved.
- Verify that paths exist from the controller to the given sub-block.
- Verify that the values required at the sub-block inputs are satisfied.

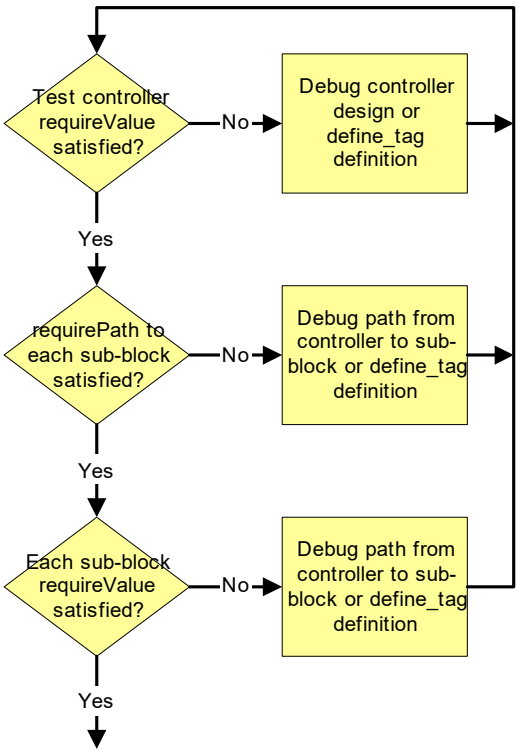


FIGURE 21. Test Connection Verification Procedure

GuideWare Methodology for DFT

The SpyGlass GuideWare methodology describes two fields of use: Block and SoC design.

Block Field of use: In this field of use, checks and goals are organized to align with the evolution and the maturity of new or re-used RTL blocks.

SoC Field of Use: The SoC integration phase includes stitching of the new RTL blocks or IPs. This field of use is divided into the following stages: Initial RTL, RTL handoff, Netlist handoff, and Layout handoff.

For a typical RTL to layout design flow, SpyGlass DFT offers the following goals:

Pre-DFT Design Stage (RTL, IP, Netlist)

- Check for DFT-Readiness of the design. This is achieved by the following goals:
  - `dft_scan_ready`: to ensure that all registers in the design can be scanned
  - `dft_best_practice`: to ensure high testability, check for estimated stuck_at fault-coverage, and means to improve it to reach the desired target
  - `dft_test_points`: to insert test points to improve testability
  - `dft_abstract`: to create a simplified abstract model of the block that can be used in lieu of the original RTL for efficient rule checking at the top/SoC level

Post-DFT Design Stage (IP, Netlist, SOC)

- For post DFT stage of the design, SpyGlass DFT offers checking using the following goals:
  - `dft_block_check`: to validate the test constraints of embedded blocks
  - `dft_scan_chain`: to validate the scan chain connectivity through the design
  - `dft_abstract_validate`: when using abstract models for one or more lower level blocks, verifies that the constraints under which the abstraction was done are met in the current design

The following tables show how the GuideWare fields of use correspond to

the DFT goals. The tables below use the following convention:

- M: Mandatory
- O: Optional (Optional goals can be accessed from *Tools/ Methodology_Configuration* menu in Console GUI)

TABLE 1 DFT: New RTL

GuideWare Stage	Goals	dft_setup	dft_scan_ready	dft_best_practice	dft_test_points	dft_abstract	dft_scan_chain
initial_rtl		O	M	M			
rtl_handoff		O	M	M	O	M	
netlist_handoff		O	M	M	O	M	M

TABLE 2 DFT: SoC

GuideWare Stage	Goals	dft_setup	dft_scab_ready	dft_best_practice	dft_test_points	dft_abstract	dft_scan_chain	dft_block_check	dft_abstract_validate
initial_rtl		O	M	M					O
rtl_handoff		O	M	M	O	M		O	O
netlist_handoff		O	M	M	O	M	O	O	O
layout_handoff		O	M	M	O		O	O	O

Step-by-step Solution

This section will describe the steps for using SpyGlass DFT in some detail.

Setup for DFT (goal name = dft_setup)

Run the setup for the dft_setup goal. This will help you to create the DFT constraints for your design, including:

- black box resolution
- test clocks
- asynchronous set/reset signals
- test modes
- PLL and clock shapers
- clock gating cells
- setting of no_scan, scan_wrap, and so on

FIGURE 22. Setup for DFT

To identify all the clocks in the design, select “Identify potential clocks in the design” tab during “Design Clocks” step, as shown below.

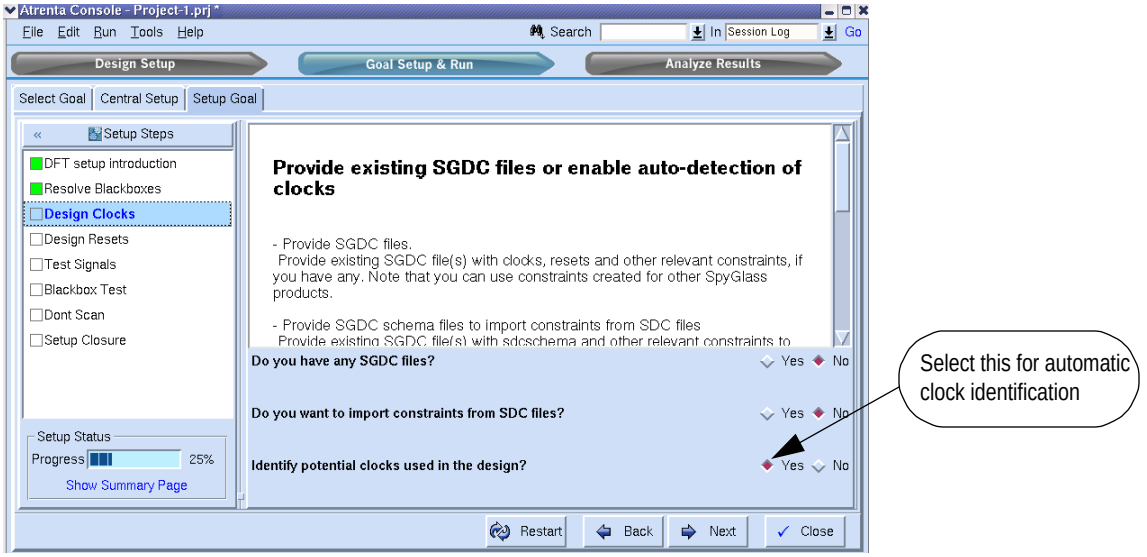


FIGURE 23. Design Clocks

This will bring up a window with different design clocks. Make sure to mark a design clock as a test clock under “DFT Mode” column, as shown below.

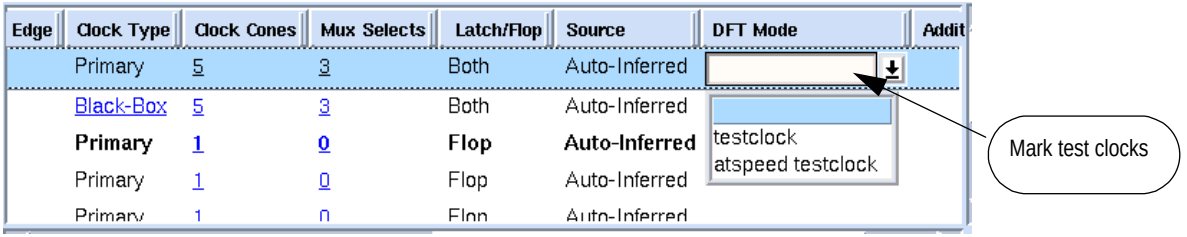


FIGURE 24. Selecting DFT Mode

The step marked as “Design Resets” can be used to automatically identify async set/reset signals in the design.

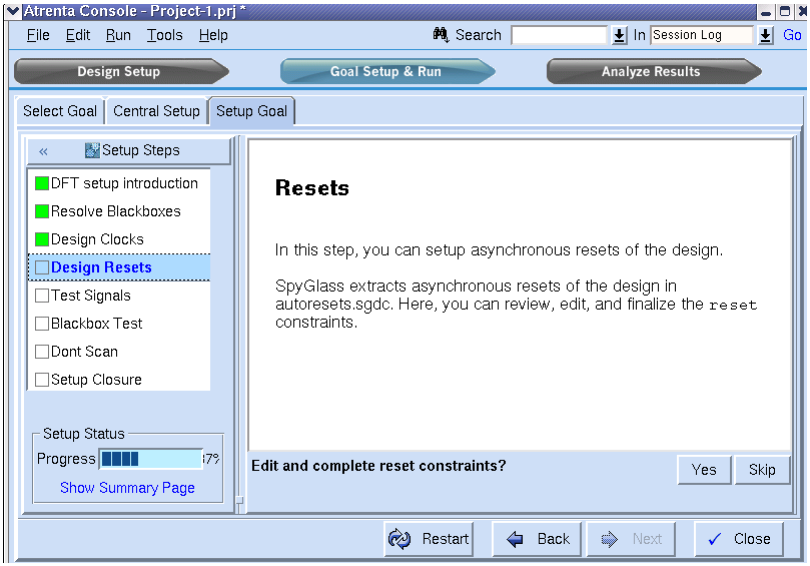


FIGURE 25. Design Resets

Any async set/reset automatically gets defined as a test mode signal of the opposite active value (to keep them disabled during scan-shift). Any additional test signals can be directly added in the test-mode file (the left-most window shown below).

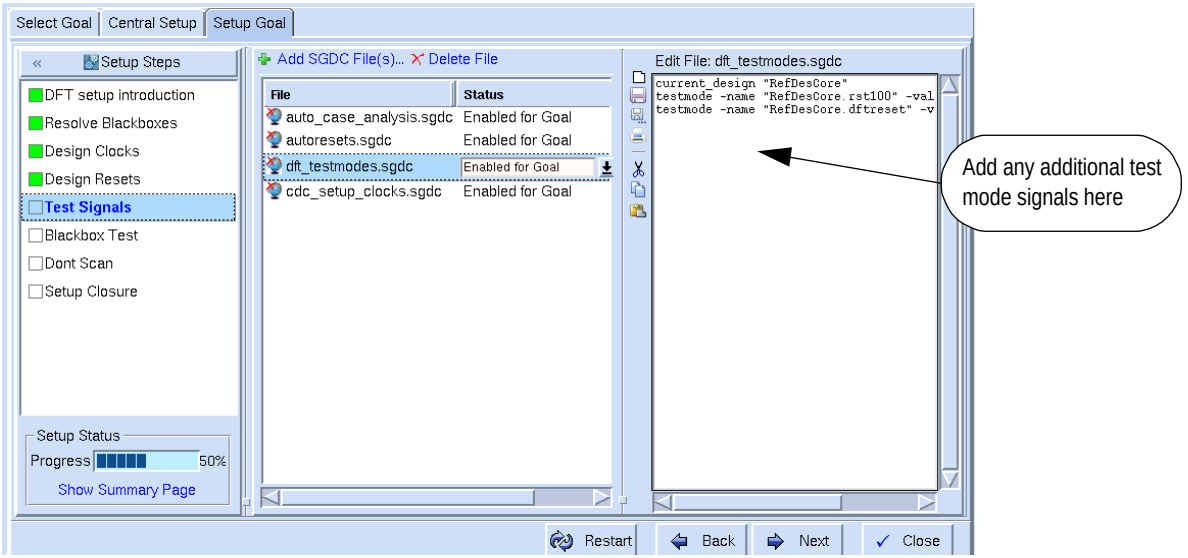


FIGURE 26. Adding additional test signals

The next step in the DFT Setup Manager will let user define various other constraints – usually applicable to black boxes in the design, such as PLL, clock_shaper, scan_wrap, gating_cell, and so on.

Finally, you can identify modules with `no_scan` constraint, and close the setup manager by saving the created SGDC constraint files. Make sure that the generated SGDC constraint files are enabled for any DFT goal that you run on the design. If necessary, you can regenerate these files, or edit them prior to running any goal through the Setup Manager.

If you are not using Console GUI and DFT Setup Manager, you can use the following steps to manually create the necessary setup for DFT analysis.

Create necessary *clock* Constraints

Identify all test clocks in the design using testclock constraint. For example:

```
clock –name tClk1 –testclock
```

Create necessary *test_mode* Constraints

Identify the test mode signals using `test_mode` constraint. Note that in addition to the normal test signals, you also need to declare set/reset signals as testmode signals. Set/reset signals are normally held off during scan shift operation.

For example, if the design has the following set/reset signal constraints -

```
reset -name "RefDesCore.rst100"   -value 0
reset -name "RefDesCore.dftreset" -value 0
```

Here the values indicate the 'active' value for reset. We will need to ensure that the signals that are acting as 'reset' are 'inactive' during scanshift mode. So we need to generate the following constraints (note the 'inactive' values):

```
test_mode -name "RefDesCore.rst100"   -value 1 -scanshift
test_mode -name "RefDesCore.dftreset" -value 1 -scanshift
```

Skip this step of creation of 'test_mode' constraints if they are already known.

Achieve Scannability (goal name = dft_scan_ready)

- Run the `dft_scan_ready` goal.
- A flip-flop is considered as scannable if during scanshift:
  - its clock can be controlled by a test clock (checked by DFT rule Clock_11) and
  - its set and reset pins (if any) are forced inactive (checked by DFT rule Async_07)
- These two rules may be violated either because test logic has not yet been designed in or because the constraint file has missing or incorrect entries. If there are violations of either rule, then diagnostic rules Diagnose_testclock and Diagnose_testmode can be used to diagnose the cause.
- Any non-scannable flip-flop will reduce the coverage for logic that only feeds that flip-flop as well as logic that is only driven by that flip-flop.

- The `dft_scan_ready` goal also checks that testmode signals that only control asynchronous set or reset pins should be unrestricted during capture. (checked by DFT rule Async_08)
  - Restricting such a dedicated signal would result in the set/reset nets not being tested thoroughly.
  - Such a violation can be fixed by adding the `-scanshift` argument to the `test_mode` constraint to indicate that the constraint only applies during shifting and is a don't care otherwise.

Clock_11 Debug

Clock_11 violations detect clock sources (see the description of Clock_11 in the SpyGlass DFT User Guide section on Clock Rules) that are not controlled by testclocks. Each violation indicates the number of flip-flops clocked by this source. Selecting any violation will highlight the source on the schematic. The following is an example shown in the Incremental Schematic.

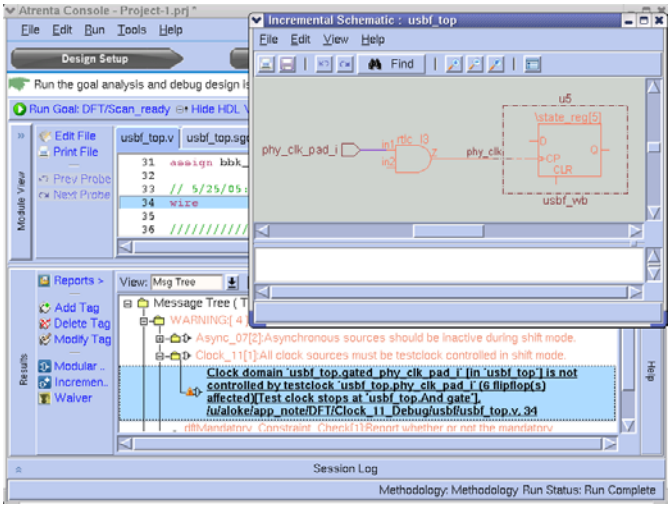


FIGURE 27. Clock_11 violations

The testclock propagation through the 'Show Case Analysis' mechanism

appears automatically in the schematic.

Points that block testclock propagation will have a pulse symbol, that is, either a ^ or a v, on a device input but no clock pulse symbol on the device output:

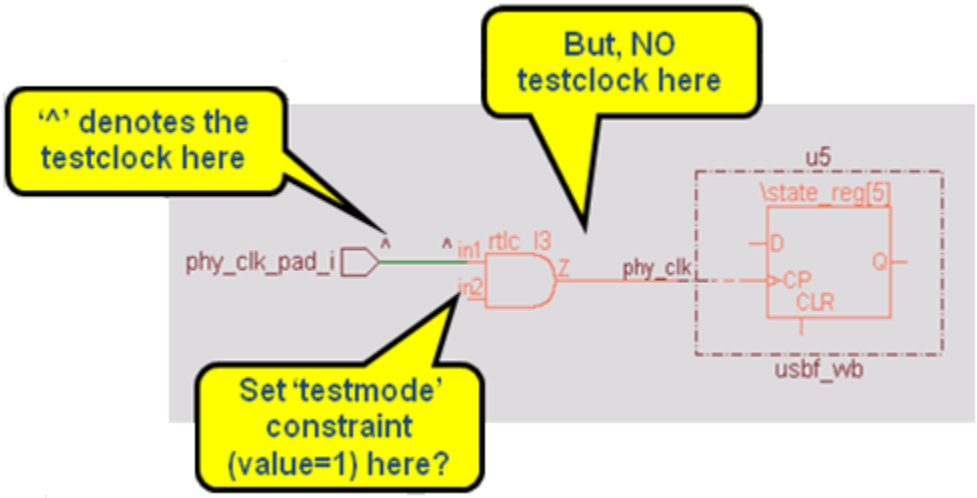


FIGURE 28. Show Case Analysis

In this example, the bottom input to the AND gate should be held at '1' to enable the clock to pass through. Hover the cursor on such a pin. Then, right-click and select Show Input Cone option to either primary inputs or flip-flops. Apply one or more constraints so that the device input pin has a value that will enable the clock path.

Diagnose_testclock

- Select a violation and display in the IS
- The testclock source and a node that blocks testclock propagation are displayed
- The input pin on the blocking and the value on that pin are also displayed

- Right clicking on that pin and selecting “Show input cone” can be used to display the logic driving the blocking pin
- Right clicking and selecting “Show debug data” and then selecting “DFT” will display values on a pin
- Use this data to determine how the clock was blocked and where constraints or the blocking logic could be modified

Info_testclock

This rule displays testclock propagation for both scanshift and capture testmode.

- Select the violation message for scanshift if there are Clock_11 violations and display in the IS
- Select the violation message for capture if there are Clock_11_capture violations and display in the IS
- The display will show either “^” for positive going clock pulse or a “V” for a negative going pulse
- The displayed propagation may be useful to determine how a testclock is reaching a particular or why a particular phase of a testclock is created

Async_07 Debug

Async_07 violations detect async sources (see the description of Async_07 in the SpyGlass DFT User Guide section on Asynchronous Rules) that are not rendered inactive during scanshift.

- Select a violation and display in the MS.
- The testmode value propagation appears automatically on the schematic.
- Visually find out the root cause why the async source of the flip-flops is held at ‘X’ or at the active value. This will lead to the conclusion about how possibly a test_mode constraint can be applied to rectify the Async_07 violation.

In the example below, the schematic shows no constraint was applied on the rst input pin of the design:

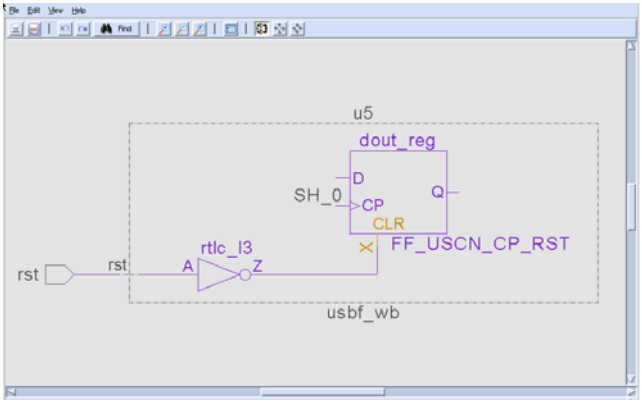


FIGURE 29. Async_07 violations

Async_08 debug

Async_08 detects connections to scannable flip-flop set or reset pins that are not fully controllable.

- Select a violation and display in the IS.
- The schematic will be back annotated with testmode capture conditions and include the effects of power and ground
- Use the displayed controllability values to determine the root cause for the incomplete controllability (nn-, ny- or yn-) at the flip-flop set or rest pin
- Consider changing a constraint or possibly modifying the design

Diagnose_testmode

- Select a violation and display in the IS
- Right click on a highlighted pin to see the values forced on that pin by either a testmode constraint or the result of a power or ground simulation

NOTE: *Since Diagnose_testmode identifies blocking gates, debug data will not be displayed on module boundaries. The display only operates on primitive gate input pins.*

- Use “Show input cone” to trace to the source of the fixed value
- Consider either changing a constraint or possibly modifying the logic

Info_testmode

- Select a violation and display in the IS
- The IS schematic will show all signals with non-x testmode values and therefore this may be difficult to use in large designs; in that case Diagnose_testmode may be more practical
- The values forced by either a testmode constraint or the result of a power or ground simulation
- This information may be useful to ensure that testmode is operating as required

Info_scanwrap debug

This informational rule lists design units that are declared in a scanwrap constraint but the enable pin, declared in the scanwrap constraint, is not active. Display the design unit and use Info_testmode. Trace the fanin cone of the enable pin to determine what change is required to force an enable value on this pin.

Info_noscan debug

This rule displays flip-flops that have been declared as noscan by a noscan constraint.

This rule may be useful when trying to determine why faults are uncontrollable or unobservable. Both conditions are necessary for fault detection.

Info_inferredNoscan debug

This rule displays flip-flops that have been inferred as noscan. This rule may be useful when trying to determine why faults are uncontrollable or unobservable. Both conditions are necessary for fault detection.

A flip-flop is inferred as noscan if it feed an asynchronous pin of another flip-flop or has a specified value from a testmode constraint on it’s output.

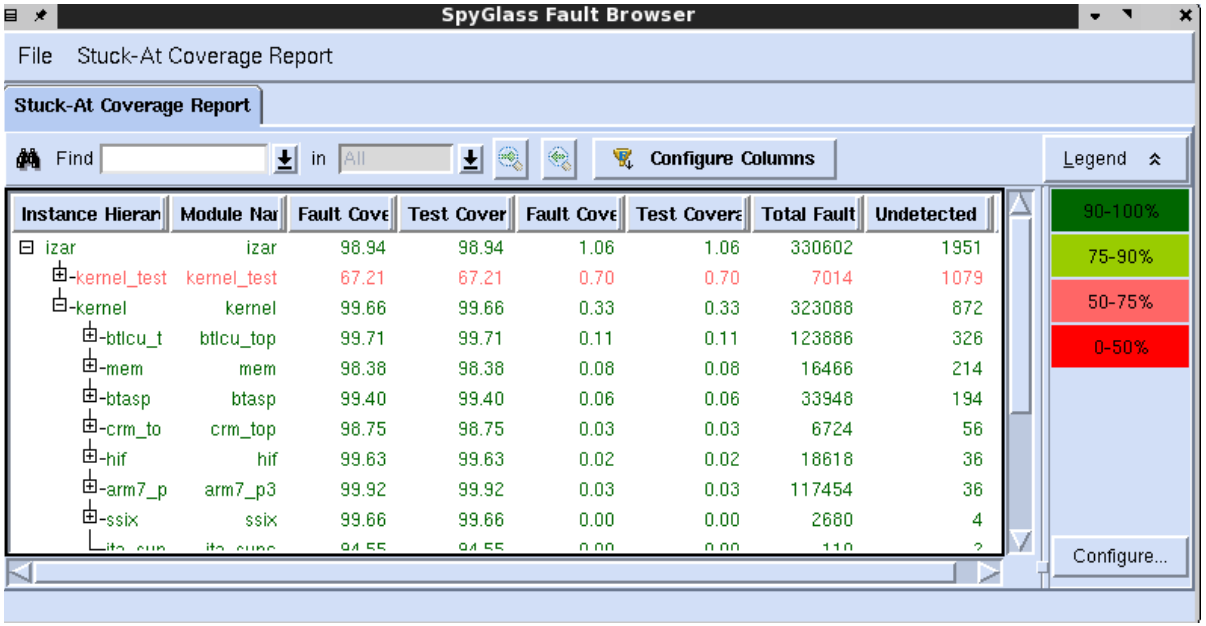
The reasons are:

- The clock -testclock constraint is specified on its output.
- The test_mode constraint is specified on its output.
- It's output gets a non-X ('0') test_mode value through sequential propagation
- It is driving an asynchronous pin of a scan flip-flop.

This rule may be useful when trying to determine why faults are uncontrollable or unobservable. Both conditions are necessary for fault detection.

Viewing the estimate of fault coverage of the design

The rule *Info_coverage* estimates the fault/test coverage of the design. The generated reports help in understanding the test health of the design. The following fault browser helps understand the relative testability scores achieved in the design.



Instance Hieran	Module Nar	Fault Cove	Test Cover	Fault Cove	Test Cover	Total Fault	Undetected
izar	izar	98.94	98.94	1.06	1.06	330602	1951
-kernel_test	kernel_test	67.21	67.21	0.70	0.70	7014	1079
-kernel	kernel	99.66	99.66	0.33	0.33	323088	872
-btlcu_t	btlcu_top	99.71	99.71	0.11	0.11	123886	326
-mem	mem	98.38	98.38	0.08	0.08	16466	214
-btasp	btasp	99.40	99.40	0.06	0.06	33948	194
-crm_to	crm_top	98.75	98.75	0.03	0.03	6724	56
-hif	hif	99.63	99.63	0.02	0.02	18618	36
-arm7_p	arm7_p3	99.92	99.92	0.03	0.03	117454	36
-ssix	ssix	99.66	99.66	0.00	0.00	2680	4

FIGURE 30. Fault coverage of the design

The following summary reported generated at this stage helps understand the fault status.

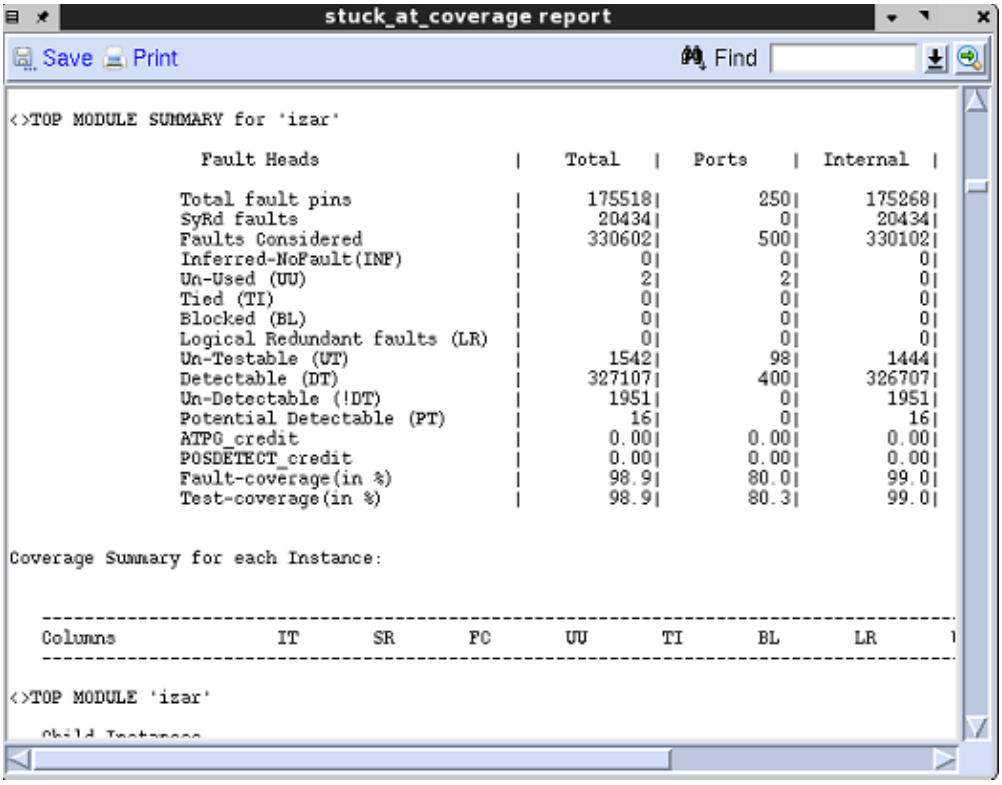
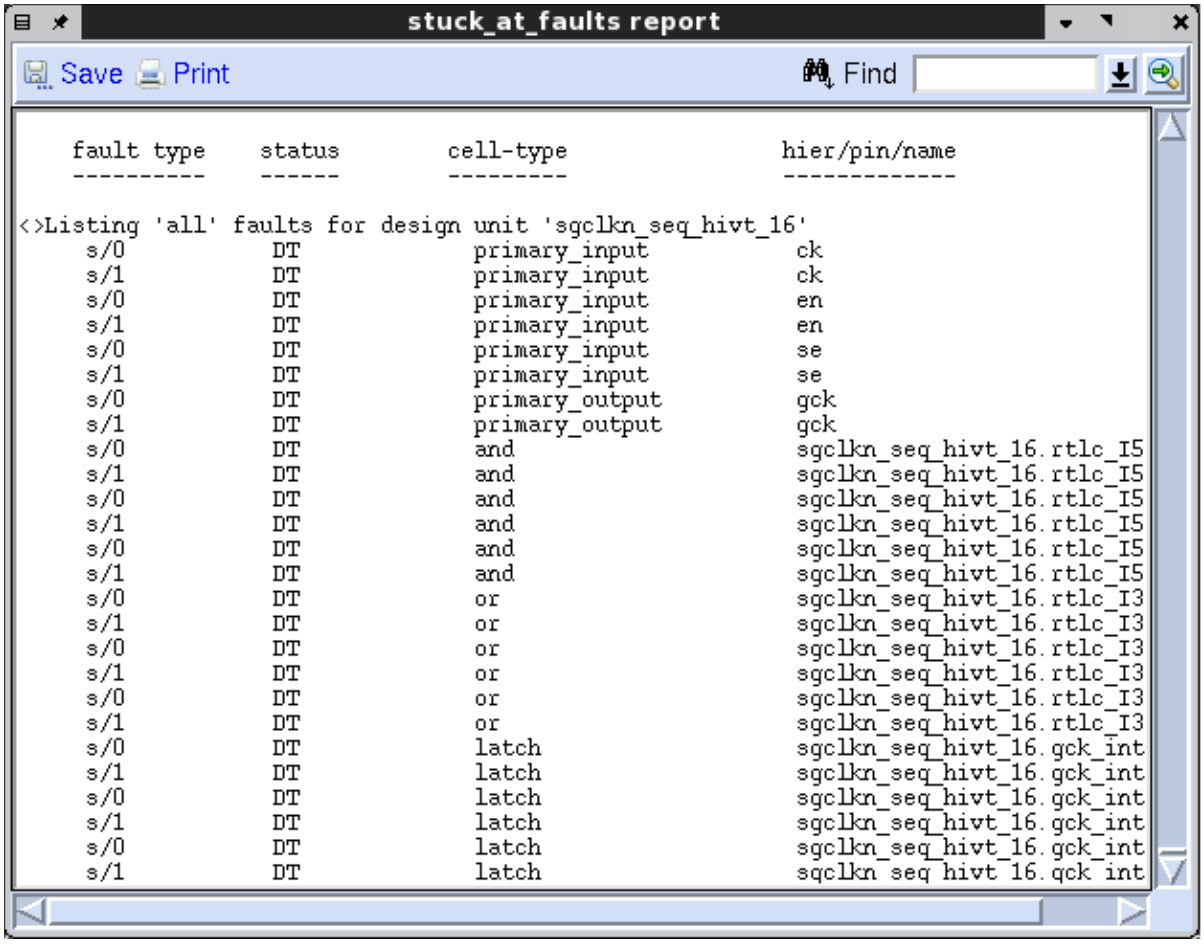


FIGURE 31. Fault status

You can generate a detailed fault report by setting the dftGenerateStuckAtFaultReport parameter using the following command:

```
set_parameter dftGenerateStuckAtFaultReport all
```




```
    fault type    status        cell-type                hier/pin/name
    ----------    ------        ---------                -------------

<>Listing 'all' faults for design unit 'sgclkn_seq_hivt_16'
     s/0           DT             primary_input            ck
     s/1           DT             primary_input            ck
     s/0           DT             primary_input            en
     s/1           DT             primary_input            en
     s/0           DT             primary_input            se
     s/1           DT             primary_input            se
     s/0           DT             primary_output           gck
     s/1           DT             primary_output           gck
     s/0           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/1           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/0           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/1           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/0           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/1           DT             and                      sgclkn_seq_hivt_16.rtlc_I5
     s/0           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/1           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/0           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/1           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/0           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/1           DT             or                       sgclkn_seq_hivt_16.rtlc_I3
     s/0           DT             latch                    sgclkn_seq_hivt_16.gck_int
     s/1           DT             latch                    sgclkn_seq_hivt_16.gck_int
     s/0           DT             latch                    sgclkn_seq_hivt_16.gck_int
     s/1           DT             latch                    sgclkn_seq_hivt_16.gck_int
     s/0           DT             latch                    sgclkn_seq_hivt_16.gck_int
     s/1           DT             latch                    sgclkn_seq_hivt_16.gck_int
```

FIGURE 32. Detailed fault report

Ensure Compliance to DFT Best Practices (goal name = dft_best_practice)

To ensure compliance to the DFT best practices, perform the following steps:

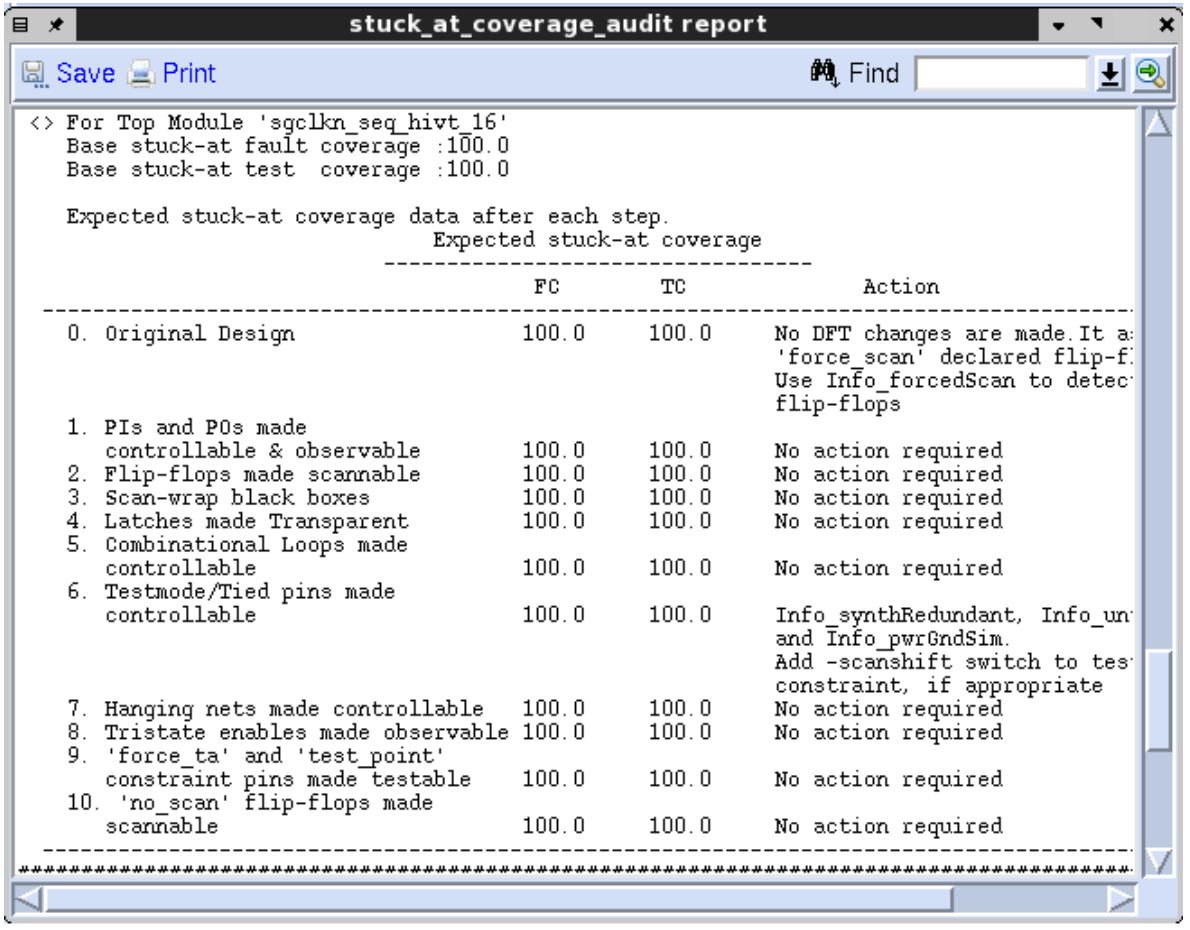
1. Run the *dft_best_practice* goal.
2. Review the *stuck_at_coverage_audit* report.

Review the stuck_at_coverage_audit report

Use the *stuck_at_coverage_audit* report to increase coverage.

The objective of the first section of this report is to increase coverage. The objective of the second section of this report is to ensure that ATPG generated tests operate as expected.

The following figure illustrates an example of the first section of this report:




```
<> For Top Module 'sgclkn_seq_hivt_16'
   Base stuck-at fault coverage :100.0
   Base stuck-at test  coverage :100.0

   Expected stuck-at coverage data after each step.
                              Expected stuck-at coverage
                            ---------------------------------
                                        FC        TC              Action
 ---------------------------------------------------------------------------------------
   0. Original Design                 100.0     100.0     No DFT changes are made. It a
                                                          'force_scan' declared flip-f
                                                          Use Info_forcedScan to detec
                                                          flip-flops
   1. PIs and POs made
      controllable & observable       100.0     100.0     No action required
   2. Flip-flops made scannable       100.0     100.0     No action required
   3. Scan-wrap black boxes           100.0     100.0     No action required
   4. Latches made Transparent        100.0     100.0     No action required
   5. Combinational Loops made
      controllable                    100.0     100.0     No action required
   6. Testmode/Tied pins made
      controllable                    100.0     100.0     Info_synthRedundant, Info_un
                                                          and Info_pwrGndSim.
                                                          Add -scanshift switch to tes
                                                          constraint, if appropriate
   7. Hanging nets made controllable  100.0     100.0     No action required
   8. Tristate enables made observable 100.0    100.0     No action required
   9. 'force_ta' and 'test_point'
      constraint pins made testable   100.0     100.0     No action required
  10. 'no_scan' flip-flops made
      scannable                       100.0     100.0     No action required
 ---------------------------------------------------------------------------------------
```

FIGURE 33. Example of the stuck_at_coverage_audit report

The report indicates the current coverage for the design and the steps that you need to follow to improve the coverage. The following sections, listed in report order, describe the diagnostic procedure for each fault category in the report.

Make flip-flops scannable

- Flip-flop scannability depends on the flip-flop clocked by a testclock and having it's set and reset pins disabled in scan shift mode.
- See section *Clock_11 Debug* for a flip-flop that is not clocked by a testclock during scan shift.
- See section *Async_07 Debug* for a flip-flop with set or rest pins not forced inactive during scan shift.

Make Latches Transparent

- The Latch_08 rule detects latches that are not transparent in the capture mode. You can use the Info_testmode rule for capture to detect the non-transparent latches.
- Select and double-click a Latch_08 violation to view the corresponding Incremental Schematic.
- If the latch enable has a non-X but an inactive value then either a test_mode constraint should have the complementary value, or some device in the fan-in to this latch enable should produce the complementary value.
- If the latch enable has no value, then either the logic feeding this enable should be modified (see *FIGURE 12. Reset bypass example* for an example) so that the enable is forced active during capture or new test_mode constraints must be defined.

Scan-wrap black boxes

- Select and double-click a Info_scanwrap violation to view the corresponding Incremental Schematic.
- Right click and select "Set SGDC constraints on module" which will invoke the SGDC Constraint Editor
- Set the Select Constraint box to scan_wrap
- Enter the black box module name
- Enter the pin(s) that control the scan wrapper
- Enter the value(s) that causes the wrapper to enter scan mode

- When finished, click “Append” to add the newly created constraint to the SGDC
- Rerun SpyGlass

Combinational Loops Made Transparent

- Select and double-click a Topology_01 violation to view the corresponding Incremental Schematic
- The circuit or constraints must be modified so the loop is broken during capture
- The logic driving loop inputs can be traced by right clicking a loop input
- Constraints may be added by right clicking on a net driving a loop input and selecting Set SGDC constraints

Testmode/Tied pins made controllable

- Select Info_untestable and then click on Incremental Schematic to view faults blocked by testmode constraints
- Consider modifying a constraint on the pin listed in the violation message
- Select Info_Info_synthRedundant and then click on Incremental Schematic to view faults blocked by connections to power or ground
- Consider modifying the logic

Info_Untestable debug

- This rule displays faults rendered untestable due to blocking by a testmode signal or forced to a constant value by a testmode signal. Blocked faults are marked as BL and tied faults are marked as TI. Marking will be displayed as xx/yy where xx and yy are BL, TI or blank. For example, a node is TI/BL if the stuck at 0 fault on this node is TI and the stuck at 1 fault on this node is BL. A node is /TI if the stuck @ 0 fault is testable and the stuck at 1 fault is tied.
- This rule does not necessarily imply a problem. It is available so that all faults may be accounted for.

- TI (Tied) and BL (Blocked) faults are caused by only power-ground simulation. Faults which are blocked by test_mode are marked as “ND or !DT” (Not Detected). Fault on the node itself (on which test_mode is specified) is marked as “UT (untestable)”

Info_SynthRedundant debug

- If the source files are RTL, then this rule detects logic that is likely to be removed by synthesis and therefore faults in this logic can be ignored since they are excluded from both the fault coverage calculation and the test coverage calculation
- If the source files are at the netlist level and therefore obtained after synthesis then synthesis redundancies have already been removed. In this case, Info_Synthredundant detects faults that are untestable due to blocked paths caused by connections tied to ground
- In either case, there is usually nothing wrong with the design. This rule is available so that all faults may be accounted for

Hanging nets made controllable

- Select and double-click a TA_09 violation to view the corresponding Incremental Schematic
- The violation message identifies the module and the unconnected pin
- Consider adding a test point to the pin or modifying the module design

Tristate nets made observable

- Select and double-click a TA_09 violation to view the corresponding Incremental Schematic
- Consider adding an observation test point to the enable net or adding a pullup to the tristate output

The *force_ta* nets and *test_point* constraint pins made testable

- Check sgdc file
- Use Info_uncontrollable to identify uncontrollable nets. Consider changing the force_ta constraint or modifying the fanout of force_ta pin

- Use Info_unobservable

Info_uncontrollable debug

- This rule displays the controllability status for nodes that are not completely controllable during capture. Uncontrollability is displayed as three y or n characters. For example, a node is ynn if it can be controlled to 0 but cannot be controlled to a 1 or to z
- This rule does not necessarily imply a problem. It is available to help understand the reason that faults are not testable

Info_unobservable debug

- This rule displays nodes that are not observable during capture. Unobservability is displayed as N
- This rule does not necessarily imply a problem. It is to help understand the reason that faults are not testable

The *no-scan* flip-flops made scannable

- Select and double-click a Info_noscan violation to view the corresponding Incremental Schematic
- Consider removing the noscan constraint

Using the Coverage_Audit report to ensure test operation

The second section of the Audit-Coverage report covers rules that detect conditions which, if not fixed, can prevent ATPG tests from operating as expected. As a result, actual coverage may be less than the result predicted by ATPG tools or tests may fail even when a chip is operating correctly.

The rules in this section of the report are sorted by the number of flip-flops affected by each rule. Flip-flop count is used instead of computing possible change in coverage because of runtime consideration.

Async_02 violations

- Select an Async_02 violation and view results in Incremental Schematic.

- A path from a flip-flop output to a flip-flop set or reset pin is displayed.
- The displayed path should be blocked in the capture mode.
- Hovering the cursor over any non-path input and schematic log window displays information about this input.
- Double click on this input to display it's fanin cone
- Eliminate this violation by either using testmode to block this path or by changing the logic

Async_11 violations

- Select an Async_11 violation opens a spreadsheet viewer window
- The spreadsheet lists all the data pins and all the flip-flop set or reset pins reached by this violation

NOTE: *An Async_11 violation occurs when a pin fans out to at least one flip-flop data pin and at least one flip-flop set or reset pin as a violation. To fix a violation, all paths to data pins must be blocked or all paths to set/reset pins must be blocked.*

NOTE: *Usually, the easiest way to fix an Async_11 violation is to select the pin type (either data or set/reset) with the fewest number of destinations. In this way, we would expect to have to make the fewest number of changes.*

- Select a pin and click on Incremental Schematic
- A path from the violation source pin to either a data pin or a set/reset pin will be displayed
- The displayed path should be blocked in the capture mode.
- Hover the cursor over any non- path input and schematic log window will provide information about this input
- Double click on this input to display it's fanin cone
- Eliminate this violation by either using testmode to block this path or by changing the logic

Clock_04 violations

- Select a Clock_04 violation that opens a spreadsheet viewer window
- The spreadsheet will list all clock pins that are used as data pins

- Select a pin in the spreadsheet and click on the Incremental Schematic
- A path from the clock and the path to a flip-flop d-pin is displayed

NOTE: *A clock_04 violation occurs for any pin declared in a clock constraint with an unblocked path to a flip-flop d-pin regardless of whether or not this pin is actually used as a clock.*

- Hover the cursor over any non- path input and schematic log window will provide information about this input
- Double-click on this input to display it's fanin cone
- Eliminate this violation by either using testmode to block this path, changing the logic or removing the clock constraint

Clock_08 violations

- Select a Clock_08 violation
- A path from a clock to a flip-flop d-pin is displayed

NOTE: *A clock_08 violation occurs for any pin declared in a clock constraint with an unblocked path to a flip-flop d-pin regardless of whether or not this pin is actually used as a clock.*

- Hover the cursor over any non-path input and schematic log window provides information about this input
- Double click on an input to display it's fan-in cone
- Eliminate Clock_08 violations by using testmode to block this path, changing the logic or removing the clock constraint

Clock_16 violations

- Select a Clock_16 violation
- Paths with opposite inversion parity from a clock to flip-flop d-pins is displayed
- Hover the cursor over any non-path input and schematic log displays information about this input
- Double-click on an input to display it's fan-in cone
- Eliminate Clock_16 violations by changing the clock logic so that inversion is removed during capture or use testmode to block this path

Clock_17 violations

- Select a Clock_17 violation
- Paths from a clock to flip-flop whose output gates the same clock to a second flip-flop is displayed
- Hover the cursor over any non- path input and schematic log window will provide information about this input
- Double-click on an input to display it's fan-in cone
- Eliminate Clock_17 violations by changing the clock logic so that the clock gating path is blocked during capture

Clock_21 violations

- Select a Clock_21 violation which will open a spreadsheet viewer window
- The spreadsheet will list all clock pins that are used as set or reset pins
- Select a pin in the spreadsheet and click on the Incremental Schematic
- A path from the clock to a flip-flop set or reset pin is displayed

NOTE: *A Clock_21 violation occurs for any pin declared in a clock constraint with an unblocked path to a flip-flop d-pin regardless of whether or not this pin is actually used as a clock.*

- Hover the cursor over any non-path input and schematic log window provides information about this input
- Double-click on this input to display it's fan-in cone
- Eliminate this violation by either using testmode to block this path, changing the logic or removing the clock constraint

Clock_27 violations

- Select a Clock_27 violation and click on the Incremental Schematic
- A path from a clock through a CGC to a flip-flop clock pin is displayed

NOTE: *A Clock_27 violation occurs when clock edge, expected at the flip-flop clock pin, cannot be produced by the CGC*

- Eliminate this violation by either changing the CGC edge type or changing the inversion parity of the clock path between the CGC and the flip-flop

Clock_28 violations

- Select a Clock_28 violation and select the Incremental Schematic
- Clocks that drive a flip-flop clock pin through re-convergent paths or clock paths that have re-convergent enables is displayed
- Hover the cursor over any non-path input and schematic log window provides information about this input
- Double-click on an input to display it's fan-in cone
- Eliminate Clock_28 violations by changing the clock logic to block all but one of the re-convergent paths during capture

Scan_07 violations

- Select a Scan_07 violation and select the Incremental Schematic
- Sequentially derived internal signals declared as a testmode is displayed
- Hover the cursor over any non-path input and schematic log window provides information about this input
- Double-click on an input to display it's fan-in cone
- Eliminate Scan_07 violations by changing the clock logic to block all but one of the re-convergent paths during capture

Scan_22 violations

- Select a Scan_22 violation and display the Incremental Schematic
- The portion of the scan chain that spans the domain crossing as well as the clocks feeding these scan cells is displayed
- Eliminate Scan_22 violations by inserting a lockup latch at the domain crossing

Topology_03 violations

- Select a Topology_03 violation and display the Incremental Schematic
- A flip-flop and an unblocked path to a second flip-flop set or reset pin is displayed
- Eliminate the Topology_03 violations by blocking the path in scan mode

Topology_05 violations

- Select a Topology_05 and display the Incremental Schematic
- The devices wire-ANDed or wire-ORed is displayed
- Eliminate the Topology_05 violations by changing the logic to eliminate the wired connection

Topology_13 violations

- Select a Topology_13 violation and display the Incremental Schematic
- The re-convergence logic and the path from the re-converge node to a flip-flop set or reset pin is displayed
- Eliminate the Topology_13 violation by changing the logic to block all but one of the re-converging paths in capture mode. Blocked paths may cause a reduction in coverage in which case consider use of test points

Tristate_06 violations

- Select a Tristate_06 violation and display the Incremental Schematic
- The wired net, it's tristate drivers and the enable control nodes are displayed
- The violation message identifies the problem such as no drivers on or more than one driver on
- Eliminate Tristate_06 violations by changing the enable decode logic so that it is fully decoded

Achieve BIST Readiness (goal name = dft_bist_ready)

- Run the `dft_bist_ready` goal. The dft_bist_ready goal can be run from the DFT sub-methodology. You can select it in the “Goal Setup and Run” tab of the GUI, by clicking on the “Select Methodology” link. In the pop-up window, select “SpyGlass Sub-Methodology” and click OK. The DFT goals will appear in the left window, as shown below.

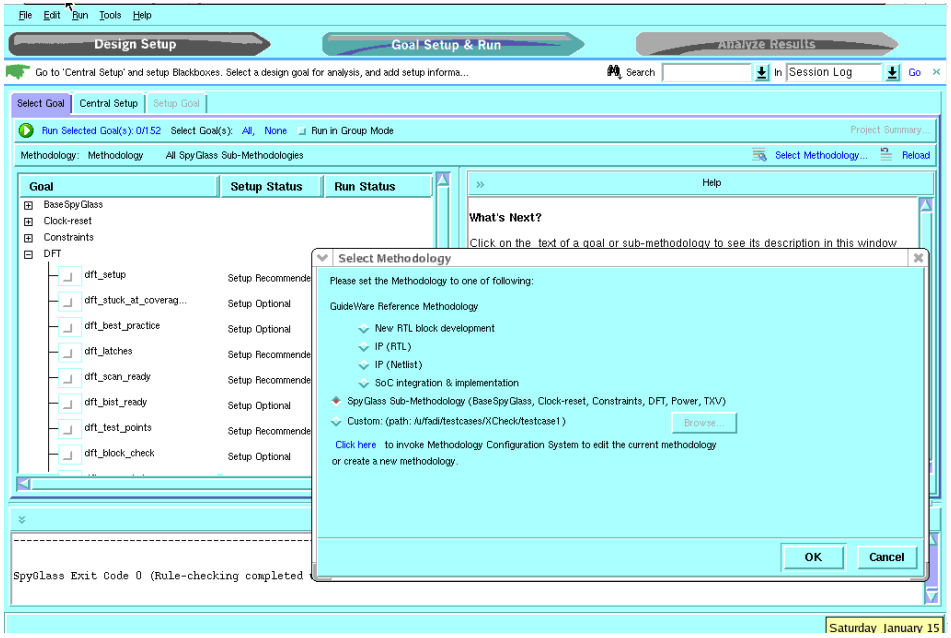


FIGURE 34. DFT goals

- In BIST (Built-In-Self-Test) designs, the scan output is compressed into a signature register, and capturing unknown values will corrupt the signature.
- A design is BIST ready when no unknown values (“X”) are captured in the scannable flip-flops of the design. This is because in BIST (Built-In-Self-Test) designs, the scan output is compressed into a signature register, and capturing unknown values will corrupt the signature. This can also be the case when ATPG Compression is used, so you should

consider running this goal also for designs using such Compression techniques.

- The dft_bist_ready goal checks the following rules:
  - **BIST_01**: Flip-flops that have more than the specified number of flip-flops and black boxes in their fan-in cones
  - **BIST_02**: Flip-flops that are driven by a gate instance with large fan-in
  - **BIST_03**: Flip-flops that remain in unknown state after initialization
  - **BIST_04**: Primary outputs or inout ports and data pins of scannable flip-flops that have unknown values in their fan-in cones
  - **BIST_05**: In scan node, have TIE-X cells outputs bypassed
- It also runs the Info_testmode and Info_testclock rules to document the propagation of the testclock and testmode signal to help debug rule violations.

Adding Testpoints (goal name = dft_test_points)

TA_09 debug

- Run the `dft_test_points` goal.
- The TA_09 generates a report test_points_selected_2.rpt that lists all suggested test points. The report is sorted in descending order for the number of nets affected by the test point and also lists the coverage that would be obtained if all test points up to that place in the list are used.
- It is often the case that not all of the test points are required. TA_09 also produces a constraint file called test_points_selected_2.rpt that contains a test point constraint (see DFT User Guide section SpyGlass DFT Constraints Currently Defined) for each test point in the report. This file can be edited to remove undesired test points--for example, test points in IP blocks or in critical timing areas of the design.
- Save the edited report in the working directory and rerun SpyGlass DFT. The Info_coverage rule indicates the coverage as if the design was modified for the selected set of test points.

- Once a selection of test points has been made, then the RTL files can be edited as illustrated in *FIGURE 17. Flip-flop added for Controllability* and *FIGURE 18. Scan chain*.

Validating Scan Chains (goal name = dft_scan_chain)

Scan chains must be specified using scan_chain constraint as described in the *Verifying Scan Chains* section.

Scan_22

Scan_22 will flag incomplete chains and highlight potential places where the chains are not properly connected.

- Click Soc_04. Select a violation of Scan_22.
- Soc_04 will highlight the scannable values defined for this chain. If the scan mux at the point of the Scan_22 violation is not correct, then either change the Define_tag definition or change the connections to this scan mux.

Scan_24

Scan_24 identifies flip-flops that are not part of any chain.

If any of these flip-flops should on a chain, verify that the scan enable conditions for this chain are defined in a define_tag constraint.

Scan_25

Scan_25 identifies chains that contain data in version. Violations may be caused by incorrectly wired scan chains scan flip-flop models.

Scan_26

Scan_26 identifies scan chains that do not have lockup latches driving the scan-out pin. The cause may be a design error in which the latch was never inserted into the design or a wiring error such that the latch is not wired in

a lockup configuration.

- Display the scan-out pin, as described in this scan_chain constraint, in the IS.
- Probing on this pin will determine whether or not a latch is in the design. If not present, then the design must be changed.
- If incorrect values or if no values are reaching the latch, then there is a connection problem associated with this latch. Probing the latch enable pin in the IS should reveal the cause.

Info_scanchain

This informational rule generates the scan chain information as available in the design for debug purpose. It generates appropriate schematic information for scan chain viewing. It also generates text report to list all the flip-flops in the scan chains as shown in the following figure.

```
#Section 1 of top module 'RefDesCore': BEGIN

scanchain_id     : chain_1
scanout          : RefDesCore.usb_susp_o
scanin           : RefDesCore.scan_in1
clock domains    : 1 (RefDesCore.clk100)
scanchain length : 25720
```

cell_order	scan_chain_id	cell_type	shift_clock	clock_polarity	hier_name
0000	chain_1	<flop>	RefDesCore.clk100	(+)	RefDesCore.wb_s2_usb2.susp_o_reg
0001	chain_1	<flop>	RefDesCore.clk100	(+)	RefDesCore.wb_s2_usb2.u4.intb_reg
0002	chain_1	<flop>	RefDesCore.clk100	(+)	RefDesCore.wb_s2_usb2.u4.\int_srcb_re
0003	chain_1	<flop>	RefDesCore.clk100	(+)	RefDesCore.wb_s2_usb2.u4.\int_srcb_reg

FIGURE 35. Flip-flops in the scan chains

Verifying Test Signal Connections in Full-chip Designs (goal name = dft_block_check)

Create test SGDC for Full-chip

- Create define_tag constraints that drive the test controller to the state necessary for each sub-block:

  ```
  define_tag –tag <condition name | sub-block name> -name
  <test controller port name> -value <value for this port>
  ```

- Create require_value constraints on the test controller ports with the values that should be produced for each sub-block setup:

  ```
  require_value -tag <condition name | sub-block name> –name
  <test controller port name> -value <value expected on this
  pin for this condition>
  ```

- Create require_value constraint for each of the sub-blocks using the syntax:

  ```
  require_value -tag <condition name | sub-block name> –name
  <sub-block pin name> -value <value expected on this pin for
  this condition>
  ```

- Create require_path constraints for each of the sub-blocks using the syntax:

  ```
  require_path -tag <condition name | sub-block name> -from
  <controller port name> –to <sub-block pin name>
  ```

- Declare all sub-blocks with require_value or require_path constraints as black boxes.
- Add all available block.sgdc files and black box these modules.

Subblock Check (goal name = dft_block_check)

- Follow the diagnostic procedure shown in Start with the Soc_01 violations for *FIGURE 21. Test Connection Verification Procedure* require_value constraints on test controller outputs. (Remember that Soc_01 simulates the values specified in define_tag –name XX and checks if the require_value constraints with same XX name are achieved.)
- Soc_01 will have a violation for nodes that do not have the value specified by a require_value constraint. If any violations are issued,

check the define_tag sequence and the corresponding require_value constraints. Complex define_tag sequences can be debugged by cutting the define_tag sequence into smaller pieces and putting require_value constraints on the state variables inside the test controller. If there are still Soc_01 violations, then repeat this process with an even smaller sequence.

- Soc_01_info will highlight nodes that do achieve the required values. The use of this rule may be an aid in debugging Soc_01 violations since it can be used to confirm that required values for a given define_tag name are achieved.
- When there are no Soc_01 violations for the test controller outputs, select Soc_02 violations for require_path constraints that define test controller and sub-block paths. Selecting an Soc_02 will highlight the violating connection in the MS. If a topological path does not exist, then either the design is not clearly understood or the enabling conditions are incorrect.
- If the require_path constraint does not specify a setup condition, then an Soc_02 violation means that no topological path exists.

  nIf the require_path constraint does specify a setup condition, then an Soc_02 violation means that either no topological path exists or the values required to sensitize this path are incorrect.
- An Soc_02_info violation for a require_path may provide useful additional information. (Refer to the *Soc_02_info* rule in the *Connectivity Verify Rules Reference Guide*.)
- Soc_05 violations highlight pins that have not achieved the values specified in their block.sgdc files. Such violations are caused by either a design error in the connection to a sub-block or in the test_mode and testclock constraints for the top-level design. Info_testmode and Info_testclock may be an aid in diagnosing these violations.

Creating and Validating an Abstract Model for a Block (goal names = dft_abstract, dft_abstract_validate)

SpyGlass DFT supports a hierarchical SoC methodology in which a simplified abstract model of the block is created when verifying the block. You can use this model instead of the original RTL for efficient verification at the top/SoC level.

The creation of the abstract model is done by running the dft_abstract goal on the block.

When using abstract models for one or more lower level blocks in a design, the dft_abstract_validate goal verifies that the constraints under which the abstraction was done are met in the current design

Refer to the *SoC Methodology User Guide* for more information on the use of the hierarchical SoC methodology for DFT.

Using Autofix/Selective Autofix

The AutoFix feature provides the capability to automatically fix the issues reported by the supported rules by modifying the RTL. The Selective Autofix feature provides the ability to fix the reported issues selectively.

You can enable the Autofix feature for a goal and for the supported rules using the dft_autofix and the rme_active parameters.

The following snippet from the project file describes method to enable the autofix feature for the TA_09 rule:

```
current_goal  dft/dft_test_points
set_parameter dftAutoFix {+RULES[TA_09]}
set_parameter rme_active 1
```

For more information on running the Selective Autofix feature, refer to the Running the Selective Autofix section in the DFT Rules Reference Guide.

Appendix A

Sample SGDC File:

```
current_design RefDesCore
// ---------------------------------------------------------
// The following clocks have been found by running
'Dft_setup' goal.
// ---------------------------------------------------------
clock -name "RefDesCore.mc_clk_i" -domain
"RefDesCore.mc_clk_i" -period 10.000000 -testclock
clock -name "RefDesCore.eth_mtx_clk_pad_i" -domain
"RefDesCore.eth_mtx_clk_pad_i" -period 10.000000 -testclock
clock -name "RefDesCore.clk100" -domain "RefDesCore.clk100"
-period 10.000000 -testclock
// ---------------------------------------------------------
// The following constraints have been found by running
Dft_setup goal
// ---------------------------------------------------------

test_mode -name "RefDesCore.rst100"   -value 1 -scanshift
test_mode -name "RefDesCore.dftreset" -value 1 -scanshift
test_mode -name "RefDesCore.usb_phy_clk_pad_i_en" -value "1"
-scanshift
```