DFT Compiler
Scan
User Guide

Version H-2013.03-SP4, September 2013

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

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

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

Copyright Notice for the Command-Line Editing Feature

Copyright Notice for the Line-Editing Library

Contents

2. Running RTL Test Design Rule Checking

3. Running the Test DRC Debugger

Preface

This preface includes the following sections:

About This Guide

The *DFT Compiler Scan User Guide* describes the processes and flows for using scan technology in DFT Compiler.

Audience

This manual is intended for ASIC design engineers who have some experience with testability concepts and for test and design-for-test engineers who want to understand how basic test automation concepts and practices relate to DFT Compiler.

Related Publications

For additional information about the DFT Compiler tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

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

- DFTMAX™
- Design Compiler®
- Design Vision™
- BSD Compiler
- TetraMAX®

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *DFT Compiler Release Notes* on the SolvNet site.

To see the *DFT Compiler Release Notes*,

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

   https://solvnet.synopsys.com/DownloadCenter

2. Select DFT Compiler, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

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

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

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

Contacting the Synopsys Technical Support Center

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

- Open a support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Key Design-for-Test Flows and Methodologies

This chapter describes the key flows, methodologies, processes, and default behavior to consider when planning your design-for-test (DFT) project. For a complete introduction to DFT processes and flows, see Chapter 2 of the *DFT Overview User Guide*.

This chapter includes the following sections:

Design-for-Test Flows in the Logic Domain

This section introduces the recommended DFT flows for simple designs in the logic domain. The flow you use depends on the state of the design before you start using DFT Compiler, as detailed in the following sections:

- Unmapped Design Flow
- Mapped Design Flow
- Mapped Designs With Existing Scan Flow
- Designing Block by Block
- Controlling Scan Replacement During Scan Insertion

Unmapped Design Flow

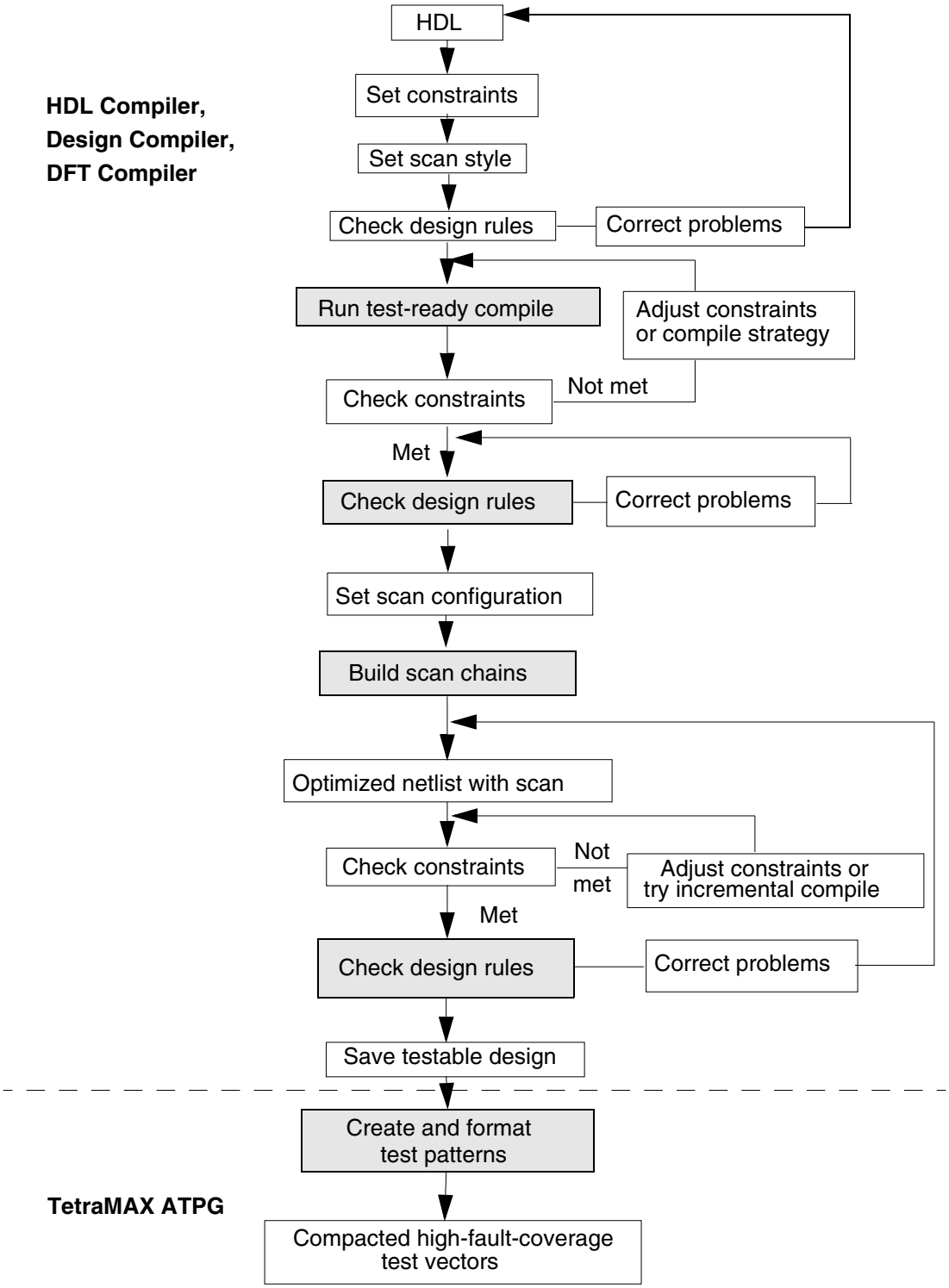
This flow applies to unmapped designs. Using the command sequence provided in this section, you can take a design from an HDL-level circuit description that does not contain existing scans to a fully optimized design with internal scan circuitry. The command sequence example applies to full-scan and partial-scan designs. The basic process consists of the following tasks:

- Synthesizing Your Design
- Postprocessing Your Design
- Building Scan Chains
- Exporting a Design to TetraMAX ATPG

  For information about how to pass the design to the TetraMAX ATPG tool to generate test patterns, see Chapter 10, “Exporting Data to Other Tools.” Also, see the *TetraMAX Online Help* for information about generating test patterns by using TetraMAX ATPG in a DFT Compiler flow.

Figure 1-1 shows a typical unmapped design flow that uses the HDL Compiler and DFT Compiler tools. At the end of the flow, TetraMAX ATPG produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

Figure 1-1 Typical Flat Design Flow From an Unmapped Design



Synthesizing Your Design

To synthesize your design, perform the following steps:

1. Select the target logic library and the link logic library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library \
              [list "*" asic_vendor.db]
```

2. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhdl
```

3. Explicitly link the design:

```
dc_shell> link
```

   If DFT Compiler is unable to resolve any references, you must provide the missing designs before proceeding. See the Design Compiler documentation for details.

4. Select the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area and runs with a 20-ns clock period. Enter the following commands:

```
dc_shell> set_max_area 1000
dc_shell> create_clock clock_port -period 20 -waveform [list 10 15]
```

   For additional related information on setting design constraints, see Chapter 6, “Architecting Your Test Design.”

5. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX ATPG to generate test patterns, and your vendor does not have specific requirements, the default settings produce the best results:

```
dc_shell> set_app_var test_default_delay 0
dc_shell> set_app_var test_default_bidir_delay 0
dc_shell> set_app_var test_default_strobe 40
dc_shell> set_app_var test_default_period 100
```

   These are the default settings; you do not need to add them to your script.

6. Select the scan style if you did not previously set the `test_default_scan_style` variable in the .synopsys_dc.setup file. In this example, the scan style is multiplexed flip-flop.

   Enter the following command:

```
dc_shell> set_app_var test_default_scan_style \
              multiplexed_flip_flop
```

   You can also use the `set_scan_configuration -style` command instead.

For more information on setting the scan style, see “Setting the Scan Style” on page 2-7.

7. Define clocks and asynchronous set and reset signals in your design, and then generate a test protocol.

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock ...
dc_shell> set_dft_signal -view existing_dft -type Reset ...
dc_shell> create_test_protocol
```

   For more information on generating a test protocol, see “Defining an Initialization Protocol” on page 5-30.

8. Check test design rules in the RTL source file, using RTL Test DRC.

```
dc_shell> dft_drc
```

   For more information, see “Understanding the Violations” on page 2-13.

9. Synthesize a design that meets the constraints you set, and map your circuit description to cells from the target logic library.

```
dc_shell> compile -scan
```

   This performs a test-ready compile. DFT Compiler synthesizes and optimizes your design relative to area and speed, and it removes redundant logic. This last feature eliminates logically untestable circuitry and is an important part of the Synopsys test methodology. Because of the `-scan` option, all flip-flops in the design are implemented as scan flip-flops.

   For details on test-ready compile, see “Test-Ready Compilation” on page 4-24.

Postprocessing Your Design

After your design is synthesized, perform the following postprocessing steps:

1. Check that you have satisfied the constraints you specified in step 5 of the previous procedure. Enter the following command:

```
dc_shell> report_constraint -all_violators
```

2. This step is optional. You can save a copy of the design at this point. Enter the following command:

```
dc_shell> write -format ddc -output design_test_ready.ddc
```

   Note:

   Do not save a copy as ASCII text (Verilog, VHDL, or EDIF formats) if you intend to start a new session, using the saved design. DFT Compiler marks the test-ready logic with attributes that are lost when you save the design as ASCII text.

3. Generate a new test protocol for the synthesized design.

4. Check the test design rules, according to the scan style you chose in step 7 of the previous procedure.

   ```
   dc_shell> dft_drc
   ```

   At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

   For more information on Test DRC, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

Building Scan Chains

To build scan chains, perform the following steps:

1. Use the `set_dft_signal` command to identify the dedicated scan-enable port. Enter the following commands:

   ```
   dc_shell> set_dft_signal -view spec -port scan_enable_port \
                -type ScanEnable \
                -active_state 1
   ```

   If you want DFT Compiler to buffer the scan-enable signal, use the `set_driving_cell` command to specify the port’s drive characteristics:

   ```
   dc_shell> set_driving_cell -lib_cell BUFX4 scan_enable_port
   ```

   If you do not want DFT Compiler to buffer the scan-enable signal, use the `set_ideal_network` command to configure the port as the driver of an ideal network:

   ```
   dc_shell> set_ideal_network scan_enable_port
   ```

   If no scan-enable port is identified, DFT Compiler creates a new scan-enable port.

2. If you are reusing functional ports as scan-in and scan-out ports, specify them with the `set_dft_signal` command:

   ```
   dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[7]
   dc_shell> ...
   dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[0]
   dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[7]
   dc_shell> ...
   dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[0]
   ```

   Otherwise, DFT Compiler creates new scan-in and scan-out ports as needed.

3. Build the scan chains in your design. Enter the following command:

   ```
   dc_shell> insert_dft
   ```

When you add scan-test circuitry to a design, its area and performance change. DFT Compiler minimizes the effect of adding scan-test circuitry on compile design rules and performance by using synthesis routines. For details of synthesis concepts such as compile design rules, see the *Design Compiler Optimization Reference Manual*. For details on how DFT Compiler fixes compile design rule violations and performance constraint violations, see Chapter 5, "Pre-Scan Test Design Rule Checking."

4. Check that you have satisfied the constraints you specified in step 5 of the preceding section, "Synthesizing Your Design" on page 1-4. Enter the following command:

```
dc_shell> report_constraint -all_violators
```

5. Recheck the test design rules, according to the scan style you chose in step 7 of the preceding section, "Synthesizing Your Design" on page 1-4.

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks that are run are more comprehensive than those in step 10 of the "Synthesizing Your Design" on page 1-4 and also include checks for the correct operation of the scan chain. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

For more information on this process, see "Post-Scan Test Design Rule Checking" on page 6-63.

The `dft_drc` command is an essential preprocess step to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

6. To generate test reports, use the `report_scan_path` command. For example, to view details of the scan chain, use the following command:

```
dc_shell> report_scan_path -view existing_dft -chain all
```

Note:

Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran the `dft_drc` command.

7. Write out the complete design in the Synopsys database format. Enter the following command:

```
dc_shell> write -format ddc -hierarchy -output design.ddc
```

Mapped Design Flow

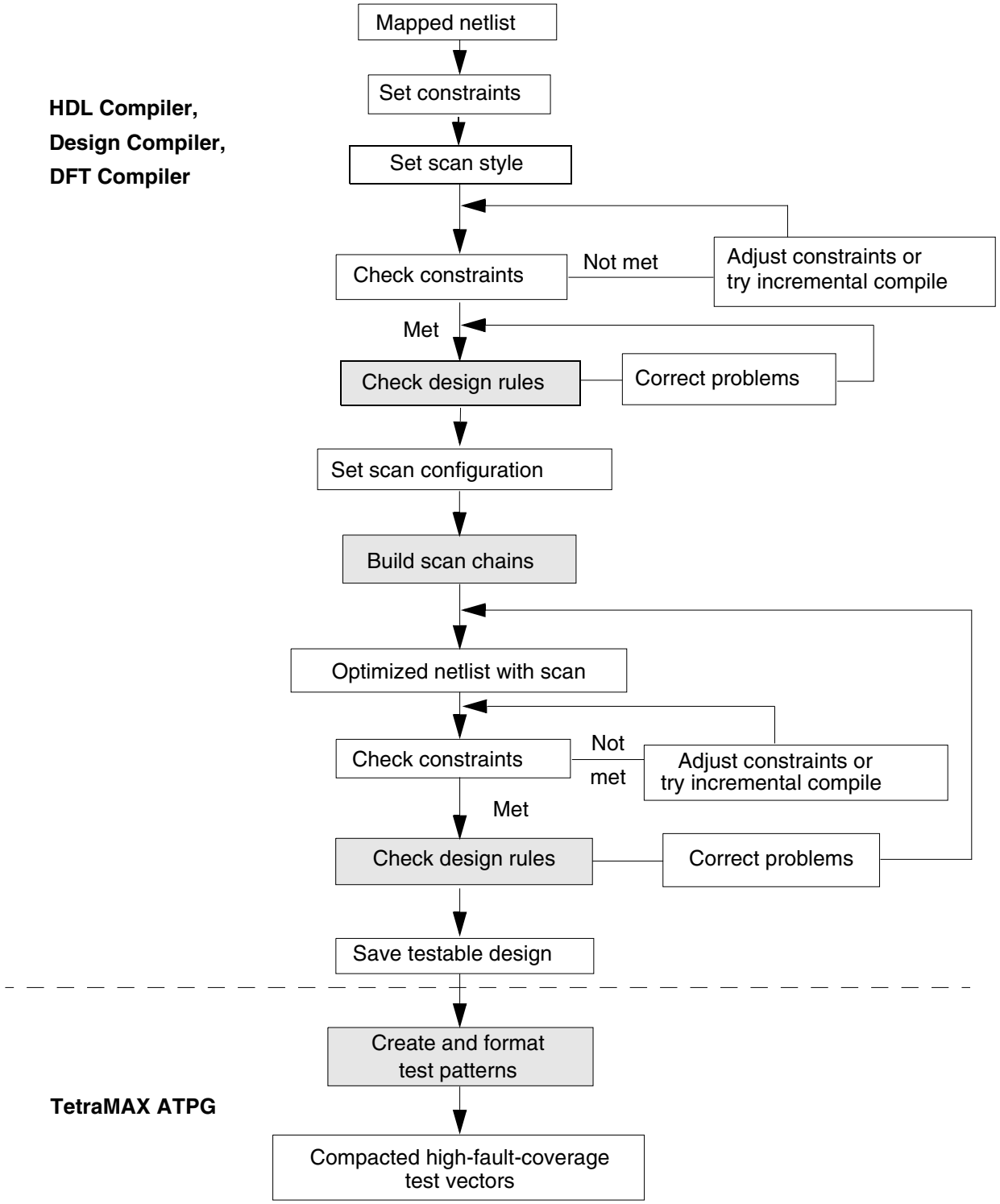
Using the command sequence provided in this section, you can take a mapped design that does not contain existing scan to a fully optimized design with internal scan circuitry. The command sequence example applies to a full-scan or partial-scan design. The basic process consists of the following tasks:

- Reading In Your Design
- Performing Scan Replacement and Building Scan Chains
- Exporting a Design to TetraMAX ATPG

  For information about how to pass the design to the TetraMAX ATPG tool to generate test patterns, see Chapter 10, “Exporting Data to Other Tools.” Also, see the *TetraMAX Online Help* for information about generating test patterns by using TetraMAX ATPG in a DFT Compiler flow.

Figure 1-2 shows a typical mapped design flow for DFT Compiler and HDL Compiler. At the end of the design flow, TetraMAX ATPG produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

Figure 1-2 Typical Flat Design Flow From a Mapped Design



Reading In Your Design

Perform the following steps to read in your design:

1. Select the target logic library and link logic library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library [list * asic_vendor.ddc]
```

2. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format ddc design_name.ddc
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhdl
```

   If you want DFT Compiler to buffer the scan-enable signal, include a dedicated scan-enable port in your design. At this stage, the scan-enable port should not be connected.

3. Explicitly link the design:

```
dc_shell> link
```

   If the tool is unable to resolve any references, you must provide the missing designs before proceeding. See the Design Compiler documentation for details.

4. Select the target logic library and the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area, and it runs with a 20-ns clock period. Enter the following commands:

```
dc_shell> set_max_area 1000
dc_shell> create_clock clock_port -period 20 -waveform [list 10 15]
```

   Note:

   If the file you read is in the Synopsys logic database format (a .ddc file), the target logic library and the design constraints might already be set.

5. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX ATPG to generate test patterns and your vendor does not have specific requirements, the default settings produce the best results:

```
dc_shell> set_app_var test_default_delay 0
dc_shell> set_app_var test_default_bidir_delay 0
dc_shell> set_app_var test_default_strobe 40
dc_shell> set_app_var test_default_period 100
```

   These are the default settings; you do not need to add them to your script.

6. Select the scan style if you did not previously set the `test_default_scan_style` variable in the .synopsys_dc.setup file. In this example, the scan style is multiplexed flip-flop.

Enter the following command:

```
dc_shell> set_app_var test_default_scan_style \
              multiplexed_flip_flop
```

You can also use the `set_scan_configuration -style` command.

For more information on setting the scan style, see "Setting the Scan Style" on page 2-7.

7. Check that you have satisfied the constraints you specified in step 4. Enter the following command:

```
dc_shell> report_constraint -all_violators
```

8. Define clocks and asynchronous set and reset signals in your design, and then generate a test protocol.

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock ...
dc_shell> set_dft_signal -view existing_dft -type Reset ...
dc_shell> create_test_protocol
```

For more information on generating a test protocol, see "Defining an Initialization Protocol" on page 5-30.

9. Check the test design rules:

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage. For more information on design rule checking, see Chapter 5, "Pre-Scan Test Design Rule Checking".

Because this design did not go through a test-ready flow, the `dft_drc` command checks the library for equivalent scan cells for all the flip-flops in your design.

Performing Scan Replacement and Building Scan Chains

To perform scan replacement and build scan chains, follow these steps:

1. To identify the dedicated scan-enable port, use the `set_dft_signal` command. Enter the following commands:

```
dc_shell> set_dft_signal -view spec -port scan_enable_port \
              -type ScanEnable \
              -active_state 1
```

If you want DFT Compiler to buffer the scan-enable signal, use the `set_driving_cell` command to specify the port's drive characteristics:

```
dc_shell> set_driving_cell -lib_cell BUFX4 scan_enable_port
```

If you do not want DFT Compiler to buffer the scan-enable signal, use the `set_ideal_network` command to configure the port as the driver of an ideal network:

```
dc_shell> set_ideal_network scan_enable_port
```

If no scan-enable port is identified, DFT Compiler creates a new scan-enable port.

2. If you are reusing functional ports as scan-in and scan-out ports, specify them with the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[7]
dc_shell> ...
dc_shell> set_dft_signal -view spec -type ScanDataIn -port DAT_IN[0]
dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[7]
dc_shell> ...
dc_shell> set_dft_signal -view spec -type ScanDataOut -port DAT_OUT[0]
```

Otherwise, DFT Compiler creates new scan-in and scan-out ports as needed.

3. Specify the scan architecture. In this example, direct DFT Compiler to build two scan chains with the following command:

```
dc_shell> set_scan_configuration -chain_count 2
```

The `set_scan_configuration` command determines most of the aspects of how DFT Compiler makes designs scannable. For more information about the `set_scan_configuration` command, see Chapter 6, “Architecting Your Test Design.”

4. Build the scan chains in your design with the following command:

```
dc_shell> insert_dft
```

Because your design has no scan cells, DFT Compiler replaces the flip-flops in your design with scannable equivalents before it builds scan chains.

When you add scan-test circuitry to a design, the design’s area and performance change. DFT Compiler minimizes the effect of adding scan-test circuitry on the compile design rules and design performance by using synthesis routines. For details about synthesis concepts such as compile design rules, see the *Design Compiler Optimization Reference Manual*. For details about how DFT Compiler fixes compile design rule violations and performance constraint violations, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

5. Check that you have satisfied the constraints you specified in step 4 of the preceding section, “Reading In Your Design” on page 1-10. Enter the following command:

```
dc_shell> report_constraint -all_violators
```

6. Recheck the test design rules, according to the scan style you chose in step 6 of the preceding section. See “Reading In Your Design” on page 1-10.

```
dc_shell> dft_drc
```

At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks run are more comprehensive than those in step 8 of the "Reading In Your Design" on page 1-10 and include checks for the correct operation of the scan chain. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

The `dft_drc` command is an essential preprocess step to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

See "Post-Scan Test Design Rule Checking" on page 6-63 for more information on this process.

7. To generate test reports, use the `report_scan_path` command. For example, to view details of the scan chain, use the command:

```
dc_shell> report_scan_path -view existing_dft -chain all
```

Note:
Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran the `dft_drc` command.

8. Write out the complete design in Synopsys database format. Enter the following command:

```
dc_shell> write -format ddc -hierarchy -output design.ddc
```

Mapped Designs With Existing Scan Flow

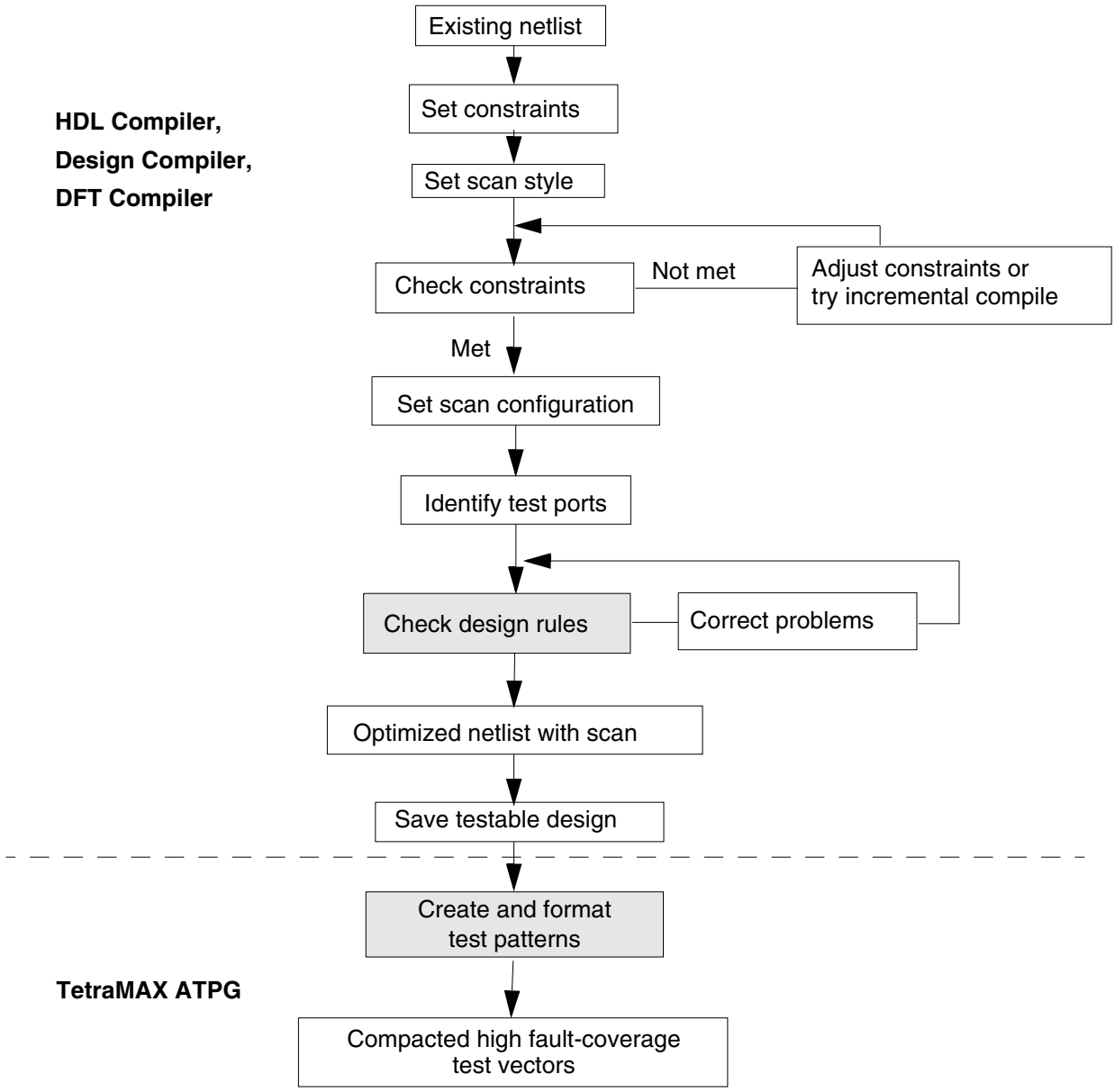
Using the command sequence provided in this section, you can take a design from an HDL-level circuit description that contains existing scans to a fully optimized design with internal scan circuitry. The command sequence example applies to a full-scan or partial-scan design. The basic process consists of the following tasks:

- Reading In Your Design
- Checking Test Design Rules
- Exporting a Design to TetraMAX ATPG

  For information about how to pass the design to TetraMAX ATPG to generate test patterns, see Chapter 10, "Exporting Data to Other Tools." Also, see the *TetraMAX Online Help* for information about generating test patterns by using TetraMAX in a DFT Compiler flow.

Figure 1-3 shows a typical flow for using DFT Compiler on a design with existing scan chains. At the end of the design flow, TetraMAX ATPG produces a set of high fault-coverage test vectors that you can readily adapt to your target tester.

Figure 1-3 Typical Flat Design Flow With Existing Scan Chains



Reading In Your Design

To read in your design, follow these steps:

1. Select the target logic library and link logic library.

```
dc_shell> set target_library asic_vendor.ddc
dc_shell> set link_library [list * asic_vendor.ddc]
```

2. Use one of the following commands to read in an HDL circuit description:

```
dc_shell> read_file -format ddc design_name.ddc
dc_shell> read_file -format verilog design_name.v
dc_shell> read_file -format vhdl design_name.vhdl
```

3. Explicitly link the design:

```
dc_shell> link
```

   If the tool is unable to resolve any references, you must provide the missing designs before proceeding. See the Design Compiler documentation for details.

4. Select the target logic library and the design constraints. In this example, the design is constrained to be no larger than 1,000 vendor units in area, and it runs with a 20ns clock period. Enter the following commands:

```
dc_shell> set_max_area 1000
dc_shell> create_clock clock_port -period 20 -waveform [list 10 15]
```

   Note:

   If the file you read in is a .ddc file, the target logic library and the design constraints might already be set.

5. Set the test timing variables to the values required by your ASIC vendor. If you are using TetraMAX ATPG to generate test patterns and your vendor does not have specific requirements, the default settings produce the best results:

```
dc_shell> set_app_var test_default_delay 0
dc_shell> set_app_var test_default_bidir_delay 0
dc_shell> set_app_var test_default_strobe 40
dc_shell> set_app_var test_default_period 100
```

   These are the default settings; you do not need to add them to your script.

6. Select the scan style if you did not previously set the `test_default_scan_style` variable in the .synopsys_dc.setup file. In this example, the scan style is multiplexed flip-flop.

   Enter the command:

```
dc_shell> set_app_var test_default_scan_style \
               multiplexed_flip_flop
```

You can also use the `set_scan_configuration -style` command. See "Architecting Your Test Design" on page 6-1 for details.

7. Check that you have satisfied the constraints you specified in step 4. Enter the command:

```
dc_shell> report_constraint -all_violators
```

Checking Test Design Rules

To check test design rules, follow these steps:

1. Identify the test ports to DFT Compiler. This example has a single scan chain. Enter the commands:

```
dc_shell> set_scan_state scan_existing

dc_shell> set_scan_path c0 -view existing_dft \
            -scan_data_in [get_ports test_si] \
            -scan_data_out [get_ports test_so] \
            -scan_enable [get_ports test_se] \
            -infer_dft_signals

dc_shell> create_test_protocol -infer_clock \
            -infer_asynch
```

   Note:

   If you read in the design from a .ddc file, this attribute might already be set. You can check this with the `report_dft_signal -view existing_dft` command.

2. Check the test design rules, according to the scan style you chose in step 6 of the preceding section, "Reading In Your Design" on page 1-15, and using the test attributes set in step 1 of this section.

```
dc_shell> dft_drc
```

   At this stage, DFT Compiler checks for and describes potential problems with the testability of your design. The checks include checking for the correct operation of the scan chain during shift and capture. After you correct all the problems that cause warning messages, you can proceed with the next step. Failure to correct the problems that cause warning messages typically results in lower fault coverage.

   Running the `dft_drc` command is an essential preprocess to guarantee that the reports generated by the `report_scan_path` command are correct. You must run the `dft_drc` command if you want to generate reports with the `report_scan_path` command.

   For more information on design rule checking, see Chapter 5, "Pre-Scan Test Design Rule Checking."

3. To generate test reports, use the `report_scan_path` command. For example, to view details of the scan chain, use the following command:

   ```
   dc_shell> report_scan_path -view existing_dft -chain all
   ```

   Note:
   Rerun the `dft_drc` command before running `report_scan_path` only if you have modified your circuit since you last ran the `dft_drc` command.

4. Write out the complete design in Synopsys database format. Enter the following command:

   ```
   dc_shell> write -format ddc -hierarchy -output design.ddc
   ```

Designing Block by Block

Many designers prefer to develop their designs on a block-by-block basis. Most of the processes described in the preceding sections can be applied to a particular block of your design, as well as to the complete design. For example, you can check the design rules and generate an initial fault coverage report on each block of your design as it is developed. The report helps you identify the blocks that have unacceptable fault coverage, which enables you to fix testability problems at an early stage. For more information on pre-scan DRC, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

Although fixing testability problems on a block-by-block basis is an important “divide-and-conquer” technique, testability problems are global in nature. A completely testable subblock might show testability problems when it is embedded in its environment.

Controlling Scan Replacement During Scan Insertion

By default, the `insert_dft` command performs scan replacement in your design before scan stitching. Here, design rule checking determines if each sequential element has the appropriate test pins (scan-in, scan-out, scan-enable, and so on) defined for the selected scan style. If no violations are found and if the appropriate test pins are found, the cells are included in the scan chain. Otherwise, DRC checks to see if each of the sequential elements in your design has a scan equivalent for the selected scan style. If no violations are found on the cells and if a scan equivalent is present, the cells are scan-replaced and included in the scan chain.

If you have a design that is already scan-replaced and you do not want the `insert_dft` command to perform scan replacement, specify the following command before stitching scan chains:

```
set_scan_configuration -replace false
```

When you specify this command, DRC only checks to see if each sequential element has the appropriate test pins for the chosen scan style. If no violations are found on the cells and the appropriate test pins are found, the cells are included in the scan chain.

However, if you have a .ddc file that is written out after test-ready compile, by using the `compile -scan` or `compile_ultra -scan` commands, then you do not need to specify again the `set_scan_configuration -replace false` command. Because the design is already scan-replaced, the scan insertion process does not perform scan replacement.

Table 1-1 summarizes the scan replacement flow.

Table 1-1 Scan Replacement Using set_scan_configuration -replace

set_scan_configuration -replace	**false**	**true (default)**
Scan equivalent check	Check based on presence of test pins for the scan style chosen	Check based on presence of test pins for the scan style chosen, and based on scan equivalence for the scan style chosen
Cells violated and message	TEST-126: cells do not have valid test pins for the scan style	TEST-120: cells have no scan equivalents based on synthesis mapping
Scan replacement	No	Yes, if needed
`preview_dft, insert_dft`	Valid cells included in scan chain	Valid cells included in scan chain

Hierarchical Scan Synthesis Flow

Hierarchical Scan Synthesis (HSS) enables you to specify scan chain designs and implement a top-down, bottom-up, or middle-out hierarchical scan insertion flow. Because HSS reduces memory usage and decreases runtime, it is useful when you are working with large designs.

HSS enables you to do the following:

- Implement automatically balanced scan chains
- Specify complete scan chains
- Generate scan chains that enter and exit a design module multiple times
- Fix certain scan rule violations automatically

- Reuse existing modules that already contain scan chains
- Control the routing order of scan chains in the hierarchy
- Perform scan insertion from the top or the bottom of the design
- Implement automatically enabling or disabling logic for bidirectional ports and internal three-state logic
- Implement automatically multiplexed and enabling or disabling logic to shared function ports as test data ports

In general, the hierarchical scan synthesis flow consists of four major steps:

- Performing DFT scan synthesis on the subdesigns of the design
- Generating Core Test Language (CTL) test models for the subdesigns
- Stitching the CTL test models together at the top-level (scan assembly)
- Performing DFT scan synthesis on the complete design with the CTL test models replacing DFT inserted subdesigns

Introduction to Test Models

HSS automatically creates test models that describe only the portions of subdesigns that are important for inserting the DFT flow. Test models store the following test information:

- Port names
- Port directions (input, output, bidirectional)
- Scan structures
- Scan clocks
- Asynchronous sets and resets
- Three-state disables
- Test attributes on ports
- Test protocol information, such as initialization, sequencing, and clock waveforms

Note the following limitations related to the use of test models:

- Because DFT Compiler saves the test model as an attribute to the design in .ddc binary format, you cannot use a text editor to view the test model.
- Test models are not adequate for the generation of test patterns, so you must have an actual netlist representation for use with automatic test pattern generation (ATPG).

Figure 1-4 shows a design schematic example.

Figure 1-4 Design Schematic Example

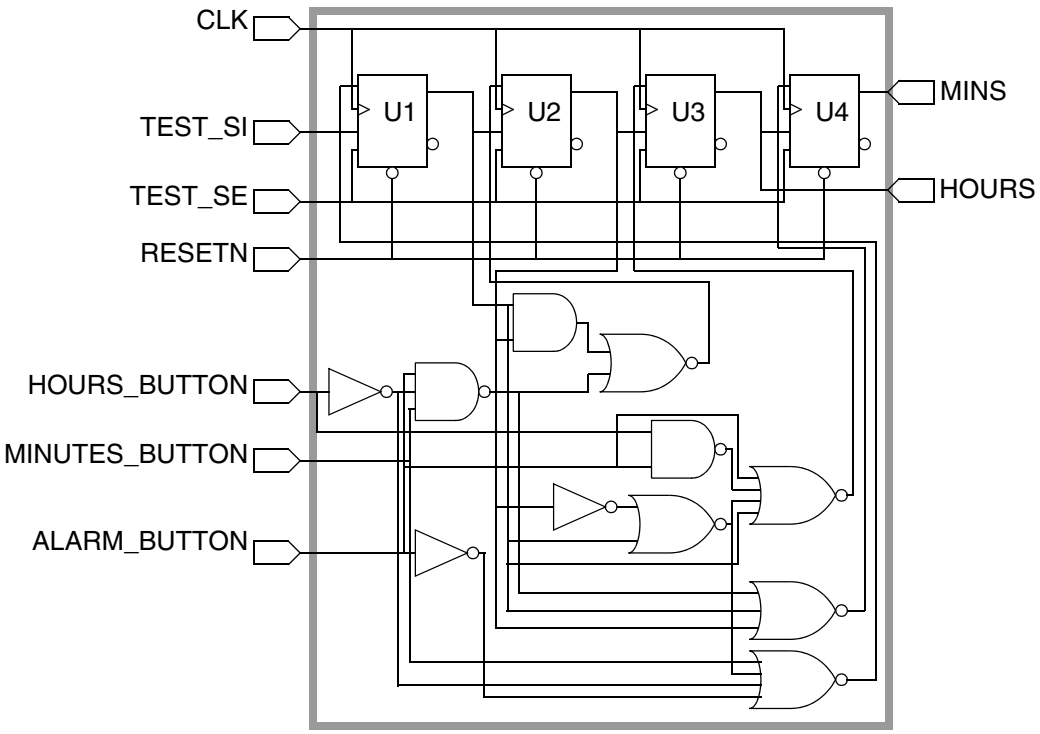


Figure 1-5 shows how a test model might represent the test attributes of that design.

Figure 1-5 Test Model Representation

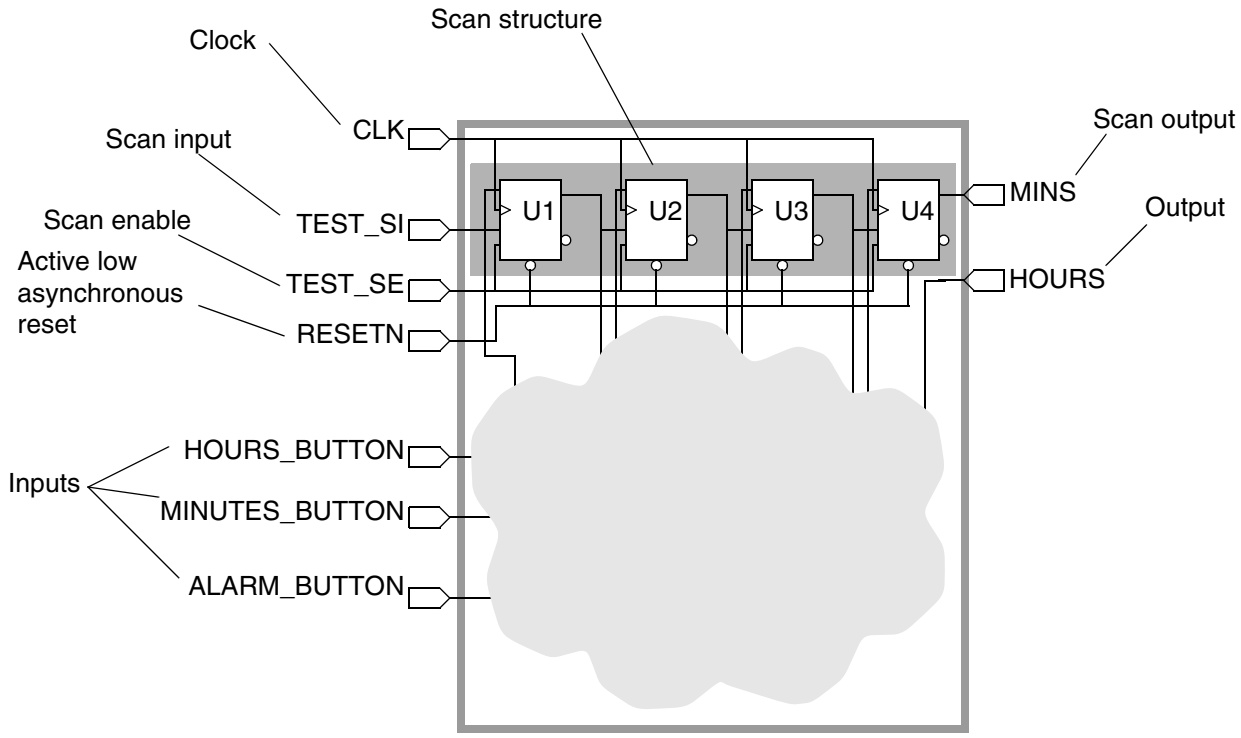
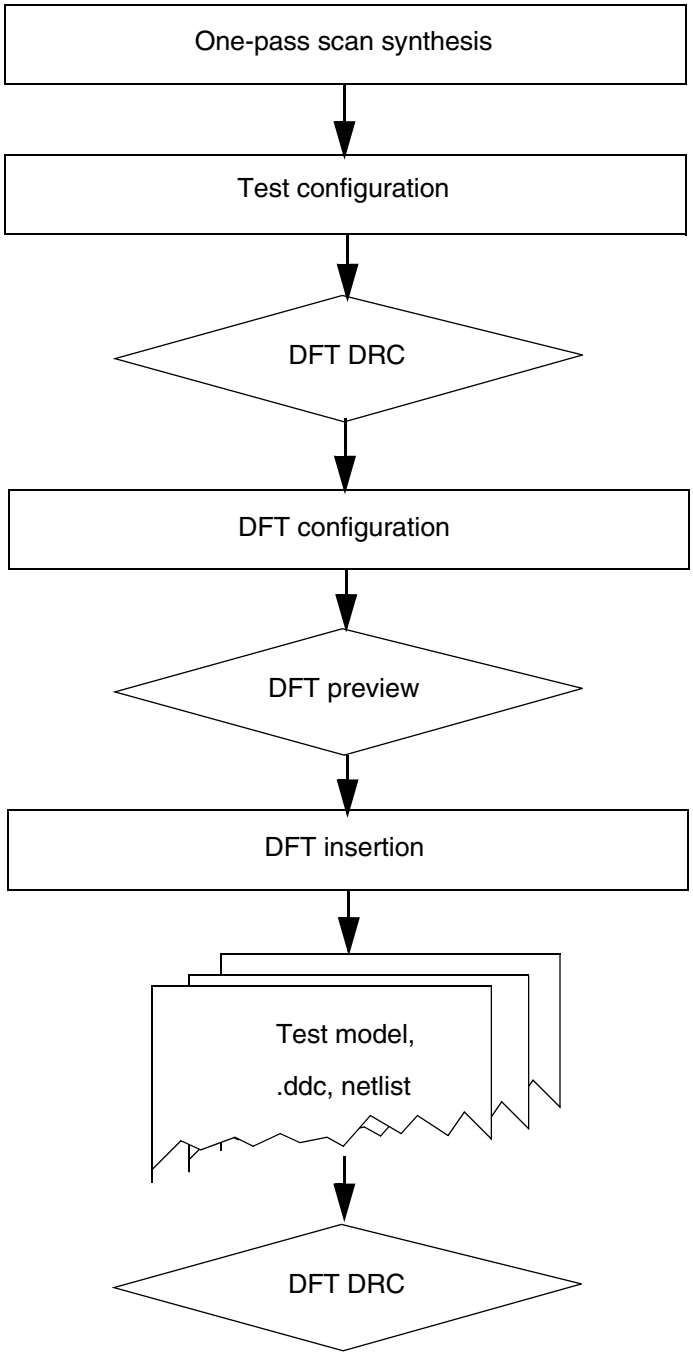


Figure 1-6 describes the module-level test synthesis flow used for HSS.

Figure 1-6 Module-Level Test Synthesis Flow



Linking Test Models to Library Cells

When complex library cells have built-in scan chains, Core Test Language (CTL) test models must be linked to the library cells for DFT Compiler to connect the scan chains at the top level.

If you have the original library source file, you can use the `read_lib -test_model` command to annotate test model information to a library cell when the library is read in. After the library has been read and annotated with the test model, a library .db file containing the test model can be written out. For example,

```
read_lib lib_file.lib -test_model [lib_cell_name:]model_file.ctl
write_lib lib -output lib_file.db
```

If the CTL model name differs from the intended library cell name, an optional library cell name prefix can be used to specify the model's intended library cell.

To link multiple test models to multiple cells within a single library, supply the model files to the `-test_model` option as a list:

```
read_lib lib_file.lib -test_model [list \
    [lib_cell_name:]model_file1.ctl \
    [lib_cell_name:]model_file2.ctl]
```

If you only have a compiled library .db file, you can use the `read_test_model` command to link a test model to an existing library cell or design in memory. Specify the library cell or design name with the `-design` option. The library cell name can be provided with or without the logic library name prefix. For example,

```
read_test_model -format ctl \
  -design design_name model_file.ctl

read_test_model -format ctl \
  -design lib_cell_name model_file.ctl

read_test_model -format ctl \
  -design logic_lib/lib_cell_name model_file.ctl
```

If you have subdesigns that are modeled using extracted timing models (ETMs), you can also link CTL test models to the ETM library cells just as you would with logic library cells.

You can use the `report_lib` command to determine if a library cell has CTL test model information applied to it, by using either the `read_lib -test_model` or `read_test_model` command.

Note:

When you link a test model to a library cell with the `read_test_model` command, it takes precedence over any existing test model information present in the library.

Checking Library Cells for CTL Model Information

To determine if a library cell has CTL model information attached to it, read in the library with the `link` command or the `compile` command, and then run the `report_lib` command. If a library cell in the library has CTL model information attached, the report will indicate `ctl` in the attributes column, as shown in Example 1-1.

Example 1-1 Simple report_lib Output

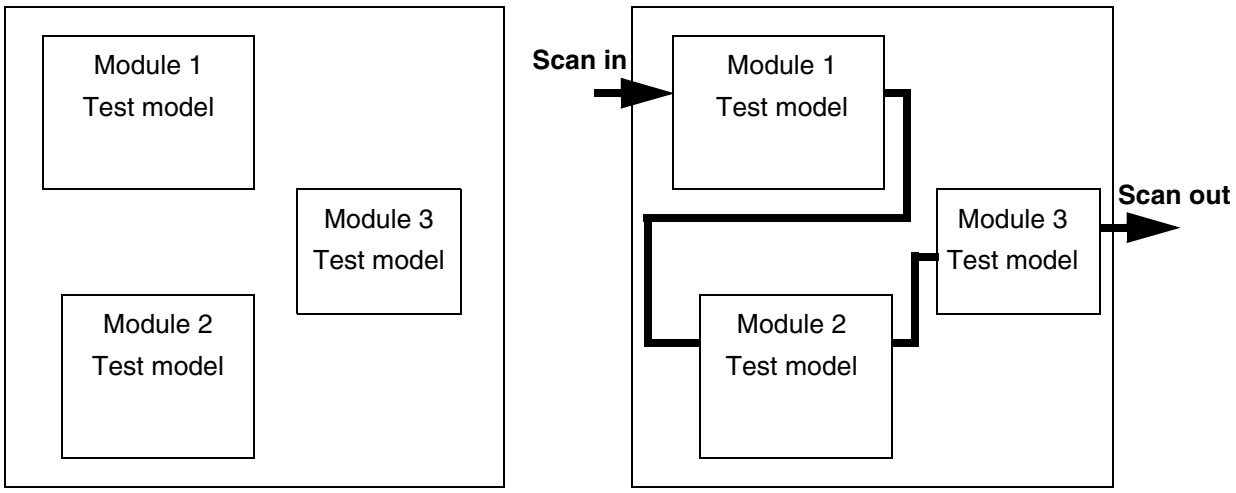
```
Cell          Footprint    Attributes
-------------------------------------
my_memory     "MEM"            b, d, s, u, ctl, t
```

Scan Assembly Using Test Models

In practice, you should use test models to represent subdesigns, and then stitch the test models together at the top level of the design. Because the test model stores less information than a full gate-level representation in a .ddc database, a design with test models uses less memory than a design with subdesigns represented by gate-level .ddc files.

Figure 1-7 shows scan assembly at the top level of a design with several subdesigns represented by test models. The top-level scan assembly process also works if some of the subdesigns are implemented at the gate level or if user-defined logic exists at the top level.

Figure 1-7 Scan Assembly Using Test Models



The following sections contain more detail about creating and using test models.

Saving Test Models for Subdesigns

At the end of the flow, use the `write_test_model` command to save the test model to disk. Alternatively, you can use the `write` command to generate a .ddc file containing both the test model and gate information.

You should also write out a Verilog or VHDL netlist of the design to use with TetraMAX ATPG.

Example 1-2 shows a command sequence for creating and saving a test model on a design that has no existing scan structures.

Example 1-2 Test Model Creation Steps for Nonscan Design

```
dc_shell> remove_design -all
dc_shell> read -format verilog ALARM_BLOCK.v
dc_shell> current_design ALARM_BLOCK
dc_shell> link
dc_shell> create_clock CLK -period 100 -waveform [list 0 50]
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
               -timing [list 45 55] -port CLK
dc_shell> set_dft_signal -view existing_dft -port \
               RESETN -type Reset -active_state 0
```

```
dc_shell> compile -scan
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> preview_dft -show all
dc_shell> insert_dft
dc_shell> dft_drc -verbose
dc_shell> write -format ddc -hierarchy -output ALARM_BLOCK.ddc
dc_shell> write -format verilog -hierarchy -output ALARM_BLOCK.v
dc_shell> write_test_model -output ALARM_BLOCK.ctlddc
dc_shell> set_app_var test_stil_netlist_format verilog
dc_shell> write_test_protocol -output ALARM_BLOCK.spf
dc_shell> exit
```

Using Test Models

At the top level of your design, you have to read in the test models that you created for the subdesigns and perform scan insertion.

To read in the test models at the top level, use the `read_test_model` command:

```
read_test_model test_model_filenames
```

Use the `list_test_models` command to create a list of the test models loaded and the associated designs, or use the `read` command for the subdesign, as with any other .ddc file. When reading this .ddc file at the top level, DFT Compiler automatically uses the test model for the subdesign.

You can use the top-level DFT flow, described in “Top-Down Scan Insertion” on page 1-44, to accomplish scan insertion at the top level of the design. DFT Compiler uses the test models loaded instead of the .ddc files for subdesigns.

Important:

DFT Compiler does not touch scan chains within the test models. It combines scan chains only at the top level to make the total lengths of the combined chains as close as possible.

DFT Compiler recognizes when lock-up latches have been inserted in a test model and does not insert another lock-up latch when connecting scan chains between two test models unless it is necessary.

After top-level scan insertion, write out the test protocol file in Standard Test Interface Language (STIL) format (the only supported format) and write out a Verilog or VHDL top-level netlist for the top-level design. For example, write out top.spf and top.v, as shown in the following commands:

```
dc_shell>  write_test_protocol -output top.spf
dc_shell>  write -format verilog -output top.v
```

If your top-level design contains test models for a subdesign, DFT Compiler does not write out a gate-level netlist for that subdesign, even if you use the `-hierarchy` option. The netlist written contains an empty module for the test model subdesign. You must write a gate-level netlist for each subdesign separately.

Example 1-3 shows how test models can be used at the top level of a design.

Example 1-3 Test Model Usage

```
dc_shell> remove_design -all
dc_shell> read_file -format verilog ALARM_BLOCK.v
dc_shell> read_file -format verilog TIME_BLOCK.v
dc_shell> read_file -format verilog COMPUTE_BLOCK.v
dc_shell> read_test_model alarm_sm_2.ctlddc
dc_shell> read_file -format verilog COMPARATOR.v
dc_shell> link
dc_shell> list_test_models
dc_shell> current_design COMPUTE_BLOCK
dc_shell> link
dc_shell> create_clock CLK -period 100 -waveform [list 0 50]
dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock -timing [list 45 55] -port CLK
dc_shell> set_dft_signal -view existing_dft \
               -port RESETN -type Reset -active_state 0
dc_shell> compile -scan
dc_shell> set_scan_configuration -chain_count 1
dc_shell> set_scan_path chain0 -view spec \
               -ordered_elements [list U1 U5/chain0 U2]
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> preview_dft -show all
dc_shell> insert_dft
dc_shell> dft_drc -verbose
dc_shell> report_scan_path -chain all
dc_shell> write -format ddc -hierarchy -output COMPUTE_BLOCK.ddc
dc_shell> write -format verilog -hierarchy -output COMPUTE_BLOCK.v
dc_shell> write_test_model -output COMPUTE_BLOCK.ctlddc
dc_shell> set_app_var test_stil_netlist_format verilog
dc_shell> write_test_protocol -output COMPUTE_BLOCK.spf
dc_shell> exit
```

Example 1-3 shows a typical script for the HSS flow. Note that the `dft_drc` command is run right after the `insert_dft` command to verify that scan insertion is correct, without errors. If the `dft_drc` command reports any serious violations, such as scan chain blockages, you must fix these and rerun the script. After the `dft_drc` command is successful, you can run the `report_scan_path -chain all` command to get more information about scan chains.

Note:

When you use the HSS flow, you must run the `dft_drc` command after scan insertion and check the results. The results from the `report_scan_path -chain all` command can be considered only after the `dft_drc` command is error-free.

Reading Designs Into TetraMAX

By default, in the TetraMAX tool, if you read in two modules with the same name, the last one takes precedence. If you have a top-level netlist with empty submodules, read it into the TetraMAX tool first, and then read in the netlists for the submodules. For example,

```
BUILD> read_netlist top.v
BUILD> read_netlist module_1.v module_2.v ... module_n.v
BUILD> run_build_model top
```

Managing Test Models

Figure 1-8 shows the test models that DFT Compiler can create after you insert scan chains. You can save the test models by using the `write` or `write_test_model` command.

Figure 1-8 Creating and Saving Test Models

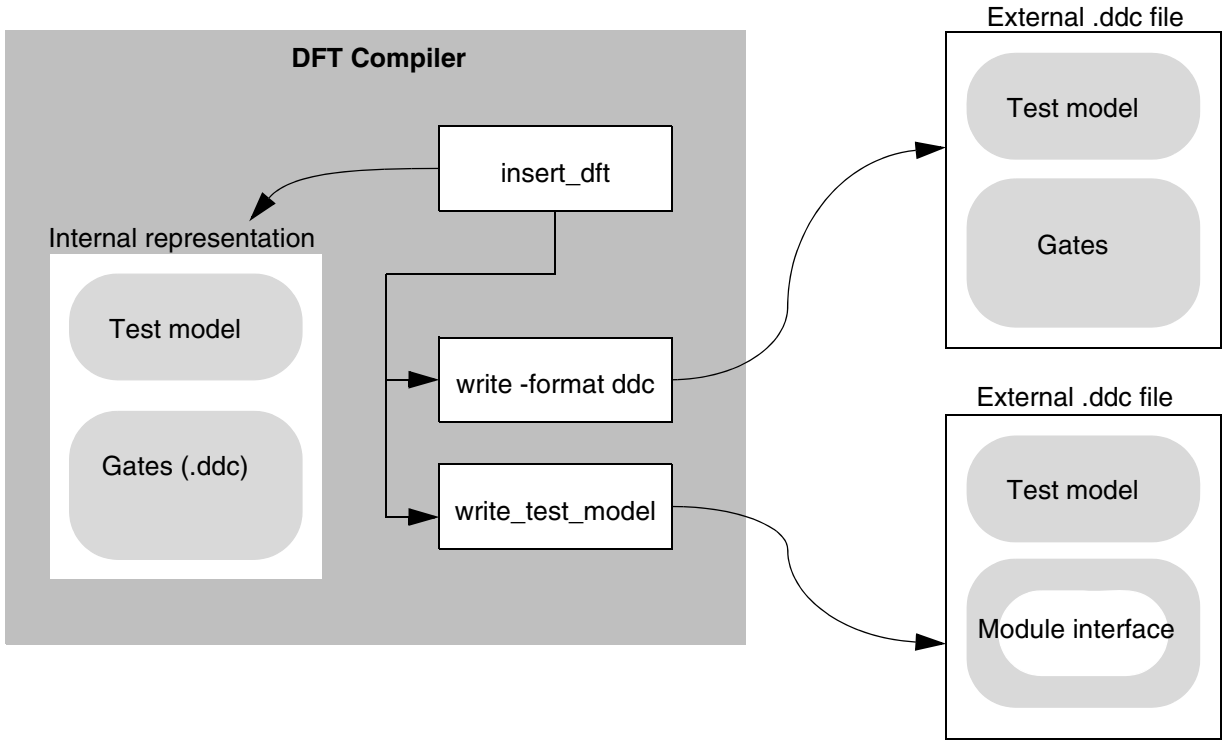
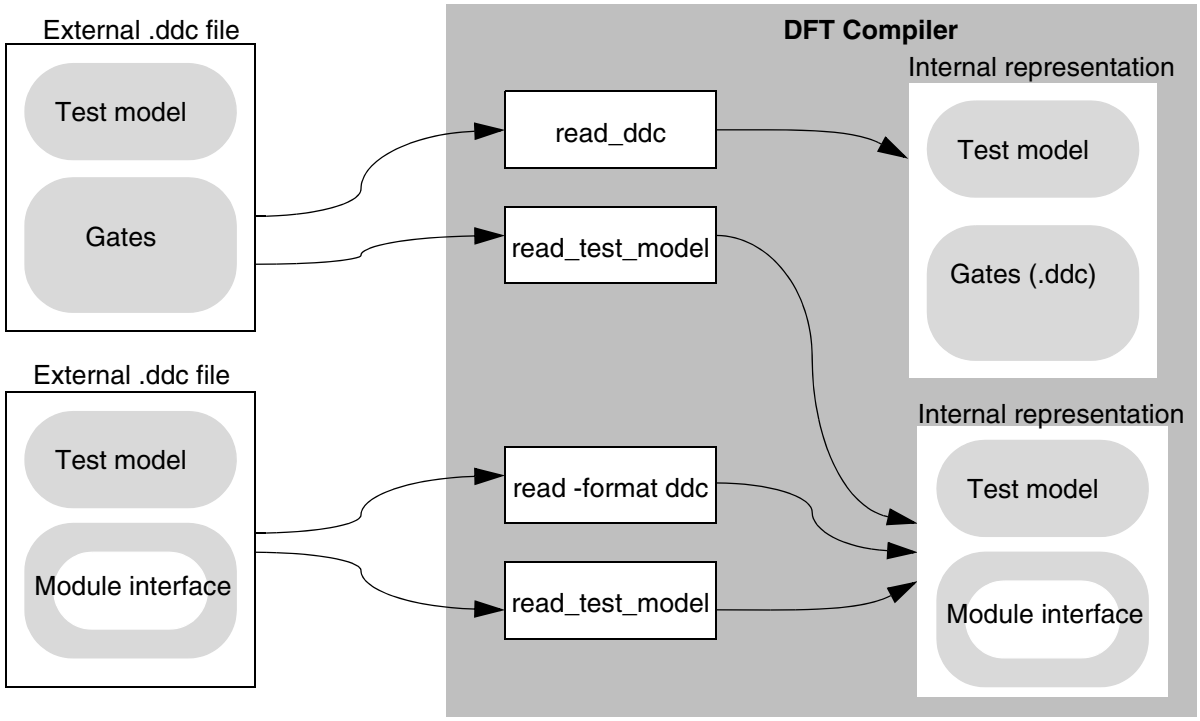


Figure 1-9 shows how the test models can be read into DFT Compiler by using the `read_file` or `read_test_model` command.

Figure 1-9 Using Test Models



The different commands that you can use to write and to read test models enable you to tailor the flow you use to meet your needs. There are several possible flows:

- Use the `write_test_model` command to save a test model, and use `read_file` or `read_test_model` to read it into DFT Compiler.

  In this case, the file created by `write_test_model` contains only the test model and module interface information, so this flow uses less memory and reduces execution time.

  This flow is recommended for the greatest reduction in runtime and memory usage, but you have the flexibility of using either of the following flows.

- Use the `write` command to save a design with a test model, and use `read_test_model` to read it into DFT Compiler.

  The file created by the `write` command contains both a test model and gate information. But the `read_test_model` command reads in only the test model portion of the file and notifies you that DFT Compiler has removed all implementation information. This flow uses less memory and reduces execution time.

- Use the `write` command to save a design with a test model, and use the `read_file` command to read it into DFT Compiler.

The `write` command creates a file containing both the test model and the gate information. The `read_file` command reads in both the test model and the gate information, but DFT Compiler uses the test model by default. Because of the test model usage, the execution time is reduced, but you do not save as much memory as with the other flows. The flat design flow does not use commands specific to test models, so it might fit into your existing flow with minimal changes.

Top-Level Integration

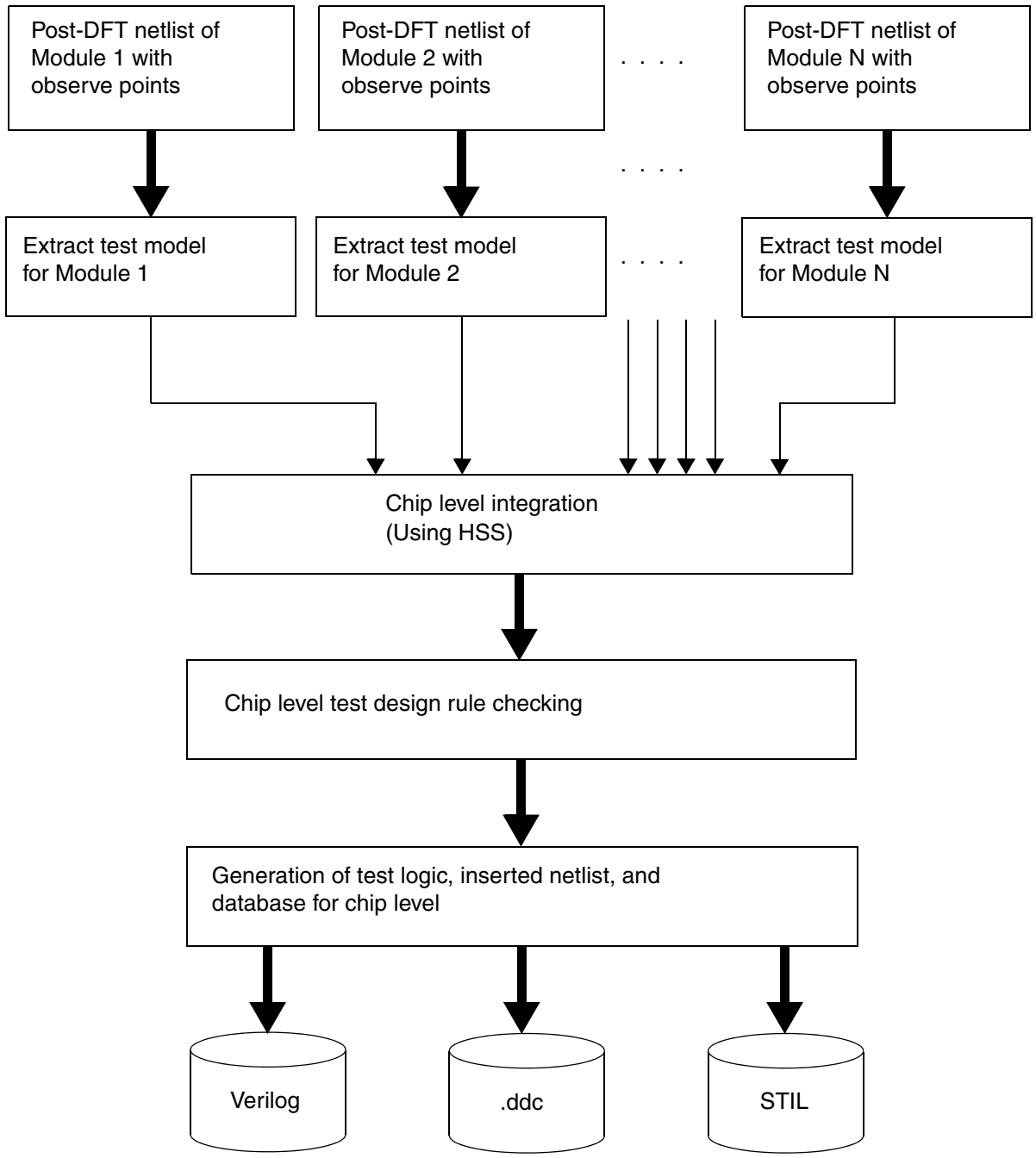
After the individual modules are complete, with scan chain and observe test logic inserted, stitch the modules together at the top level.

When you use the HSS flow, keep the following points in mind:

- If you are inserting observe points, no observe analysis is performed for any user-defined logic or glue logic at the top level.
- If dedicated clocks exist in some of the modules, they need to be stitched with a chip-level pin to avoid creating another chip-level port.
- You cannot integrate blocks that were created using the internal pin flow.

Figure 1-10 shows the HSS flow.

Figure 1-10 Chip Level Integration Using the HSS Flow



Checking Connectivity to Test Models During Integration

In an integration flow, you must ensure that the TestMode and Constant signals entering a block match what is required to shift the scan chains through the block. If conditions do not match, scan blockages can result. Use design rule checking to confirm that these conditions are met before DFT insertion. Before running the `dft_drc` command, set the following variable to enable this connectivity checking functionality:

```
dc_shell> set_app_var test_validate_test_model_connectivity true
```

The `dft_drc` command simulates the test_setup procedure and reports any mismatches between actual and expected values on the TestMode and Constant signals of instances represented by the test models.

If a mismatch is detected, the scan segments represented in the test model are not stitched onto the scan chains.

Using Test Fast Feedthrough Analysis

The `dft_drc` command can now better infer feedthrough logic through complex design blocks or cells during pre-DFT checking. If you have complex design blocks that are described using Verilog models through the `test_simulation_library` variable, you should enable this new capability. Instances containing nested test models also benefit from this new inference capability. To enable this feature, set the following variable:

```
dc_shell> set_app_var test_fast_feedthrough_analysis true
```

Hierarchical ScanEnable Integration

When you insert DFT at a top level that contains cores, the cores already contain complete scan-enable networks. Instead of connecting the top-level ScanEnable signal to target pins inside the core, DFT Compiler must connect to ScanEnable signal pins at the core boundary.

When ScanEnable signals at the core and/or top level are defined with the `-usage` option of the `set_dft_signal` command, DFT Compiler attempts to determine which top-level signal should drive each core-level signal, using the priorities shown in Table 1-2. The column headers along the top denote various core-level scan-enable usages. For each usage, that

column shows the priorities used to determine how top-level signals are connected to that core-level signal.

Table 1-2 Core-Level and Top-Level Scan-Enable Usages With Priorities

Core / Top	scan	clock_gating	scan plus clock_gating	No usage specified
scan	1		2	1
clock_gating		1		
scan plus clock_gating	2	2	1	2
No usage specified	3	3	3	3
New port created (test_se)	4[1]	4[1]	4[1]	4[1]

1. Only one new port is created for all core pin configurations requiring a new port; this new port will behave like a scan-enable signal with no usage specified for all further DFT integration purposes.

These connection priorities propagate signal usages upward through the hierarchy while preserving the original usage intent as much as possible. You can use the `set_dft_signal -connect_to` command and related options to specify specific source-to-pin signal connections that override these default signal connection behaviors. For more information, see "Connecting the Scan-Enable Signal in Hierarchical Flows" on page 6-36.

DFT Flows in Design Compiler Topographical Mode

DFT Compiler also works within the Design Compiler topographical domain shell (`dc_shell -topographical_mode`). Whereas `dc_shell` uses wire load models for timing and area power optimizations, Design Compiler topographical mode uses placement timing values instead. For more information, see the *Design Compiler User Guide*.

This section describes the DFT features that are supported in Design Compiler topographical mode. The following topics are covered:

- Supported DFT Features
- DFT Insertion in Design Compiler Topographical Mode
- Hierarchical Support in Design Compiler Topographical Mode

Supported DFT Features

The following DFT features are supported:

- Basic scan and adaptive scan flows
- Multivoltage
- Multiple timing modes, that is, Design Compiler “multimode” (MM) timing for different functional modes (see Design Compiler documentation for further information)
- Clock and asynchronous AutoFix flow for uncontrollable clocks and asynchronous set and reset signals
- Automatic test point insertion
- User-defined test point insertion
- Internal pins flow
- Multiple user-defined test modes
- Hierarchical scan flows
- Boundary scan insertion with BSD Compiler
- On-chip clocking (OCC) controller support
- Memories with test models

DFT Insertion in Design Compiler Topographical Mode

In Design Compiler topographical mode, DFT Compiler creates an initial topographical ordering during DFT insertion. The virtual layout information is used, where available, to ensure that there is no severe impact on the functional timing.

Example 1-4 shows a script that performs DFT insertion in a Design Compiler topographical mode flow.

Example 1-4 DFT Insertion Script for Design Compiler Topographical Mode

```
compile_ultra -scan
create_test_protocol
dft_drc
preview_dft

insert_dft
dft_drc

# perform incremental optimization after DFT insertion
compile_ultra -incremental -scan
```

When you run DFT Compiler in this mode, the `preview_dft` and `insert_dft` commands issue the following message to indicate that topographical information is used:

```
Running DFT insertion in topographical mode.
```

In Design Compiler topographical mode, only DFT insertion is supported by the `insert_dft` command. New logic is mapped but not optimized. A warning message is issued if you attempt to enable optimization:

```
dc_shell-topo> set_dft_insertion_configuration \
                  -synthesis_optimization all

Warning: Synthesis optimizations for DFT are not allowed in
DC-Topographical flow. Turning off all the optimizations.
Accepted insert_dft configuration specification.
```

After the `insert_dft` command completes, perform an incremental topographical compile to optimize the design. Design Compiler synthesis optimizes the newly inserted DFT logic, and it optimizes the design to accommodate the additional area and timing overhead of the DFT logic.

For DFTMAX compressed scan designs, if you are using Design Compiler Graphical by specifying the `-spg` option of the `compile_ultra` command, the tool

- Builds congestion-aware decompressor and compressor structures to reduce congestion.
- Performs scan chain reordering in the incremental compile to reduce scan chain wire length.

For more information on scan compression, see the *DFTMAX Compression User Guide*.

When the design is transferred to the IC Compiler tool, use the SCANDEF-based scan chain reordering flow to perform reordering using detailed placement information. See “SCANDEF-Based Reordering Flow” on page 10-15.

Hierarchical Support in Design Compiler Topographical Mode

Design Compiler topographical mode supports two flows:

- Top-Level Design Stitching Flow
- Performing a Bottom-up or Hierarchical Compile

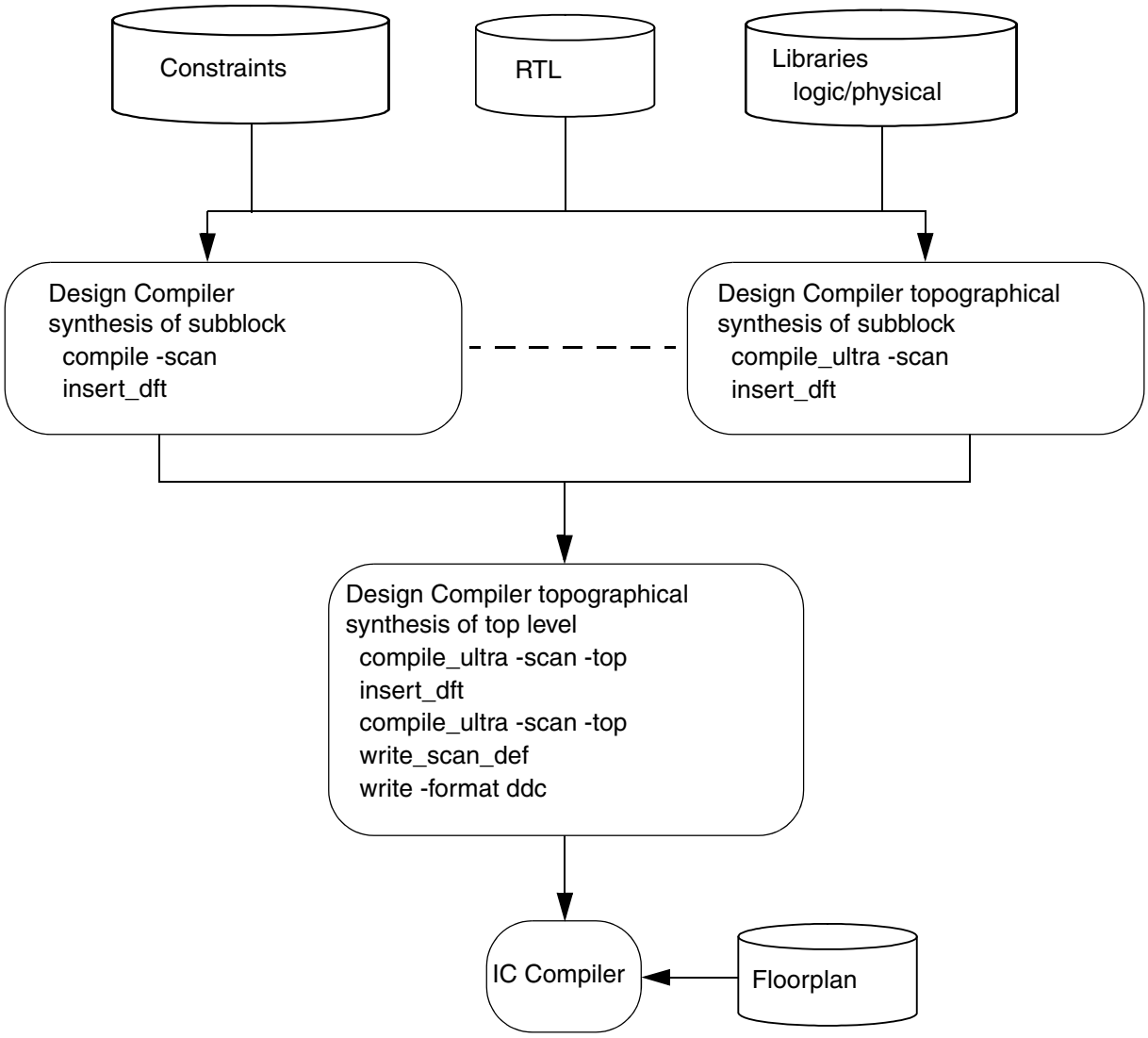
The top-level design stitching flow is used to stitch compiled physical blocks into the top-level design.

The bottom-up hierarchical flow is used when you need to address design and runtime challenges or when you want to use a divide-and-conquer synthesis approach. For more information on the bottom-up HSS flow or the hierarchical adaptive scan synthesis (HASS) flow, see “Hierarchical Scan Synthesis Flow” on page 1-18 and Chapter 3, “Hierarchical Adaptive Scan Synthesis,” in the *DFTMAX Compression User Guide*, respectively.

Top-Level Design Stitching Flow

Figure 1-11 shows the top-level design stitching flow in Design Compiler topographical mode.

Figure 1-11 Top-Level Design Stitching Flow



To perform the top-level design stitching flow:

1. Set up the design and libraries.
2. Read in the top-level design.
3. Compile each subblock by performing the following steps:
   a. Set the current design to the subblock.
   b. Apply timing constraints and power constraints.

c. Perform test-ready compile by using the `compile_ultra -scan` command.

d. Apply the DFT constraints.

e. Use the `insert_dft` command in the subblock so that the inserted scan chains can be included in the top-level scan chain.

f. Run the `compile_ultra -scan -incremental` command.

4. Set the current design to the top-level design, link the design, and apply the top-level timing constraints.
5. Run the `compile_ultra -scan -top` command.

   The `-top` option maps the top-level logic, and the `-scan` option enables the tool to map sequential cells to the appropriate scan flip-flops.
6. Apply DFT constraints.
7. Run the `insert_dft` command to insert scan chains at the top level, followed by `compile_ultra -scan -top` to map any additional unmapped logic that might have been introduced.

Performing a Bottom-up or Hierarchical Compile

In a hierarchical topographical compile flow, you compile the subdesigns separately and then incorporate them in the top-level design. This is also known as a bottom-up flow.

In topographical mode, the tool can read the following types of hierarchical blocks:

- Netlists generated in topographical mode
- Block abstractions generated in topographical mode or in the IC Compiler tool
- Interface logic models (ILMs) generated in topographical mode or in the IC Compiler tool

You cannot combine block abstractions and ILMs in the same flow. Figure 1-12 provides an overview of the hierarchical flow for designs containing block abstractions. Figure 1-13 provides an overview of the hierarchical flow for designs containing ILMs. For detailed information on the steps used in a bottom-up topographical compile flow, see the topographical synthesis chapter of the *Design Compiler User Guide*.

Figure 1-12 Overview of the Hierarchical Flow for Designs Containing Block Abstractions

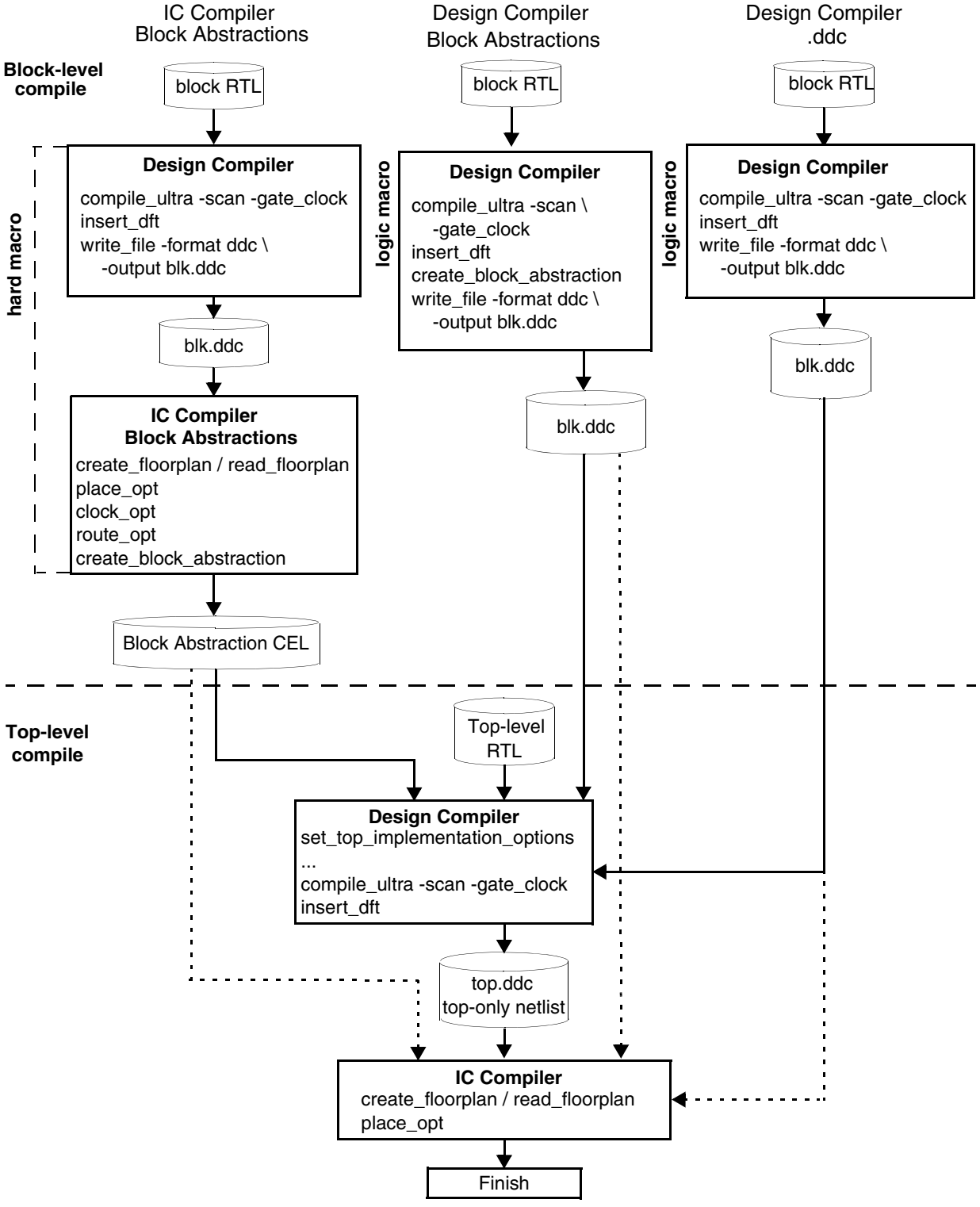
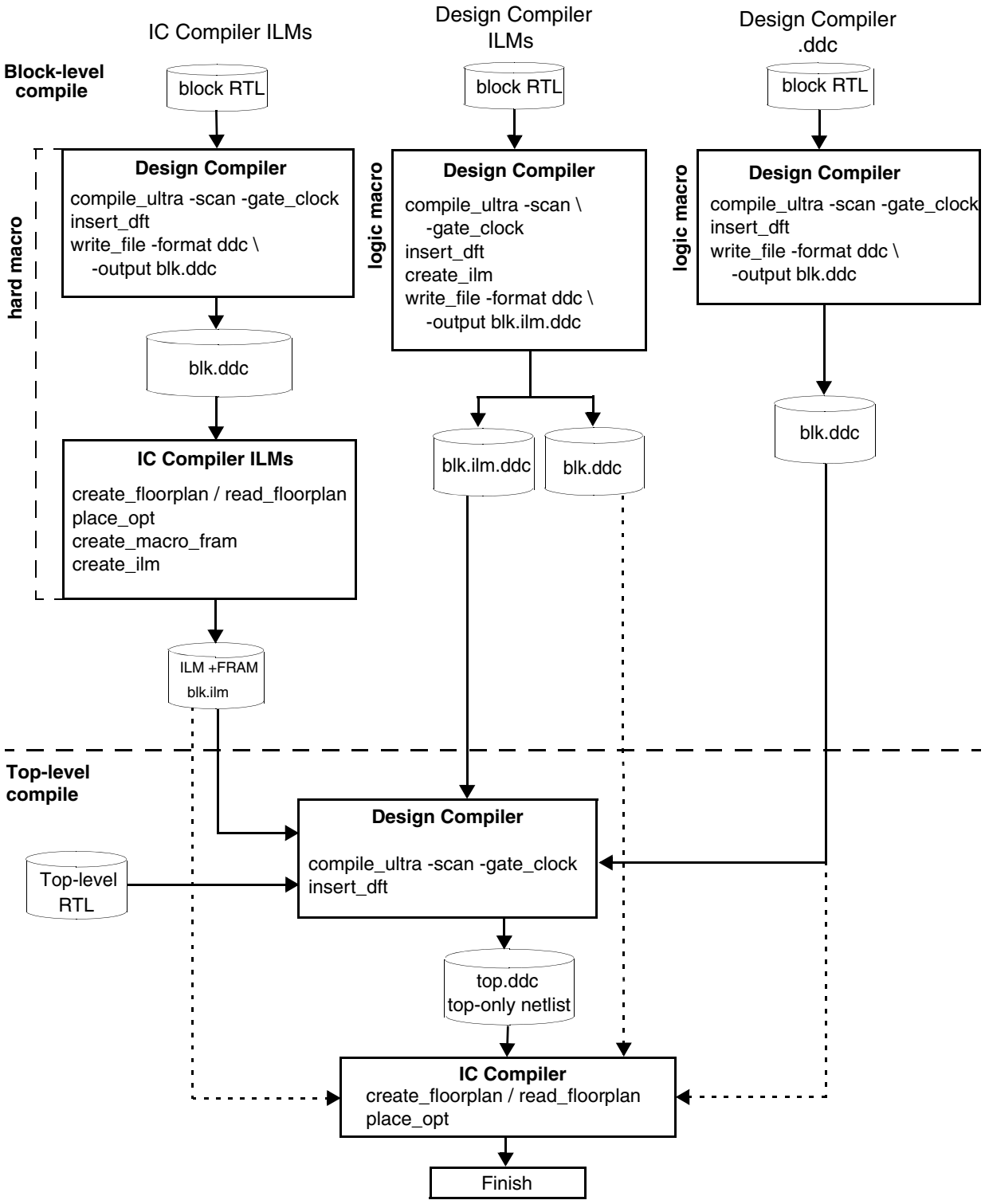


Figure 1-13 Overview of the Hierarchical Flow for Designs Containing ILMs



In these flows, DFT is inserted as each block is compiled. As a result, a hierarchical topographical flow is also a hierarchical DFT flow. Hierarchical DFT flows include the hierarchical scan synthesis (HSS), hierarchical adaptive scan synthesis (HASS), and Hybrid flows. The hierarchical DFT flow depends on whether scan compression is used, and if so, where it is applied. For more information on the HSS flow, see “Hierarchical Scan Synthesis Flow” on page 1-18. For more information on the HASS and Hybrid flows, see Chapter 3, “Hierarchical Adaptive Scan Synthesis,” in the *DFTMAX Compression User Guide*.

At the block level, SCANDEF and CTL model files are generated for each block. The SCANDEF file contains information about scan ordering requirements, and the .ctl file contains information about the DFT logic. The information in these files is also stored in the .ddc file, to be used by top-level topographical DFT insertion and reordering. However, the ASCII files can be used for reference.

At the top level, a SCANDEF file is created for the layout tool. For topographical full-netlist blocks, block abstractions, and ILMs, use the `-expand_elements` option of the `write_scan_def` command to include all scan elements in the SCANDEF. This allows detailed reordering in layout. The scan elements of IC Compiler block abstractions and ILMs are abstracted in the SCANDEF file using the BITS construct, as detailed reordering has already been performed for these blocks.

For more information on SCANDEF files, see “SCANDEF-Based Reordering Flow” on page 10-15.

The top level can also contain hierarchical blocks that are not physically aware. Examples are hierarchical blocks that are elaborated from the RTL at the top level or full .ddc netlist files created in the Design Compiler tool. These blocks can be DFT-inserted, but it is not a requirement; DFT Compiler can insert DFT in these blocks during top-level DFT insertion.

Scan Insertion Methodologies

DFT Compiler supports bottom-up and top-down scan insertion. To improve DFT Compiler performance on multimillion-gate designs, use the hierarchical scan synthesis (HSS) flow. The HSS flow reduces runtime and memory usage, and significantly increases the capacity of DFT Compiler.

For more information on HSS, see “Hierarchical Scan Synthesis Flow” on page 1-18.

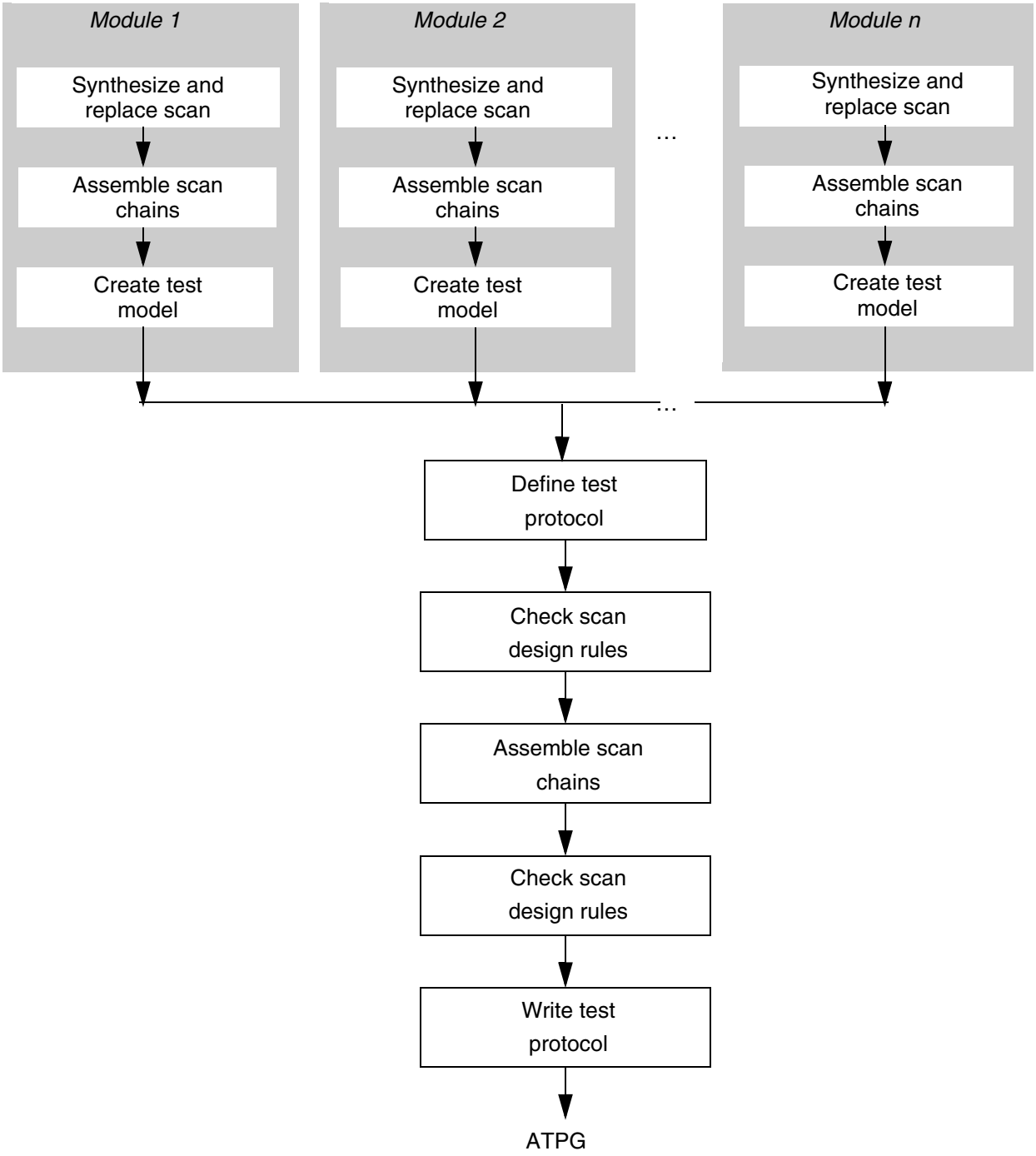
This section discusses the following scan insertion methodologies:

- Bottom-Up Scan Insertion
- Top-Down Scan Insertion

Bottom-Up Scan Insertion

For bottom-up scan insertion, follow the flows outlined earlier in this chapter for each module of the design. For each subdesign, save a test model for the scan-inserted design. At higher levels, stitch together the test models instead of using the .ddc files containing the gate information of the subdesigns. Figure 1-14 shows the bottom-up scan insertion flow.

Figure 1-14 Bottom-Up Scan Insertion Flow



To use the bottom-up scan insertion flow, follow these steps:

1. Synthesize your design, performing scan replacement.
2. Build scan chains in each module of the design.
3. Define the test protocol at the top level.

   The test protocol provides information about your design to the test design rule checker.
4. Check design rules at the top level.

   Evaluate the scan conformance of your design, and determine if any sequential cells violate the test design rules. Test design rule checking also highlights design issues that can lower your fault coverage results.

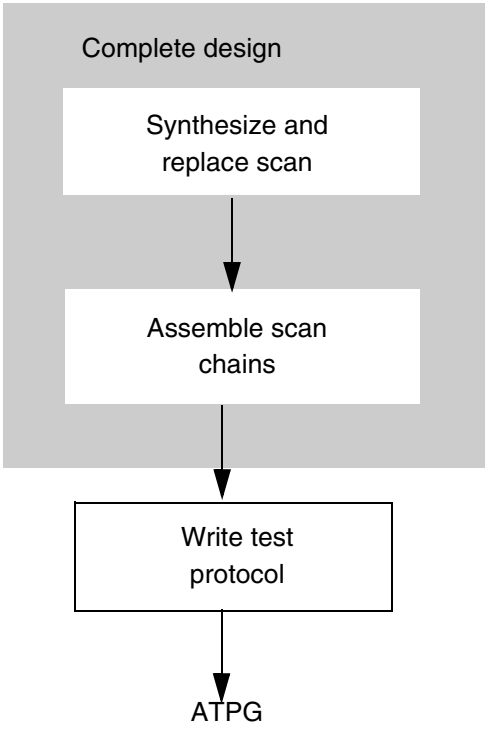
   If the results of your analysis show that the design does not meet your requirements, you can often improve the results by modifying the test protocol. However, in some cases, you might have to modify the HDL description.
5. Assemble the scan chains.

   After inserting scan cells in all modules, assemble the scan chains at the top level of the design. Use the same procedure for assembling scan chains at the top level that you used at the module level, and then run `dft_drc` to make sure the resulting scan chains operate properly.
6. Write out the Verilog netlist and the test protocol, and then proceed to the TetraMAX tool to perform DRC, and ATPG.

Top-Down Scan Insertion

With top-down scan insertion, perform scan insertion only at the top level of the design. Figure 1-15 shows the top-down scan insertion flow.

Figure 1-15 Top-Down Scan Insertion



To use the top-down scan insertion flow, follow these steps:

1. Synthesize your design, performing scan replacement.
2. Build scan chains at the top level of the design.
3. Write out the Verilog netlist and the test protocol, and then proceed to the TetraMAX tool to perform DRC, and ATPG.

DFT Compiler Default Scan Synthesis Approach

If you do not specify scan detail, the `insert_dft` command implements a default scan design by using the full-scan methodology. This section describes the ground rules that the `preview_dft` and `insert_dft` commands apply to generate a default scan design.

The section includes the following subsections:

- Scan Replacement
- Scan Element Allocation

- Test Signals
- Pad Cells
- Area and Timing Optimization

Scan Replacement

DFT Compiler performs the following scan replacement tasks during the `insert_dft` command:

- Scan-replaces sequential elements if a scan replacement on the sequential elements was not performed previously, and the cell does not violate test DRC.
- Converts the scan elements that resulted from a test-ready compile or a previous scan insertion back to nonscan elements if test DRC violations prevent their inclusion in a scan chain, and the `set_dft_insertion_configuration -unscan true` command has been issued.

Scan Element Allocation

DFT Compiler allocates scan elements to scan chains in the following manner:

- Allocates scan elements to produce the minimum number of scan chains consistent with clock domain requirements. By default, the `insert_dft` command generates a scan design with the number of scan chains being equal to the number of clock domains. The resulting design contains one scan chain for each set of sequential elements clocked by the same edge of the same test clock.
- Automatically infers existing scan chains both in the current design and in subdesigns. This is true only if the design has the proper attributes.
- Does not reroute existing scan chains built by the `insert_dft` command or subdesign scan chains built by the `insert_dft` command, even if the existing routing does not conform to default behavior.
- Orders scan elements in scan chains alphanumerically. By default, the `insert_dft` command alphanumerically orders scan elements within scan chains across the full hierarchical path specification of the scan element name.

Test Signals

DFT Compiler inserts and routes test signals in the following manner:

- Automatically inserts and routes global test signals to support the specified scan style. These test signals include clocks and enable signals.
- Allocates ports to carry test signals. Where possible, the `insert_dft` command uses “mission” ports (that is, normal function ports) to carry scan-out ports and inserts multiplexing logic, if required. The `insert_dft` command performs limited checking for existing multiplexing logic to prevent redundant insertion.
- Inserts three-state and bidirectional disabling logic during default scan synthesis. The `insert_dft` command checks for existing disabling logic to prevent redundant insertion.

Pad Cells

If the current design includes pad cells, the `insert_dft` command identifies the pad cells and correctly inserts test structures next to them by

- Ensuring correct core-side hookup to all pad cells and three-state drivers
- Inserting required logic to force bidirectional pads carrying scan-out signals into output mode during scan shift
- Inserting required logic to force bidirectional pads carrying scan-in, control, and clock signals into input mode during scan shift
- Determining requirements and, if necessary, inserting required logic to force all other nondegenerated bidirectional ports into input mode during scan shift
- Inserting required logic to enable three-state output pads associated with scan-out ports during scan shift
- Inserting required logic to disable three-state outputs that are not associated with scan-out ports during scan shift

Area and Timing Optimization

By default, the `insert_dft` command uses constraint-optimized scan insertion to reduce the scan-related impact on performance and area. This process minimizes constraint violations and eliminates compile design rule errors. The `insert_dft` command uses the clock waveforms described by the `create_clock` command to determine whether a logic path meets performance constraints. The `insert_dft` command does not use the timing

values described by using `set_dft_signal` for constraint optimization. The `insert_dft` command also selects scan-out signal connections (Q or QN) to minimize constraint violations.

Scan chain synthesis is concerned primarily with the scan shift operation. The `dft_drc` command identifies problems associated with scan capture that might require you to resolve problems caused by functional clock waveforms.

Scan chains synthesized by the `insert_dft` command are functional under zero-delay assumptions. Before you perform scan synthesis, you can specify test clocks, using the `create_clock` command. All test clocks have the same period.

Getting the Best Results With Scan Design

To get the best scan design results, your ATPG tool must be able to control the inputs and observe the outputs of individual cells in a circuit. By observing all the states of a circuit (complete fault coverage), the ATPG tool can check whether the circuitry is good or faulty for each output. The quality of the fault coverage depends on how well a device's circuitry can be observed and controlled.

If the ATPG tool cannot observe the states of individual sequential elements in the circuit, fault coverage is lowered because the distinction between a good circuit and a faulty circuit is not visible at a given output.

To maximize your fault coverage, follow these recommendations:

- Use full scan.
- Fix all design rule violations.
- Follow these design guidelines:
  - Be careful when you use gated clocks. If the clock signal at a flip-flop or latch is gated, a primary clock input might not be able to control its state. If your design has extensive clock gating, use AutoFix or provide another way to disable the gating logic in test mode.

    Note:
    DFT Compiler supports gated-clock structures inserted by the Power Compiler tool.

  - Generate clock signals off-chip or use clock controllers compatible with DFT Compiler. If uncontrollable clock signals are generated on-chip, as in frequency dividers, you cannot control the state of the sequential cells driven by these signals. If your design includes internally generated, uncontrollable clock signals, use AutoFix or provide another way to bypass these signals during testing.

- Minimize combinational feedback loops. Combinational feedback loops are difficult to test because they are hard to place in a known state.
- Use scan-compatible sequential elements. Be sure that the library you select has scannable equivalents for the sequential cells in your design.
- Avoid uncontrollable asynchronous behavior. If you have asynchronous functions in your design, such as flip-flop preset and clear, use AutoFix so that you can control the asynchronous pins or make sure you can hold the asynchronous inputs inactive during testing.
- Control bidirectional signals from primary inputs.

The scan design technique does not work well with certain circuit structures, such as

- Large, nonscan macro functions, such as microprocessor cores
- Compiled cells, such as RAM and arithmetic logic units
- Analog circuitry

For these structures, you must provide a test method that you can integrate with the overall scan-test scheme.

DFT Compiler and Power Compiler Interoperability

This section discusses issues associated with the interoperability of DFT Compiler and Power Compiler. It includes the following subsections:

Improving Testability in Clock Gating

Clock gating raises some concerns for ATPG. These concerns involve enabling clock controllability and scan testing as well as optimizing ATPG results.

DFT Compiler might not include a clock-gated register in a scan chain during scan synthesis if gating the register clock makes it uncontrollable for test. When the register is excluded from the scan chain, test controllability is reduced at the register input and test observability

is reduced at the register output. If many registers are gated, the fault coverage can be reduced significantly.

If you add control points during clock gating, you can enable scan testing, thus improving the testability of the design. However, by controlling the clock signal, you might introduce untestable faults into the design. To optimize the ATPG results, you might need to insert observation points during clock gating.

This section includes the following subsections:

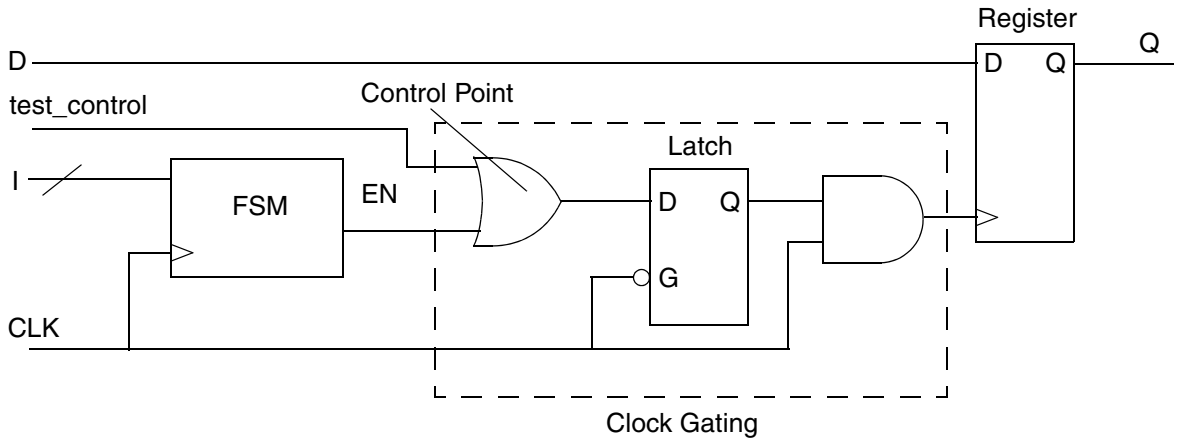
- Inserting Control Points in Control Clock Gating
- Scan-Enable Signal Versus Test-Mode Signal
- Inserting Observation Points to Control Clock Gating
- Choosing a Depth for Observability Logic

Inserting Control Points in Control Clock Gating

A control point is an OR gate driven by a control signal. This signal eliminates the gating of the clock signal during test, thus restoring the controllability of the clock signal.

Figure 1-16 shows a control point (OR gate) placed before a latch connected to a test_control port. When the test_control signal is high, it overrides clock gating, thus making the gated clock and CLK signals identical.

Figure 1-16 Control Point in Gated Clock Circuitry



You can place the control point OR gate in front of or behind the latch. Latch-based clock gating requires that the enable signal always arrives after the trailing edge of the clock signal or the rising edge for a falling-edge clock signal. If the control point is inserted before the latch, it is impossible for the control point to violate this requirement. If it is inserted behind the latch, it benefits the circuit by adding an extra hold time margin for the clock glitch hold

check. It does this by delaying the data signal, which helps the clock pulse arrive ahead of the data.

Inserting the control point after the latch causes performance degradation because a gate is added between the latch and the register. In addition, the `test_control` signal must then transition after the trailing edge (rising edge for falling-edge signal) of the clock signal during test because it does not go through the latch; otherwise glitches in the resulting signal corrupt the clock signal.

You can choose to make the `test_control` port either a scan-enable signal or a test-mode signal. The scan-enable signal is active only during scan shifting. The test-mode signal is active during the entire test.

The `set_clock_gating_style` command has two options for determining the location and type of the control point for test:

- The `-control_point` option can be `none`, `before`, or `after`. The default is `none`. When you use a latch-free clock-gating style, the `before` and `after` options are equivalent.
- The `-control_signal` option can be `test_mode` or `scan_enable`. The default is `scan_enable`. This option uses an existing test port or creates a new test port and connects the test port to the control point OR gate.

  You can use the `-control_signal` option only if the `-control_point` option is used. For existing test ports, you must use the `set_dft_signal` command to define the scan-enable or test-mode signal.

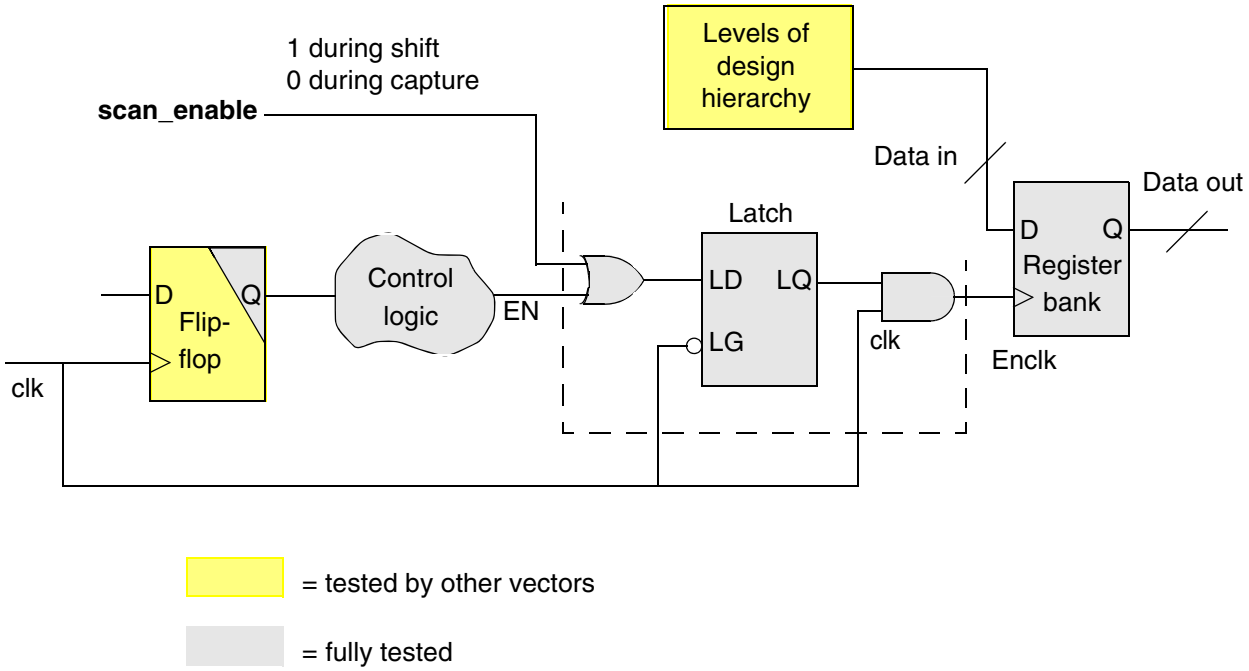
Scan-Enable Signal Versus Test-Mode Signal

The scan-enable and test-mode signals differ in the following ways:

- A scan-enable signal is active only during scan mode.
- A test-mode signal is active during the entire test (scan mode and parallel mode).

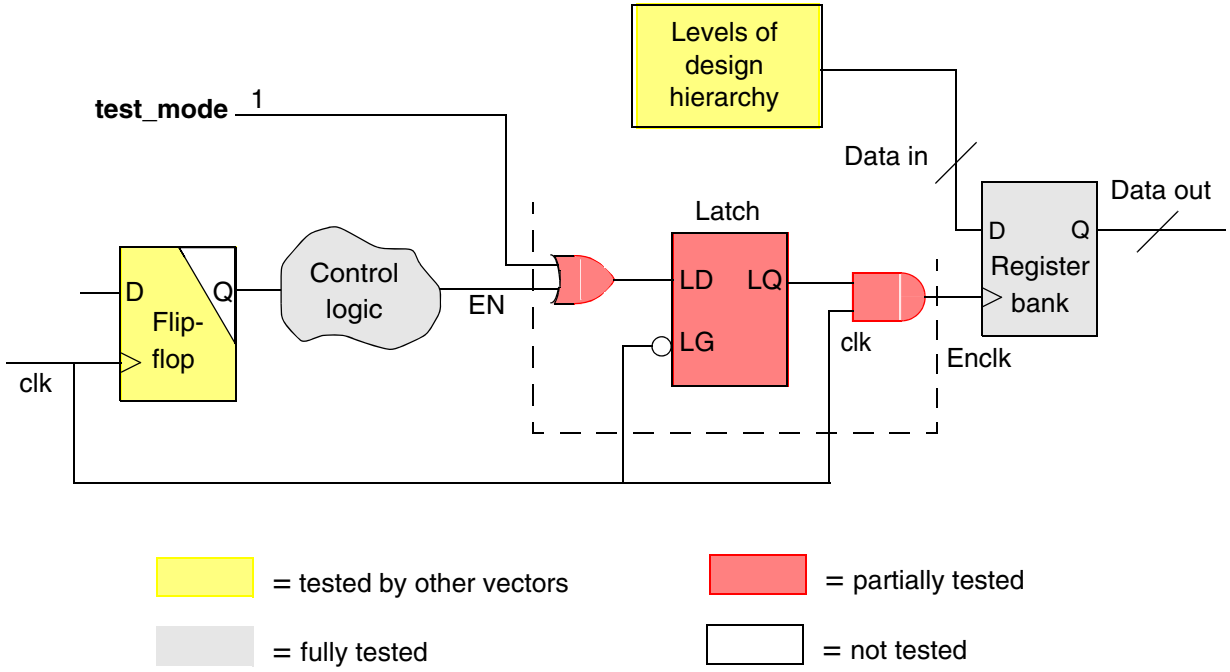
The scan-enable signal typically provides higher fault coverage than the test-mode signal. Fault coverage with the scan-enable signal is comparable to a circuit without clock gating, as shown in Figure 1-17. During the ATPG capture mode, the register bank is clocked whenever the output of the flip-flop is a 1; this behavior matches functional operation.

Figure 1-17 Test Coverage With Scan-Enable Signal



In some situations, you must use a test-mode signal. The control logic preceding the clock-gating circuitry is not tested, as shown in Figure 1-18. In addition, the clock-gating logic can be tested only for stuck-at-1 faults. During test, the test-mode signal is always a logic 1, which forces the clock at the register bank to switch at all times. One way to test the clock-gating logic circuitry is to add observability logic to the design.

Figure 1-18 Test Coverage With Test-Mode Signal

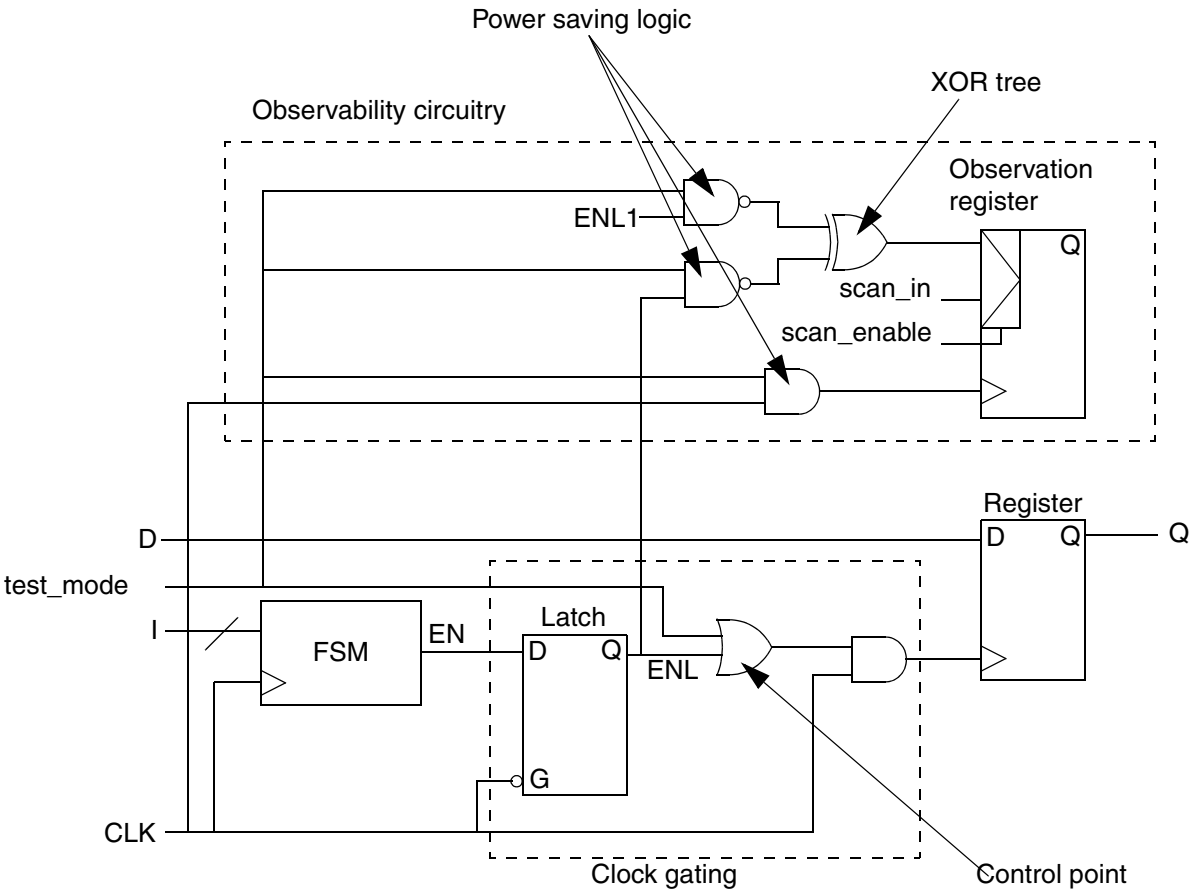


Inserting Observation Points to Control Clock Gating

When you use the test-mode signal, the EN signal and other signals in the control logic are untestable. You can add observation points during clock gating to increase the fault coverage.

Figure 1-19 shows clock-gating circuitry, including an XOR tree and a scan register, to observe the ENL and ENL1 signals, which are outputs of the clock-gating latches.

Figure 1-19 Observability Circuitry in Clock-Gating Circuits



During testing, observability circuitry allows observation of the ENL and ENL1 signals. In normal operation of the circuit, the XOR tree does not consume power, because the NAND gates block all signal transitions of the ENL and ENL1 signals.

The insertion of observation test points has high testability and is power-efficient because the XOR tree consumes power only during test and the clock of the observation register is also gated.

The `set_clock_gating_style` command has two options for increasing observability when you use the `-control_signal test_mode` option:

- The `-observation_point` option can be `true` or `false`. The default is `false`. When you set this option to `true`, the `set_clock_gating_style` command adds a cell that contains at least one observation register and an appropriate number of XOR trees. The

scan chain includes the observation register. The output of the observation register is not functionally connected to the circuit.

- The `-observation_logic_depth` option allows a `depth_value` specification. The default is 5. The value determines the depth of the XOR tree.

Choosing a Depth for Observability Logic

Use the `-observation_logic_depth` option of the `set_clock_gating_style` command to set the logic depth of the XOR tree in the observability cell. The default for this option is 5.

Clock-gating software builds one observability cell for each clock-gated design. Each gated register in the design provides a gated enable signal as input to the XOR tree in the observability cell.

If you set the logic depth of your XOR tree too small, clock gating creates more XOR trees and associated registers to provide enough XOR inputs to accommodate signals from all the gated registers. Each additional XOR tree adds some overhead for area and power. Using one XOR tree adds the least amount of overhead to the design.

If you set the logic depth of your XOR tree too high, clock gating can create one XOR tree with plenty of inputs. However, too large a tree can cause the delay in the observability circuitry to become critical.

Use a value that meets the following two criteria in choosing or changing the XOR logic tree depth:

1. High enough to create the fewest possible XOR trees
2. Low enough to prevent critical delay in the observability circuitry

Power Compiler/DFT Compiler Interoperability Flows

This section suggests methodologies that can help you avoid interoperability problems between Power Compiler and DFT Compiler. You can apply these methodologies directly within the current Power Compiler/DFT Compiler flow.

Using Test-Mode Signals With Power Compiler

The `-control_signal test_mode` option of the `set_clock_gating_style` command relies on the DFT signal definition to stitch an existing test-mode signal to the clock-gating control point.

Power Compiler identifies the test-mode signal in the following way:

1. Power Compiler looks for an existing test-mode signal, defined using the `set_dft_signal -type TestMode` command. The signal can be defined with either `-active_state 1` or `-active_state 0`.
2. If a test-mode signal has not been defined with the `set_dft_signal` command, Power Compiler creates a test_cgtm signal, driven by an input port of the same name.

Using Scan-Enable Signals With Power Compiler

To achieve high-fault-coverage in clock-gated designs, the scan chain must be shifted when the scan-enable signal is asserted. The easiest way to ensure this is to disable all gating elements when the scan-enable signal is asserted. You can achieve this by placing an OR gate or a NOR gate on the EN input, which forces EN on when the scan-enable signal is asserted.

In parallel capture mode, the scan chain must also be clocked to ensure that data is captured in the scan registers. However, using the scan-enable signal to control latch-based and latch-free clock gating might not guarantee this. Furthermore, using the signal to control latch-free clock gating might make the clock pin of the clock-gated registers uncontrollable.

In Figure 1-20, the latch-based clock gating does not have violations that prevent scan insertion. The clock pin is correctly forced to a known state at time = 0, and it correctly changes state as a result of the test clock toggling (toggling to an unknown state X is a change of state). Test DRC verifies that the test clock pulse arrives at the test clock pin to check the scan shift process. Because the scan-enable port is constrained to logic state 1 during this process, the clock pin is controllable.

Figure 1-20 Violation-Free Clock Gating

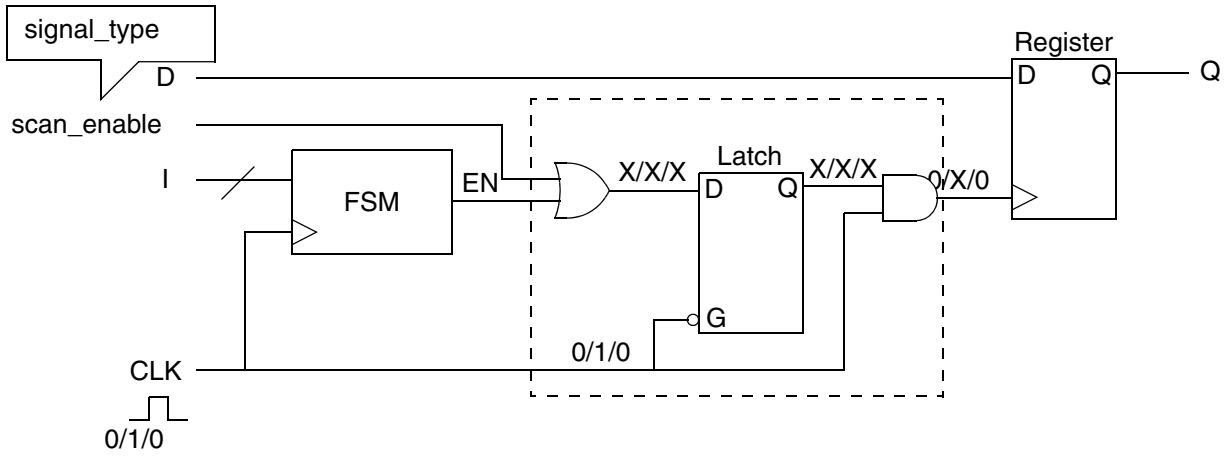
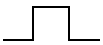
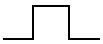
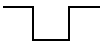
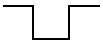
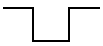
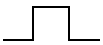
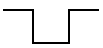


Table 1-3 shows the possible combinations of latch-based clock gating, clock waveforms, control signals and control point location. For each combination, the last column indicates whether scan insertion can be performed on the clock-gated register. Two special cases (marked by **) require that you modify the test protocol.

Table 1-3 Latched-Based Clock-Gating Configurations

Clock gating	CLK	Control signal	Control point location	Register can be inserted?
High Latch-based		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes**
			After latch	Yes
Low Latch-based		test_mode	Before latch	Yes
			After latch	Yes
		scan_enable	Before latch	Yes
			After latch	Yes
		test_mode	Before latch	Yes**
			After latch	Yes
High Latch-based		scan_enable	Before latch	Yes
			After latch	No
Low Latch-based		scan_enable	Before latch	Yes
			After latch	No

Consider the case of high latch-based clock gating controlled by a test-mode signal and driven by a return-to-1 clock. If the control point is inserted before the latch, then when the clock port is off (time = 0), the latch is blocked and the clock pin is not controllable. This violation also occurs in the case of a low latch-based clock gating controlled by a test-mode signal and driven by return-to-0.

To achieve a known state in the latch, add a clock pulse to the `test_setup` section of the test protocol. Use the following `set_dft_drc_configuration` command to update the `test_setup` section with the clock pulse:

```
set_dft_drc_configuration -clock_gating_init_cycles 1
```

If you have multiple cascaded latches and the first latch is loaded with the test-mode signal, use the following `set_dft_drc_configuration` command to update the `test_setup` section with the specified number of clock pulses:

```
set_dft_drc_configuration -clock_gating_init_cycles n
```

Here, *n* equals the number of clock pulses required to initialize clock-gating latches.

Note the following:

- The set of clock cycles should equal the depth of the chain of latches. Make sure this is the case.
- Violations still occur when there are multiple cascaded latches and the scan-enable and control point location are used as before, with mixed active-high and active-low latches.

Consider a latch state that is known, as in the case of a high latch-based clock gating controlled by the scan-enable signal with a return-to-1 clock. When the clock port is off (time = 0), the register does not hold its state. During pre-DFT DRC capture check, the EN signal goes through the control point and the system clock pulse might not arrive at the register clock pin. Because the register might receive invalid data during the capture phase of ATPG, it will not be scan-inserted. The same DRC violations occur for the low latch-based clock gating controlled by the scan-enable signal with a return-to-0 clock.

Latch-Free Clock-Gating Configurations

Table 1-4 shows combinations of latch-free clock gating, clock waveforms, and control signals. For each combination, the last column indicates when the clock-gated register can be scan-inserted and when it cannot.

Table 1-4 Latched-Free Clock-Gating Configurations

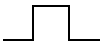
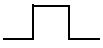
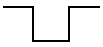
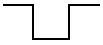
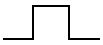
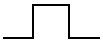
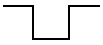
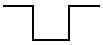
Clock gating	CLK	Control signal	Control point location	Register can be inserted?
High Latch-based		test_mode	Before	Yes
		scan_enable	Before	Yes

Table 1-4 Latched-Free Clock-Gating Configurations (Continued)

Clock gating	CLK	Control signal	Control point location	Register can be inserted?
		test_mode	Before	Yes
		scan_enable	Before	Yes
Low Latch-based		test_mode	Before	Yes
		scan_enable	Before	Yes
		test_mode	Before	Yes
		scan_enable	Before	Yes

Connecting Test Pins to Clock-Gating Cells Using the insert_dft Command

You can use the `insert_dft` command to connect clock-gating cells to the test ports.

The `insert_dft` command has a hookup-test-ports capability that makes clock-gating cell connections as well as carries out the construction of DFT logic in one step. The command makes the connections of the top level test ports to the test pins of the clock-gating cells through the hierarchy. If the design does not have a test port at any level of hierarchy, a new test port is created. If a test port exists, it is used.

Note:

The `insert_dft` command will honor and preserve any clock-gating connections previously made; the command will create only the missing connections.

This feature is enabled by default. You can control whether this functionality is enabled by using the following command:

```
set_dft_configuration -connect_clock_gating enable | disable
```

After you have inserted the clock-gating cells, if this functionality is enabled, you only need to use the `insert_dft` command to connect test ports in a flow similar to the following:

```
read_ddc design.ddc
set_clock_gating_style -control_signal test_mode
compile_ultra -scan -gate_clock
set_dft_signal -type TestMode -port test_mode
. . .
create_test_protocol
preview_dft
insert_dft
```

You can use the `report_dft_configuration` command to report what was previously set and the `reset_dft_configuration` command to restore the setting to the default.

All existing DFT flows except those mentioned in "Limitations" on page 1-64 are supported.

Design Requirements

For the feature to work, the design must have the clock-gating cells inserted before you run the `insert_dft` command. Insertion of clock-gating cells can be done either by using either the `compile_ultra -gate_clock` command or the `insert_clock_gating` command. The design must have the clock-gating attributes present in order for `dft_drc` and `insert_dft` to recognize them as valid clock-gating cells. The attributes help to distinguish between a design that has an erroneous connectivity for which DRC should flag violations and a design that is correct but for which the connectivity is yet to be established by the `insert_dft` command.

If you start with the .ddc format, the clock-gating cell attributes will generally be present. However, if clock-gating cell attributes are not present (for example, if you start with a Verilog netlist), you need to ensure that the required attributes are present for the clock-gating cells so that the `dft_drc` and `insert_dft` commands can recognize them. You can do this by using the `identify_clock_gating` command or by following a Power Compiler recommended flow and manually identifying the clock-gating cells.

Hookup Testport Connections

Table 1-5 describes the connections made by the `insert_dft` command.

Table 1-5 Connections Made to the Clock-Gating Cells by insert_dft

	Clock-gating control signal	DFT command	Top-level port used
1	`scan_enable`	No `set_dft_signal` defined with `-type ScanEnable`	`test_se` created and connected to test pin of clock-gating cell.
2	`scan_enable`	`set_dft_signal -view spec|exist -type ScanEnable -port test_se -active_state 0|1`	No new port created. `test_se` used to connect to test pin of clock-gating cell.
3	`test_mode`	No `set_dft_signal` defined with `-type TestMode`	New port. `test_cgtm` created to connect to test pin of clock-gating cell.
4	`test_mode`	`set_dft_signal -view spec|exist -type TestMode -port test_mode -active_state 0|1`	No new port created. `test_mode` used to connect to test pin of clock-gating cell.

Note:

The DFT signal specifications intended for clock-gating cell connections are not mode-specific. Therefore you cannot specify a test mode using the `-test_mode` option of the `set_dft_signal` command.

Design Rule Checking Changes

The test pin connections of valid clock-gating cells, that is, those identified with Power Compiler clock-gating attributes, are checked by the `dft_drc` command.

After the clock-gating cells are identified, the `dft_drc` command performs the following checks:

1. If the test pin is already connected, no violation will be reported by the `dft_drc` command, and the flip-flops controlled by the clock-gating cell are put onto the scan chain during the `insert_dft` command provided no other violations exist for the cell.
2. If the test pin is not connected, the `dft_drc` command will generate the following warning message for the clock-gating cells:

```
Warning: Clock gating cell %s has unconnected test pin. (TEST-130)
```

This warning is issued irrespective of whether the `set_dft_configuration -connect_clock_gating` option is enabled or disabled.

For all cells controlled by clock-gating cells that are flagged with TEST-130 violations for which the `set_dft_configuration -connect_clock_gating` option is set to `disable`, the `dft_drc` command will not report that flip-flops driven by these clock-gating cells will have clock violations (D1 or D9) and will not be included in scan chains (unless AutoFix is enabled).

For all cells controlled by clock-gating cells that are flagged with TEST-130 violations for which the `set_dft_configuration -connect_clock_gating` option is set to `enable`, the `dft_drc` command will not report that these flip-flops as having clock violations and will stitch these flip-flops into the scan chains.

Specifying Signals for Automatic Clock-Gating Cell Test Pin Connections

By default, DFT Compiler chooses an available ScanEnable or TestMode signal to connect to clock-gating cell test pins, depending on the type of control signal specified with the `set_clock_gating_style command`. However, you can also define a dedicated ScanEnable or TestMode signal to use for these test pin connections.

Specifying a Global Clock-Gating Control Signal

You can define a global clock-gating ScanEnable or TestMode control signal by using the `-usage clock_gating` option when defining the signal with the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable | TestMode
    -view spec
    -usage clock_gating
    -port port_list
```

To define a signal with the `clock_gating` usage, the `-type` option must be set to `ScanEnable` or `TestMode`, and the `-view` option must be set to `spec`.

When you define a clock-gating control signal with the `clock_gating` usage, the `insert_dft` command is limited to using only that signal to connect to the test pins of clock-gating cells. If there are insufficient ScanEnable or TestMode signals for other purposes, DFT Compiler creates additional ScanEnable or TestMode signals as needed.

You can use the `report_dft_signal` and `remove_dft_signal` commands for reporting and removing the specification, respectively.

Specifying Object-Specific Clock-Gating Control Signals

You can also define dedicated ScanEnable or TestMode clock-gating control signals for specific parts of the design by using the `-connect_to` option of the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable | TestMode
    -view spec
    -usage clock_gating
    -port port_list
    [-connect_to object_list]
    [-exclude object_list]
```

The `-connect_to` option specifies a list of design objects that are to use the specified clock-gating control signal. The supported object types are

- Clock-gating cells

  For Power Compiler clock-gating cells, specify the hierarchical clock-gating cell. For existing clock-gating cells identified with the `set_dft_clock_gating_pin` command, specify the leaf clock-gating cell.

- Hierarchical cells
- Designs
- Test clock ports

  This allows you to make clock-domain-based signal connections. It includes clock-gating cells that gate the specified test clocks. The functional clock behavior is not considered.

  Note:
  This specification requires that a functional clock also be defined on the test clock port.

- Scan-enable or test-mode pins of CTL-modeled cores

You can also use the `-exclude` option to specify a list of clock-gating cells, hierarchical cells, or design names to exclude from the object-specific control signal.

The following example defines a ScanEnable signal named SE_CG to connect to the test pins of existing clock-gating cells ICG_CLK100 and ICG_CLK200:

```
dc_shell> set_dft_signal \
            -type ScanEnable -view spec -port SE_CG \
            -usage clock_gating -connect_to {ICG_CLK100 ICG_CLK200}
```

Limitations

Note the following limitations:

- The `insert_dft` command cannot be used on an unmapped design to connect to clock-gating cells, that is, when a design is still in the RTL stage.
- Only the clock-gating cells recognized by Power Compiler are supported for automatic test pin connection. The methodologies recommended for Power Compiler must be followed to enable successful recognition of these clock-gating cells and their test pins.
- For user-instantiated clock-gating cells, test pins must be manually specified. For more information, see “Connecting User-Instantiated Clock-Gating Cells” on page 7-84.
- Clock-gating cell connections are not mode-specific.
- The `insert_dft` and `preview_dft` commands do not report connections made to clock-gating cells.

2

Running RTL Test Design Rule Checking

This chapter describes how to prepare for and run RTL test design rule checking (DRC) and analyze DRC violations.

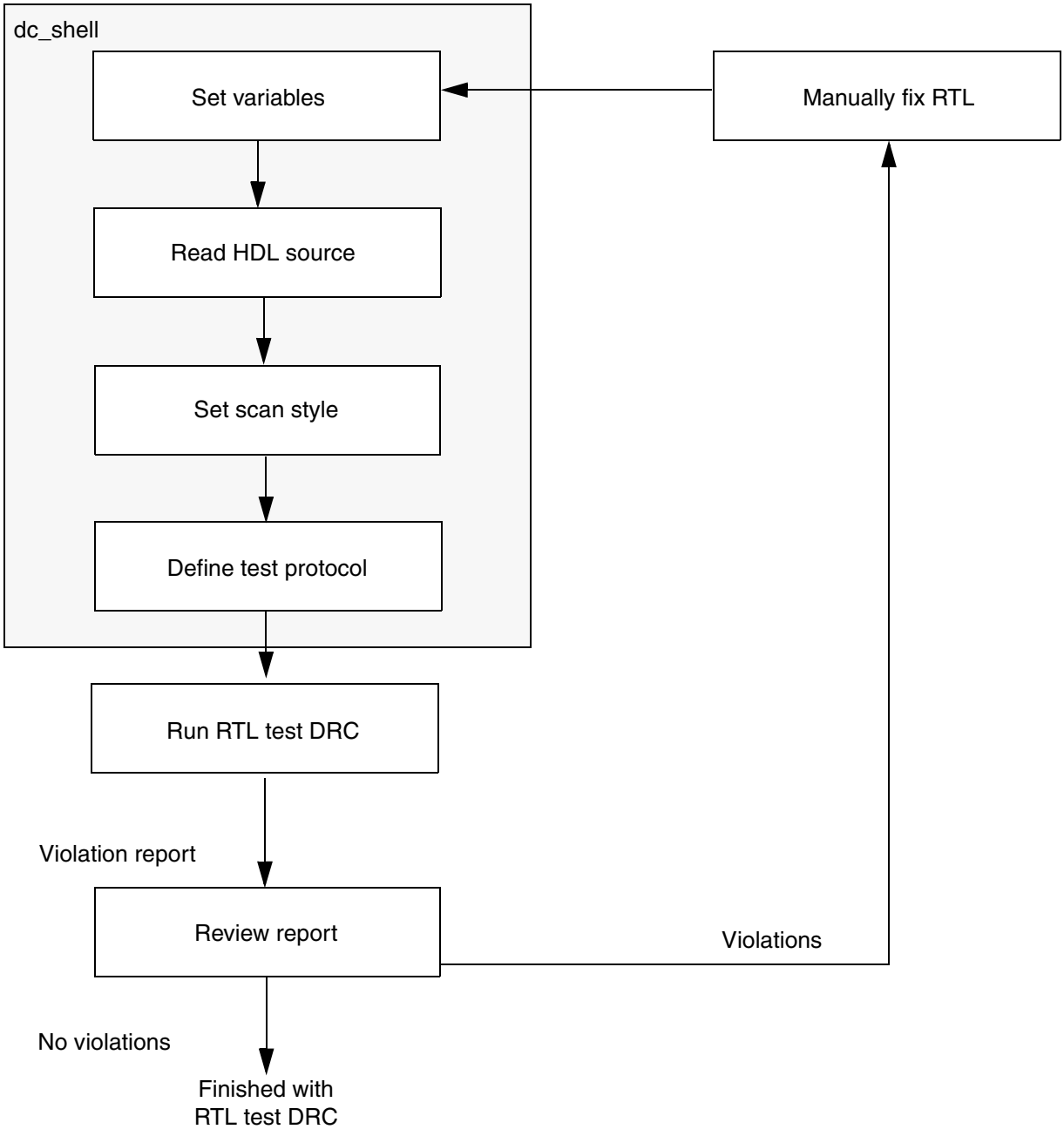
The RTL test design rule checking (DRC) process provides early warnings of test-related issues. This feedback is crucial because it provides you with an opportunity to correct your RTL code before the compile phase of the design flow. By correcting these problems during this stage, you can reduce time-consuming iterations that would occur later in the design process.

This chapter includes the following sections:

Understanding the Flow

Figure 2-1 shows a typical RTL test DRC flow.

Figure 2-1 RTL Test DRC Design Flow



Specifying Setup Variables

To begin preparing for RTL test DRC checking, you need to specify a series of setup variables, as described in the following steps:

1. Set the `hdlin_enable_rtldrc_info` variable to `true`. This variable reports file names and line numbers associated with each violation, which makes it easier for you to later edit the source code and fix violations.

   ```
   dc_shell> set hdlin_enable_rtldrc_info true
   ```

2. Make sure you define the list of searched logic libraries by using the `link_library` variable.

3. Read in your HDL source code by using the `read` variable. The following variable reads in a Verilog file called `my_design.v`:

   ```
   dc_shell> read_file -format verilog my_design.v
   ```

Generating a Test Protocol

A test protocol is required for specifying signals and initialization requirements associated with design rule checking. This section has the following subsections related to generating a test protocol:

- Defining a Test Protocol
- Setting the Scan Style
- Design Examples

Defining a Test Protocol

To define the test protocol, you need to

- Identify all test clock signals by using the `set_dft_signal` command, as shown in the following example:

  ```
  dc_shell> set_dft_signal -view existing_dft \
                -type ScanClock -timing {45 55}
  ```

  Make sure you identify a clock signal as a clock and not as any other signal type, even if it has more than one attribute. An error message will appear if you identify a clock signal with any other attribute.

- Identify all nonclock control signals, such as asynchronous presets and clears or scan-enable signals, using the `set_dft_signal` command.

You should identify the following nonclock control signals:

- Reset
- ScanEnable
- Constant
- ScanDataIn
- ScanDataOut
- TestData
- TestMode

For example,

```
dc_shell> set_dft_signal -view existing_dft \
                -type Reset -active_state 1
```

- Define constant logic value requirements.

  If a signal must be set to a fixed constant value, use the `set_dft_signal` command, as shown in the following example:

  ```
  dc_shell> set_dft_signal -view existing_dft \
                  -type constant -active_state 1
  ```

- Define test-mode initialization requirements.

  Your design might require initialization to function in test mode. Use the `read_test_protocol` command to read in a custom initialization sequence. You can define a custom initialization sequence by modifying the protocol created by the `create_test_protocol` command.

Reading in an Initialization Protocol in STIL Format

The following example reads in a STIL initialization protocol:

```
dc_shell> read_test_protocol -section test_setup my_protocol_file.spf
```

Example 2-1 shows a complete STIL procedure file, including an initialization sequence. The initialization sequence is found in the `test_setup` section of the `MacroDefs` block.

Example 2-1 Complete Protocol File (init.spf)

```
STIL 1.0 {
   Design P2000.9;
}
Header {
   Title "DFT Compiler 2000.11 STIL output";
   Date "Wed Jan  3 17:36:04 2001";
```

```
   History {
   }
}
Signals {
   "ALARM" In; "BSD_TDI" In; "BSD_TEST_CLK" In; "BSD_TMS" In;
   "BSD_TRST" In; "CLK" In; "HRS" In; "MINS" In; "RESETN" In;
   "SET_TIME" In; "TEST_MODE" In; "TEST_SE" In; "TEST_SI" In;
   "TOGGLE_SWITCH" In;
   "AM_PM_OUT" Out; "BSD_TDO" Out; "HR_DISPLAY[0]" Out;
   "HR_DISPLAY[1]" Out; "HR_DISPLAY[2]" Out; "HR_DISPLAY[3]"
Out;
   "HR_DISPLAY[4]" Out; "HR_DISPLAY[5]" Out; "HR_DISPLAY[6]"
Out;
   "HR_DISPLAY[7]" Out; "HR_DISPLAY[8]" Out; "HR_DISPLAY[9]"
Out;
   "HR_DISPLAY[10]" Out; "HR_DISPLAY[11]" Out;
"HR_DISPLAY[12]" Out;
   "HR_DISPLAY[13]" Out; "MIN_DISPLAY[0]" Out;
"MIN_DISPLAY[1]" Out;
   "MIN_DISPLAY[2]" Out; "MIN_DISPLAY[3]" Out;
"MIN_DISPLAY[4]" Out;
   "MIN_DISPLAY[5]" Out; "MIN_DISPLAY[6]" Out;
"MIN_DISPLAY[7]" Out;
   "MIN_DISPLAY[8]" Out; "MIN_DISPLAY[9]" Out;
"MIN_DISPLAY[10]" Out;
   "MIN_DISPLAY[11]" Out; "MIN_DISPLAY[12]" Out;
"MIN_DISPLAY[13]" Out;
   "SPEAKER_OUT" Out;
}
SignalGroups {
   "all_inputs"   '"ALARM" + "BSD_TDI" + "BSD_TEST_CLK" +
"BSD_TMS" +
   "BSD_TRST" + "CLK" + "HRS" + "MINS" + "RESETN" + "SET_TIME" +
   "TEST_MODE" + "TEST_SE" + "TEST_SI" + "TOGGLE_SWITCH"'; //
#signals=14
   "all_outputs"   '"AM_PM_OUT" + "BSD_TDO" + "HR_DISPLAY[0]"
+
   "HR_DISPLAY[1]" + "HR_DISPLAY[2]" + "HR_DISPLAY[3]" +
   "HR_DISPLAY[4]" + "HR_DISPLAY[5]" + "HR_DISPLAY[6]" +
   "HR_DISPLAY[7]" + "HR_DISPLAY[8]" + "HR_DISPLAY[9]" +
   "HR_DISPLAY[10]" + "HR_DISPLAY[11]" + "HR_DISPLAY[12]" +
   "HR_DISPLAY[13]" + "MIN_DISPLAY[0]" + "MIN_DISPLAY[1]" +
   "MIN_DISPLAY[2]" + "MIN_DISPLAY[3]" + "MIN_DISPLAY[4]" +
   "MIN_DISPLAY[5]" + "MIN_DISPLAY[6]" + "MIN_DISPLAY[7]" +
   "MIN_DISPLAY[8]" + "MIN_DISPLAY[9]" + "MIN_DISPLAY[10]" +
   "MIN_DISPLAY[11]" + "MIN_DISPLAY[12]" + "MIN_DISPLAY[13]"
+
   "SPEAKER_OUT"'; // #signals=31
   "all_ports"   '"all_inputs" + "all_outputs"'; //
#signals=45
   "_pi"   '"all_inputs"'; // #signals=14
   "_po"   '"all_outputs"'; // #signals=31
}
```

```
ScanStructures {
   ScanChain "c0" {
      ScanLength 40;
      ScanIn "TEST_SI";
      ScanOut "SPEAKER_OUT";
   }
}
Timing {
   WaveformTable "_default_WFT_" {
      Period '100ns';
      Waveforms {
         "all_inputs" { 0 { '5ns' D; } }
         "all_inputs" { 1 { '5ns' U; } }
         "all_inputs" { Z { '5ns' Z; } }
         "all_outputs" { X { '0ns' X; } }
         "all_outputs" { H { '0ns' X; '95ns' H; } }
         "all_outputs" { T { '0ns' X; '95ns' T; } }
         "all_outputs" { L { '0ns' X; '95ns' L; } }
         "CLK" { P { '0ns' D; '45ns' U; '55ns' D; } }
         "BSD_TEST_CLK" { P { '0ns' D; '45ns' U; '55ns' D; } }
         "RESETN" { P { '0ns' U; '45ns' D; '55ns' U; } }
      }
   }
}
PatternBurst "__burst__" {
   "__pattern__" {
   }
}
PatternExec {
   Timing "";
   PatternBurst "__burst__";
}
Procedures {
   "load_unload" {
      W "_default_WFT_";
      V {  "BSD_TEST_CLK"=0; "BSD_TRST"=0; "CLK"=0; "RESETN"=1;
       "TEST_MODE"=1; "TEST_SE"=1; "_so"=#; }
      Shift {
         W "_default_WFT_";
         V {  "BSD_TEST_CLK"=P; "BSD_TRST"=0; "CLK"=P;
"RESETN"=1;
       "TEST_MODE"=1; "TEST_SE"=1; "_so"=#; "_si"=#; }
   }
}
   "capture" {
      W "_default_WFT_";
      F {  "BSD_TRST"=0; "TEST_MODE"=1; }
      V { "_pi"=\r14 #; "_po"=\r31 #; }
   }
   "capture_CLK" {
      W "_default_WFT_";
```

```
      F {  "BSD_TRST"=0; "TEST_MODE"=1; }
      "forcePI": V { "_pi"=\r14 #; }
      "measurePO": V { "_po"=\r31 #; }
      "pulse": V { "CLK"=P; }
   }
   "capture_BSD_TEST_CLK" {
      W "_default_WFT_";
      F {  "BSD_TRST"=0; "TEST_MODE"=1; }
      "forcePI": V { "_pi"=\r14 #; }
      "measurePO": V { "_po"=\r31 #; }
      "pulse": V { "BSD_TEST_CLK"=P; }
   }
   "capture_RESETN" {
      W "_default_WFT_";
      F {  "BSD_TRST"=0; "TEST_MODE"=1; }
      "forcePI": V { "_pi"=\r14 #; }
      "measurePO": V { "_po"=\r31 #; }
      "pulse": V { "RESETN"=P; }
   }
}
MacroDefs {
   "test_setup" {
      W "_default_WFT_";
      V {  "BSD_TEST_CLK"=0; "CLK"=0; }
      V {  "BSD_TEST_CLK"=0; "BSD_TRST"=0; "CLK"=0; "RESETN"=1;
       "TEST_MODE"=1; }
   }
}
```

Note:

The `read_test_protocol -section test_setup` command imports only the `test_setup` section of the protocol file and ignores the remaining sections.

Setting the Scan Style

The scan style setting affects messages generated by test design rule checking. This is because some design rules apply only to specific scan styles. To set the scan style, use the following syntax for the `set_scan_configuration` command:

```
set_scan_configuration -style scan_style
```

You can use any of the following arguments for the *`scan_style`* value:

- `multiplexed_flip_flop`
- `clocked_scan`
- `lssd`

- `scan_enabled_lssd`
- `combinational`

Alternatively, you can set the `test_default_scan_style` variable instead of using the `set_scan_configuration` command.

If you do not set the scan style before performing test design rule checking, `multiplexed_flip_flop` is used as the default scan stye.

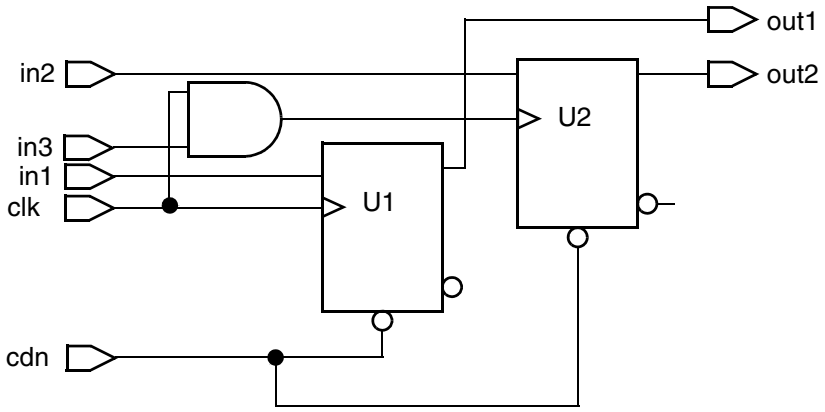
Design Examples

This section contains two simple design examples that illustrate how to generate test protocols. The first example shows how to use the `set_dft_signal` command to control the clock signal, the scan-enable signal, and the asynchronous reset. The second example describes a two-pass process for defining an initialization sequence in a test protocol.

Test Protocol Example 1

Figure 2-2 shows a schematic and the Verilog code for a simple RTL design that needs a test protocol.

Figure 2-2 RTL Design That Needs a Simple Protocol




```
module tcrm (in1, in2, in3, clk, cdn, out1, out2);
input in1, in2, in3, clk, cdn;
output out1, out2;
reg U1, U2;
wire gated_clk;

always @(posedge clk or negedge cdn) begin
```

```
    if (!cdn) U1 <= 1'b0;
    else U1 <= in1;
end

assign gated_clk = clk & in3;

always @(posedge gated_clk or negedge cdn) begin
  if (!cdn) U2 <= 1'b0;
   else U2 <= in2;
  end

assign out1 = U1;
assign out2 = U2;

endmodule
```

In this design, you must define the clock signal, `clk`. You must also specify that `in3` be held at 1 during scan input to enable the clock signal for U2. Finally, you must hold the `cdn` signal at 1 during scan input so that the reset signal is not applied to the registers.

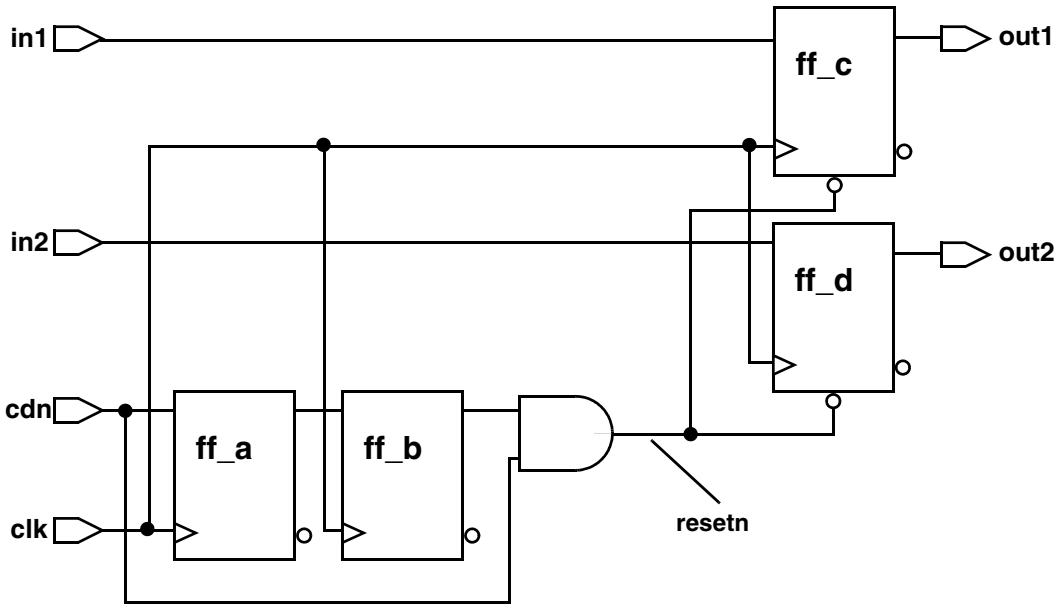
The following command sequence specifies a test protocol for the design example:

```
dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock -timing [list 45 55] \
               -port clk

dc_shell> set_dft_signal -view existing_dft -port cdn \
               -type Reset -active_state 0

dc_shell> set_dft_signal -view spec -port in3 \
               -type ScanEnable -active_state 1

dc_shell> create_test_protocol

dc_shell> write_test_protocol -output design.spf
```

Test Protocol Example 2

Figure 2-3 shows a schematic and the corresponding Verilog code for an RTL design that requires initialization.

Figure 2-3 Design That Requires an Initialization Sequence




```
module ssug (in1, in2, clk, cdn, out1, out2);
input in1, in2, clk, cdn;
output out1, out2;
reg ff_a, ff_b, ff_c, ff_d;
wire resetn;

  always @(posedge clk) begin
    ff_b <= ff_a;
    ff_a <= cdn;
  end
  assign resetn = cdn & ff_b;
  always @(posedge clk or negedge resetn) begin
    if (!resetn) begin
      ff_c <= 1'b0;
      ff_d <= 1'b0;
    end
    else begin
      ff_c <= in1;
      ff_d <= in2;
    end
  end
  assign out1 = ff_c;
  assign out2 = ff_d;
endmodule
```

In this design, you must define the clock signal, `clk`. You must also make sure that `cdn` and the Q output of `ff_b_reg` remain at 1 during the test cycle, so that the `resetn` signal remains at 1.

If you do not initialize the design, test DRC assumes that the `resetn` signal is not controllable and marks the `ff_c` and `ff_d` flip-flops as having design rule violations.

To initialize the design, you must hold `cdn` at 1 and pulse the `clk` signal twice so that the `resetn` signal is at 1.

For this example, the protocol is generated in a two-pass process. In the first pass, the generated protocol contains an initialization sequence based on the test attributes placed on `clk` and `cdn` ports. The command sequence that defines the preliminary protocol is as follows:

```
dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock -timing [list 45 55] \
               -port clk

dc_shell> set_dft_signal -view existing_dft \
                -type Constant -active_state 1 -port cdn

dc_shell> create_test_protocol

dc_shell> write_test_protocol -output first.spf
```

The resulting protocol contains the initialization steps shown in Example 2-2.

Example 2-2 Preliminary Initialization Sequence

```
MacroDefs {
   "test_setup" {
      W "_default_WFT_";
      V {  "clk"=0; }
      V {  "cdn"=1; "clk"=0; }
   }
}
```

If you run test design rule checking without modifying these initialization steps, it reports the following violation:

```
Warning: Reset input of DFF ff_d_reg was not controlled. (D3-1)
```

For the second pass of the protocol generation process, modify the initialization sequence as shown:

1. Add the three lines shown in bold to the `test_setup` section of the `MacroDefs` block:

```
MacroDefs {
   "test_setup" {
      W "_default_WFT_";
      V {  "clk"=0; }
```

```
        V {  "cdn"=1; "clk"=0; }
        V {  "cdn"=1; "clk"=P; }
        V {  "cdn"=1; "clk"=P; }
        V {  "cdn"=1; "clk"=0; }   }
}
```

The added steps pulse the clock signal twice while holding the `cdn` port to 1. The final step holds `clk` to 0 because the test design rule checker expects all clocks to be in an inactive state at the end of the initialization sequence.

2. Save the protocol into a new file. In this case, the file is called `second.spf`.
3. Read in the new macro in one of two ways:
   a. Reread the whole modified protocol file:

      ```
      read_test_protocol second.spf
      ```

   b. Read just the initialization portion of the protocol, and use the `create_test_protocol` command to fill in the remaining sections of the protocol:

      ```
      remove_test_protocol
      read_test_protocol -section test_setup second.spf
      create_test_protocol
      ```

4. After you have read in the initialization protocol, perform test DRC again. The following violation is reported:

   ```
   Warning: Cell ff_b_reg has constant 1 value. (TEST-505)
   ```

   This is to be expected because the outputs of `ff_a` and `ff_b` did not reach 0. Constant flip-flops are not included in the scan chain.

Running RTL Test DRC

After generating the test protocol, you are ready to run the test DRC process. To do this, specify the `dft_drc` command at the shell prompt, as shown in the following example:

```
dc_shell> dft_drc
```

This command generates a set of report files containing all known design violations. You'll need to review these reports and manually fix any violations before advancing to design compilation and scan insertion.

Understanding the Violations

The Test DRC process checks your design to determine if you have any test design rule violations. Before you can fix your design, you must understand what types of violations are checked and why these checks are necessary.

This section explains the test design rule checks that are performed on your design, describes messages you see when you encounter test design rule violations, and describes the methods you can use to fix the violations.

The section has the following subsections:

Violations That Prevent Scan Insertion

Scan design rules require that in test mode the registers have the functionality to operate as cells within a large shift register. This enables data to get into and out of the chip. The following violations prevent a register from being scannable:

- The flip-flop clock signal is uncontrollable.
- The latch is enabled at the beginning of the clock cycle.
- The asynchronous controls of registers are uncontrollable or are held active.

Uncontrollable Clocks

This violation can be caused by undefined or unconditioned clocks. DFT Compiler considers a clock to be controlled only if both of these conditions are true:

- The clock is forced to a known state at time = 0 in the clock period, which is the same as the “clock off state” in the TetraMAX tool.
- The clock changes state as a result of the test clock toggling.

  Going to an unknown state (X) is considered to be a change of state. However, if the clock stays in a single known state no matter what state the test clock is in, the clock will generate a violation for not being reached by any test clock.

You must use the `set_dft_signal` command to define test clocks in your design. For more information, see “Defining a Test Protocol” on page 2-3.

Also use the set_dft_signal command to condition gated clocks to reach their destinations, as shown in the following example:

```
dc_shell> set_dft_signal -view existing_dft \
                -type constant -active_state 1
```

The violation message provides the name of the signal that drives the clock inputs and the registers that ATPG cannot control.

If a design has an uncontrollable register clock pin, it generates one of the following warning messages:

```
Warning: Clock input I of DFF S was not controlled. (D1-N)

Warning: Clock input I of DLAT S was not controlled. (D4-N)
```

Asynchronous Control Pins in Active State

Asynchronous pins of a register must be capable of being disabled by an input of the design. If they cannot be disabled, this is reported as a violation. This violation can be caused by asynchronous control signals, such as the preset or clear pin of the flip-flop or latch, that are not properly conditioned before you run DFT Compiler. You might be able to fix this by setting a signal as active_state that has a hold value of 0 during scan shift or by defining a signal as active_state that has a hold value of 1. If you create all signal definitions correctly before running DFT Compiler, this violation indicates registers that ATPG cannot control.

If a register has an asynchronous pin that is not controlled by an asynchronous control signal, you get one of the following warning messages:

```
Warning: Set input I of DFF S was not controlled. (D2-N)

Warning: Reset input I of DFF S was not controlled. (D3-N)

Warning: Set input I of DLAT S was not controlled. (D5-N)

Warning: Reset input I of DLAT S was not controlled. (D6-N)
```

Violations That Prevent Data Capture

After DFT Compiler checks for violations that prevent scan insertion, the next step is to verify that your design can get valid data during the capture phase of ATPG.

Note that ATPG does not consider timing when generating vectors for a scan design. If you do not fix the violations in this section, ATPG might generate vectors that fail functional simulation or fail on the tester, although in the TetraMAX tool you would also have to override the TetraMAX default settings.

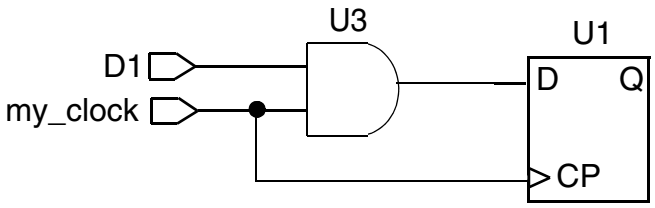
The violations are described in the following sections:

- Clock Used As Data
- Black Box Feeds Into Clock or Asynchronous Control
- Source Register Launch Before Destination Register Capture
- Registered Clock-Gating Circuitry
- Three-State Contention
- Clock Feeding Multiple Register Inputs

Clock Used As Data

When a clock signal drives the data pin of a cell, as in Figure 2-4, ATPG tools cannot determine the captured value. Modify the logic leading to the datapath to eliminate dependency on the clock.

Figure 2-4 Clock Signal Used As Data Input



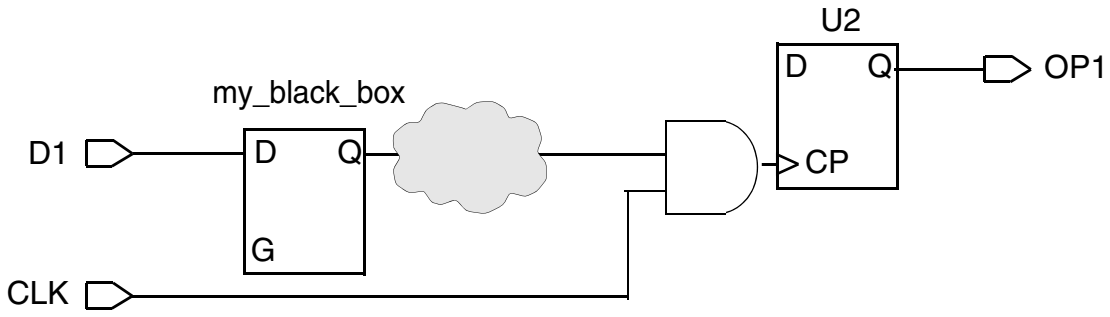
If the clock and data input to a register are interdependent, you might get the following warning message:

```
Warning: Clock C connects to clock and data inputs I1/I2 of DFF S.
  (D11-N)
```

Black Box Feeds Into Clock or Asynchronous Control

If the output of a black box indirectly feeds into the clock of a register, the register might not be able to capture data. An example is shown in Figure 2-5.

Figure 2-5 Black Box Feeds Clock Input



For more information about black boxes, see "Black Boxes" on page 2-21.

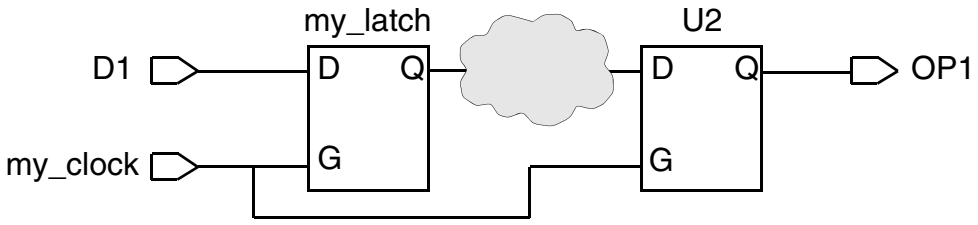
Source Register Launch Before Destination Register Capture

This section describes the violations caused by source registers that launch new data to the destination registers before they can capture and shift out the original data.

When two latches are enabled by the same clock but have a combinational datapath between them, data can propagate through both latches in a single clock cycle. This reduces the ability of ATPG to observe logic along this path. Modify the logic leading to the affected latches to eliminate any paths affected by latches that are enabled by the same clock.

An example of this violation is shown in Figure 2-6. When the clock turns off, that is, pulses from an inactive state to an active state and then back, the second latch (U2) can capture the value originally on port D1 or on its data pin, depending on the relationship between the clock width and the delay on the datapath. The possibility of data feedthrough causes the destination latch (U2) to capture data unreliably.

Figure 2-6 Latch-Based Circuit With Source Register Launch Before Destination Register Capture



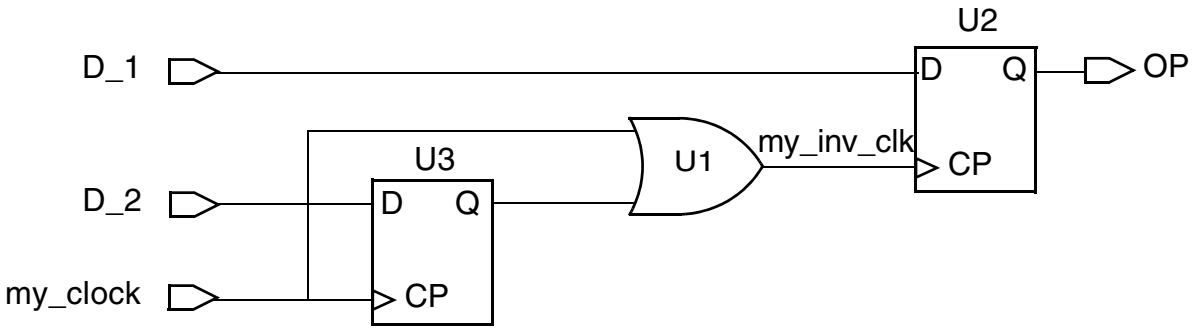
If multiple latches are enabled so that the latches feed through capture data, you get the following warning message:

```
Warning: Clock C cannot capture data with other clocks off. (D8-N)
```

Registered Clock-Gating Circuitry

If you gate the register output with the same clock signal that is used to clock the register, you cannot use the same phase of the resulting signal as a clock. An example is shown in Figure 2-7.

Figure 2-7 Invalid Clock-Gating Circuit



The U1 output invalidly clocks register U2. The OR gate, U1, has two inputs, where one is the output of register U3 and the other is the signal used to clock U3.

Note that Power Compiler clock gating does not lead to this violation because Power Compiler uses opposite edge-triggered flip-flops or latches to create the clock-gating signals.

This circuit configuration results in timing hazards, including clock glitches and clock skew. Modify the clock-gating logic to eliminate this type of logic.

If you implement this type of clock-gating circuitry, you get the following warning message:

```
Warning: Clock input I of DFF S was not controlled. (D1-N)
```

Three-State Contention

DFT Compiler can check to see if your RTL code contains three-state contention conditions. If floating or contention is found, one of the following three warning messages is issued:

```
Warning: Bus gate N failed contention ability check for drivers G1 and
G2. (D20-N)

Warning: Bus gate N failed Z state ability check. (D21-N)
```

```
Warning: Wire gate N failed contention ability check for drivers G1 and
G2. (D22-N)
```

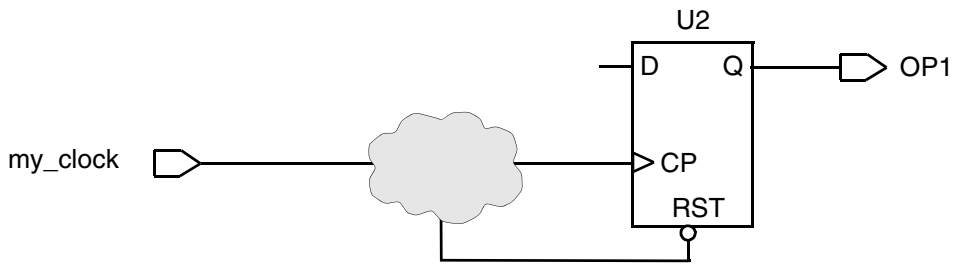
Clock Feeding Multiple Register Inputs

A clock that feeds multiple register inputs reduces the fault coverage attainable by ATPG. The signal can be one of the following:

- A clock signal that feeds into more than one register clock pin
- A clock signal that feeds into a clock pin and an asynchronous control of a register

The logic that feeds the same clock into multiple clock pins or asynchronous pins should be modified so that the clock reaches only one port on the register. Figure 2-8 shows an example of this violation.

Figure 2-8 Clock Signal Feeds Register Clock Pin and Asynchronous Reset



If you implement this type of design circuitry, you get the following warning message:

```
Warning: D12 Clock C connects to clock/set/reset inputs (G1 / G2) of
DFF I. (D12-N)
```

Violations That Reduce Fault Coverage

Violations that can reduce your fault coverage are discussed in the following sections:

- Combinational Feedback Loops
- Clocks That Interact With Register Input
- Multiple Clocks That Feed Into Latches and Flip-Flops
- Black Boxes

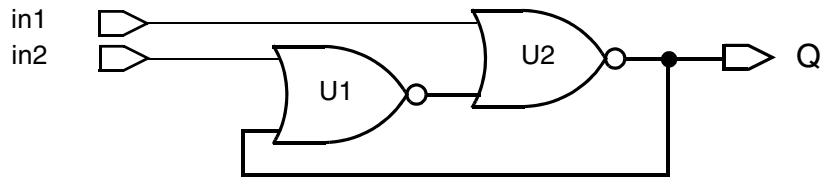
Combinational Feedback Loops

An active (or *sensitizable*) feedback loop reduces the fault coverage that ATPG can achieve by increasing the difficulty of controlling values on paths containing parts of the loop.

A loop that oscillates causes severe problems for ATPG and for fault simulation. You can break these loops by placing test constraints on the design. This creates a feedback loop that is not active. DFT Compiler does not report violations on loops that you have broken by setting constraints.

If you are using the loop as a latch, convert the combinational elements that make up this feedback loop into a latch from your ASIC vendor library. Figure 2-9 shows this type of loop.

Figure 2-9 Highlighted Combinational Feedback Loop



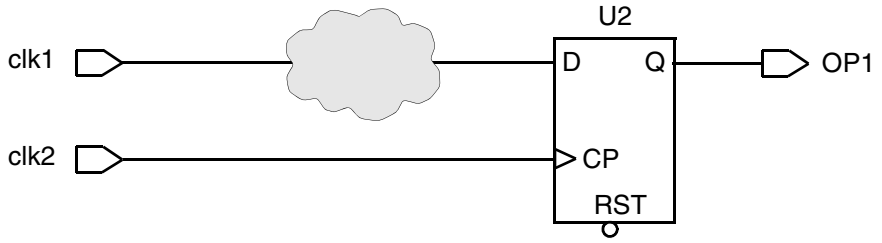
If your design contains a sensitizable feedback loop, you get the following warning message:

```
Warning: Feedback path network X is sensitizable through source gate G.
(D23-N)
```

Clocks That Interact With Register Input

A clock that affects the data input of a register reduces the fault coverage attainable by ATPG, because ATPG pulses only one clock at a time, keeping all other clocks in their off states. Attempting to fix this purely in the ATPG setup can result in timing hazards. Do not use the circuit shown in Figure 2-10, because testing this logic requires multiple ATPG iterations and might also require special scan chain design considerations (not discussed here). Redesign the logic feeding the data inputs of the registers to eliminate dependency on other clocks.

Figure 2-10 Clock Interacting With Register Input



If a clock affects the data of a register, you might get the following warning message:

`Warning: Clock C connects to data input of DFF S. (D10-N)`

Multiple Clocks That Feed Into Latches and Flip-Flops

This section describes the types of clock-gating configurations that can reduce fault coverage. For more information on other clock-gating configurations that prevent scan insertion and data capture, see "Violations That Prevent Scan Insertion" on page 2-13 and "Violations That Prevent Data Capture" on page 2-14.

Latch Requires Multiple Clocks to Capture Data

For a latch to be usable as part of a scan chain, it must be enabled by one clock or by a clock ANDed with data derived from sources other than that clock. Multiple clocks and gated clocks must be ORed together so that any one of the clocks can capture data. ATPG forces all but one clock off at any time. Latches that can capture data as a result of more than one clock must be able to capture data with one clock active and all others off.

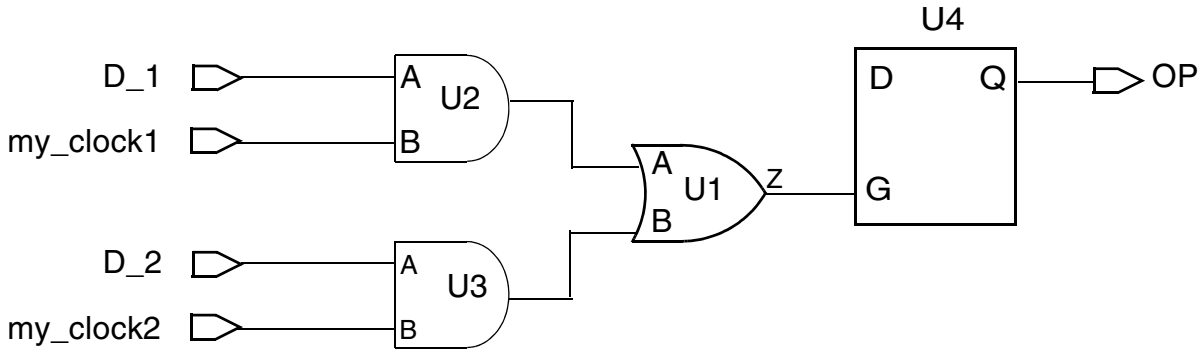
If your design has an OR gate with clock and data inputs, the output clock of the OR gate has extra pulses that depend on the data input. If your design has an AND gate with more than one clock input, the output of the AND gate never generates a clock pulse. Both of these cases are violations, and DFT Compiler generates a warning message.

You can create valid clock-gating logic for latches if your circuitry contains an

- AND gate with only one clock input and one or more data inputs
- OR gate with clock or gated clock inputs

A combination of these valid clocking rules is shown in Figure 2-11.

Figure 2-11 Valid Latch Clock Gating



If you generate logic that violates these clock rules, you get the following warning message:

```
Warning: Clock C cannot capture data with other clocks off. (D8-N)
```

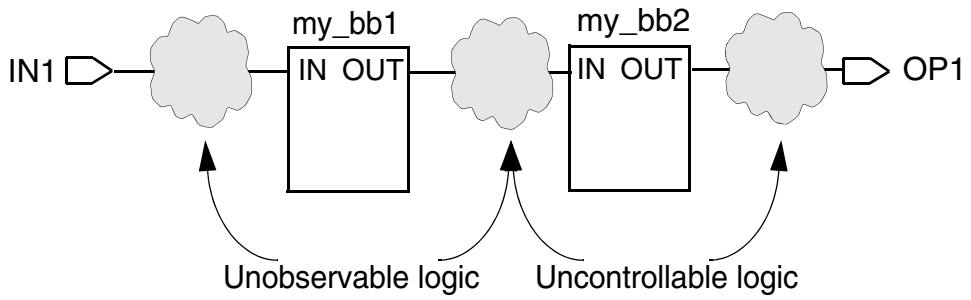
Latches Are Not Transparent

Latches should be transparent in certain types of scan styles. If a latch is not transparent, ATPG might have more difficulty controlling it. This could cause a loss of fault coverage on the path through the latch.

Black Boxes

Logic that drives or is driven by black boxes cannot be tested because it is unobservable or uncontrollable. This violation can drastically reduce fault coverage, because the logic that surrounds the black box is unobservable or uncontrollable. Figure 2-12 shows an example.

Figure 2-12 Effect of Black Boxes on Surrounding Logic



If there are any black boxes in your design, the `dft_drc` command issues the following warning message:

```
Warning: Cell U0 (black_box) is unknown (black box) because
functionality for output pin Z is bad or incomplete. (TEST-451)
```

Limitations

Note the following limitations:

- The `set_svf` command is not supported in the RTL test DRC flow. You should comment out any `dft_drc` command that performs test DRC checking on elaborated RTL before you perform design synthesis, which generates the verification setup file.
- The `compile_ultra -gate_clock -scan` command is not supported in the RTL test DRC flow. When the `create_test_protocol` command is run on the elaborated RTL, subsequent `compile_ultra -gate_clock -scan` commands might not properly incorporate clock-gating cells into the scan chains. You should comment out any `create_test_protocol` commands performed on elaborated RTL before you perform design synthesis with this command.

3

Running the Test DRC Debugger

This chapter describes debugging design rule checking (DRC) violations by using the Design Vision graphical user interface (GUI).

Design Vision provides analysis tools for viewing and analyzing your design. It allows you to view the design violations, and it can provide an early warning of test-related issues. The GUI provides the debug environment for pre-DFT DRC violations, post-DFT DRC violations, and Core Test Language (CTL) models.

This chapter includes the following sections:

Starting and Exiting the Graphical User Interface

To invoke Design Vision and view test DRC results, you need to

- Execute the `design_vision` command or, for topographical mode, the `design_vision -topographical_mode` command from the command line.
- Choose File > Execute Script to run `dc_shell` script.
- Choose Test > Run DFT DRC to check the design for DRC violations. This brings up the violation browser.

Alternatively,

- Enter the `dft_drc` command on the Design Vision command line. Then choose Test > Browse Violations to invoke the violation browser.

To exit Design Vision,

- Choose File > Exit

You can also enter exit or quit on the command line or press Control-c three times in the UNIX or Linux shell.

To invoke Design Vision directly from dc_shell, enter

```
dc_shell> gui_start
```

To use options with this command, see the *Design Vision User Guide* for further information.

Note:

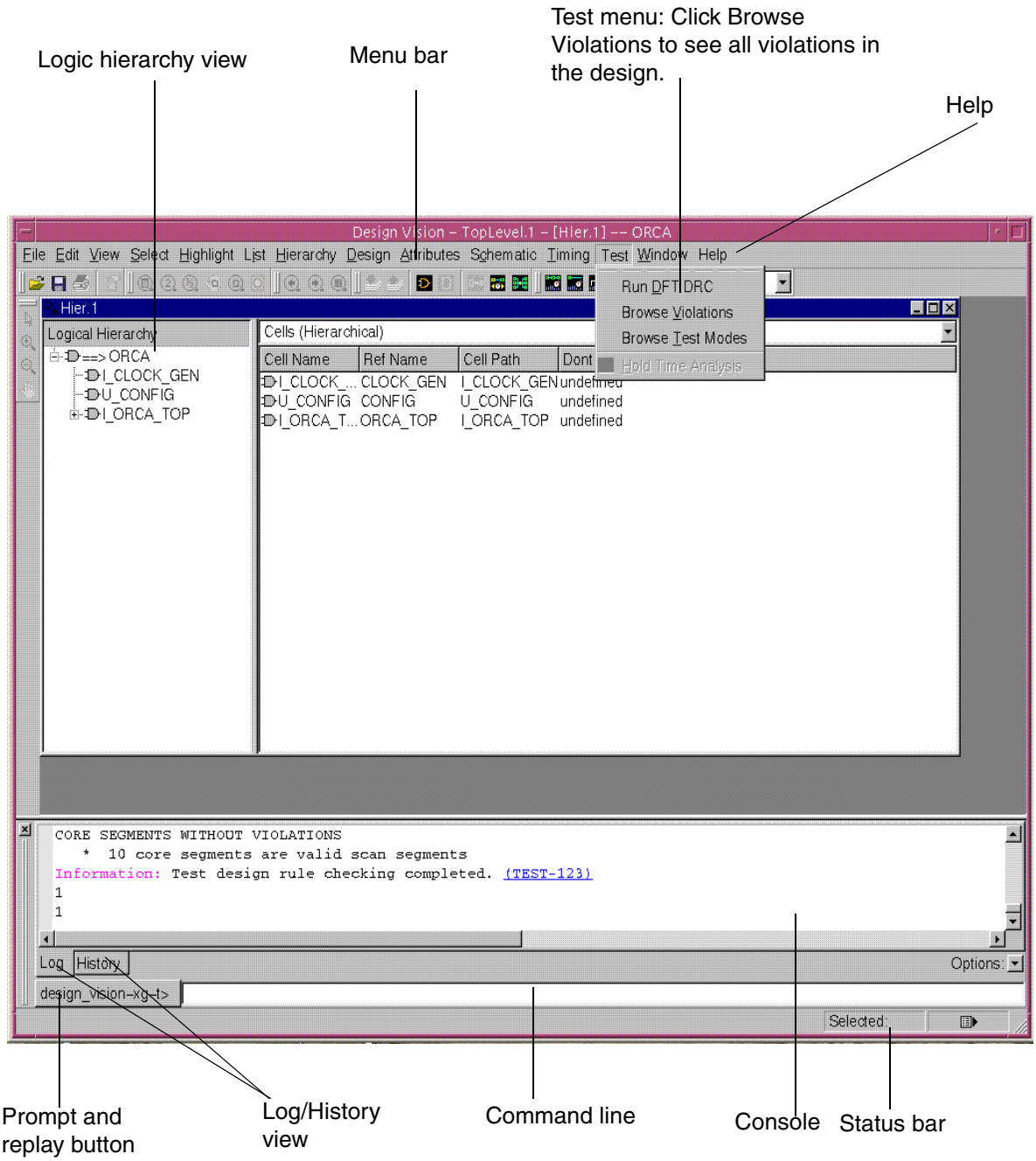
Before invoking Design Vision or opening the GUI, make sure you have correctly set your display environment variable. See the *Design Compiler User Guide* for information on setting this variable.

Exploring the Graphical User Interface

The Design Vision window is a top-level window in which you can display design information in various types of analysis views. The GUI functions as a visual analysis tool to help you to visualize and analyze the violations in your design.

Figure 3-1 shows the Design Vision window running in the Design Vision foreground.

Figure 3-1 The Design Vision Window



The window consists of a title bar, a menu bar, and several toolbars at the top of the window and a status bar at the bottom of the window.

You use the workspace between the toolbars and the status bar to display view windows containing graphical and textual design information. You can open multiple windows and use them to compare views, or different design information within a view, side by side.

Logic Hierarchy View

The logic hierarchy view helps you navigate through your design and gather information. The view is divided into the following two panes:

- Instance tree
- Objects list

The instance tree lets you quickly navigate the design hierarchy and see the relationships among its levels. If you select the instance name of a hierarchical cell (one that contains subblocks), information about that instance appears in the object table. You can Shift-click or Control-click instance names to select combinations of instances.

By default, the object table displays information about hierarchical cells belonging to the selected instance in the instance tree. To display information about other types of objects, select the object types in the list above the table. You can display information about hierarchical cells, all cells, pins and ports, pins of child cells, and nets.

Console

The console provides a command-line interface and displays information about the commands you use in the session in the following two views:

- Log view
- History view

The log view is displayed by default when you start Design Vision. The log view provides the session transcript. The history view provides a list of the commands that you have used during the session. To select a view, click the tab at the bottom of the console.

Command Line

You can enter dc_shell commands on the command line at the bottom of the console. Enter these commands just as you would enter them at the dc_shell prompt in a standard UNIX or Linux shell. When you issue a command by pressing Return or clicking the prompt button to the left of the command line, the command output, including processing messages and any warnings or error messages, is displayed in the console log view.

You can display, edit, and reissue commands on the console command line by using the arrow keys to scroll up or down the command stack and to move the insertion point to the left or right on the command line. You can copy text in the log view and paste it on the command line.

You can also select commands in the history view and edit or reissue them on the command line.

Viewing Man Pages

The GUI provides an HTML-based browser that lets you view, search, and print man pages for commands, variables, and error messages.

To view a man page in the man page viewer,

1. Choose Help > Man Pages.
2. Click the category link for the type of man page you want to view: Commands, Variables, or Messages.
3. Click the title link for the man page you want to view.

Menus

The menu bar provides menus with the commands you need to use the GUI. Choose commands on the Test menu to view design violations and to open the violation browser.

Checking Scan Test Design Rules

Check the current design for DRC violations in your scan test implementation before you perform other DFT Compiler operations such as inserting scan cells. You can use the violation browser and the violation inspector to examine and debug any DRC violations that you find.

To view DRC violations,

- Choose Test > Run DFT DRC.

DFT Compiler checks the design for DRC violations and displays messages in the console log view. If violations exist, Design Vision automatically opens a new Design Vision window and displays the violation messages in the violation browser.

Examining DRC Violations

You can use the DRC violation browser to search for and view information about DFT unified DRC violations in the current design. The violation browser can display both static and dynamic violation messages. Static violations occur as a result of the design topology. To detect dynamic violations, you must simulate the design in the violation inspector.

To open the violation browser and view violations,

- Choose Test > Browse Violations

The violation browser view window appears in a new Design Vision window, docked to the left side of the window.

See "Viewing Design Violations" on page 3-6.

Viewing Test Protocols

You can view details about the default test protocol and any user-defined test protocols that you created for the design.

To view test protocols,

- Choose Test > Browse Test Modes. The Test Modes Details dialog box appears. Alternatively, you can open this dialog box by clicking the Test Modes button in the violation inspector.
- Select a test protocol name in the Test Modes list.

Viewing Design Violations

This section covers the following topics:

Examining DRC Violations

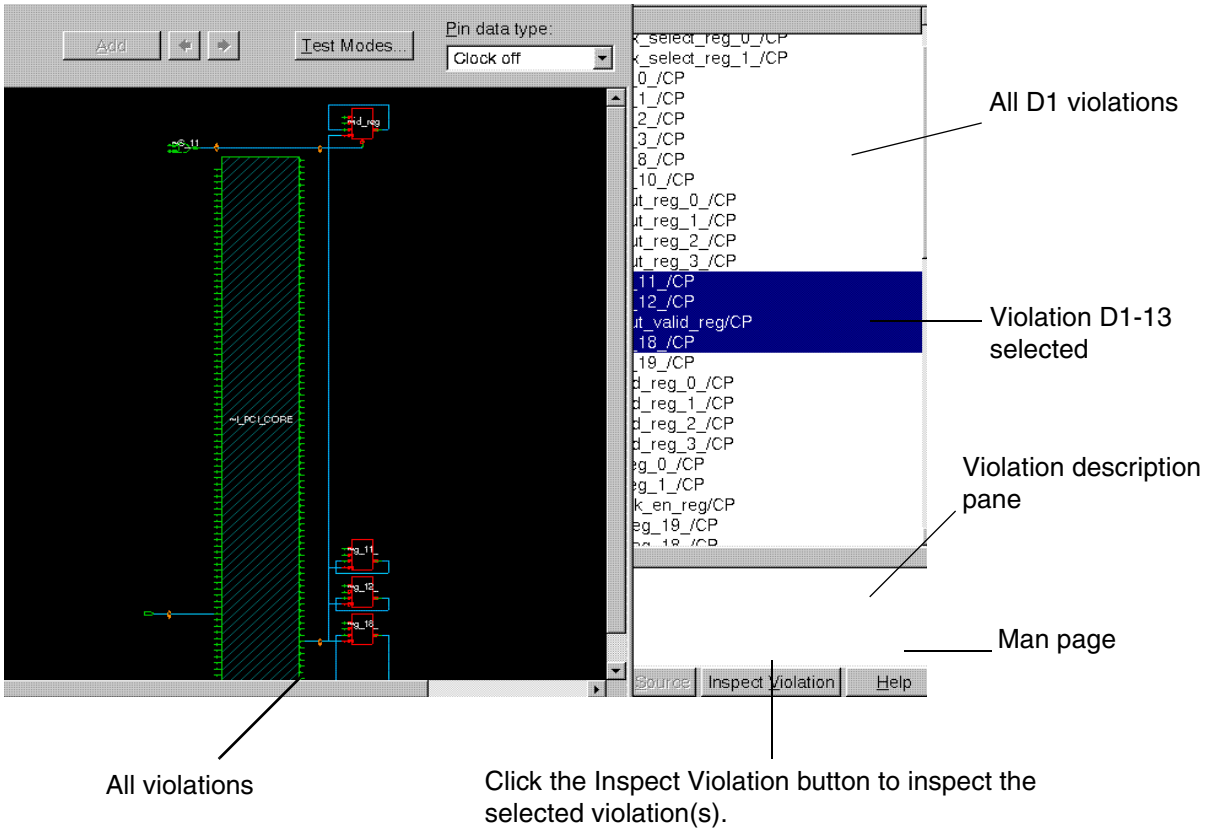
The violation browser lets you examine detailed information about violations and also provides a variety of tools for viewing the different aspects of a violation. The violation browser groups the warning and error messages into categories that help you find the problems you are concerned about.

To view a violation,

- Choose Test > Browse Violations.

This opens the violation browser view window, as shown in Figure 3-2.

Figure 3-2 Violation Browser View Window



The violation browser window consists of two panes: a violation category tree on the left and a violation pin list on the right. The Violation Categories pane lists different categories of violations, for example, Modeling and Pre-DFT.

To see the violations:

- Click the expansion button (plus sign) of the violation category to display the violations of that group.

  The expanded view displays the types and number of violations.

When you select a violation in the left pane, a list of pins where the violations occur appears in the right pane.

For example,

- Expand the Pre-DFT category view.
- Select violation D1.

  The resulting D1 violations are shown in the right-side pane.
- Click a specific violation pin or violation ID, and the corresponding description is displayed in the description pane.
- Click the Inspect Violation button to view the violation schematic. For more details, see "Inspecting Static DRC Violations" on page 3-8.

  (Optional) To view the man page of a violation, click the Help button.

Inspecting DRC Violations

You can analyze and debug DFT unified DRC violations by inspecting them in a violation inspector window. You can inspect one or more violations of the same type. The violation inspector provides both a schematic view for inspecting static violations and a coordinated waveform view for simulating dynamic violations.

This section has the following subsections:

- Inspecting Static DRC Violations
- Inspecting Dynamic DRC Violations

Inspecting Static DRC Violations

The violation inspector integrates the schematic viewer with the waveform viewer to debug violations. However, the waveform view within the violation inspector is useful only for debugging dynamic violations and only for pin data that consists of data values over a sequence of events or a period of time. For pin data types that are constant or do not fit the concept of values over time or sequence of events, pin data values are represented as strings on schematics, and the waveform view in the violation inspector is hidden in such cases.

This section has the following subsections:

Viewing a Violation

When you select a violation in the violation browser, the corresponding path schematic is displayed in the violation inspector so that you can investigate the violations. A path schematic can contain cells, pins, ports, and nets.

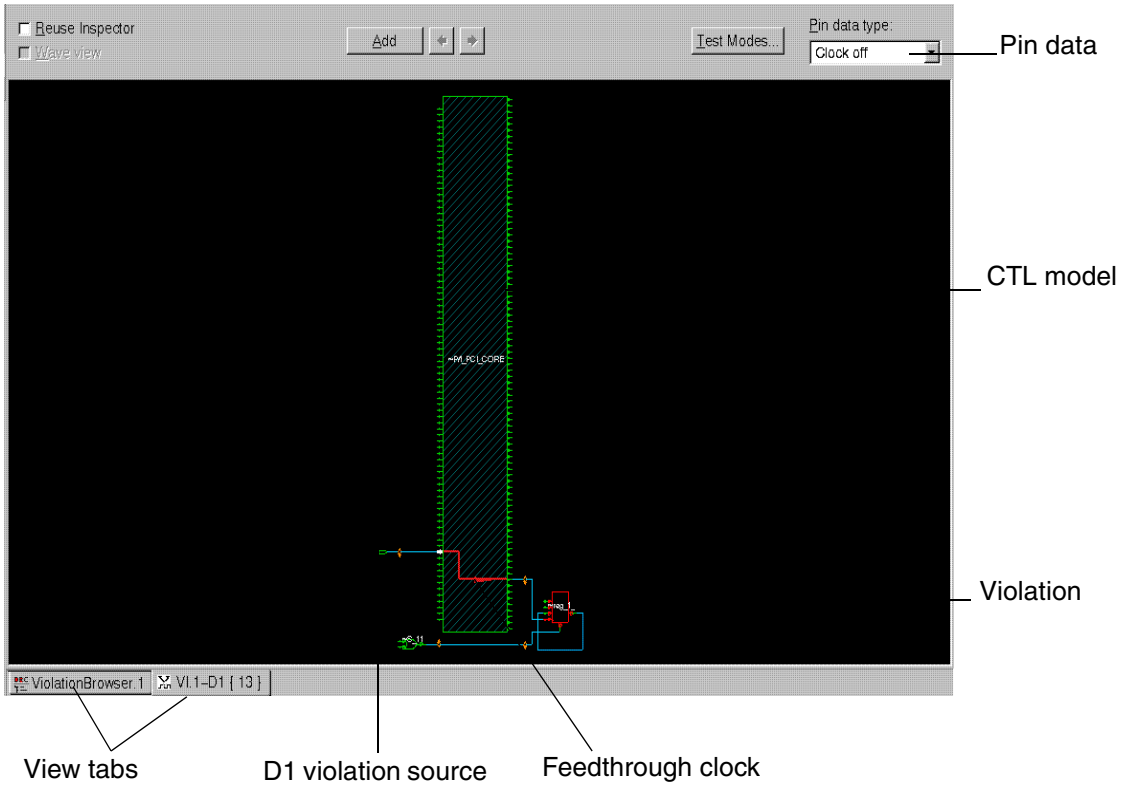
To open the violation schematic,

- Click the Inspect Violation tab at the bottom of the console.

  This opens the violation inspector window, as shown in Figure 3-3.

  This window displays the path schematic for visually examining any violations and the violation source. A path schematic provides contextual information and details about the path and its components. Red-colored cells indicate pins with violations.

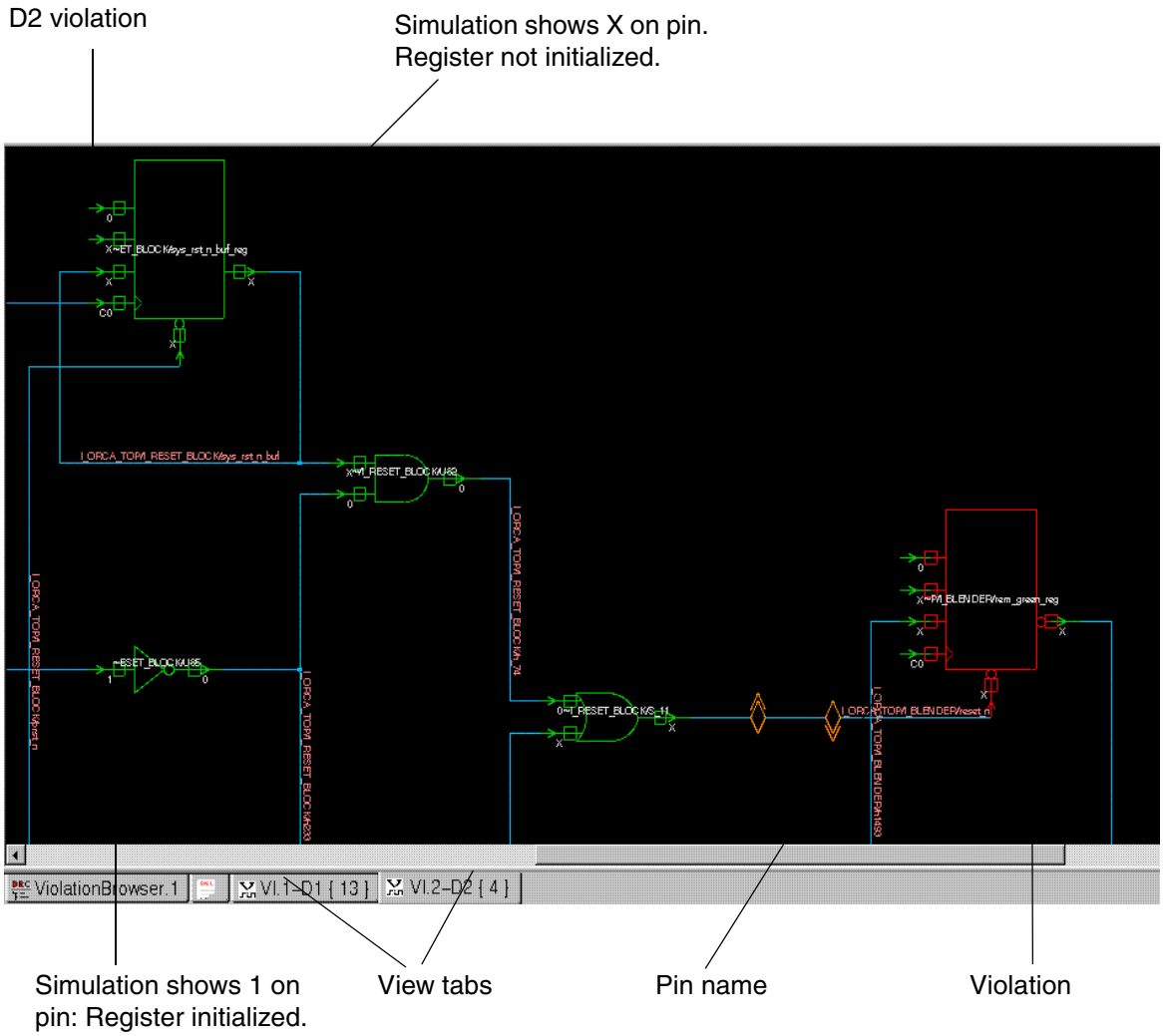
Figure 3-3 Schematic Viewer: Viewing a Violation



- (Optional) Select the data type name in the “Pin data type” menu to display a different pin data type.
- Display object information in an InfoTip by moving the pointer over a pin, cell, net, or other type of object.
- Pin information includes the cell name, pin direction, and simulation values.
- Cell information includes the cell name and the names and directions of the attached pins.
- Net information includes the net name, local fanout value, and fanout value.

  Observe the simulation values displayed at the input and output pins in Figure 3-4.

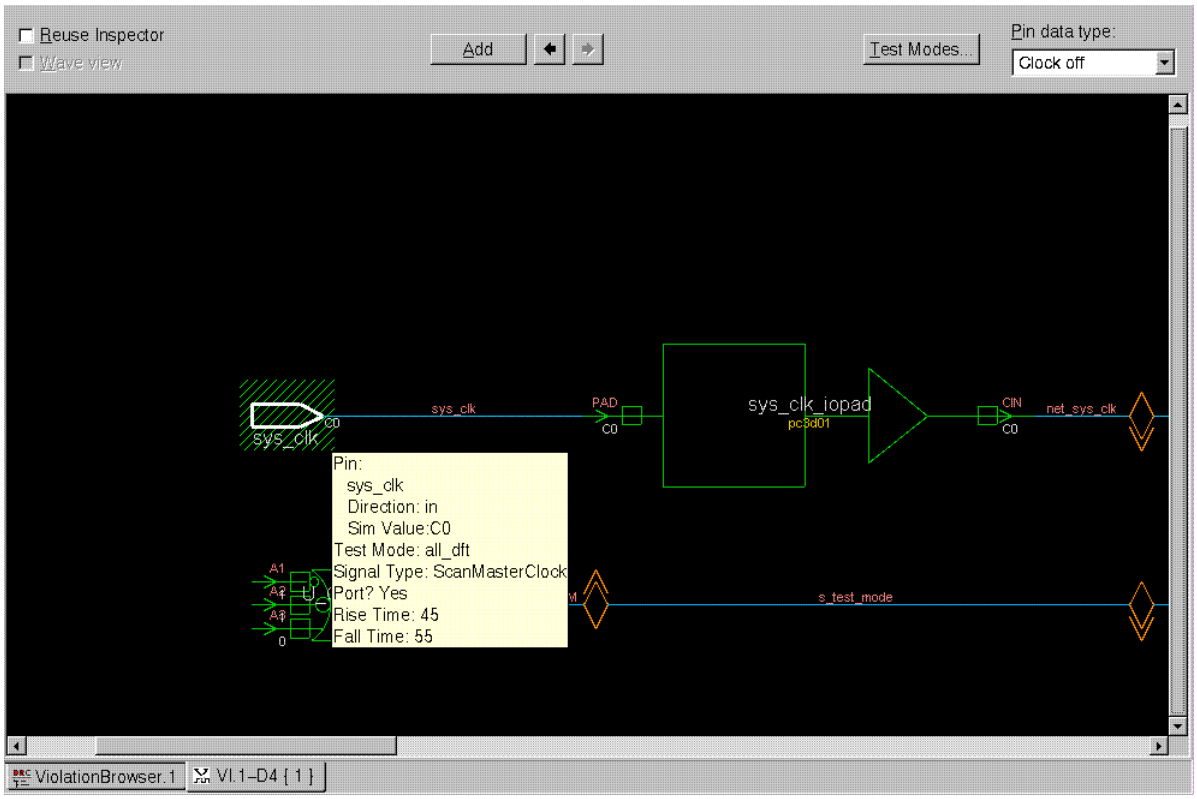
Figure 3-4 Viewing Simulation Values



Note that you can browse back and forth between the path schematics of D1 and D2 violations by clicking the corresponding D1 and D2 tabs at the bottom of the console window.

- If you define a test pin with the `set_dft_signal` command, the pin appears with a hatched fill pattern and the InfoTip displays the pin information. See how the test pins are specified in Figure 3-5.

Figure 3-5 Test Pin Defined



Viewing Multiple Violations

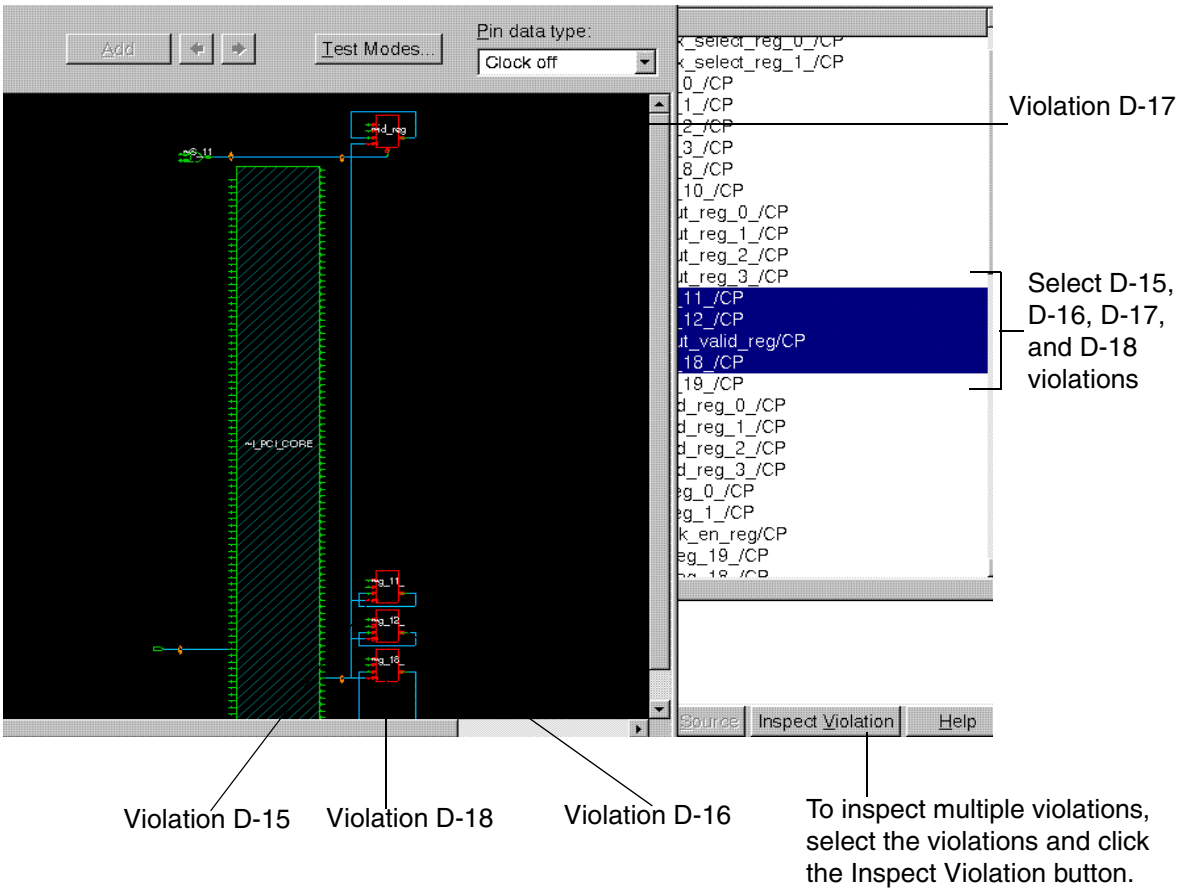
To view the path schematics for multiple violations,

1. Shift-click to select multiple violation IDs in the violation browser.
2. Click the Inspect Violation button at the bottom of the window.

   The schematics of the selected violations are displayed in the schematic viewer of the violation inspector.
3. (Optional) Click the Help button to bring up the Man Page Viewer.

Figure 3-6 shows the selection of multiple violations.

Figure 3-6 Viewing Multiple Violations



Viewing CTL Models

A CTL model provides information about scan cells and the test modes in which they are active. It also describes the number of scan cells in a chain and the pins associated with the particular scan chain.

If your design contains CTL models, the violation schematic displays them as black boxes with a hatched fill pattern to distinguish them from other cells.

A CTL model allows you to view the feedthrough signals, like clocks and asynchronous signals. You can view the fanin and fanout for the path schematics. This step can provide useful information about the logic that drives, or is driven by, the problem path.

The following pins are displayed in a CTL model:

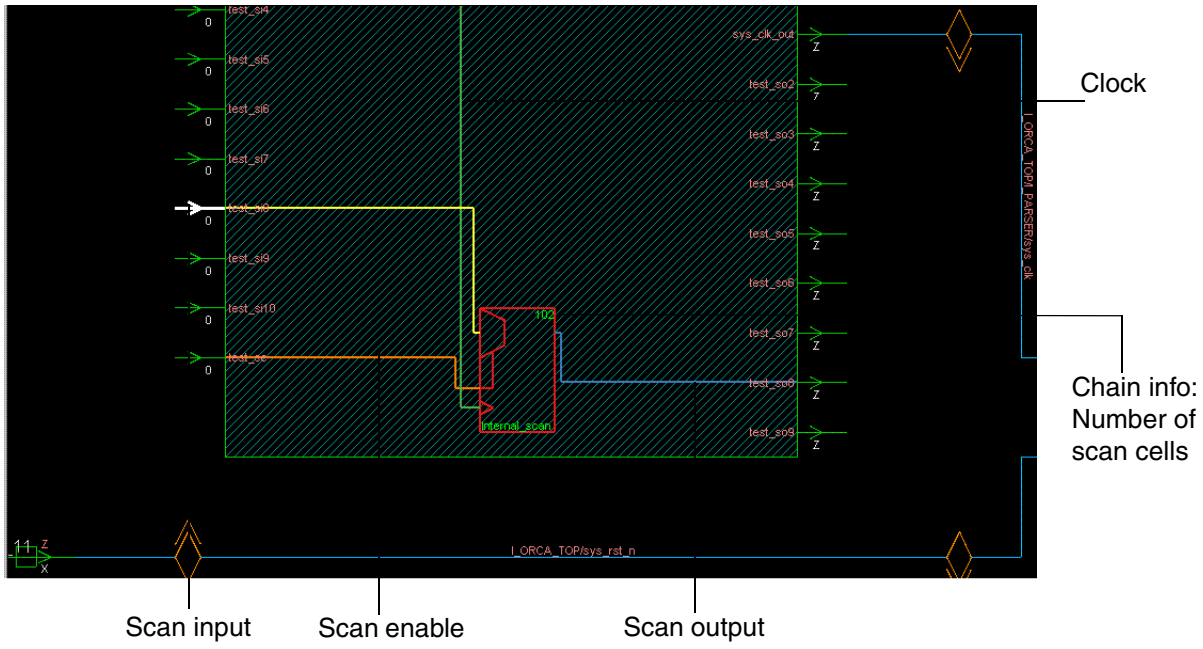
- Scan input
- Scan output
- Scan enable
- Scan clock

To display the feedthrough flylines for a CTL model,

1. Select an input or output pin on the model.
2. Right-click and choose “Show feed throughs”.

See the design schematics of the CTL model in Figure 3-7.

Figure 3-7 CTL Scan Chain Information



Inspecting Dynamic DRC Violations

The waveform viewer displays a coordinated waveform view for simulating dynamic violations. The violation inspector provides both a schematic view (violation schematic) for inspecting static violations and a coordinated waveform view for simulating dynamic violations.

The pin data for dynamic violations represents simulation values for a series of initialization cycles. To debug dynamic violations, you can select pins in the violation schematic and view their simulation values in the waveform view. Simulation values can be constant, or they can vary over time in a series of simulation “events.”

The violation inspector displays the pin data that corresponds to the most suitable pin data type for debugging the violation. To simulate the pin data for a dynamic violation, you must change to a pin data type that supports simulation values.

To display waveforms for pins with dynamic violations,

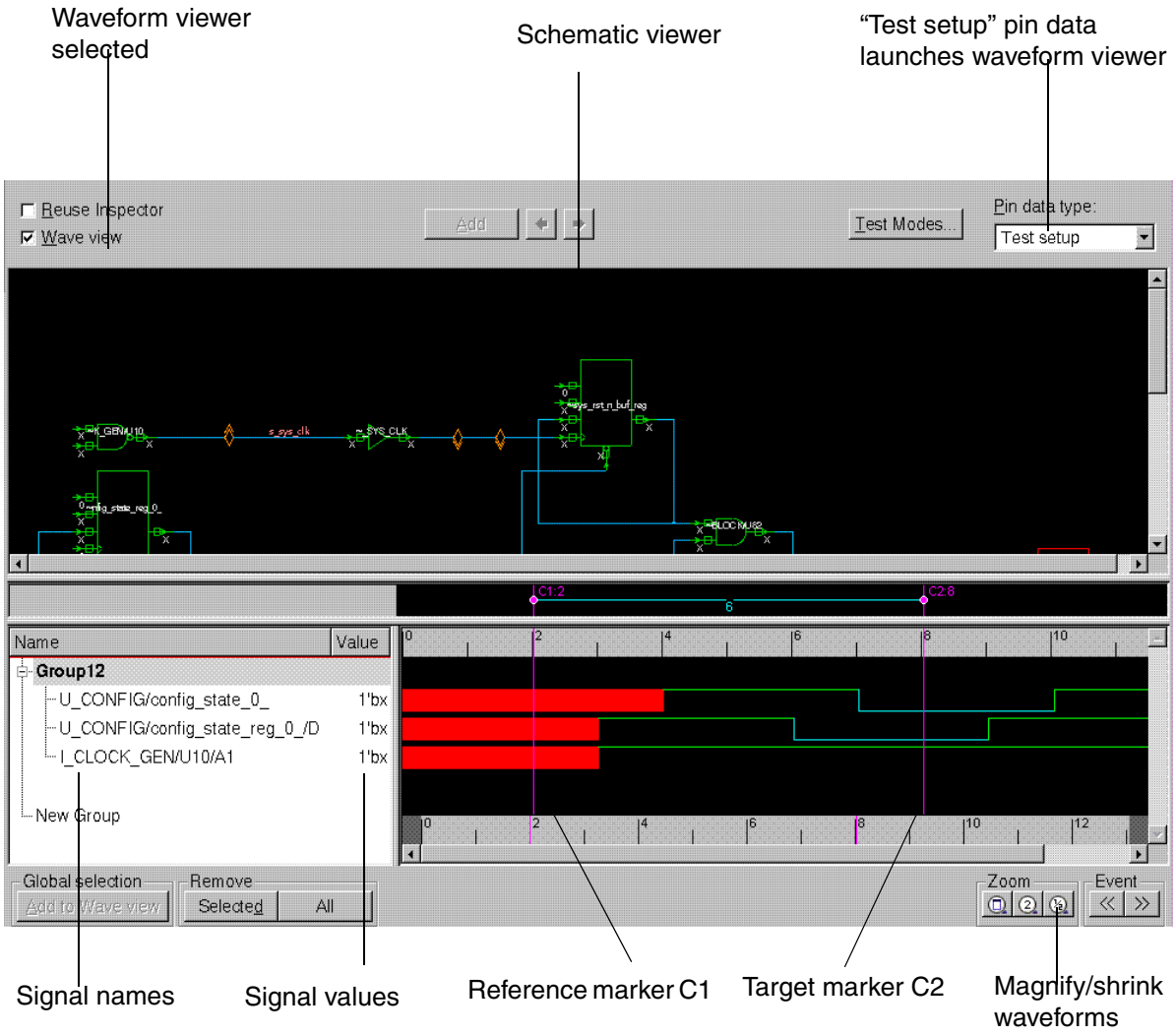
- Select one or more pin names in the violation browser.
- Click the Inspect Violation button.
- Select “Test setup” in the “Pin data type” list.

  The waveform view appears below the schematic view in the violation inspector window, as shown in Figure 3-8. You can adjust the relative heights of these views by dragging the split bar up or down.

  The waveform view consists of two panes: an expandable signal list on the left and the waveform viewer on the right. You can adjust the relative widths of the panes by dragging the split bars left or right.
- Select one or more objects (pins, cells, nets, or buses) for the signals that you want to inspect.
- Click the “Add to Wave View” button.

  The signal names and values appear in the signal list, and a waveform for each signal appears in the waveform viewer.

Figure 3-8 Waveform Viewer



To change the visible time range,

- Drag the pointer left or right over the portion of the global time range that you want to view.

You can use the reference and target markers, C1 and C2, to measure the time between events. C1 marks the current event and C2 marks the event you want to measure. The

number of events or time units between the markers appears in the marker region above the upper time scale.

- To move C1, click or drag the pointer in the marker region.
- To move C2, middle-click or drag the pointer with the middle mouse button in the marker region

You can move or copy signals into a group or from one group to another. You can also remove selected signals or clear the waveform view.

To move signals into a group or from one group to another,

1. Select the signal names in the signal list pane.
2. Drag the selected signals over the group name.

To copy signals into a group or from one group to another,

1. Select the signal names in the signal list pane.
2. Shift-drag the selected signals over the group name.

To remove signals from the waveform view,

1. Select the signal names in the signal list pane.
2. Click the Selected button.

To clear the waveform view, click the All button.

Commands Specific to the DFT Tools in the GUI

Detailed descriptions of the DFT-specific commands and options in the GUI are listed in this section.

gui_inspect_violations

The `gui_inspect_violations` command brings up the specified DFT DRC violations in a new violation inspector window unless a violation inspector window has been marked for reuse. If no violation inspector window exists, a new violation inspector window is created as a new top-level window. Subsequent windows are created in the active top-level window. The new violation inspector window that is created is not marked reusable.

The syntax for this command is

```
gui_inspect_violations -type violation_type violation_list
```

To inspect multiple violations (5, 9,13) of type D1, for example, use the following syntax:

```
gui_inspect_violations -type D1 {5 9 13}
```

To inspect a single violation 4 of type D2, for example, use the following syntax:

```
gui_inspect_violations -type D2 4
```

or

```
gui_inspect_violations D2-4
```

gui_wave_add_signal

The `gui_wave_add_signal` command adds specified objects to the waveform view of a specified violation inspector window. If you specify a cell, a group is created in the waveform view and all the pins of the cell are added to this group as a list of signals. For a bus, all nets are added. The objects that are added will be selected.

The syntax of the command is

```
gui_wave_add_signal
     [-window inspector_window]
     [-clct list]
```

To add a port object i_rd, for example, use the following syntax:

```
# This command adds the port object to the first violation in
# the inspector window with a waveform view
gui_wave_add_signal i_rd
```

To add selected objects, use the following syntax:

```
# Adds selected objects to the waveform view of the violation inspector
# named ViolationInspector.3
gui_wave_add_signal -window ViolationInspector.3 -clct [get_selection]
```

gui_violation_schematic_add_objects

The `gui_violation_schematic_add_objects` command adds specified objects to the schematic view of a specified violation inspector window and selects them.

The syntax of this command is

```
gui_violation_schematic_add_objects
     [-window inspector_window]
     [-clct list]
```

Table 3-1 gui_violation_schematic_add_objects Command Syntax

Options	Descriptions
`-window` *`inspector_window`*	Specifies a signal to be added to the specified violation inspector window. If `inspector_window` is not a valid violation in the inspector window, an error message displays and the command exits. If no `-window` option is specified, the signal is added to the waveform viewer of the first launched violation inspector.
`-clct` *`list`*	Specifies that `list` is to be considered as a collection of object handles. In the absence of the `-clct` option, `list` is considered as a collection of object names.

4

Performing Scan Replacement

This chapter describes the scan replacement process, including constraint-optimized scan insertion.

The scan replacement process inserts scan cells into your design by replacing nonscan sequential cells with their scan equivalents. If you start with an HDL description of your design, scan replacement occurs during the initial mapping of your design to gates. You can also start with a gate-level netlist; in this case, scan replacement occurs as an independent process.

With either approach, scan synthesis considers the design constraints and the impact of both the scan cells themselves and the additional loading due to scan chain routing to minimize the impact of the scan structures on the design.

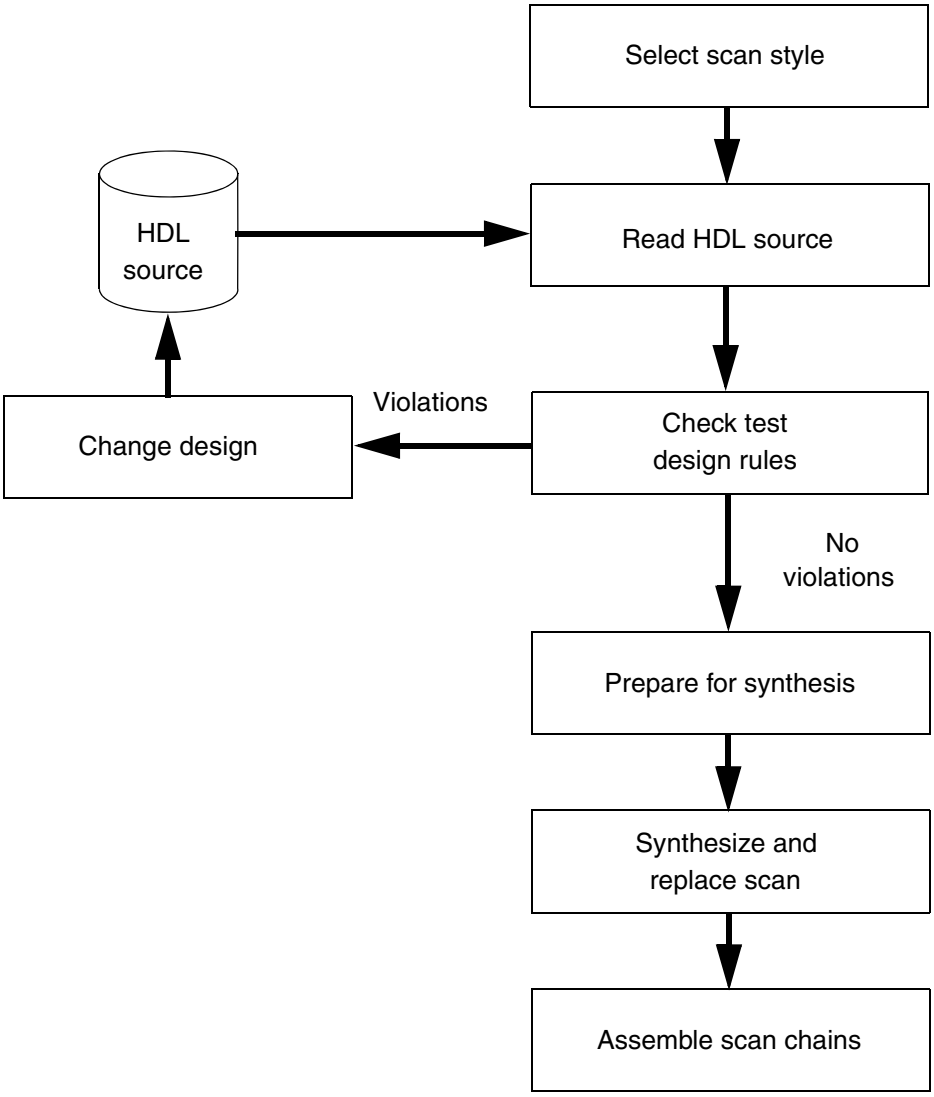
This chapter includes the following sections:

- Validating Your Netlist
- Performing Constraint-Optimized Scan Insertion

Scan Replacement Flow

Figure 4-1 shows the flow for the scan replacement process. This flow assumes that you are starting with an HDL description of the design. If you are starting with a gate-level netlist, you must use constraint-optimized scan insertion. (See "Preparing for Constraint-Optimized Scan Insertion" on page 4-35).

Figure 4-1 Synthesis and Scan Replacement Flow



The following steps explain the scan replacement process:

1. Select a scan style.

   DFT Compiler requires a scan style to perform scan synthesis. The scan style dictates the appropriate scan cells to insert during optimization. You must select a single scan style and use this style on all the modules of your design.

2. Check test design rules of the HDL-level design description.

3. Synthesize your design.

   Test-ready compile maps all sequential cells directly to scan cells. During optimization, DFT Compiler considers the design constraints and the impact of both the scan cells themselves and the additional loading due to scan chain routing to minimize the impact of the scan structures on the design.

Preparing for Scan Replacement

This section discusses what to consider before starting the scan replacement process, and has the following subsections:

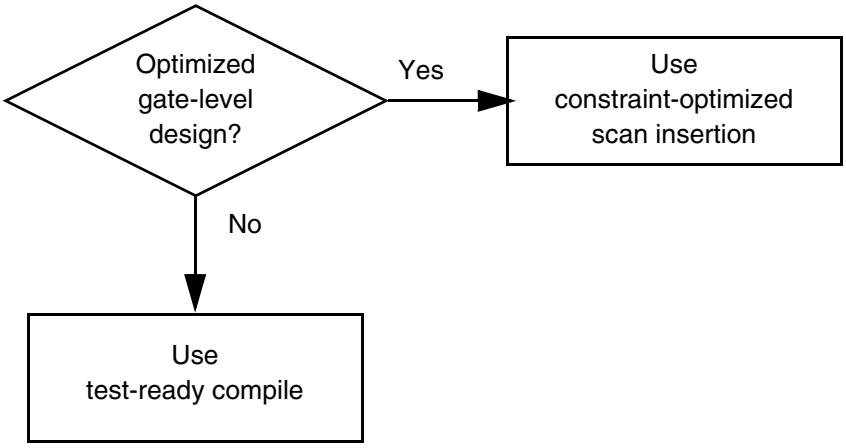
- Selecting a Scan Replacement Strategy
- Identifying Barriers to Scan Replacement
- Preventing Scan Replacement

Selecting a Scan Replacement Strategy

You should select the scan replacement strategy based on the status of your design. If you have an optimized gate-level design and will not be using the `compile` command to perform further optimization, you should use constraint-optimized scan insertion. In all other cases, you should use test-ready compile to insert the scan cells.

Figure 4-2 shows how to determine the appropriate scan replacement strategy.

Figure 4-2 Selecting a Scan Replacement Strategy



Test-ready compile offers the following advantages:

- Single-pass synthesis

  With test-ready compile, the Synopsys tools converge on true one-pass scan synthesis. As a practical matter, design constraints usually result in some cleanup and additional optimization after compile, but test-ready compile is more straightforward compared with other methods.

- Better quality of results

  Test-ready compile offers better quality of results (QoR) compared with past methods. Including scan cells at the time of first optimization results in fewer design rule violations and other constraint violations due to scan circuitry.

- Simpler overall flow

  Test-ready compile requires fewer optimization iterations compared with previous methods.

Note:
For details on constraint-optimized scan insertion, see “Performing Constraint-Optimized Scan Insertion” on page 4-32.

Identifying Barriers to Scan Replacement

You should perform test DRC to identify conditions that prevent scan replacement. Test DRC identifies the following conditions that prevent scan replacement:

- The logic library does not contain an appropriate scan cell for the sequential cell.
- DFT Compiler does not support scan replacement for the sequential cell.
- The sequential cell has an attribute that prevents scan replacement.
- An invalid net drives the clock pin of the sequential cell.
- An invalid net drives the asynchronous pin of the sequential cell.

The following sections provide details about each of these conditions.

Note:

For full details on performing test DRC, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

Logic Library Does Not Contain Appropriate Scan Cells

If a scan equivalent does not exist for a sequential cell, scan replacement cannot occur for that cell. DFT Compiler generates the following warning message when a scan equivalent does not exist for a sequential cell:

```
Warning: No scan equivalent exists for cell %s (%s). (TEST-120)
```

This warning message can occur when

- The logic library does not contain scan cells.
- The logic library contains scan cells, but it does not provide a scan equivalent for the indicated nonscan cell.
- The logic library contains scan cells, but it incorrectly models the scan equivalent for the nonscan cell.
- The logic library contains scan cells, but all possible scan equivalents have the `dont_use` attribute applied.

If DFT Compiler cannot find scan equivalents for any sequential cell in the logic library, it generates the following warning message:

```
Warning: Target library for design contains no scan-cell models.
(TEST-224)
```

If you see this warning, check with your library vendor to see if the vendor provides a logic library that supports scan synthesis.

If DFT Compiler finds a scan cell in the logic library that is not the obvious replacement cell you expect, the reason could be that

- The chosen scan equivalent results in a lower-cost implementation overall.
- The exact scan equivalent does not exist in the logic library or it has the `dont_use` attribute applied.
- The logic library has a problem. In that case, contact the library vendor for more information.

Support for Different Types of Sequential Cells and Violations

DFT Compiler supports sequential cells that have the following characteristics:

- During functional operation, the cell functions as a D flip-flop, a D latch, or a master-slave latch.
- During scan operation, the cell functions as a D flip-flop or a master-slave latch.
- The cell stores a single bit of data.

  Edge-triggered cells that violate this requirement cause DFT Compiler to generate the following warning message:

  ```
  Warning: Cell %s (%s) is not supported because it has
  too many states (%d states). This cell is being
  black-boxed. (TEST-462)
  ```

  Master-slave latch pairs with extra states cause DFT Compiler to generate one of these warning messages, depending on the situation:

  ```
  Warning: Master-slave cell %s (%s) is not supported
  because state pin %s is neither master nor slave. This
  cell is being black-boxed. (TEST-463)

  Warning: Master-slave cell %s (%s) is not supported
  because there are two or more master states. This cell
  is being black-boxed. (TEST-464)

  Warning: Master-slave cell %s (%s) is not supported
  because there are two or more slave states. This cell
  is being black-boxed. (TEST-465)
  ```

- The cell has a three-state output.

  Cells that violate this requirement cause DFT Compiler to generate this warning message:

  ```
  Warning: Cell %s (%s) is not supported because it is a
  sequential cell with three-state outputs. This cell is
  being black-boxed. (TEST-468)
  ```

- The cell uses a single clock per internal state.

  Cells that violate this requirement cause DFT Compiler to generate one of these warning messages:

  ```
  Warning: Cell %s (%s) is not supported because state
  pin %s has no clocks. This cell is being black-boxed. (TEST-466)

  Warning: Cell %s (%s) is not supported because state
  pin %s is multi-port. This cell is being black-boxed. (TEST-467)
  ```

  Note that the cell might use different clocks for functional and test operations.

Note:
Your design will contain unsupported sequential cells only if you explicitly instantiate them. DFT Compiler does not insert unsupported sequential cells.

Attributes That Can Prevent Scan Replacement

The following attributes affect scan replacement:

- `scan_element == false`

  The `scan_element` attribute is applied by the `set_scan_element` command. When set to `false`, it excludes sequential cells from scan replacement and scan stitching. The behavior depends on the type of cell the attribute is applied to:

  - If the `scan_element` attribute is set to `false` on an unmapped sequential cell or a nonscan cell, the cell is never scan-replaced or scan-stitched.
  - If the `scan_element` attribute is set to `false` on a test-ready cell,
    - Subsequent test-ready compile commands do not unscan it.
    - In wire load mode, the `insert_dft` command unscans it (unless the `set_dft_insertion_configuration -synthesis_optimization` option is set to `none`) and does not stitch it into scan chains.
    - In topographical mode, the `insert_dft` command keeps the cell scan-replaced (to minimize layout disturbance) but does not stitch it into scan chains.

- `dont_touch == true`

  The `dont_touch` attribute is applied by the `set_dont_touch` command. When set to `true`, it prevents the cell type from being changed, which indirectly affects scan replacement. The behavior depends on the type of cell the attribute is applied to:

  - Test-ready cell: If the `dont_touch` attribute is set to `true` on a test-ready cell, the cell is kept as a scan-replaced cell. If the cell is a valid scan cell, it is stitched into scan chains. If not, due to other directives such as `set_scan_element false`, it remains as an unstitched test-ready cell.

- Nonscan cell: If the `dont_touch` attribute is set to `true` on a nonscan cell, the cell is kept as a nonscan cell. It is not scan-replaced or stitched into scan chains. Pre-DFT DRC notes such cells with the following information message:

  ```
  Information: Cell %s (%s) could not be made scannable as
  it is dont_touched. (TEST-121)
  ```

  Nonscan cells identified as shift register elements can be stitched into scan chains.

- Unmapped sequential cell: If the `dont_touch` attribute is set to `true` on an unmapped sequential cell before the initial compile, the attribute prevents the cell from being mapped. As a result, DFT insertion fails with the following error message:

  ```
  Error: DFT insertion isn't supported on designs with unmapped cells.
  (TEST-269)
  ```

The `dont_touch` attribute is ignored when an identified shift register is split by the scan architect; the head scan flip-flops of any new shift register segments are scan-replaced even if they have the `dont_touch` attribute applied.

A `dont_touch` attribute on the top-level design does not affect scan replacement of the design.

Note:

Although the `-exclude_elements` option of the `set_scan_configuration` excludes cells from scan stitching, it does not prevent scan replacement, and it does not cause excluded test-ready cells to be unscanned. To prevent cells from being scan-replaced, use the `set_scan_element false` command.

Nonscan sequential cells generally reduce the fault coverage results for full-scan designs. If you do not want to exclude affected cells from scan replacement, remove the script commands that apply the attributes, then rerun the script.

Invalid Clock Nets

The term *system clock* refers to a clock used in the parallel capture cycle. The term *test clock* refers to a clock used during scan shift. Multiplexed flip-flop designs use the same clock as both the system clock and the test clock.

In a nonscan design, an invalid clock net, whether a system clock or a test clock, prevents scan replacement of all sequential cells driven by that clock net.

The requirements for valid clocks in DFT Compiler include the following:

- A system or test clock used during scan testing must be driven from a single top-level port.

  An active clock edge at a sequential cell must be the result of a clock pulse applied at a top-level port, not the result of combinational logic driving the clock net.

- A system or test clock used during scan testing can be driven from a bidirectional port.

  DFT Compiler supports the use of bidirectional ports as clock ports if the bidirectional ports are designed as input ports during chip test. If a bidirectional port drives a clock net but the port is not designed as an input port during chip test mode, DFT Compiler forces the net to X and cells clocked by the net to become black-box sequential cells.

- A system or test clock used during scan testing must be generated in a single tester cycle.

  The clock pulse applied at the clock port must reach the sequential cells in the same tester cycle. DFT Compiler does not support sequential gating of clocks, such as clock divider circuitry.

- A system or a test clock used during scan testing cannot be the result of multiple clock inputs.

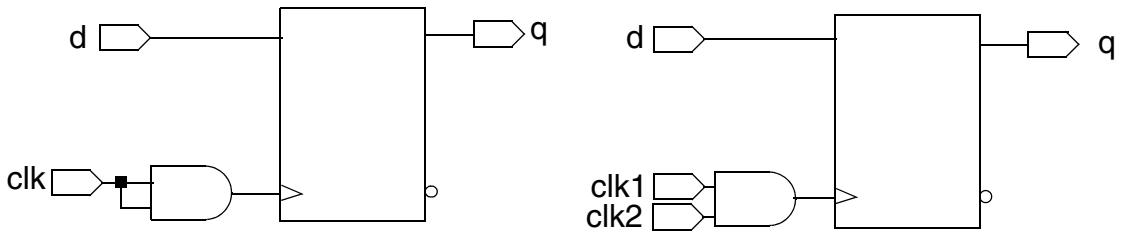
  DFT Compiler does not support the use of combinationally combined clock signals, even if the same port drives the signal.

  Note:

  If the same port drives the combinationally combined clock signal, as shown in the design on the left in Figure 4-3 or the design in Figure 4-4, DFT Compiler does not detect the problem in nonscan or unrouted scan designs.

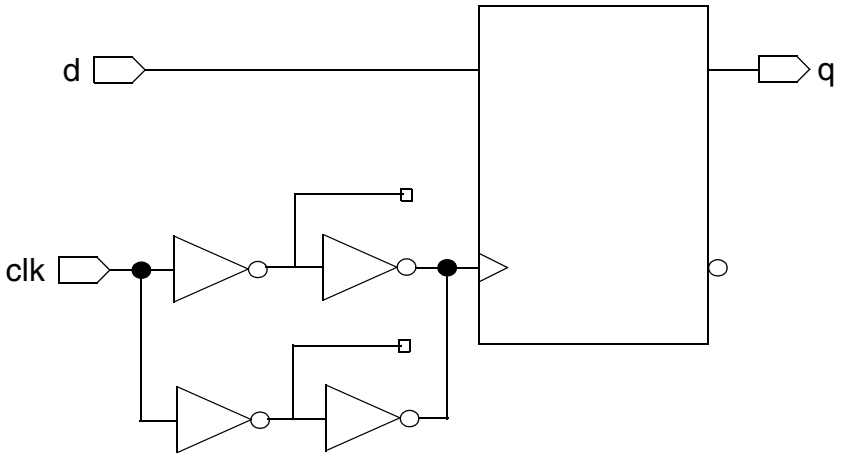
  Figure 4-3 shows design examples that use combinationally combined clocks. When multiple clock signals drive a clock net, DFT Compiler forces the net to X and cells clocked by the net become black-box sequential cells.

Figure 4-3 Examples of Combinationally Combined Clock Nets



DFT Compiler supports the use of reconvergent clocks, such as clocks driven by parallel clock buffers. Figure 4-4 shows a design example that uses a reconvergent clock net.

Figure 4-4 Example of a Reconvergent Clock Net



- A test clock must remain active throughout the scan shift process.

  To load the scan chain reliably, make sure the test clock remains active until scan shift completes. For combinationally gated clocks, you must configure the design to disable the clock gating during scan shift.

  DFT Compiler supports combinational clock gating during the parallel capture cycle.

Test design rule checking on a nonscan design might not detect invalid clock nets. DFT Compiler identifies all invalid clock nets only in existing scan designs.

DFT Compiler cannot control the clock net when

- A sequential cell drives the clock net
- A multiplexer with an unspecified select line drives the net of a test clock
- Combinational clock-gating logic can generate an active edge on the clock net

DFT Compiler generates this warning message when it detects an uncontrollable clock:

```
Warning: Normal mode clock pin %s of cell %s (%s) is
uncontrollable. (TEST-169)
```

Because uncontrollable clock nets prevent scan replacement, you should correct uncontrollable clocks. Sequentially driven clocks require test-mode logic to bypass the violation. You can bypass violations caused by other sources of uncontrollable clocks by using test configuration or test-mode logic.

DFT Compiler can control a combinationally gated test clock that cannot generate an active clock edge. However, DFT Compiler considers this type of clock invalid, because the clock

might not remain active throughout scan shift. In this case, DFT Compiler generates this warning message:

```
Warning: Shift clock pin %s of cell %s (%s) is illegally
gated. (TEST-186)
```

Because invalid gated-clock nets prevent scan replacement, you should correct invalid gated clocks. You can use AutoFix to bypass invalid gated clocks when using the multiplexed flip-flop scan style. You might also be able to change the test configuration to bypass the violation. For more information on AutoFix, see "Using AutoFix" on page 7-22.

Invalid Asynchronous Pins

DFT Compiler considers a net that drives an asynchronous pin as valid if it can disable the net from an input port or from a combination of input ports. DFT Compiler cannot control an asynchronous pin driven by ungated sequential logic.

In a nonscan design, a net with an uncontrollable asynchronous pin prevents scan replacement of all sequential cells connected to that net.

DFT Compiler generates this warning message when it detects an uncontrollable asynchronous pin:

```
Warning: Asynchronous pins of cell FF_A (FD2) are uncontrollable.
(TEST-116)
```

Because nets with an uncontrollable asynchronous pin prevent scan replacement, you should correct uncontrollable nets. Use AutoFix if you are using the multiplexed flip-flop scan style, test configuration, or test-mode logic to bypass uncontrollable asynchronous pin violations.

Preventing Scan Replacement

You can use the following attributes to prevent scan replacement during test-ready compile and DFT insertion:

- `scan_element == false`
- `dont_touch == true`

Setting the `scan_element` attribute to `false` prevents a cell from being scan-replaced and scan stitched. Setting the `dont_touch` attribute to `true` on a nonscan cell prevents it from being scan replaced and scan stitched. For more information on these attributes, see "Attributes That Can Prevent Scan Replacement" on page 4-8.

Specifying a Scan Style

This section explains the process for selecting and specifying a scan style for your design. It has the following subsections:

- Types of Scan Styles
- Scan Style Considerations
- Setting the Scan Style

Types of Scan Styles

DFT Compiler supports the scan styles listed in the following subsections:

- Multiplexed Flip-Flop Scan Style
- Clocked Scan Style
- LSSD Scan Style
- Scan-Enabled LSSD Scan Style

Note:
The LSSD scan style includes the LSSD and clocked LSSD scan styles.

This section briefly describes each scan style. For more detailed information on scan styles, see the “Scan Styles” chapter of the *DFT Overview User Guide*.

Multiplexed Flip-Flop Scan Style

DFT Compiler supports multiplexed flip-flop scan equivalents for D flip-flops and master-slave flip-flops. The multiplexed flip-flop scan equivalents for all flip-flop styles must be fully functionally modeled in the logic library. This scan style has the following advantages:

- Multiplexed flip-flop is the most widely known and understood scan style.
- Multiplexed flip-flop scan cells are easy to design and characterize, as they consist of a conventional flip-flop plus a data selection MUX.

The multiplexed flip-flop scan style has the disadvantage that hold time or clock skew problems can occur on the scan path because of a short path from a scan cell’s scan output pin to the next scan cell’s scan input pin. DFT Compiler can reduce the occurrence of these problems by considering hold time violations during optimization.

Clocked Scan Style

DFT Compiler supports clocked-scan equivalents for D flip-flops and latches. The clocked-scan style is well suited for use in multiple-clock designs because of the dedicated test clock.

The clocked-scan style also has some disadvantages:

- Hold time or clock skew problems can occur on the scan path because the path from a scan cell's scan output pin to the next scan cell's scan input pin is too short. DFT Compiler can reduce the occurrence of these problems by considering hold time violations during optimization.
- This scan style requires the routing of two edge-triggered clocks. Routing clock lines is difficult because you must carefully control the clock skew.

LSSD Scan Style

DFT Compiler supports level-sensitive scan design (LSSD) equivalents for D flip-flops, master-slave flip-flops, and D latches. Timing problems on the scan path are unlikely in LSSD designs because of the use of nonoverlapping two-phase clocks during the scan operation.

The LSSD scan style also has some disadvantages:

- This scan style requires a greater wiring area than the multiplexed flip-flop or clocked-scan styles.
- DFT Compiler does not support the scan replacement of more complex LSSD cells, such as multiple data port master latches.

When you use the LSSD scan style, define the clock waveforms so that the master and slave clocks have nonoverlapping waveforms because master and slave latches should never be active simultaneously.

Scan-Enabled LSSD Scan Style

DFT Compiler supports scan-enabled level-sensitive scan design (LSSD) equivalents for D flip-flops. This scan style is similar to the LSSD scan style, except that a global scan-enable signal is used to repurpose the functional clock as the slave test clock in test mode. Timing problems on the scan path are unlikely in LSSD designs because of the use of nonoverlapping two-phase clocks during the scan operation.

The scan-enabled LSSD scan style also has some disadvantages:

- This scan style requires a greater wiring area than the multiplexed flip-flop or clocked-scan styles.
- DFT Compiler only supports the scan replacement of flip-flops.

When you use the scan-enabled LSSD scan style, define the clock waveforms so that the master and slave clocks have nonoverlapping waveforms because master and slave latches should never be active simultaneously.

Scan Style Considerations

You must select a single scan style and use this style for all modules of your design. Consider the following questions when selecting a scan style:

- Which scan styles are supported in your logic library?

  To make it possible to implement internal scan structures in the scan style you select, appropriate scan cells must be present in the technology libraries specified in the `target_library` variable.

  Use of sequential cells that do not have a scan equivalent always results in a loss of fault coverage in full-scan designs. Techniques to verify scan equivalents are discussed in "Verifying Scan Equivalents in the Logic Library" on page 4-17.

- What is your design style?

  If your design is predominantly edge-triggered, use the multiplexed flip-flop, clocked scan, clocked LSSD, or scan-enabled LSSD scan style.

  If your design has a mix of latches and flip-flops, use the clocked scan or LSSD scan style.

  If your design is predominantly level-sensitive, use the LSSD scan style.

- How complete are the models in your logic library?

  The quality and accuracy of the scan and nonscan sequential cell models in the Synopsys logic library affect the behavior of DFT Compiler. Incorrect or incomplete library models can cause incorrect results during test design rule checking.

  DFT Compiler requires a complete functional model of a scan cell to perform test design rule checking. The Library Compiler UNIGEN model supports complete functional modeling of all supported scan cells. However, the usual Library Compiler sequential modeling syntax supports only complete functional modeling for multiplexed flip-flop scan cells.

  When the logic library does not provide a functional model for a scan cell, the cell is a black box for DFT Compiler.

For information on the scan cells in the logic library you are using, see your ASIC vendor. For information on creating logic library elements or to learn more about modeling scan cells, see the information about defining test cells in the Library Compiler documentation.

Setting the Scan Style

You can specify the scan style by using either the `test_default_scan_style` variable or the `set_scan_configuration -style` command:

- `test_default_scan_style` – applies to all designs in the current session. The syntax is as follows:

  `set_app_var test_default_scan_style` *`style`*

- `set_scan_configuration -style` – applies only to the current design. If your selected scan style differs from the default scan style, you must execute this command for each module. The syntax is as follows:

  `set_scan_configuration -style` *`style`*

Table 4-1 shows the scan style keywords used to specify the scan style. Use these keywords with either the `test_default_scan_style` variable or the `set_scan_configuration -style` command.

Table 4-1 Scan Style Keywords

Scan style	Keyword
Multiplexed flip-flop	`multiplexed_flip_flop`
Clocked scan	`clocked_scan`
Level-sensitive scan design	`lssd`
Scan-enabled level-sensitive scan design	`scan_enabled_lssd`

The following examples show how to specify the scan style by using either the `test_default_scan_style` variable or the `set_scan_configuration` command:

```
dc_shell> set_app_var test_default_scan_style \
              multiplexed_flip_flop

dc_shell> set_scan_configuration -style clocked_scan
```

Verifying Scan Equivalents in the Logic Library

Before starting scan synthesis, you need to confirm that your logic library contains scan cells and then verify that the scan cells are suitable for the selected scan style.

This section has the following subsections:

- Checking the Logic Library for Scan Cells
- Checking for Scan Equivalents

Checking the Logic Library for Scan Cells

You can determine whether the logic library contains scan cells by using either of the following methods:

- Search the library .ddc file.

  Every scan cell, regardless of the scan style, must have a scan input pin and a scan output pin. You can determine whether the logic library contains scan cells by using the `filter` command to search for scan input or scan output pins.

  Depending on its polarity, a scan input pin can have a `signal_type` attribute of either `test_scan_in` or `test_scan_in_inverted` in the logic library. A scan output pin can have a `signal_type` attribute of either `test_scan_out` or `test_scan_out_inverted` in the logic library, depending on its polarity.

  The following command sequence shows the use of the `filter` command:

  ```
  dc_shell> read_ddc class.ddc

  dc_shell> get_pins class/*/* -filter "@signal_type = test_scan_in"
  ```

  If the library contains scan cells, the `filter` command returns a list of pins; if the library does not contain scan cells, the `filter` command returns an empty list.

- Check the test design rules.

  As one of the first checks it performs, the `dft_drc` command determines the presence of scan cells in the logic library. If the technology libraries identified in the `target_library` variable do not contain scan cells, the `dft_drc` command generates the following warning message:

  ```
  Warning: Target library for design contains no scan-cell models.
  (TEST-224)
  ```

  You must define the `current_design` before you run the `dft_drc` command.

If your logic library does not contain scan cells, check with your semiconductor vendor to see if the vendor provides a logic library that supports test synthesis.

Checking for Scan Equivalents

To verify that the logic library contains scan equivalents for the sequential cells in your design, run the `dft_drc` command on your design or on a design containing the sequential cells likely to be used in your design.

If the logic library does not contain a scan equivalent for a sequential cell in a nonscan design or the scan equivalent has the `dont_use` attribute applied, the `dft_drc` command generates the following warning message:

```
Warning: No scan equivalent exists for cell instance (reference).
(TEST-120)
```

In verbose mode (`dft_drc -verbose`), the TEST-120 message lists all scan equivalent pairs available in the target library in the selected scan style. If the target library contains no scan equivalents in the chosen scan style, no scan equivalents are listed.

Suppose you have a design containing D flip-flops but the target logic library contains scan equivalents only for JK flip-flops. Example 4-1 shows the warning message issued by the `dft_drc` command, along with the scan equivalent mappings to the available scan cells.

Example 4-1 Scan Equivalent Listing

```
Warning: No scan equivalent exists for cell q_reg (FD1P). (TEST-120)

Scan equivalent mappings for target library are:

        FJK3      -> FJK3S
        FJK2      -> FJK2S
        FJK1      -> FJK1S
```

Scan Cell Replacement Strategies

This section describes how to select the set of scan cells and multibit components to use in your scan replacement strategy. It includes the following subsections:

- Specifying Scan Cells
- Multibit Components

Specifying Scan Cells

Before you perform scan cell replacement, you need to specify the set of scan cells to be used by DFT Compiler. This section has the following subsections:

- Restricting the List of Available Scan Cells
- Scan Cell Replacement Strategies
- Mapping Sequential Gates in Scan Replacement

Restricting the List of Available Scan Cells

The `set_scan_register_type` command lets you specify which flip-flop scan cells are to be used by `compile -scan` to replace nonscan cells. The command restricts the choices of scan cells available for scan replacement. You can apply this restriction to the current design, to particular designs, or to particular cell instances in the design.

Note:

The `set_scan_register_type` command applies to the operation of both the `compile -scan` command and the `insert_dft` command.

The `set_scan_register_type` command has the following syntax:

```
set_scan_register_type [-exact]
    -type scan_flip_flop_list [cell_or_design_list]
```

The *`scan_flip_flop_list`* value is the list of scan cells that the `compile -scan` command is allowed to use for scan replacement. There must be at least one such cell named in the command. Specify each scan cell by its cell name alone, without the library name.

The *`cell_or_design_list`* value is a list of designs or cell instances where the restriction on scan cell selection is to be applied. In the absence of such a list, the restriction applies to the current design, set by the `current_design` command, and to all lower-level designs in the design hierarchy.

The `-exact` option determines whether the restriction on scan cell selection also applies to back-end delay and area optimization done by the `insert_dft` command or by subsequent synthesis operations such as the `compile -incremental` command. If the `-exact` option is used, the restriction still applies to back-end optimization. In other words, scan cells can be replaced only by other scan cells in the specified list. If the `-exact` option is not used, the optimization algorithm is free to use any scan cell in the target library.

Scan Cell Replacement Strategies

Here are some examples of `set_scan_register_type` commands:

```
dc_shell> set_scan_register_type -exact -type FD1S
```

This command causes the compile -scan command to use only FD1S scan cells to replace nonscan cells in the current design. Because of the -exact option, this restriction applies to both initial scan replacement and subsequent optimization.

```
dc_shell> set_scan_register_type -exact \
               -type {FD1S FD2S} {add2 U1}
```

This command causes compile -scan to use only FD1S or FD2S scan cells to replace each nonscan cell in all designs and cell instances named add2 or U1. In all other designs and cell instances, compile -scan can use any scan cells available in the target library. The -exact option forces any back-end delay optimization to respect the scan cell list, thus allowing only FD1S and FD2S to be used.

```
dc_shell> set_scan_register_type \
               -type {FD1S FD2S} {add2 U1}
```

This command is the same as the one in the previous example, except that the -exact option is not used. This means that the back-end optimization algorithm is free to replace the FD1S and FD2S cells with any compatible scan cells in the target library.

If you use the set_scan_register_type command on generic cell instances, be sure to use the -scan option with the compile command. Otherwise, the scan specification will be lost.

To report scan paths, scan chains, and scan cells in the design, use the report_scan_path command, as shown in the following examples:

```
dc_shell> report_scan_path -view existing_dft \
               -chain all

dc_shell> report_scan_path -view existing_dft -cell all
```

To cancel all set_scan_register_type settings currently in effect, execute the following command:

```
dc_shell> remove_scan_register_type
```

Mapping Sequential Gates in Scan Replacement

To use the set_scan_register_type command effectively, understanding the scan replacement process is important.

The compile -scan command maps sequential gates into scan flip-flops and latches, using three steps:

1. The compile -scan command maps each sequential gate in the generic design description into an initial nonscan latch or flip-flop from the target library. In the absence of any set_scan_register_type specification, compile -scan chooses the smallest area-cost flip-flop or latch. For a design or cell instance that has a

`set_scan_register_type` setting in effect, `compile -scan` chooses the nonscan equivalent of a scan cell in the *`scan_flip_flop_list`*.

2. The `compile -scan` command replaces the nonscan flip-flops with scan flip-flops, using only the scan cells specified in the `set_scan_register_type` command, where applicable. If `compile -scan` is unable to use a scan cell from the *`scan_flip_flop_list`*, it uses the best matching scan cell from the target library and issues a warning.
3. If the `-exact` option is not used in the `set_scan_register_type` command, the Design Compiler and DFT Compiler tools attempt to remap each scan flip-flop into another component from the target library to optimize the delay or area characteristics of the circuit. If the `-exact` option is used, optimization is restricted to using the scan cells in the *`scan_flip_flop_list`*.

The operation of step 1 can be controlled by the `set_register_type` command. The `set_register_type` command specifies a list of allowed cells for implementing functions specified in the HDL description of the design, but you need to be careful about using this command in conjunction with scan replacement. For example, if you tell the `compile` command to use a sequential cell that has no scan equivalent, DFT Compiler will not be able to replace the cell with a corresponding scan cell.

The `set_scan_register_type` command affects only the replacement of nonscan cells with scan cells. It cannot be used to force existing scan cells to be replaced by new scan cells. To make this type of design change, you need to go back to the original nonscan design and apply a new `set_scan_register_type` specification, followed by a new `compile -scan` or `insert_dft` operation.

Multibit Components

Multibit components are supported by DFT Compiler during scan replacement. This section has the following subsections:

- What Are Multibit Components?
- How DFT Compiler Assimilates Multibit Components
- Controlling Multibit Test Synthesis
- Performing Multibit Component Scan Replacement
- Disabling Multibit Component Support

What Are Multibit Components?

A multibit component is a sequence of cells with identical functionality. It can consist of single-bit cells or the set of multibit cells supported by Design Compiler synthesis. Cells can have identical functionality even if they have different bit-widths. Multibit synthesis ensures regularity and predictability of layout.

HDL Compiler infers multibit components through HDL directives. See the *HDL Compiler for Verilog User Guide* for more information about multibit inference. Specify multibit components by using the Design Compiler `create_multibit` command and `remove_multibit` command. Control multibit synthesis by using the `set_multibit_options` command. For further information, see the *Design Compiler Optimization Reference Manual*.

When you create a new multibit component with the `create_multibit` command, choose a name that is different from the name of any existing object in your design. This will prevent possible conflicts later when you use the `set_scan_path` command.

Structured logic synthesis is a special case of multibit synthesis in which the individual bits of a multibit component are implemented as distinct elements. Use the `set_multibit_options -mode structured` command to enable structured logic synthesis.

How DFT Compiler Assimilates Multibit Components

Multibit components have the following properties:

- All the synthesis and optimization that DFT Compiler performs is as prescribed by the multibit mode in effect.
- Scan chain allocation and routing result in a layout that is as regular as possible.

To achieve these goals, DFT Compiler assimilates sequential multibit components into synthesizable segments.

A synthesizable segment, an extension of the user segment concept, has the following properties:

- Its implementation is not fixed at the time of specification.
- It consists of a name and a sequence of cells that implicitly determine an internal routing order.
- It lacks access pins and possibly internal routing.
- It does not need to be scan-replaced.
- Test synthesis controls the implementation.

A synthesizable segment that cannot be synthesized into a valid user segment is invalid. Only multibit synthesizable segments are supported.

Controlling Multibit Test Synthesis

You control multibit test synthesis through the specification of the scan configuration by using the following commands:

- `set_scan_configuration`
- `reset_scan_configuration`
- `set_scan_path`
- `set_scan_element`

Commands that accept segment arguments also accept multibit components. You can refer by instance name to multibit components from the top-level design through the design hierarchy. Commands that accept sets of cells also accept multibit components. When you specify a multibit component as being a part of a larger segment, the multibit component is included in the larger user-defined segment without modification.

Performing Multibit Component Scan Replacement

Use the `compile -scan` command or the `insert_dft` command to perform multibit component scan replacement. These commands perform a homogeneous scan replacement. Bits of a multibit component are either all scan-replaced or all not scan-replaced. Bits are then assembled into multibit cells as specified by the `set_multibit_options` command.

The number of cells after scan replacement can change. For example, a 4-bit cell can be scan-replaced by two 2-bit cells. If this occurs, the two 2-bit cells get new names. If the cell is scan-replaced with a cell of equal width, a 4-bit cell replaced by a 4-bit cell for example, the name of the cell remains the same.

You control the scan replacement of multibit components by using the `set_scan_element` command.

When specifying individual cells by using either of these commands, do not specify an incomplete multibit component unless you previously disabled multibit optimization.

Disabling Multibit Component Support

You can disable structured logic and multibit component support by doing one of the following:

- Remove some or all of the multibit components by using the `remove_multibit` command.
- Turn off scan synthesis by using the `set_scan_configuration -preserve_multibit_segment false` command.
- Remove a previously defined scan path by using the `remove_scan_path` command.

Test-Ready Compilation

Scan cell replacement works most efficiently if it is done when you compile your design. This section describes the following topics related to the test-ready compilation process:

- What Is Test-Ready Compile?
- Preparing for Test-Ready Compile
- Controlling Test-Ready Compile
- Comparing Default Compile and Test-Ready Compile
- Complex Compile Strategies

What Is Test-Ready Compile?

Test-ready compile integrates logic optimization and scan replacement. During the first synthesis pass of each HDL design or module, test-ready compile maps all sequential cells directly to scan cells. The optimization cost function considers the impact of the scan cells themselves and the additional loading due to the scan chain routing. By accounting for the timing impact of internal scan design from the start of the synthesis process, test-ready compile eliminates the need for an incremental compile after scan insertion.

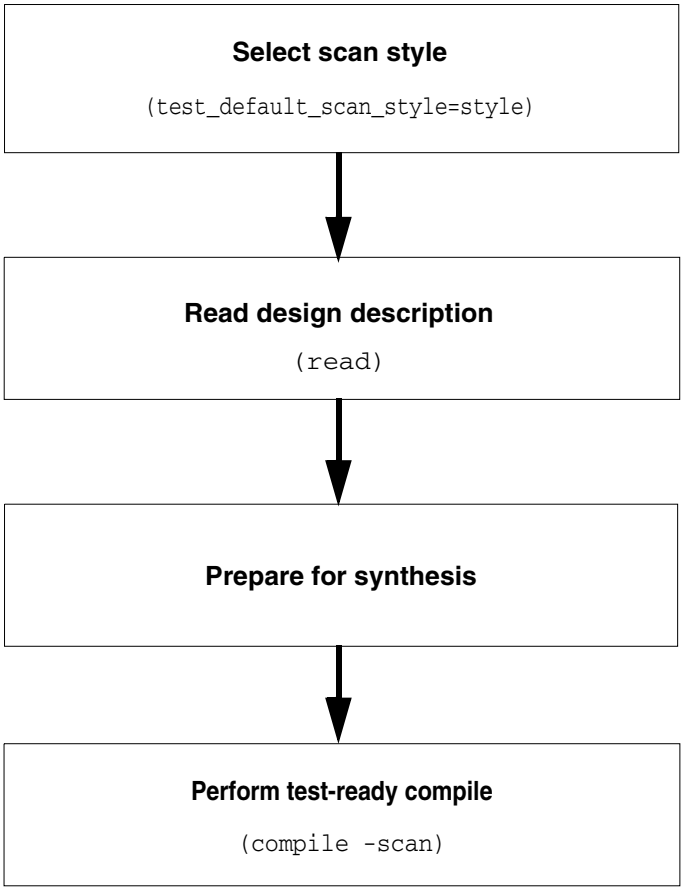
During optimization, DFT Compiler cannot determine whether the sequential cells in your HDL description meet the test design rules, so it maps all sequential cells to scan cells. Later in the scan synthesis process, DFT Compiler can convert some sequential cells back to nonscan cells. For example, test design rule checking might find scan cells with test design rule violations. In other circumstances, you might manually specify some sequential cells as nonscan elements. In such cases, DFT Compiler converts the scan cells to nonscan equivalents during execution of the `insert_dft` command.

Typically, the input to test-ready compile is an HDL design description. You can also perform test-ready compile on a nonscan gate-level netlist that requires optimization. For example, a gate-level netlist resulting from technology translation usually requires logic optimization to meet constraints. In such a case, use test-ready compile to perform scan replacement.

The Test-Ready Compile Flow

Figure 4-5 shows the test-ready compile flow and the commands required to complete this flow.

Figure 4-5 Test-Ready Compile Flow



Before performing test-ready compile:

- Select a scan style

  For information about selecting a scan style, see “Specifying a Scan Style” on page 4-13.

- Prepare for logic synthesis

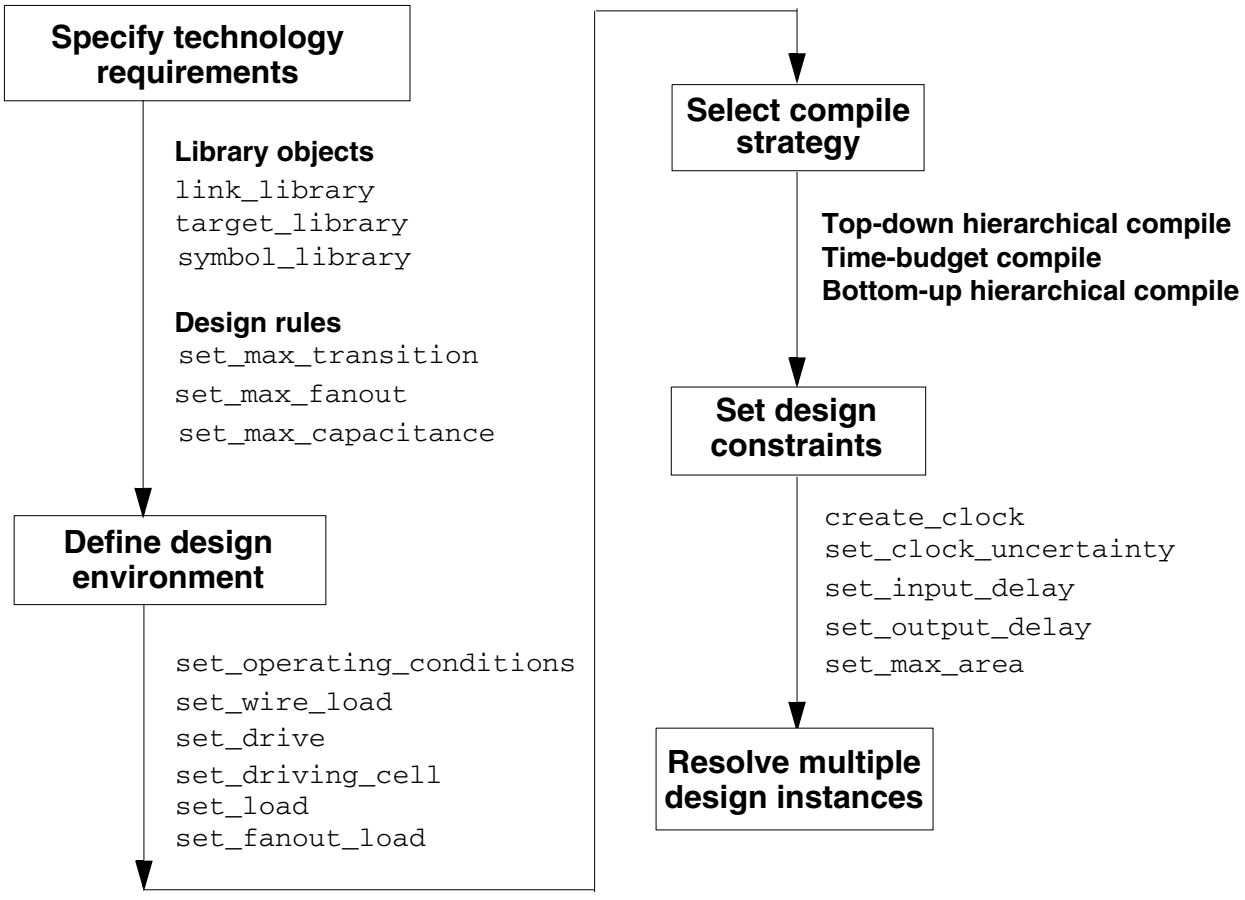
  For information about preparing for logic synthesis, see “Preparing for Test-Ready Compile” on page 4-26.

The result of test-ready compile is an optimized design that contains unrouted scan cells. The optimization performed during test-ready compile accounts for both the impact of the scan cells and the additional loading due to the scan chain routing. A design in this state is known as an *unrouted scan design*.

Preparing for Test-Ready Compile

Figure 4-6 shows the synthesis preparation steps. For more information about these steps, see the *Design Compiler User Guide*.

Figure 4-6 Synthesis Preparation Steps



Performing Test-Ready Compile in the Logic Domain

The `compile -scan` command invokes test-ready compile. You must enter this command from the `dc_shell-xg-t` command line; the Design Analyzer menus do not support the `-scan` option.

```
dc_shell> compile -scan
```

For details of how to constrain your design and for other compile options, see the *Design Compiler Optimization Reference Manual*.

Controlling Test-Ready Compile

You can use the following variable and commands to control scan implementation by `compile -scan`:

- `test_default_scan_style` or `set_scan_configuration -style`
- `set_scan_element` *`element_name`* `true | false`
- `set_scan_register_type [-exact] -type` *`scan_flip_flop_list`* [*`cell_or_design_list`*]
- `set_scan_configuration -preserve_multibit_segment`

The `test_default_scan_style` variable determines which scan style the `compile -scan` command uses for scan implementation. You can also use the `set_scan_configuration -style` command for the same purpose.

You might not want to include a particular element in a scan chain. If this is the case, first analyze and elaborate the design. Then, use the `set_scan_element false` command in the generic technology (GTECH) sequential element. Subsequently, when you use the `compile -scan` command, this element is implemented as an ordinary sequential element and not as a scan cell. The following example shows a script that uses the `set_scan_element false` command on generics:

```
analyze -format VHDL -library WORK switch.vhd
elaborate -library WORK -architecture rtl switch
set_scan_element false Q_reg
compile -scan
```

Note:

Use the `set_scan_element false` statement sparingly. For combinational ATPG, using nonscan elements generally results in lower fault coverage.

You might want to specify which flip-flop scan cells are to be used for replacing nonscan cells in the design. In that case, use the `set_scan_register_type` command as described in “Specifying Scan Cells” on page 4-19.

Comparing Default Compile and Test-Ready Compile

The following example shows the effect of test-ready compile on a small design. The Verilog description shown in Example 4-2 describes a small design containing two flip-flops: one a simple D flip-flop and one a flip-flop with a multiplexed data input.

Example 4-2 Verilog Design Example

```
module example (d1,d2,d3,sel,clk,q1,q2);
input d1,d2,d3,sel,clk;
output q1,q2;
reg q1,q2;
   always @ (posedge clk) begin
      q1 = d1;
      if (sel) begin
         q2=d2;
      end else begin
         q2=d3;
      end
   end
endmodule
```

The following command sequence performs the default compile process on the Verilog design example:

```
dc_shell> set target_library class.db
dc_shell> read_file -format verilog example.v
dc_shell> set_max_area 0
dc_shell> compile
```

Example 4-3 shows the VHDL equivalent to the Verilog design example provided in Example 4-2.

Example 4-3 VHDL Design Example

```
-------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
-------------------------------------------------
entity EXAMPLE is
   port( d1:in       STD_LOGIC;
         d2:in       STD_LOGIC;
         d3:in       STD_LOGIC;
         sel:in      STD_LOGIC;
         clk:in      STD_LOGIC;
         q1:out      STD_LOGIC;
         q2:out      STD_LOGIC
   );
end EXAMPLE;
-------------------------------------------------
architecture RTL of EXAMPLE is
```

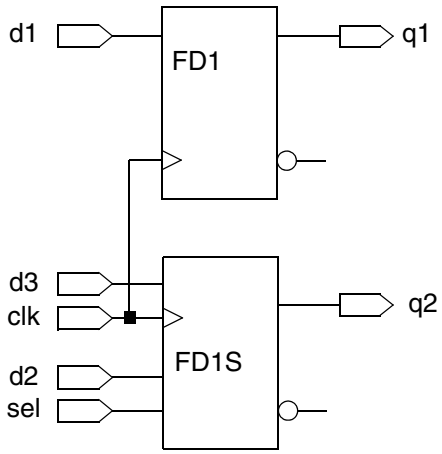
```
begin
   process
   begin
     wait until (clk'event and clk = '1');
     q1 <= d1;
     if (sel = '1') then
        q2 <= d2;
     else
        q2 <= d3;
     end if;
   end process;
end RTL;
```

The following command sequence performs the default compile process on the VHDL design example:

```
dc_shell> set target_library class.db
dc_shell> analyze -format vhdl \
               -library work example.vhd
dc_shell> elaborate -library work EXAMPLE
dc_shell> set_max_area 0
dc_shell> compile
```

Figure 4-7 shows the result of the default compile process on the design example. Design Compiler synthesis uses the D flip-flop (FD1) and the multiplexed flip-flop scan cell (FD1S) from the class logic library to implement the specified functional logic.

Figure 4-7 Gate-Level Design: Default Compile



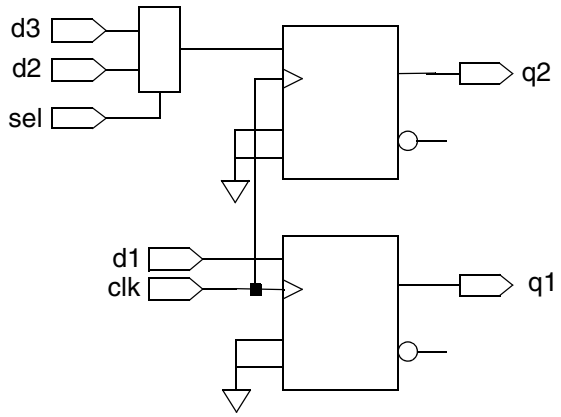
Using default compile increases the scan replacement runtime and can result in sequential cells that do not have scan equivalents.

To invoke test-ready compile, specify the scan style before optimization and use the `-scan` option of the `compile` command.

```
dc_shell> set_app_var test_default_scan_style \
                multiplexed_flip_flop
dc_shell> compile -scan
```

Figure 4-8 shows the result of the test-ready compile process on the design example.

Figure 4-8 Gate-Level Design: Test-Ready Compile



During test-ready compile, DFT Compiler

- Implements the scan equivalent cells by using the multiplexed flip-flop scan cell (FD1S)
- Ties the scan-enable pins (SE) to logic 0 so that the functional data input pins are active
- Ties the inactive scan input pins (SI) to logic 0

During scan routing, DFT Compiler replaces the temporary scan connections with the final scan connections.

A scan equivalent might not exist for the exact implementation defined, such as for the simple D flip-flop in the previous example. In that case, DFT Compiler might use a scan cell that can be logically modified to meet the required implementation. For example, if the target library contains a scan cell with asynchronous pins that can be tied off, test-ready compile automatically uses that scan cell.

Complex Compile Strategies

For larger designs or for designs with more aggressive timing goals, you might need to use more complex compile strategies, such as bottom-up compile, or you might need to use incremental compile a number of times. To include test-ready compile in your compile scripts, always use the `-scan` option of the `compile` command when compiling each current design, even if there are no sequential elements in the top level of the current design.

Example 4-4 illustrates this guideline. It shows you how to perform a bottom-up compile for the a design, TOP, that has no sequential elements at the top level but instantiates two sequential modules A and B. (For clarity, details on how you might constrain the designs are omitted.) Note that the `compile -scan` command is used at the top level even though there are no sequential elements at the top level of the design.

Example 4-4 Bottom-Up Compile Script

```
dc_shell> current_design A
dc_shell> compile -scan

dc_shell> current_design B
dc_shell> compile -scan

dc_shell> current_design TOP
dc_shell> compile -scan
```

Validating Your Netlist

Before you assemble the scan structures, you need to use the `link` and `check_design` commands to check the correctness of your design. You should fix any errors reported by these commands to guarantee the maximum fault coverage.

This section discusses the procedures for running the `link` and `check_design` commands.

Running the link Command

The `link` command attempts to find models for the references in your design. The command searches the design files and library files defined by the `link_library` variable. If the `link_library` variable does not specify a path for a design file or library file, the `link` command uses the directory names defined in the search path. Specifying the asterisk character (*) in the `link_library` variable forces the `link` command to search the designs in memory. See the man pages for more information about the `link` command.

If the `link` command reports unresolved references, such as missing designs or library cells in the netlist, resolve these references to provide a complete netlist to DFT Compiler. DFT Compiler operates on the complete netlist. DFT Compiler does not know the functional

behavior of a missing cell, so it cannot predict the output of that cell. As a result, output from the missing reference is not observable. Each missing reference results in a large number of untestable faults in the vicinity of that cell and lower total fault coverage.

If the unresolved reference involves a simple cell, you can often fix the problem by adding the cell to the library or by replacing the reference with a valid library cell.

Handling a compiled cell requires a more complex solution. If the compiled cell does not contain internal gates, such as a ROM or programmable logic array, you can compile a behavioral model of the cell into gates and then run DFT Compiler on the equivalent gates.

For more information about missing references or link errors, see the *Design Compiler User Guide*.

Running the check_design Command

The `check_design` command reports electrical design errors, such as port mismatches and shorted outputs that might lower fault coverage. For best fault coverage results, correct any design errors identified in your design. For more information about the `check_design` command, see the *Design Compiler User Guide*.

Performing Constraint-Optimized Scan Insertion

During the scan replacement process, constraint-optimized scan insertion does the following:

- Inserts the scan cells
- Optimizes the scan logic, based on the design constraints
- Fixes all compile-related design rule violations

Scan insertion is the process of performing scan replacement and scan assembly in a single step. You use the `insert_dft` command to invoke constraint-optimized scan insertion. However, you can also perform scan replacement and scan assembly in separate steps.

This section contains the following subsections related to constraint-optimized scan insertion:

Supported Scan States

Constraint-optimized scan insertion supports mixed scan states during scan insertion. Modules can have the following scan states:

- Nonscan

  The design contains nonscan sequential cells. Constraint-optimized scan insertion must scan-replace and route these cells.

- Unrouted scan

  The design contains unrouted scan cells. These unrouted scan cells can result from test-ready compile or from the scan replacement phase of constraint-optimized scan insertion. Constraint-optimized scan insertion must include these cells in the final scan architecture.

- Scan

  The design contains routed scan chains. Constraint-optimized scan insertion must include these chains in the final scan architecture.

Because the focus of this chapter is the scan replacement process, this discussion assumes that

- The input to constraint-optimized scan insertion is an optimized nonscan gate-level design
- The output from constraint-optimized scan insertion is an optimized design that contains unrouted scan cells. Note that constraint-optimized scan insertion performs scan replacement only.

Note:
When you do not route the scan chains, the optimization performed during constraint-optimized scan insertion accounts for the timing impact of the scan cell, but it does not take into account the additional loading due to the scan chain routing.

Locating Scan Equivalents

To perform scan replacement, constraint-optimized scan insertion first locates simple scan equivalents by using the identical-function method. If the `insert_test_map_effort_enabled` variable is true and if constraint-optimized scan insertion does not achieve scan replacement using this method, it then uses sequential-mapping-based scan replacement. See the scan insertion information in the *Design Compiler Optimization Reference Manual* for details about scan replacement methods.

Like test-ready compile, constraint-optimized scan insertion supports degeneration of scan cells to create the required scan equivalent functionality.

Replacing sequential cells with their scan equivalents modifies the design timing, as shown in Figure 4-9. DFT Compiler performs scan-specific optimizations to reduce the timing impact of scan replacement. By using focused techniques, constraint-optimized scan insertion optimizes the scan logic faster than the incremental compile process could.

Figure 4-9 Timing Changes Due to Scan Replacement

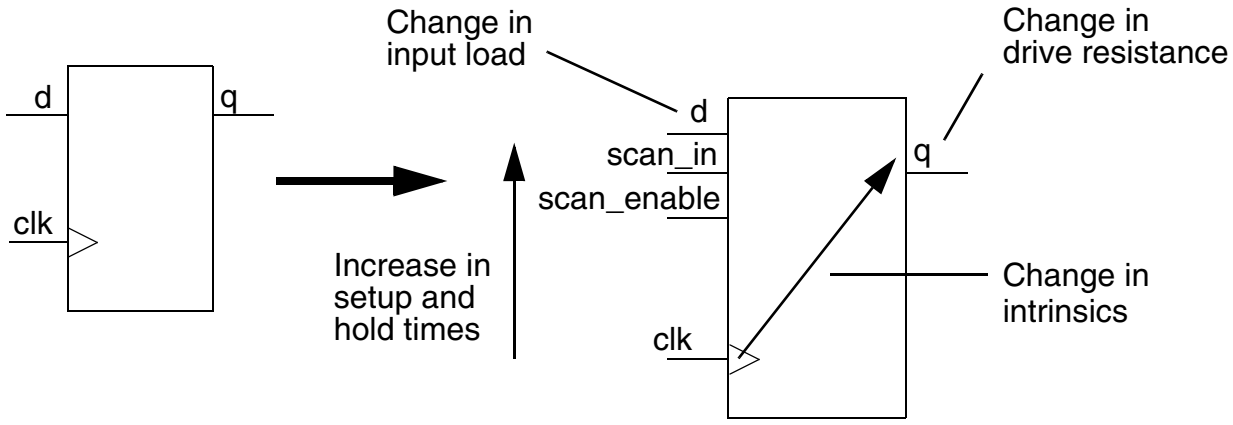
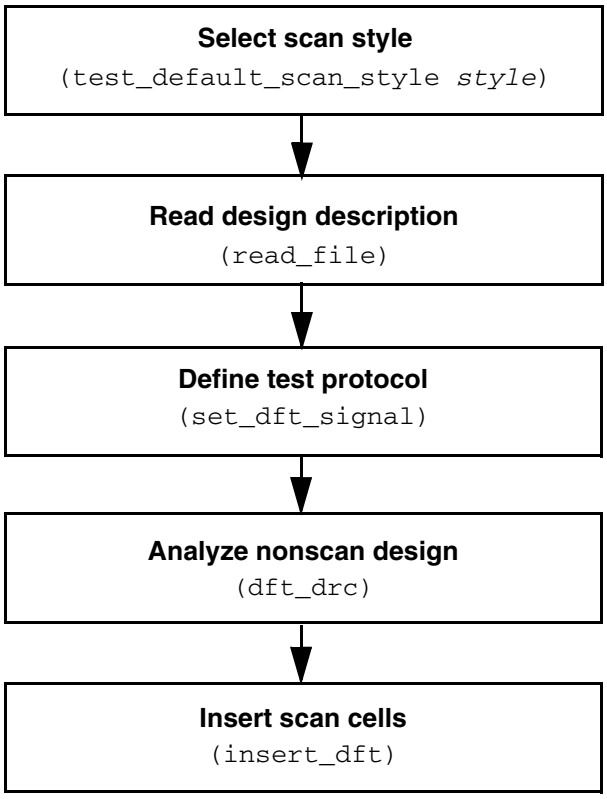


Figure 4-10 shows the flow used to insert scan cells with constraint-optimized scan insertion and the commands required to complete this flow.

Figure 4-10 Constraint-Optimized Scan Insertion Flow (Scan Replacement Only)



Preparing for Constraint-Optimized Scan Insertion

Before performing constraint-optimized scan insertion,

- Verify the timing characteristics of the design.

  Constraint-optimized scan insertion results in a violation-free design when the design has the following timing characteristics:

  - The nonscan design does not have constraint violations.
  - The timing budget is good.
  - You have properly applied realistic path constraints.
  - You have described the clock skew.

Note:
If your design enters constraint-optimized scan insertion with violations, long runtimes can occur.

- Select a scan style.
- Identify barriers to scan replacement.

Scan Insertion

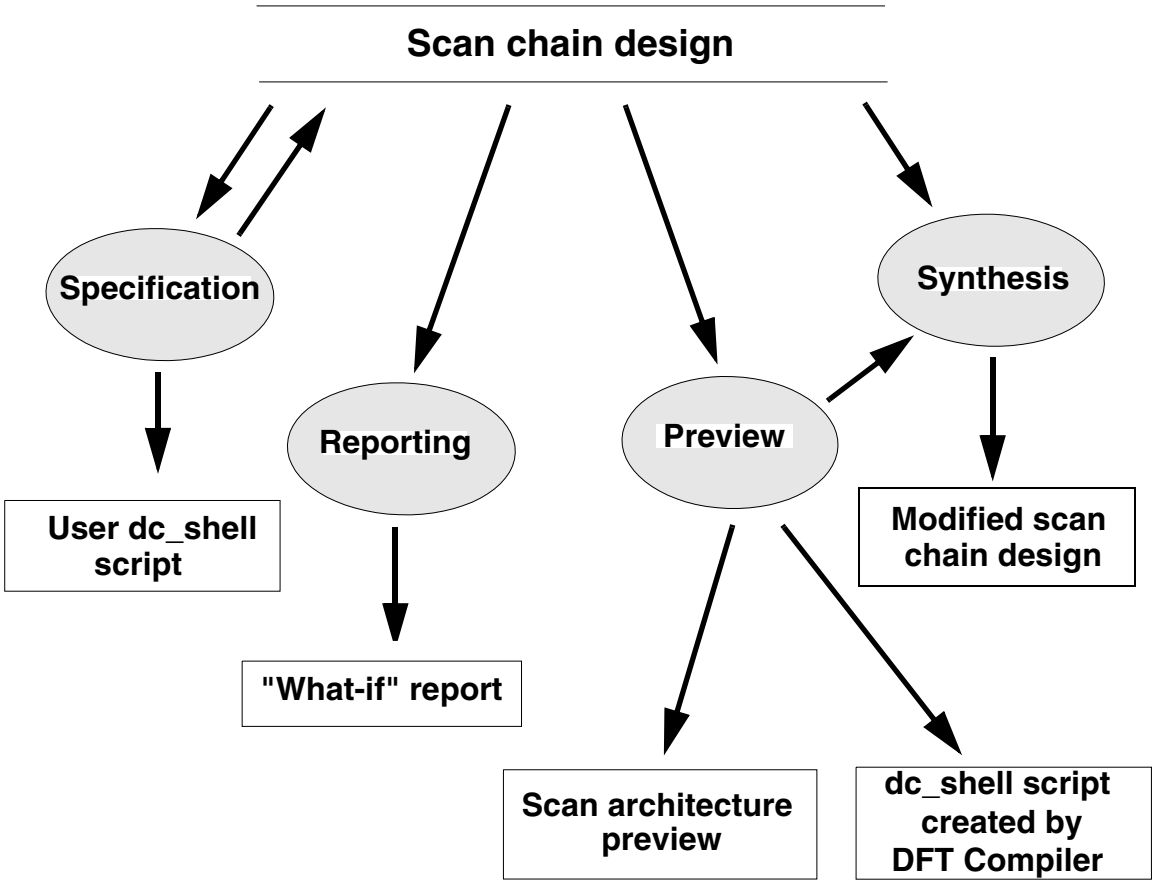
To alter a default scan design, you must specify changes to the scan configuration. You can make specifications at any point before scan synthesis. This section describes the specification commands you can use.

With the DFT Compiler scan insertion capability, you can

- Implement automatically balanced scan chains
- Specify complete scan chains
- Generate scan chains that enter and exit a design module multiple times
- Automatically fix certain scan rule violations
- Reuse existing modules that already contain scan chains
- Control the routing order of scan chains in a hierarchy
- Perform scan insertion from the top or the bottom of the design
- Implement automatically enabling or disabling logic for bidirectional ports and internal three-state logic
- Share functional ports as test data ports. DFT Compiler inserts enabling and disabling logic, as well as multiplexing logic, as necessary

You can design scan chains by using a specify-preview-synthesize process, which consists of multiple specification and preview iterations to define an acceptable scan design. After the scan design is acceptable, you can invoke the synthesis process to insert scan chains. Figure 4-11 shows this specify-preview-synthesize process.

Figure 4-11 The Scan Insertion Process



Example 4-5 is a basic scan insertion script.

Example 4-5 Basic Scan Insertion Script

```
current_design Top
set_dft_configuration -fix_set enable -fix_reset enable
set_scan_configuration -chain_count ...
create_test_protocol -infer_clock -infer_asynch
dft_drc
preview_dft
insert_dft
dft_drc
report_scan_path  -view existing_dft -chain all
report_constraints -all_violators
```

In this example, the following DFT configuration command enables AutoFix. For more information, see "Using AutoFix" on page 7-22.

```
set_dft_configuration -fix_reset enable -fix_set enable
```

The scan specification command is `set_scan_configuration -chain_count 1`. It specifies a single scan chain in the design.

The `preview_dft` command is the preview command. It builds the scan chain and produces a range of reports on the proposed scan architecture.

The `insert_dft` command is the synthesis command. It implements the proposed scan architecture.

The following sections describe these steps in the design process.

Specification Phase

During the specification phase, you use the scan and DFT specification commands to describe how the `insert_dft` command should configure the scan chains and the design. You can apply the commands interactively from the dc_shell or use them within design scripts. The specification commands annotate the database but do not otherwise change the design. They do not cause any logic to be created or any scan routing to be inserted.

The specification commands apply only to the current design and to lower-level subdesigns within the current design.

If you want to do hierarchical scan insertion by using a bottom-up approach, use the following general procedure:

1. Set the current design to a lower-level subdesign (`current_design` command).
2. Set the scan specifications for the subdesign (`set_scan_path`, `set_scan_element`, and so on).
3. Insert the scan cells and scan chains into the subdesign (`dft_drc`, `preview_dft`, and `insert_dft`).
4. Repeat steps 1, 2, and 3 for each subdesign, at each level of hierarchy, until you finish scan insertion for the whole design.

By default, the `insert_dft` command recognizes and keeps scan chains already inserted into subdesigns at lower levels. Thus, you can use different sets of scan specifications for different parts or levels of the design by using the `insert_dft` command separately on each part or level.

Note that each time you use the `current_design` command, any previous scan specifications no longer apply. This means that you need to enter new scan specifications for each newly selected design.

Scan Specification

Using the scan specification commands, you can specify as little or as much scan detail as you want. If you choose not to specify any scan detail, the `insert_dft` command implements the default full-scan methodology. If you choose to completely specify the scan design that you require, you explicitly assign every scan element to a specific position in a specific scan chain. You can also explicitly define the pins to use as scan control and data pins.

Alternatively, you can create a partial specification, where you define some elements but do not issue a complete specification. If you issue a partial specification, the `preview_dft` command creates a complete specification during the preview process.

The scan specification commands are

- `set_scan_configuration`
- `set_scan_path`
- `set_dft_signal`
- `set_scan_element`
- `reset_scan_configuration`

These commands are described in detail later in this section.

DFT Configuration

The DFT configuration commands are as follows:

- `reset_dft_configuration`
- `set_autofix_configuration`
- `set_autofix_element`
- `set_dft_configuration`
- `set_dft_signal`

Preview

The `preview_dft` command produces a scan chain design that satisfies scan specifications on the current design and displays the scan chain design for you to preview. If you do not like the proposed implementation, you can iteratively adjust the specification and rerun preview until you are satisfied with the proposed design.

The `preview_dft` command performs the following tasks:

- It checks the specification for consistency. For example, you cannot assign the same scan element to two different chains.
- It creates a complete specification if you have specified only a partial specification.
- It runs AutoFix.
- It produces a list of test points that are to be inserted into the design, based on currently enabled utilities.

When you use the `preview_dft` command, you can use the `-script` option to create a dc_shell script that completely specifies the proposed implementation. You can edit this script and use the edited script as an alternative means of iterating to a scan design that meets your requirements.

Limitation:

The `preview_dft` command does not annotate the design database with test information. If you want to annotate the database with the completed specification, use the `-script` option to create a specification dc_shell script and then run this script. The specification commands in this script add attributes to the database.

Synthesis

You invoke the synthesis process by using the `insert_dft` command, which implements the scan design determined by the preview process. If you issue this command without explicitly invoking the preview process, the `insert_dft` command transparently runs `preview_dft`.

Execute the `dft_drc` command at least one time before executing the `insert_dft` command. Executing the `dft_drc` command provides information on circuit testability before inserting scan into your design.

5

Pre-Scan Test Design Rule Checking

This chapter describes the process for preparing for and running test design rule checking (DRC), and checking violations before scan insertion.

This chapter includes the following sections:

Test DRC Basics

This section discusses the test DRC flow, the types of messages generated as a result of running the process, and the effects of violations on scan replacement.

Test DRC Flow

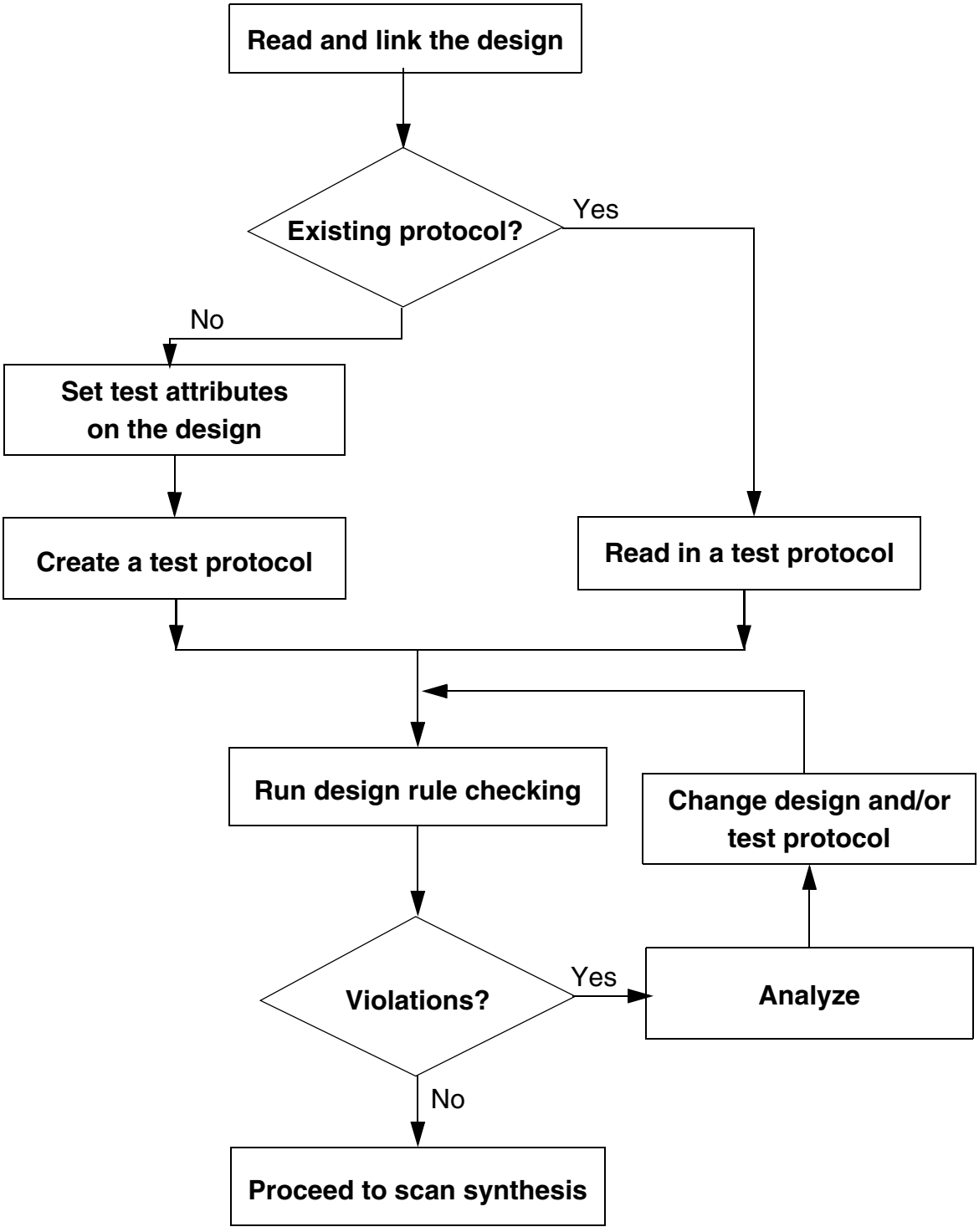
You use the `dft_drc` command to activate test design rule checking. However, before running this process, you must first create a test protocol that includes timing information (see "Creating the Test Protocol" on page 5-5 for information on creating test protocols).

After running the `dft_drc` command, violation messages are reported in three categories:

- Information messages – no action is required.
- Warning messages – you should analyze the violations; however, you can still run certain DFT commands.
- Error messages – these indicate serious errors that you need to correct before you can use the DFT commands.

Figure 5-1 illustrates a general test design rule checking flow.

Figure 5-1 Test DRC Flow



The following steps outline the test DRC process:

1. Read and link the design into DFT Compiler.

   For details, see the “Preparing Your Design” on page 5-4.

2. Determine if you have an existing test protocol for the design.

   If so, read the test protocol into DFT Compiler.

   If not, do the following:

   - Set the appropriate test attributes on the design.
   - Create the test protocol.

3. Run design rule checking.

   - If test DRC reports no violations, you can insert DFT structures into your design.
   - If test DRC reports violations, you can graphically analyze the violations by using Design Vision, as described in Chapter 3, “Running the Test DRC Debugger.”

     To fix the violations, either change your design, change your test protocol, or do both.

Preparing Your Design

To prepare your design-for-test DRC, follow these steps:

1. Set the `search_path` variable to point to directory paths that contain your design and library files.
2. Set the `link_library` variable to point to the logic library files referred to by your design.
3. Set the `target_library` variable to point to logic library files you want mapped to your design.
4. Use the `read_file` command to read your design into DFT Compiler.
5. Run the `link` command to link your design with your logic library.

See the Design Compiler documentation for more information on how to read and link your design.

Creating the Test Protocol

If you have an existing test protocol, read the test protocol into DFT Compiler by using the `read_test_protocol` command. If you do not have an existing test protocol, create it by following these steps:

1. Identify the test-related ports in your design. Such signals include
   - Clocks
   - Asynchronous sets and resets
   - Scan inputs
   - Scan outputs
   - Scan enables
2. Define DFT signals on these ports by using the `set_dft_signal` command.
3. Run the `create_test_protocol` command to create the test protocol for your design.

Assigning a Known Logic State

You can use the `set_test_assume` command to assign a known logic state to output pins of black-box sequential cells. The command syntax is

```
set_test_assume value pin_list
```

The *value* argument specifies the assumed value, 1 or 0, on this output pin or pins.

The *pin_list* argument specifies the names of output pins of unknown black-box cells, including nonscan sequential cells in full-scan designs. The hierarchical path to the pin should be specified for pins in subblocks of the current design.

The `dft_drc` command takes into account the conditions you define with the `set_test_assume` command.

Performing Test Design Rule Checking

After you create or read in a test protocol, perform test design rule checking by running the `dft_drc` command.

If you run the `insert_dft` command without first running the `dft_drc` command, the tool will implicitly run the `dft_drc` command before proceeding with DFT insertion.

In either case, the following message indicates that test DRC checking is performed:

```
Information: Starting test design rule checking. (TEST-222)
```

In the AutoFix flow, the first DRC analysis determines what test points are needed. The `insert_dft` command inserts the test point logic into the design database, then implicitly runs the `dft_drc` command again to determine the final DRC results. In this case, you will see an additional TEST-222 message issued during DFT insertion.

Reporting All Violating Instances

By default, the `dft_drc` command generates a message only for the first violating instance of a given violation type. To see all violations, use the `dft_drc -verbose` command.

Analyzing and Debugging Violations

You can graphically analyze the cause of a violation by using Design Vision, as described in Chapter 3, "Running the Test DRC Debugger."

After you have located the cause of the violation, you can either change the design, change the test protocol, or do both. Then rerun the steps outlined above to see if the violations have been fixed.

You can also use AutoFix to fix uncontrollable clocks and asynchronous sets and resets. For more information on AutoFix, see "Using AutoFix" on page 7-22.

Summary of Violations

At the completion of design rule checking, the `dft_drc` command displays a violation summary. Example 5-1 shows the format of the violation summary.

Example 5-1 Violation Summary

```
-------------------------------------------------------
  DRC Report
  Total violations: 6
-------------------------------------------------------
6 PRE-DFT VIOLATIONS
 3 Uncontrollable clock input of flip-flop violations (D1)
 3 DFF set/reset line not controlled violations (D3)

Warning: Violations occurred during test design rule
checking. (TEST-124)
-------------------------------------------------------
  Sequential Cell Report
  3 out of 5 sequential cells have violations
-------------------------------------------------------
SEQUENTIAL CELLS WITH VIOLATIONS
      *   3 cells have test design rule violations
SEQUENTIAL CELLS WITHOUT VIOLATIONS
      *   2 cells are valid scan cells
```

The total number of violations for the circuit appears in the header. If there are no violations in the circuit, the `dft_drc` command displays only the violation summary header. Within the summary, violations are organized by category. A violation category appears in the summary only if there are violations in that category. For each category, the `dft_drc` command displays the number (n) of violations, along with a short description of each violation and the corresponding error message number. Using the error message number, you can find the violation in the `dft_drc` run.

Unknown cell violations have message numbers in the TEST-451 to TEST-456 range. Unsupported cell violations have message numbers in the TEST-464 to TEST-469 range. The following is an excerpt from a violation summary for unknown cells:

```
-------------------------------------------------------
  DRC Report
  Total violations: 4
-------------------------------------------------------

3 MODELING VIOLATIONS
     1 Cell has unknown model violation (TEST-451)
```

Enhanced Reporting Capability

You can enable enhanced DRC reporting by setting the `test_disable_enhanced_dft_drc_reporting` variable to `false`. When enhanced reporting is enabled, the reporting and formatting of rule violations are changed to provide a better understanding of the respective rules.

Example 5-2 provides a typical enhanced DRC report:

Example 5-2 Enhanced DRC Report Example

```
In mode: all_dft...
  Pre-DFT DRC enabled
Information: Starting test design rule checking. (TEST-222)
  Loading test protocol
  ...basic checks...
  ...basic sequential cell checks...
      ...checking for scan equivalents...
  ...checking vector rules...
  ...checking pre-dft rules...
Simulation library files used for DRC
------------------------------------------------------------------
./core_slow_lvds_pads.v
./core_slow_special_cells.v
Cores and modes used for DRC in mode: all_dft
------------------------------------------------------------------
SUB_1: U1, U3, U4 mode: Internal_scan
SUB_2: U5, U6 mode: Internal_scan
Modeling and user constraints that will prevent scan insertion
------------------------------------------------------------------
Warning: Cell U34 will not be scanned due to set_scan_element command.
```

```
(TEST-202)
DRC violations which will prevent scan insertion
-------------------------------------------------------------
Warning: Cell U1 has constant 1 value. (TEST-505)
Warning: Reset input RN of DFF U53 was not controlled. (D3-1)
Information: There are 10 other cells with the same violation. (TEST-171)
DRC Violations which can affect ATPG coverage
-------------------------------------------------------------
Warning: Clock CCLK can capture new data on LS input of DFF U25. (D13-1)
         Source of violation: input CLK of DLAT U13/clk_gate_flop/latch.
Warning: CCLK clock path affected by new capture on LS input of DFF U17
(D15-1)
         Source of violation: input CLK of DLAT U18/clk_gate_flop/latch.
-------------------------------------------------------------
DRC Report
Total violations: 14
-------------------------------------------------------------
1 MODELING AND USER VIOLATIONS AFFECTING SCAN INSERTION
  1 cell with set_scan_element constraint (TEST-202)
11 DRC VIOLATIONS AFFECTING SCAN INSERTION
  1 Constant cell (TEST-505)
11 DFF reset line not controlled violations (D3)
2 DRC VIOLATIONS AFFECTING ATPG coverage
  1 Data path affected by clock captured by clock in level sensitive
      clock_port violations
(D13)
  1 Clock path affected by clock captured by clock in level sensitive
      clock_port violations
(D15)
-------------------------------------------------------------
Sequential Cell Report
                             Cells   Core    core_cells
-------------------------------------------------------------
Sequential elements detected:   50      5      50
Clock gating cells:             0
Synchronizing cells:            0
Non scan elements:              1       0      0
Excluded scan elements:         0       0      0
Violated scan elements:         11      1      10
Scan elements:                  39      4      40
-------------------------------------------------------------
```

Test Design Rule Checking Messages

When you invoke the `dft_drc` command, it generates messages to assist you in determining problems with your scan design. These messages fall into three categories:

- Information

  Information messages give you the status of the design rule checker or more detail about a particular rule violation.

- Warning

  A warning message indicates a testability problem that lowers the fault coverage of the design. Most of the violations reported by the `dft_drc` command are warning messages. The warnings allow you to evaluate the effect of violations and determine acceptable violations, based on your test requirements.

  Many warnings reported by `dft_drc` reduce fault coverage. Try to correct all violations, because a cell that violates a design rule, as well as the cells in its neighborhood, is not testable. A cell's neighborhood can be as large as its transitive fanin and its transitive fanout.

- Error

  An error message indicates a serious problem that prevents further processing of the design in DFT Compiler until you resolve the problem.

Understanding Test Design Rule Checking Messages

You can access online help for most warning messages generated by `dft_drc`. Online help provides information about the violation and information about how to proceed. Use the help command to access online help:

```
dc_shell> man message_id
```

Replace the *message_id* argument with the string shown in the parentheses that follow the warning text.

To keep a record of the information, warning, and error messages for your design, direct the output from the `dft_drc` command to a file with a command such as

```
dc_shell> dft_drc > my_drc.out
```

In this example, `my_drc.out` is the name of the output file.

Effects of Violations on Scan Replacement

When violations occur, the `dft_drc` command issues the following message:

```
Warning: Violations occurred during test design rule checking. (TEST-124)
```

For designs that are synthesized with the `compile -scan` command, the default behavior is that violations on scan-replaced cells cause the `insert_dft` command to unscan those cells. Sequential cells with violations are not included in a scan chain because they would probably prevent the scan chain from working as intended.

For designs that are not synthesized with the `compile -scan` command, violations on sequential cells cause the `insert_dft` command not to perform scan replacement for those cells.

For certain violation types, you can configure DFT insertion to include violating sequential cells in scan chains. See “Masking Capture DRC Violations” on page 5-37.

Viewing the Sequential Cell Summary

When the `dft_drc` command completes DRC, it provides a summary of the test status of the sequential cells in your design. Example 5-3 shows an example of the summary.

Example 5-3 Sequential Cell Summary

```
------------------------------------------------------------
  Sequential Cell Report

  2 out of 133721 sequential cells have violations
------------------------------------------------------------

SEQUENTIAL CELLS WITH VIOLATIONS
      *   2 cells have capture violations
SEQUENTIAL CELLS WITHOUT VIOLATIONS
      *133719 cells are valid scan cells
```

To get a complete listing of all the cells in each category, run the `dft_drc -verbose` command.

For information about classifying sequential cells, see the next section.

Classifying Sequential Cells

After the violation summary, the `dft_drc` command displays a summary of sequential cell information.

Example 5-4 shows the syntax of the sequential cell summary.

Example 5-4 Sequential Cell Summary

```
------------------------------------------------------------
  Sequential Cell Report

  2 out of 133721 sequential cells have violations
------------------------------------------------------------

SEQUENTIAL CELLS WITH VIOLATIONS
      *   2 cells have capture violations
SEQUENTIAL CELLS WITHOUT VIOLATIONS
      *133719 cells are valid scan cells
```

The number of sequential cells with violations appears in the header. This number is the sum of the cells with scan shift violations, capture violations, and constant values, along with

the cells that are black boxes. If a design has no sequential cells, only a header with the following message appears:

```
There are no sequential cells in this design
```

Within the summary, the sequential cells are divided into two groups: those with violations and those without. Only the categories of sequential cells that are found in the design are listed in the summary. In verbose mode, cell names are listed within each category. More information about the sequential cell categories is provided in the following sections.

Sequential Cells With Violations

This section of the sequential cell summary points to problematic sequential cells. The cells in this group have corresponding violations that can be found in the DRC output of the `dft_drc` command.

Cells With Scan Shift Violations

This category includes cells with scan-in and scan connectivity violations. Within this category, cells are listed by the type of scan shift violation.

- Not scan-controllable

  The `dft_drc` command cannot transport data from a scan-in port into the cell.

- Not scan-observable

  The `dft_drc` command cannot transport data from the cell to a scan-out port.

  Note:

  Cells in multibit components are homogeneous. If a cell in a multibit component has violations, all of the cells in that multibit component have violations.

After `dft_drc` has run, you can invoke the `report_scan_path -view existing_dft -chain all` command to observe the scan chains as extracted by the `dft_drc` command.

Black-Box Cells

Included in the black-box cells category are sequential cells that cannot be used for scan shift. Unknown cells and unsupported cells are classified as black boxes. These cells are not scan-replaced when you run the `insert_dft` command.

Constant Value Cells

The constant value category includes sequential cells that are constant during scan testing. These cells are assumed to hold constant values; they are not scan-replaced by `insert_dft`. For every constant value sequential cell, there is a corresponding TEST-504 or TEST-505 violation.

Sequential Cells Without Violations

The valid scan cells category displays the number of sequential cells that have no test design rule violations. ATPG tools can use these cells for scan shift and for measuring circuit response data. Valid scan cells can be scan-replaced by `insert_dft`.

Note:

Valid scan cells can have capture violations. Valid cells with capture violations only are scan-replaced.

The number of synchronization latches is listed in the last category.

Checking for Modeling Violations

If you instantiate a cell that DFT Compiler doesn't understand, you can get modeling violations. The `dft_drc` command performs modeling checks locally, one cell at a time.

Modeling violations are discussed in the following subsections:

- Black-Box Cells
- Unsupported Cells
- Generic Cells
- Scan Cell Equivalents
- Latches

Black-Box Cells

A cell whose output is considered unknown is classified as a *black-box* cell. These cells might lack a functional description in the logic library. Such cells are marked as black-box by the `report_lib` command. Also, the `dft_drc` command identifies black-box sequential cells.

The dft_drc command requires that you have a functional model in your library for each leaf cell in your design. If you use cells that do not have functional models, the dft_drc command displays the following warning:

```
Warning: Cell %s (%s) is unknown (black box) because functionality for
output pin %s is bad or incomplete. (TEST-451)
```

You do not need to correct black-box violations for memory macro cells; they are always modeled as black-box cells by the dft_drc command. In TetraMAX, you can use memory models so that sequential ATPG can obtain fault coverage around the memories.

See the Library Compiler reference manuals for more information on modeling the behavior of cells.

Correcting Black-Box Cells

DFT Compiler models a cell as a black box in these cases:

- The link command cannot resolve the cell reference by using the technology libraries or designs in the search_path (unresolved reference).
- The logic library model for the cell reference does not contain a functional description (black-box library cell).

In the following cases, a black-box cell can have a severe impact on fault coverage:

- The black-box cells are pad cells.

  The dft_drc command completely fails and prevents insert_dft from working. This occurs during scan stitching at the top level.

- A black-box cell controls the enable signal of an internal three-state driver or a bidirectional signal.

  The insert_dft command inserts three-state and bidirectional control logic if the existing control logic is a black box, even if doing so is unnecessary.

DFT Compiler generates this warning message when it models a cell as a black box:

```
Warning: Cell %s (%s) is unknown (black box) because functionality for
output pin %s is bad or incomplete. (TEST-451)
```

The method for correcting the violation depends on the source of the violation and the complexity of the cell.

Note:
Use the link command to correct unresolved references.

Black-Box Library Cell

If no functional description of the cell exists in the logic library, you need to obtain either a functional model or a structural model of the cell.

If the cell can be functionally modeled by the Library Compiler tool, obtain an updated logic library that includes a functional model of the cell.

If you have a simulation model for the black box, declare it by using the following variable:

```
dc_shell> set_app_var test_simulation_library simulation_library_path
```

Note the following license-related requirements:

- If you have a Library Compiler license and the library source code, add the functional description to the library cell model.

  See the Library Compiler documentation for information about cell modeling.

- If you do not have a Library Compiler license or library source code, ask your semiconductor vendor for a library that contains a functional model of the cell.

If the Liberty syntax does not support functional modeling of the cell, create a structural model for the cell and link the design to this structural model instead of the library cell model.

Note:

You should only use the `test_simulation_library` variable to replace leaf cells that do not have functional models. Do not use the variable to replace any arbitrary module in the design. If you want to replace the entire design module that consists of leaf cells, you should use the `remove_design` command to remove the module and then read the Verilog netlist description of that module into memory.

Unsupported Cells

Cells can have a functional description and still not be supported by the `dft_drc` command. Using state table models, library developers can describe cells that violate the current assumptions for test rule checking. The `dft_drc` command detects those cells and flags them as black boxes.

DFT Compiler supports single-bit cells or multibit cells that have identical functionality on each pin; these cells have the following characteristics:

- The functional view, which Design Compiler synthesis understands and manipulates, is either a flip-flop, a latch, or a master-slave cell with `clocked_on` and `clocked_on_also` attributes.
- The test view, used for scan shifting, is either a flip-flop or a master-slave cell.
- The functional view and the test view each have a single clock per internal state.

The multibit library cell interfaces must be either fully parallel or fully global. For cells that do not meet these criteria, DFT Compiler uses single-bit cells.

For example, if you want to infer a 4-bit banked flip-flop with an asynchronous clear signal, the clear signal must be either different for each bit or shared among all 4 bits. If the first and second bits share one asynchronous reset but the third and fourth bits share another reset, DFT Compiler does not infer a multibit flip-flop. Instead, DFT Compiler uses 4 single-bit flip-flops. For more information about multibit cells and multibit components, see the *Design Compiler Optimization Reference Manual*.

DFT Compiler does not support registers or duplicate sequential logic within a cell. The nonscan equivalent of a scan cell must have only one state. A scan cell can have multiple states in shift mode.

If the `dft_drc` command detects such a cell, it issues the following warning:

```
Cell %s (%s) is not supported because it has too many
states (%d states). This cell is being black-boxed.(TEST-462)
```

When the `dft_drc` command recognizes part of a cell as a master-slave latch pair but finds extra states, it issues one of the following warnings, depending on the situation:

```
Master-slave cell %s (%s) is not supported because the state
pin %s is neither a master nor a slave. This cell is being
black-boxed,(TEST-463)

Master-slave cell %s (%s) is not supported because there
are two or more master states. This cell is being
black-boxed,(TEST-464)

Master-slave cell %s (%s) is not supported because there
are two or more slave states. This cell is being
black-boxed,(TEST-465)
```

If the `dft_drc` command detects a state with no clocks or with multiple clocks, it issues one of the following warnings:

```
Cell %s (%s) is not supported because the state pin %s has no
clocks. This cell is being black-boxed,
(TEST-466)

Cell %s (%s) is not supported because the state pin %s is
multi-port. This cell is being black-boxed.(TEST-467)
```

In addition, the `dft_drc` command detects and rejects sequential cells with three-state outputs and issues the following warning:

```
Cell %s (%s) is not supported because it is a sequential
cell with three-state outputs. This cell is being
black-boxed,(TEST-468)
```

Black-box cells have an adverse effect on fault coverage. To avoid this effect, you must replace unsupported cells with cells that DFT Compiler can support.

Note:

Unsupported cells can originate only from explicit instantiation. They are not used by the Design Compiler or DFT Compiler tools. For more information on modeling sequential cells, see *Library Compiler Methodology and Function Modeling User Guide* and *Library Compiler Timing, Signal Integrity, and Power Modeling User Guide*.

Generic Cells

Your design should be a mapped netlist. In the RTL stage, the `dft_drc` command will map your design into an internal representation.

Some generic cells, such as unimplemented DesignWare parts and operators, have implicit functional descriptions. The `dft_drc` command treats them as black-box cells and displays the following warning message:

```
Warning: Cell %s (%s) is unknown (black box) because
functionality for output pin %s is bad or incomplete. (TEST-451)
```

If you instantiate generic cells after running `compile -scan`, you must recompile your design.

Scan Cell Equivalents

When checking test design rules in a design without scan chains, the `dft_drc` command verifies that each sequential element has not been explicitly marked by using the `set_scan_element false` command. If a scan cell equivalent does not exist or it has the `dont_use` attribute applied, the `dft_drc` command issues the following warning message:

```
Warning: No scan equivalent exists for cell %s (%s).
(TEST-120)
```

Note:

Use the `set_scan_element false` command to prevent scan replacement.

The cells in violation are marked as nonscan. In the full-scan methodology, these cells are black boxes. If these cells are not valid nonscan, they are in violation and are black boxes. You can suppress the TEST-120 warning with the `set_scan_element` command. For example, to ensure that a nonscan latch cell is not made scannable, enter the command

```
dc_shell> set_scan_element false latch_name
```

If you use the `set_scan_element` command, the `dft_drc` command issues the following information message:

```
Information: Cell %s (%s) will not be scanned due to a
set_scan_element command. (TEST-202)
```

If the dft_drc command cannot find scan cell equivalents in the target library, the probable reason is that the target library does not contain test cells. In such cases, the dft_drc command issues the following warning:

```
Warning: Target library for design contains no scan-cell
models. (TEST-224)
```

Scan Cell Equivalents and the dont_touch Attribute

If you set the dont_touch attribute on a nonscan cell before scan cell replacement, that cell is not modified or scan-replaced when you optimize the design. In this case, the dft_drc command produces the following warning:

```
Warning: Cell %s (%s) can't be made scannable because it is
dont_touched. (TEST-121)
```

If you apply the dont_touch attribute to scan-replaced cell, the cell can still be added to a scan chain.

Note:

Use the dont_touch attribute carefully, because it can increase the number of nonscan cells, and nonscan cells lower fault coverage.

Use the set_scan_element false command if you do not want to make a sequential cell scannable but you do want to be able to modify the cell during optimization.

Latches

DFT Compiler replaces latches with scannable latches whenever possible. If the dft_drc command cannot find scan cell equivalents for the latches, it marks the latches as nonscan and issues the TEST-120 warning as previously explained.

Nonscan Latches

DFT Compiler models nonscan latches in two ways:

- As black boxes
- As synchronization elements

If you do not scan replace your latches, you can ignore “no-scan equivalent” messages for latches.

A nonscan latch is treated by default as a black box. However, if the latch satisfies the requirements for a synchronization element, the dft_drc command treats the latch as a synchronization element.

Note:
The `dft_drc` command allows synchronous elements to be on the scan chain.

Setting Test Timing Variables

This section discusses the process for setting test timing variables for your design. The timing variables are used by the test protocol for design rule checking and for DFT preview and insertion.

This section has the following subsections:

- Protocols for Common Design Timing Requirements
- Setting Timing Variables

Protocols for Common Design Timing Requirements

Before creating a test protocol and checking test design rules, you need to identify the timing information for your design. You do this by setting a number of timing variables and, if necessary, by defining test clock requirements. Timing variables are discussed in detail in "Setting Timing Variables" on page 5-19.

Defining test clock requirements is discussed in detail in Chapter 6, "Architecting Your Test Design".

If you intend to use postclock strobing, you need to change the default variable values. If your design's timing variable values are the same as the variables' defaults, you do not need to make any changes.

Strobe-Before-Clock Protocol

The timing requirements for a strobe-before-clock protocol are shown in the following example. Check with your semiconductor vendor for specific timing information.

```
test_default_period :        100 ;
test_default_delay :           0 ;
test_default_bidir_delay :     0 ;
test_default_strobe :         40 ;
test_default_strobe_width :    0 ;
```

Strobe-After-Clock Protocol

To use a strobe-after-clock protocol, set the timing values described in the following example. Although the strobe-after-clock protocol works with TetraMAX ATPG, the strobe-before-clock protocol is more efficient. Use the following timing values:

```
test_default_period :       100 ;
test_default_delay :          5 ;
test_default_bidir_delay :   55 ;
test_default_strobe :        95 ;
test_default_strobe_width :   0 ;
```

If you intend to use a strobe-after-clock protocol with TetraMAX ATPG, use the timing values shown in the following example.

```
test_default_period :      100 ;
test_default_delay :         0 ;
test_default_bidir_delay :   0 ;
test_default_strobe :       90 ;
test_default_strobe_width :  0 ;
```

Setting Timing Variables

Before you run the `create_test_protocol` command, you need to define timing variables. The command uses the following test variables to determine the values in the test protocol timing variables:

```
test_default_period
test_default_delay
test_default_bidir_delay
test_default_strobe
test_default_strobe_width
```

The requirements from your semiconductor vendor, together with the basic scan test requirements, drive the specification of test timing parameters. If you intend to use postclock strobing, you need to change the default variable values. You can do this every time you create a new design, or you can add these variable values to your local .synopsys_dc.setup file.

test_default_period Variable

The `test_default_period` variable defines the default, in ns, for the period in the test protocol. The period value must be a positive real number.

By default, DFT Compiler uses a 100 ns test period. If your semiconductor vendor uses a different test period, specify the required test period by using the `test_default_period` variable.

The syntax for setting the variable is

```
set_app_var test_default_period period
```

For example,

```
dc_shell> set_app_var test_default_period 100
```

In the .synopsys_dc.setup file, the `test_default_period` variable is set to 100 ns.

test_default_delay Variable

The `test_default_delay` variable defines the default, in ns, for the input delay in the inferred test protocol. The delay value must be a nonnegative real number less than the strobe value. See the default timing in Figure 5-2 on page 5-24.

By default, DFT Compiler applies data to all nonclock input ports 0 ns after the start of the cycle. If your semiconductor vendor requires different input timing, specify the required input delay by using the `test_default_delay` variable.

The syntax for setting the variable is

```
set_app_var test_default_delay delay
```

For example,

```
dc_shell> set_app_var test_default_delay 5
```

In the .synopsys_dc.setup file, `test_default_delay` is 0 ns.

test_default_bidir_delay Variable

The `test_default_bidir_delay` variable defines the default, in ns, for the bidirectional delay in the inferred test protocol. The *bidir_delay* must be a positive real number less than the strobe value and can be less than, greater than, or equal to the delay value. See the default timing in Figure 5-2 on page 5-24.

By default, DFT Compiler applies data to all bidirectional ports in input mode 0 ns after the start of the parallel measure cycle. In any cycle where a bidirectional port changes from input mode to output mode, DFT Compiler releases data from the bidirectional port 0 ns after the start of the cycle. If your semiconductor vendor requires different bidirectional timing, specify the required bidirectional delay by using the `test_default_bidir_delay` variable.

The risks associated with incorrect specification of the bidirectional delay time include

- Test design rule violations
- Bus contention
- Simulation mismatches

Minimize these risks by carefully specifying the bidirectional delay time.

DFT Compiler uses the bidirectional delay time as

- The data application time for bidirectional ports in input mode during the parallel measure cycle and during scan-in for bidirectional ports used as scan inputs or scan-enable signals
- The data release time for bidirectional ports in input mode during cycles in which the bidirectional port changes from input mode to output mode

DFT Compiler performs relative timing checks during test design rule checking. The following requirements must be met:

- The bidirectional delay time must be less than the strobe time.

  If you change the strobe time from the default, confirm that the bidirectional delay value meets this requirement.
- If the bidirectional port drives sequential logic, the bidirectional delay time must be equal to or greater than the active edge of the clock.

The syntax for setting the variable is

```
set_app_var test_default_bidir_delay bidir_delay
```

For example,

```
dc_shell> set_app_var test_default_bidir_delay 40
```

In the .synopsys_dc.setup file, `test_default_bidir_delay` is 0 ns.

test_default_strobe Variable

The `test_default_strobe` variable defines the default, in ns, for the strobe in the inferred test protocol. The strobe value must be a positive real number less than the period value and greater than the `test_default_delay` value (see the default timing in Figure 5-2 on page 5-24).

By default, DFT Compiler compares data at all output ports 40 ns after the start of the cycle. If your semiconductor vendor requires different strobe timing, specify the strobe time by using the `test_default_strobe` variable.

The syntax for setting the variable is

```
set_app_var test_default_strobe strobe
```

For example:

```
dc_shell> set_app_var test_default_strobe 100
```

In the .synopsys_dc.setup file, `test_default_strobe` is 40 ns.

test_default_strobe_width Variable

The `test_default_strobe_width` variable defines the default, in ns, for the strobe width in the inferred test protocol. The strobe width value must be a positive real number. The strobe value plus the strobe width value must be less than or equal to the period value. See the default timing in Figure 5-2 on page 5-24.

Clocking requirements specified by semiconductor vendors include

- Clock waveform timing
- Maximum number of unique clock waveforms
- Minimum delay between different clock waveforms, which allows for clock skew on the tester

DFT Compiler provides the capability to specify clock waveform timing but does not place any restrictions on the number of unique waveforms that can be defined or the minimum time between clock waveforms. By determining what restrictions the semiconductor vendor places on these timing parameters, you can define clock waveforms that meet the restrictions.

When DFT Compiler infers clock ports during `dft_drc`, the clock type determines the default timing for each clock edge. Table 5-1 provides the default clock timing for each clock type.

Table 5-1 Default Clock Timing for Each Clock Type

Clock type	First edge	Second edge
Edge-triggered or D-latch enable	45	55
Master clock	30	40
Slave clock	60	70
Edge-triggered	45	60
Master clock1	50	60
Slave clock	40	70

DFT Compiler determines the polarity of the first edge (rise or fall) so that the first clock edge triggers the majority of cells on a clock. The timing arcs in the logic library specify each cell's trigger polarity. The polarity of the second edge is opposite the polarity of the first edge, that is, if the first edge is rising (falling), the second edge is falling (rising).

Use the `set_dft_signal` command to specify clock waveforms if your semiconductor vendor's requirements differ from the default timing.

The `set_dft_signal` command has a time period associated with it. That period has to be identical to the `test_default_period` value. If you change the value of one, you must check the value of the other.

The syntax for setting the variable is

`set_app_var test_default_strobe_width` *`strobe_width`*

If you need a window strobe in your STIL procedure file (SPF) or STIL patterns, set the default of test_default_strobe_width to 1 ns, as shown in the following command:

```
dc_shell> set_app_var test_default_strobe_width 1
```

In the .synopsys_dc.setup file, `test_default_strobe_width` is 0 ns.

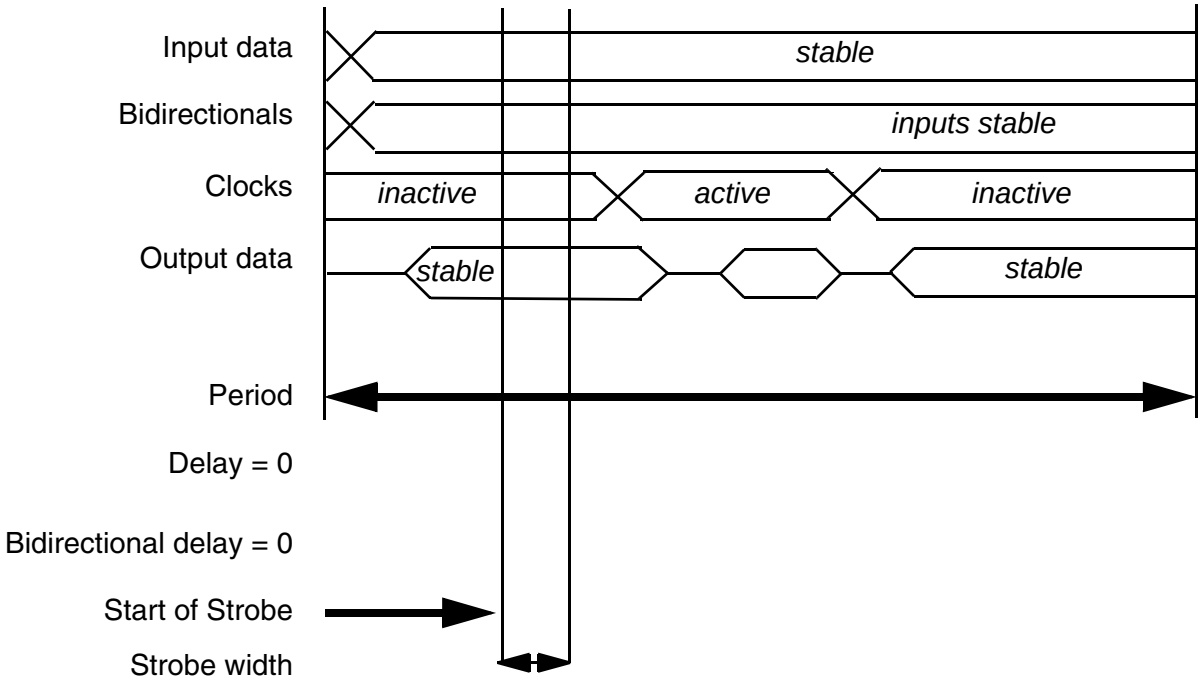
Note:

When `test_default_strobe_width` is 0 ns, the strobe width is equal to one of two values: the difference between the strobe time and the end of the period, or the difference between the strobe time and the first input event after the strobe occurs, whichever occurs first.

The Effect of Timing Variables on Vector Formatting

Figure 5-2 shows a timing diagram for a strobe-before-clock scheme.

Figure 5-2 Effect of Timing Variables on Vector Formatting



Creating Test Protocols

Test protocols are an intrinsic part of your design-for-test process and must be created before you run the `dft_drc` command. This section has the following subsections related to creating test protocols:

- Design Characteristics for Test Protocols
- STIL Test Protocol File Syntax
- Defining an Initialization Protocol
- Scan Shift and Parallel Measure Cycles
- Examining a Test Protocol File

Design Characteristics for Test Protocols

A test protocol is based on certain characteristics of a design. The following subsections discuss how a protocol is affected by these design characteristics:

- Scan Style
- New DFT Signals
- Existing Clock Ports
- Existing Asynchronous Control Ports
- Bidirectional Ports

Scan Style

Each scan style has a unique method of performing scan shift, which must be reflected in the test protocol. For more information on how scan style influences the scan shift process, see “Scan Shift and Parallel Measure Cycles” on page 5-32.

New DFT Signals

The DFT signal attributes are set automatically for each new test port created by the `insert_dft` command. The DFT signal attributes are preserved if you save the design in Synopsys .ddc format. If you have an existing scan design that is not saved in Synopsys .ddc format, you must re-identify each test port with the appropriate `set_dft_signal` command.

Existing Clock Ports

You specify existing clock ports (and their timing attributes) by using the `set_dft_signal` command.

Tracing back from the clock pins on all sequential elements to the ports driving these pins interesting Clock ports can also be inferred by. The default timing for the clock signals is determined by the `set_scan_configuration -style` command.

Existing Asynchronous Control Ports

You specify existing asynchronous control ports by using the following command:

```
dc_shell> set_dft_signal -view existing_dft \
              -type Reset -port RSTN -active_state 0
```

Asynchronous control ports can also be inferred by tracing back from the asynchronous pins on all sequential elements to the ports controlling these pins. Asynchronous control ports must be identified because all asynchronous inputs must be disabled during scan shift to allow predictable loading and unloading of the scan data.

Bidirectional Ports

In all cycles except parallel measure and capture, all nondegenerated bidirectional ports are assumed to be in output (driving) mode and are appropriately masked. During parallel measure and capture cycles, ATPG data controls the bidirectional ports as normal input or output ports but the `test_default_bidir_delay` variable controls the timing.

STIL Test Protocol File Syntax

DFT Compiler reads test protocols written in the Standard Test Interface Language (STIL). The STIL format is also used by TetraMAX ATPG.

Although the STIL procedure file syntax is the same as that used by TetraMAX ATPG, DFT Compiler cannot read some of the STIL elements that are available in TetraMAX ATPG.

The following STIL elements are *not* available in DFT Compiler:

- Post `load_unload` vectors
- Multiple scan groups in the `load_unload` procedure
- Multiple waveforms in the timing section

For general information on STIL standards (IEEE Std. 1450.0-1999), see the STIL home page at

http://grouper.ieee.org/groups/1450/index.html

Note that both the DFT Compiler and TetraMAX tools use the IEEE P1450.1 extensions to STIL. For details, see Appendix E, “STIL IEEE P1450.1 Extensions,” in the *TetraMAX Online Help*.

Defining the test_setup Macro

The `test_setup` macro is optional. It defines any initialization sequences that the design might need for test mode or to ensure that the device is in a known state. A `test_setup` macro example is shown in Example 5-5.

Example 5-5 Defining the test_setup Macro in the SPF

```
STIL;
     ScanStructures {
          ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
```

```
          ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
          ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
          ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
     }
Procedures {
     "load_unload" {
          V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
          Shift {
               V { _si=####; _so=####; CLOCK=P;}
          }
     }
}
MacroDefs {
     test_setup {
          V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1; }
          V {PLL_RESET = 0; }
          V {PLL_RESET = 1; }
     }
}
```

If you need to initialize a port to X in the `test_setup` macro, the STIL assignment character for this is N. An X indicates that outputs are measured and the result is masked.

Defining Basic Signal Timing

If you do not define the signal timing explicitly, DFT Compiler uses its own defaults.

Example 5-6 contains many additions to define signal timing. Line numbers have been added for reference. Note:

- Lines 6–9. Defines some additional signal groups so that timing for all inputs or outputs can be defined in just a few lines, instead of explicitly naming each port and its timing.
- Lines 12–28. Defines a waveform table with a period of 1,000 ns that defines the timing to be used during nonshift cycles.
- Line 37. Adds the W statement to ensure that BROADSIDE_TIMING is used for V cycles during the `load_unload` procedure.
- Line 48. Causes the `test_setup` macro to use BROADSIDE_TIMING.

Example 5-6 Defining Timing in the SPF

```
1. STIL;
2. UserKeywords PinConstraints;
3. PinConstraints { "TEST_MODE" 1; "PLL_TEST_MODE" 1; }
4. SignalGroups {
5.   bidi_ports = '"D[0]" + "D[1]" + "D[2]" + "D[3]" + "D[4]"
       + "D[5]" + "D[6]" + "D[7]" + "D[8]" + "D[9]" + "D[10]" +
       "D[11]" + "D[12]" + "D[13]" + "D[14]" + "D[15]" `;
6.   input_grp1 = 'SCAN_ENABLE + BIDI_DISABLE + TEST_MODE +
       PLL_TEST_MODE' ;
```

```
7.   input_grp2 = 'SDI1 + SDI2 + DIN + "IRQ[4]"' ;
8.   in_ports = 'input_grp1 + input_grp2';
9.   out_ports = 'SDO2 + D1 + YABX + XYZ';
10. }
11. Timing {
12.   WaveformTable "BROADSIDE_TIMING" {
13.   Period '1000ns';
14.   Waveforms {
15.     CLOCK { P { '0ns' D; '500ns' U; '600ns' D; } }
         //  clock
16.     CLOCK { 01Z { '0ns' D/U/Z; } }
17.     RESETB { P { '0ns' U; '400ns' D; '800ns' U; } }
         // async reset
18.     RESETB { 01Z { '0ns' D/U/Z; } }
19.     input_grp1 { 01Z { '0ns' D/U/Z; } }
20.     input_grp2 { 01Z { '10ns' D/U/Z; } }
    // outputs are to be measured at t=350
21.     out_ports { HLTX { '0ns' X; '350ns' H/L/T/X; } }
    // bidirectional ports as inputs are forced at t=20
22.     bidi_ports { 01Z { '0ns' Z; '20ns' D/U/Z; } }
23.     // bidirectional ports as outputs are measured at
         t=350
24.     bidi_ports { X { '0ns' X; } }
25.     bidi_ports { HLT { '0ns' X; '350ns' H/L/T; } }
26.     }
27.   } // end BROADSIDE_TIMING
28. }
29. ScanStructures {
30.   ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
31.   ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
32.   ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
33.   ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
34. } // end scan structures
35. Procedures {
36. "load_unload" {
37.   W "BROADSIDE_TIMING" ;
38.  V {CLOCK=0; RESETB=1; SCAN_ENABLE=1; BIDI_DISABLE=1;
    bidi_ports = \r16 Z;}
39.   V {}
40.   V { bidi_ports = \r4 1010 ; }
41.   Shift {
42.   V { _si=####; _so=####; CLOCK=P;}
43.   }
44. } // end load_unload
45. } //end procedures
46. MacroDef {
47.   "test_setup" {
48.     W "BROADSIDE_TIMING" ;
49.     V {TEST_MODE = 1; PLL_TEST_MODE = 1; PLL_RESET = 1;
50.     BIDI_DISABLE = 1; bidi_ports = ZZZZZZZZZZZZZZZZ; }
51.     V {PLL_RESET = 0; }
52.     V {PLL_RESET = 1; }
```

```
53.    } // end test_setup
54. } //end procedures
```

Defining the load_unload Procedure

The `load_unload` procedure contains information about placing the scan chains into a shiftable state and shifting 1 bit through them. DFT Compiler creates this procedure if you define the scan-enable information before you write out the STIL file. Example 5-7 shows the syntax used to define scan chains.

Example 5-7 Defining Scan Chain Loading and Unloading in the SPF

```
STIL;
ScanStructures {
  ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
  ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
  ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
  ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
  "load_unload" {
    V { CLOCK=0; RESETB=1; SCAN_ENABLE=1; }
  }
}
```

Defining the Shift Procedure

The shift procedure specifies how to shift the scan chains within the definition of the `load_unload` procedure. The bold text shown in Example 5-8 defines the shift procedure.

Example 5-8 Defining the Scan Chain Shift Procedure in the SPF

```
STIL;
ScanStructures {
  ScanChain "c1" { ScanIn SDI2; ScanOut SDO2; }
  ScanChain "c2" { ScanIn SDI1; ScanOut D1; }
  ScanChain "c3" { ScanIn DIN; ScanOut YABX; }
  ScanChain "c4" { ScanIn "IRQ[4]"; ScanOut XYZ; }
}
Procedures {
  "load_unload" {
    V { CLOCK=0; RESETB=1; SCAN_ENABLE = 1; }
    Shift {
    V { _si=####; _so=####; CLOCK=P;}
    }
  }
}
```

Defining an Initialization Protocol

If your design requires an initialization sequence to configure it for scan testing, you can provide the initialization vectors through an initialization protocol. With an initialization protocol, you provide specific vectors to initialize the design while letting the `create_test_protocol` command complete the scan shifting steps of the protocol.

Use the following process to generate an initialization protocol:

1. Analyze the design to determine its test configuration requirements.
   - Determine the initial state required and the initialization sequence necessary to achieve this state.
   - Determine the test configuration required to maintain this initial condition throughout scan testing.
2. Generate a default test protocol file.
   - Specify timing parameters if you require values other than the default.
   - Specify test configuration requirements determined in the analysis step by using the `set_dft_signal` command.
   - Run `create_test_protocol` to generate the default protocol.
   - Use the `write_test_protocol` command to write the ASCII protocol file.
3. Create the initialization protocol file.

   Modify the initialization sequence in the `test_setup` section of the test protocol file.
4. Read in the initialization protocol.

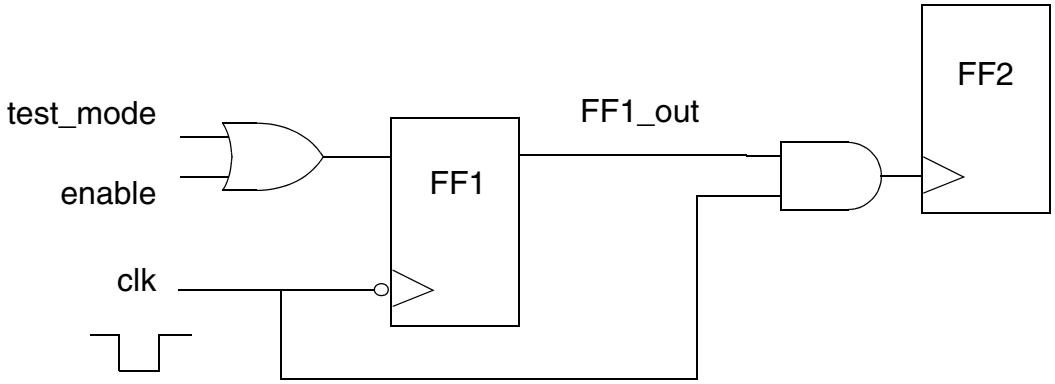
   First remove the existing protocol by using the `remove_test_protocol` command, and then read the initialization protocol using the `following` command.

   ```
   read_test_protocol -section test_setup
   ```
5. Rerun `create_test_protocol` to complete the test protocol.

   Run test DRC.

See the design in Figure 5-3 for an illustration of the use of an initialization protocol.

Figure 5-3 Design That Needs an Initialization Protocol



In this design, the clock signal, clk, is active low. For the clock signal to reach FF2, you need to initialize it by pulsing clk one time so that the enable signal FF1_out is asserted. Because the `create_test_protocol` command has no knowledge of this requirement, you need to modify the generated protocol to include this special initialization sequence.

The initialization sequence generated by the `create_test_protocol` command looks like the following:

```
"test_setup" {
   W "_default_WFT_";
   V {  "CLK"=1; }
   V {  "CLK"=1; "test_mode"=1; }
 }
```

If this initialization sequence has not been modified, test DRC gives the following violations:

```
4 PRE-DFT VIOLATIONS
     3 Uncontrollable clock input of flip-flop violations (D1)
     1 Clock not able to capture violation (D8)
```

The initialization sequence that is necessary to initialize the circuit is as follows:

```
"test_setup" {
   W "_default_WFT_";
   V {  "CLK"=1; }
   V {  "CLK"=1; "test_mode"=1; }
   V {  "CLK"=P; "test_mode"=1; }
   V {  "CLK"=1; "test_mode"=1; }
```

Test DRC requires that all clock signals are in their inactive state at the end of the initialization sequence. When this initialization sequence is applied, test DRC indicates that there are no test design rule violations.

However, after the `insert_dft` command completes, this initialization sequence is lost. You must reapply the same initialization sequence to ensure that post-scan insertion test DRC reports no violations.

Table 5-2 shows the flows you should use with various types of test protocols.

Table 5-2 Initialization Protocol Flows

If you have	Use this flow

No test protocol	`set_dft_signal...`
	`create_test_protocol`
	`dft_drc`

Only the `test_setup` section in the protocol	`set_dft_signal...`
	`read_test_protocol -section test_setup`
	`create_test_protocol`
	`dft_drc`

Full protocol	`read_test_protocol` (no `-section test_setup`) `set_dft_signal` (for clocks and asynchronous signals) `dft_drc`

Scan Shift and Parallel Measure Cycles

The standard strobe-before-clock protocol shifts all scan chains simultaneously. This protocol allows scan shift output for the current pattern and scan shift input for the next pattern to overlap. If scan groups are used, not all scan chains are required to shift simultaneously. For more information on scan groups, see “Creating Scan Groups” on page 7-100.

The process DFT Compiler uses to perform scan shift is determined by the scan style you selected with the `set_scan_configuration -style` command or with the `test_default_scan_style` variable.

For all scan styles, the parallel measure cycle is performed by application of data to nonclock input ports, holding clocks inactive, and comparing data at output ports. The capture cycle involves pulsing a clock. Nonclock input ports remain unchanged from the parallel measure cycle; output ports and bidirectional ports are masked.

Multiplexed Flip-Flop Scan Style

For the multiplexed flip-flop scan style, scan shift is performed by execution of the following steps *n* times, where *n* is the number of bits in the longest scan chain:

1. Assert the scan-enable signals.
2. Apply scan data at the scan input ports.
3. Compare scan data at the scan output ports.
4. Pulse the system clocks.

   System clock ports are identified with the `set_dft_signal` command as `ScanClock` signals.

During the parallel measure and capture cycles, test design rule checking treats the scan-enable signal like any other parallel input; in some capture cycles, the captured data can be from the scan path rather than the functional path. Because fault detection occurs only during the parallel measure cycle and during comparison of captured data at the scan output ports, treating the scan-enable signal as a parallel input allows inclusion of scan logic and clock logic in the fault list and detection of faults on these nodes.

Clocked-Scan Scan Style

For the clocked-scan scan style, scan shift is performed by execution of the following steps *n* times, where *n* is the number of bits in the longest scan chain:

1. Apply scan data at the scan input ports.
2. Compare scan data at the scan output ports.
3. Pulse the scan clock ports.

   Scan clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` signals.

LSSD Scan Style

For the LSSD scan style, scan shift is performed by execution of the following steps *n* times, where *n* is the number of bits in the longest scan chain:

1. Apply scan data at the scan input ports.
2. Compare scan data at the scan output ports.
3. Pulse the test master clock, and then pulse the slave clock.

   Test master and slave clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` and `ScanSlaveClock` signals, respectively.

Scan-Enabled LSSD Scan Style

For the scan-enabled LSSD scan style, scan shift is performed by execution of the following steps *n* times, where *n* is the number of bits in the longest scan chain:

1. Assert the scan-enable signals.
2. Apply scan data at the scan input ports.
3. Compare scan data at the scan output ports.
4. Pulse the test master clock, and then pulse the slave clock.

   Test master clock ports are identified with the `set_dft_signal` command as `ScanMasterClock` signals. System clock ports, which are repurposed as slave test clock ports in test mode, are also identified as `ScanMasterClock` signals but must have slave test clock timing waveforms defined.

Examining a Test Protocol File

You can convert a test protocol file into an ASCII file that you can view and edit. To print this test protocol file to a file, use the `write_test_protocol` command. The command syntax is as follows:

```
write_test_protocol [-output test_protocol_file_name]
     [-test_mode mode_name]
     [-names verilog | verilog_single_bit]
```

Table 5-3 write_test_protocol Command Syntax

Option	Description
`-output` *`test_protocol_file_name`*	Specifies the name of the ASCII output file. The default file name is *`design_name`*`.spf`, where *`design_name`* is the current design, and the `.spf` extension identifies the file type as a STIL format test protocol file.
`-test_mode` *`mode_name`*	Specifies the CTL model test mode from which the protocol is generated.

Table 5-3 write_test_protocol Command Syntax (Continued)

Option	Description
`-names verilog | verilog_single_bit`	Specifies the form of the names used in the STIL procedure file. Names can be unchanged from internal representation (the default). They can also be modified as Verilog names or as Verilog names compatible with the usage of the `verilogout_single_bit` environment variable. In all cases, the internal representation is not changed. This option takes effect only in conjunction with `-test_mode` options, when HSS is used. In all other cases, the form of the names is determined by the setting of the `test_stil_netlist_format` variable.

Note:

Do not use the `write_test_protocol` command before you run `create_test_protocol`. If you do, you will get an error message to the effect that no test protocol exists.

Example 5-9 shows the test protocol file for a multiplexed flip-flop design. This file was generated by use of the `write_test_protocol` command after execution of test design rule checking on the design.

Example 5-9 Test Protocol for Multiplexed Flip-Flop Design Example

```
STIL 1.0 {
   Design P2000.9;
}
Header {
   Title DFT Compiler 2003.06 STIL output;
   Date  Thu Apr 10 14:30:34 2003 ;
   History {
   }
}
Signals {
    CDN  In;  CLK  In;  DATA  In;  IN1  In;  TEST_SE  In;
TEST_SI  In;
    OUT1  Out;  OUT2  Out;
}
SignalGroups {
    all_inputs  =  ' CDN  +  CLK  +  DATA  +  IN1  +  TEST_SE  +
    TEST_SI '; // #signals=6
    all_outputs  =  ' OUT1  +  OUT2 '; // #signals=2
    all_ports  =  ' all_inputs  +  all_outputs '; // #signals=8
    _pi  =  ' all_inputs '; // #signals=6
    _po  =  ' all_outputs '; // #signals=2
}
Timing {
   WaveformTable  _default_WFT_  {
```

```
      Period '100ns';
      Waveforms {
          all_inputs  { 0 { '5ns' D; } }
          all_inputs  { 1 { '5ns' U; } }
          all_inputs  { Z { '5ns' Z; } }
          all_inputs  { N { '5ns' N; } }
          all_outputs  { X { '0ns' X; } }
          all_outputs  { H { '0ns' X; '95ns' H; } }
          all_outputs { T { '0ns' X; '95ns' T; } }
          all_outputs  { L { '0ns' X; '95ns' L; } }
          CLK  { P { '0ns' D; '45ns' U; '55ns' D; } }
          CDN  { P { '0ns' U; '45ns' D; '55ns' U; } }
      }
   }
}
PatternBurst  __burst__  {
   PatList {
       __pattern__  {
      }
   }
}
PatternExec {
   PatternBurst  __burst__ ;
}
Procedures {
    capture  {
      W  _default_WFT_ ;
      V {  _pi =\r6 #;  _po =\r2 #; }
   }
    capture_CLK  {
      W  _default_WFT_ ;
       forcePI : V {  _pi =\r6 #; }
       measurePO : V {  _po =\r2 #; }
       pulse : V {  CLK =P; }
   }
    capture_CDN {
      W  _default_WFT_ ;
       forcePI : V {  _pi =\r6 #; }
       measurePO : V {  _po =\r2 #; }
       pulse : V {  CDN =P; }
   }
}
MacroDefs {
    test_setup  {
      W  _default_WFT_ ;
      V {   CLK =0; }
      V {   CDN =1;  CLK =0; }
   }
}
```

Updating a Protocol in a Scan Chain Inference Flow

If you import an existing-scan netlist without any test attributes, test DRC can infer the scan structures if you perform the following steps:

1. Specify test clocks and other test attributes in the design.
2. Create a test protocol.
3. Run the `dft_drc` command to infer scan structures.

If scan chain inference is successful, the protocol is updated to contain procedures to shift the scan chain.

Masking Capture DRC Violations

By default, DFT Compiler excludes sequential cells with DRC violations from scan chains. In some cases, such as spare cells that have constant data inputs, you can include violating cells in scan chains.

DFT Compiler allows you to mask cells with certain capture DRC violation types, as described in the following sections:

- Configuring Capture DRC Violation Masking
- Reporting Capture DRC Violation Masking
- Resetting Capture DRC Violation Masking

Configuring Capture DRC Violation Masking

You can mask the following capture DRC violation types during pre-DFT DRC:

- `TEST-504` – Cell always captures constant zero value
- `TEST-505` – Cell always captures constant one value
- `D17` – Cell has a clock, set, or reset input pin that cannot capture data

When a violation type is masked, violating cells are included in scan chains. To mask a violation type, use the `set_dft_drc_rules` command. The syntax is

```
set_dft_drc_rules
  [-allow drc_list]
  [-ignore drc_list]
  [-cell cell_list]
```

The -allow and -ignore options both allow you to specify one or more violation types to mask. The difference is as follows:

- -allow – DFT allows violating cells to be included in scan chains, but the violations are still reported
- -ignore – DFT completely ignores the violating cells; the violating cells are included in scan chains and the violations are not reported

For example, the following command includes constant-capturing sequential cells in scan chains (with warnings issued during pre-DFT DRC):

```
dc_shell> set_dft_drc_rules -allow {TEST-504 TEST-505}
```

By default, the specification applies globally to the entire design. To limit the specification to certain cells, use the -cell option. For example, the following command includes specific noncapturing sequential cells in scan chains (with no warnings):

```
dc_shell> set_dft_drc_rules -ignore {D17} \
           -cell [get_object_name [get_cells {CONFIG_reg[*]}]]
```

You can issue multiple set_dft_drc_rules commands. The tool applies all command specifications cumulatively. Cell-specific specifications take precedence over global specifications.

Reporting Capture DRC Violation Masking

You can use the report_dft_drc_rules command to report masking specifications previously applied with the set_dft_drc_rules command. The syntax is

```
report_dft_drc_rules
  [-violation drc_list]
  [-cell cell_list]
```

By default, all previously applied command specifications are reported. For example,

```
dc_shell> report_dft_drc_rules

  Violation       Default         Specified       Range/
    Name          Action          Action          Cell list
  ------------------------------------------------------
  TEST-504        omit            allow           all cells
  TEST-505        omit            allow           BLK1
                  omit            allow           BLK2
                  omit            ignore          USPAREGATES
```

You can use the -violation option to restrict the report to certain violation types. For example,

```
dc_shell> report_dft_drc_rules -violation {TEST-504}
```

```
  Violation       Default         Specified       Range/
    Name          Action          Action          Cell list
  ---------------------------------------------------------
  TEST-504        omit            allow           all cells
```

You can use the `-cell` option to restrict the report to certain cell-specific command specifications. For example,

```
dc_shell> report_dft_drc_rules -cell {BLK1 BLK2}

  Violation       Default         Specified       Range/
    Name          Action          Action          Cell list
  ---------------------------------------------------------
  TEST-505        omit            allow           BLK1
                  omit            allow           BLK2
```

Resetting Capture DRC Violation Masking

You can use the `reset_dft_drc_rules` command to remove masking specifications previously applied with the `set_dft_drc_rules` command. The syntax is

```
reset_dft_drc_rules
  [-violation drc_list]
  [-cell cell_list]
```

By default, all previously applied command specifications are removed. For example,

```
dc_shell> reset_dft_drc_rules
```

You can use the `-violation` option to remove only specifications for certain violation types. For example,

```
dc_shell> reset_dft_drc_rules -violation {TEST-504 TEST-505}
```

You can use the `-cell` option to remove only certain cell-specific command specifications. For example,

```
dc_shell> reset_dft_drc_rules -cell {BLK1 BLK2}
```

6

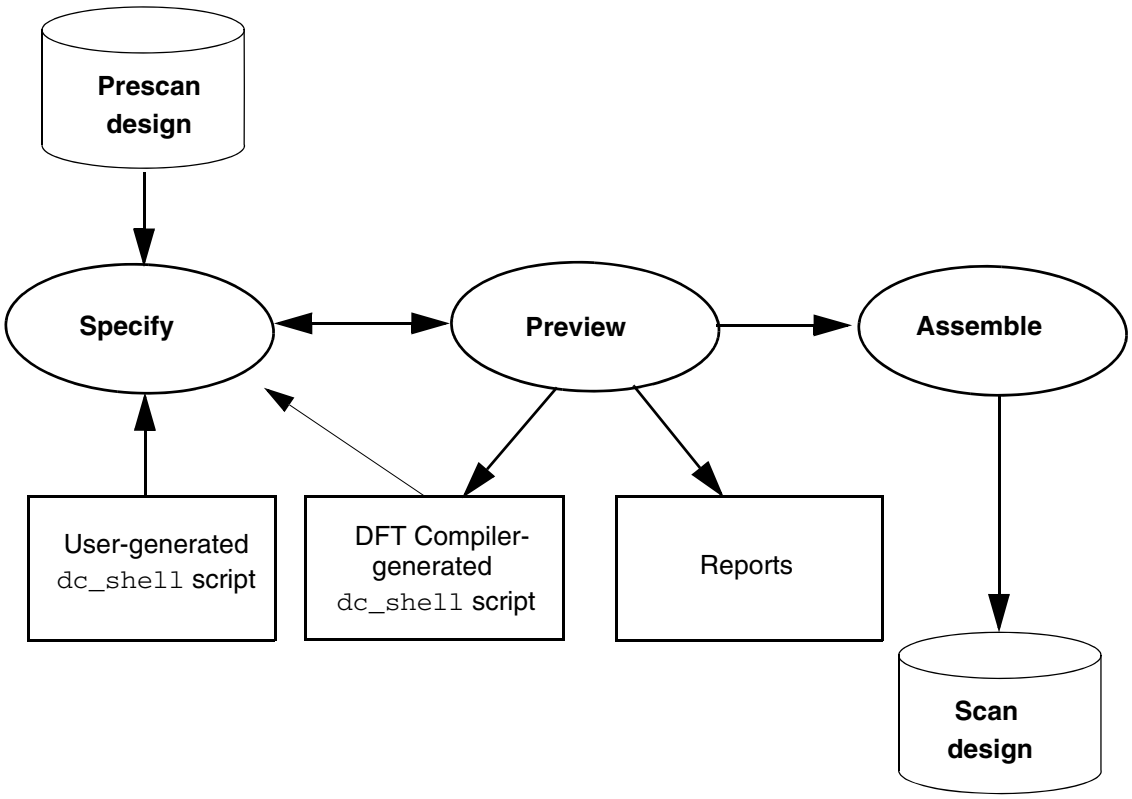
Architecting Your Test Design

This chapter describes the basic processes involved in configuring and architecting your test design for scan insertion.

This chapter includes the following sections:

The standard DFT architecture process consists of configuring your architecture, building scan chains, connecting test signals, setting test clocks, and analyzing your configurations before and after scan insertion. Figure 6-1 shows the basic flow of the scan chain generation process.

Figure 6-1 Scan Chain Generation Process



Configuring Your DFT Architecture

Before you run scan insertion, you need to configure your DFT architecture. This section includes the following subsections related to the configuration process:

- Defining Your Scan Architecture
- Specifying Individual Scan Paths
- Previewing Your Scan Design

Defining Your Scan Architecture

To define your scan architecture, you need to set design constraints, define any test modes, specify test ports, and identify and mark any cells that you do not want to have scanned.

Use the following script for the basic scan assembly flow:

```
current_design top

# specify the scan architecture
set_scan_configuration -chain_count 4

# create the test protocol
create_test_protocol

# check pre-DFT DRC test design rules
dft_drc

# preview the scan structures
preview_dft

# assemble the scan structures
insert_dft

# check post-DFT DRC test design rules
dft_drc
```

Scan configuration is the specification of global scan properties for the current design. Use the `set_scan_configuration` command to specify global scan properties such as

- Scan style and methodology
- Length and number of scan chains
- Handling of multiple clocks
- Internal and external three-state nets
- Bidirectional ports

Note:

This list of the `set_scan_configuration` command's options is not exhaustive. For a complete listing, as well as a description of each option's purpose, see the man page.

DFT Compiler automatically generates a complete scan architecture from the global properties that you have defined.

Setting Design Constraints

You should set constraints before running the `insert_dft` command because it minimizes constraint violations. Use Design Compiler commands to set area and timing constraints on your design. If you have already compiled your design, you do not need to reset your constraints. For more information about setting area and timing constraints on your design, see the *Synopsys Timing Constraints and Optimization User Guide*.

Defining Constant Input Ports During Scan

If your design requires a signal to be held constant to satisfy design rules or to enable circuit paths, you need to use the `set_dft_signal` command. For more information, see Chapter 5, "Pre-Scan Test Design Rule Checking".

Specifying Test Ports

The `insert_dft` command adds scan signals that use existing ports. These ports are identified by using the `set_dft_signal` command. If the tool cannot find existing ports that it can use as test ports, it adds new ports to the design. The `insert_dft` command names the new ports according to the following variables:

- `test_clock_port_naming_style`
- `test_scan_clock_a_port_naming_style`
- `test_scan_clock_b_port_naming_style`
- `test_scan_clock_port_naming_style`
- `test_scan_enable_inverted_port_naming_style`
- `test_scan_enable_port_naming_style`
- `test_clock_in_port_naming_style`
- `test_clock_out_port_naming_style`

Specifying Individual Scan Paths

DFT Compiler supports detailed specification of individual scan paths. Use the following commands to specify the scan architecture:

- `set_scan_element`

  Use this command to specify sequential elements that are to be excluded from scan replacement. By default, all nonviolating sequential cells with equivalent scan cells are scan-replaced. You can specify leaf cells, hierarchical cells, references, library cells, and designs.

  Use the `set_scan_element` command sparingly. For best results, use the command only on leaf or hierarchical cells.

- `set_scan_path`

  Use this command to specify properties specific to a scan chain, such as name, membership, chain length, clock association, and ordering.

- `set_dft_signal`

  Use this command to specify desired port connections and scan chain assignments for test signals.

- `set_autofix_element`

  Use this command to control particular bidirectional ports on the top level of the current design.

In case you are unfamiliar with some of the scan path components used in the scan specification commands, Figure 6-2 illustrates the scan path components.

Figure 6-2 Scan Path Components

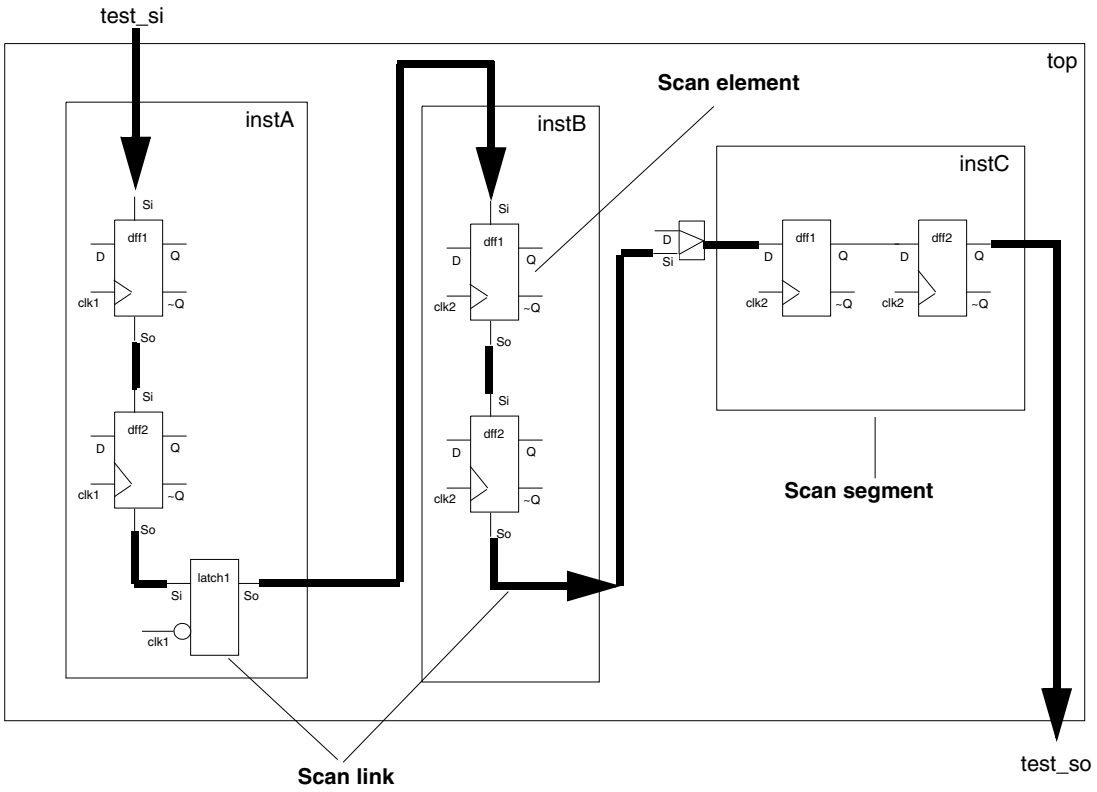


Figure 6-2 shows a single scan path that starts at port test_si, which receives the test_scan_in scan signal, and ends at port test_so, which drives the test_scan_out scan signal. Cells instA/dff1, instA/dff2, instB/dff1, and instB/dff2 are examples of scan elements. The shift register in instC is a defined scan segment. In the bottom-up flow, the scan chains in instA and instB are considered subchains or inferred scan segments. The thick lines represent scan links. The latch (instance latch1) is also a scan link.

The following sections discuss some of the situations you might encounter during scan specification. See Chapter 4, “Performing Scan Replacement,” for scan style selection considerations.

Previewing Your Scan Design

Use the `preview_dft` command to preview your scan design. The command generates a scan chain design that satisfies scan specifications on the current design and displays the scan chain design. This allows you to preview your scan chain designs without synthesizing them and to change your specifications to explore the design space as necessary.

When you are using the dft_drc command, a valid protocol must exist before you run the preview_dft command. This is created by the create_test_protocol command. If no DFT DRC has been performed, the preview_dft command runs it automatically. The preview_dft and insert_dft commands use the same algorithms to design scan chains.

Example 6-1 shows a report example generated by the preview_dft command.

Example 6-1 Report Generated by the preview_dft Command

```
**************************************
Preview_dft report
For    : 'Insert_dft' command
Design : sub
Version: G-2012.06
Date   : Wed Jun 20 04:49:18 2012
**************************************

Number of chains: 2
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix

Scan chain '1' (test_si1 --> Z[15]) contains 8 cells

Scan chain '2' (test_si2 --> Z[7]) contains 8 cells
```

You can use the preview_dft command before DFT insertion to report a summary of the scan chains you are going to get. You can also use the report_scan_path -view existing_dft -chain all command after DFT insertion to report a summary of the scan chains that have been inserted. However, the preview_dft command displays additional types of information beyond scan path information, such as test point, test signal, and scan compression information.

Architecting Scan Chains

The set_scan_configuration command enables you to specify the scan chain design. This command controls almost all aspects of how the insert_dft command makes designs scannable. Exceptions are specific to particular scan chains and are specified in the set_scan_path command options.

This section has the following subsections related to architecting scan chains:

- Controlling the Scan Chain Length
- Determining the Scan Chain Count
- Defining Individual Scan Chain Characteristics
- Balancing Scan Chains

Controlling the Scan Chain Length

You can globally specify the length of scan chains. Controlling the length of the scan chain can help to balance the scan configuration in a design that has bottom-up or system-on-a-chip (SoC) scan insertion.

Specifying the Global Scan Chain Length Limit

Setting the scan chain length limit helps with bottom-up scan insertion by balancing scan chains more efficiently at the top level. Setting a limit on the length of scan chains allows for design constraints related to pin availability or test vector depth.

Use the `set_scan_configuration -max_length` command to specify the length of a scan chain:

```
dc_shell> set_scan_configuration -max_length 7
```

For example, if you set the scan chain length limit to 7 registers for a single-clock, single-edge design with 29 flip-flops, the `insert_dft` command creates five scan chains with lengths of 6, 6, 6, 6, and 5 registers. This scan chain allocation meets the scan chain length limit while also balancing the scan chain lengths as closely as possible.

Note:

Specifying both the `-max_length` option and the `-chain_count` option (described in the next section) might result in conflicting scan chain allocations. In such a case, the `-max_length` option takes precedence.

Specifying the Global Scan Chain Exact Length

You can specify an exact length for all scan chains by using the `-exact_length` option of the `set_scan_configuration` command.

For example, suppose your design has 420 flip-flops, and you want an exact length of 80 flip-flops per scan chain. In this case, specifying `set_scan_configuration -exact_length 80` creates five chains with 80 flip-flops and one chain with 20 flip-flops.

Warning:

The exact length feature is meant to be used only with standard scan, including standard scan configured for multiple test modes. It is not currently supported with DFTMAX compressed scan modes. Do not use this feature with DFTMAX scan compression.

Note the following properties of this feature:

- This option disables scan chain balancing.
- The `-exact_length` option takes precedence over both the `-max_length` and `-chain_count` options.
- The `report_scan_configuration` command reports the value of the exact length configuration.
- The user-specified chain configuration is preserved.
- The quality of results cannot be guaranteed when this option is used on designs containing complex segments.

Determining the Scan Chain Count

You can specify the number of scan chains. DFT Compiler attempts to create the specified number of scan chains while minimizing the longest scan chain length. Use these questions to decide how many scan chains to request:

- How many scan chains does your semiconductor vendor allow?

  Many semiconductor vendors restrict the maximum number of scan chains due to software or tester limitations. Before performing scan specification, check with your semiconductor vendor for the maximum number of scan chains supported.

- How many clock domains exist in your design?

  To prevent timing problems on the scan path in multiplexed flip-flop designs, allocate a scan chain for each clock domain (DFT Compiler default behavior). DFT Compiler considers each edge of a clock a unique clock domain. Multiple clock domains do not affect the number of scan chains in scan styles other than multiplexed flip-flop.

- How much time will it take to test your design?

  Because the test time is proportional to the length of the longest scan chain, increasing the number of scan chains reduces the test time for a design.

Use the set_scan_configuration -chain_count command to specify the number of scan chains.

```
dc_shell> set_scan_configuration -chain_count 7
```

By default, DFT Compiler generates

- One scan chain per clock domain if you select the multiplexed flip-flop scan style
- One scan chain if you select any other scan style

Note:

The -max_length and -chain_count options are mutually exclusive. If you use both options, the -max_length option takes precedence over the -chain_count option.

Defining Individual Scan Chain Characteristics

Typically, scan chains are configured using global chain count or chain length settings. Use the set_scan_path command to specify one or more additional requirements for individual scan chains in the current design.

The set_scan_path command enables you to

- Specify a name for a scan chain
- Allocate scan cells, scan segments, scan links, and subdesign scan chains to scan chains and specify the ordering of the scan chain
- Specify a dedicated scan-out port
- Limit a scan chain's elements to only those components you specify or enable DFT Compiler to balance scan chains by adding more elements
- Specify individual exact scan chain lengths
- Assign scan chains to clock domains

Scan chain elements cannot belong to more than one chain. The command options are not incremental. Where set_scan_path commands conflict, the preview command (preview_dft) and scan insertion command (insert_dft) apply the most recent command.

For example, the following command sets the length of an individual scan chain:

```
dc_shell> set_scan_path C1 -exact_length 40
```

Balancing Scan Chains

The `insert_dft` command always attempts to balance the number of cells in each scan chain. However, some scan chain requirements can limit or disable balancing, such as

- Disabling clock mixing or clock edge mixing
- Defining scan chains with the `set_scan_path` command
- Using test models
- Using a hierarchical DFT insertion flow
- Specifying a global scan chain exact length

When overriding the default behavior, always use the `preview_dft` command to verify that the result meets your requirements.

Multiple Clock Domains

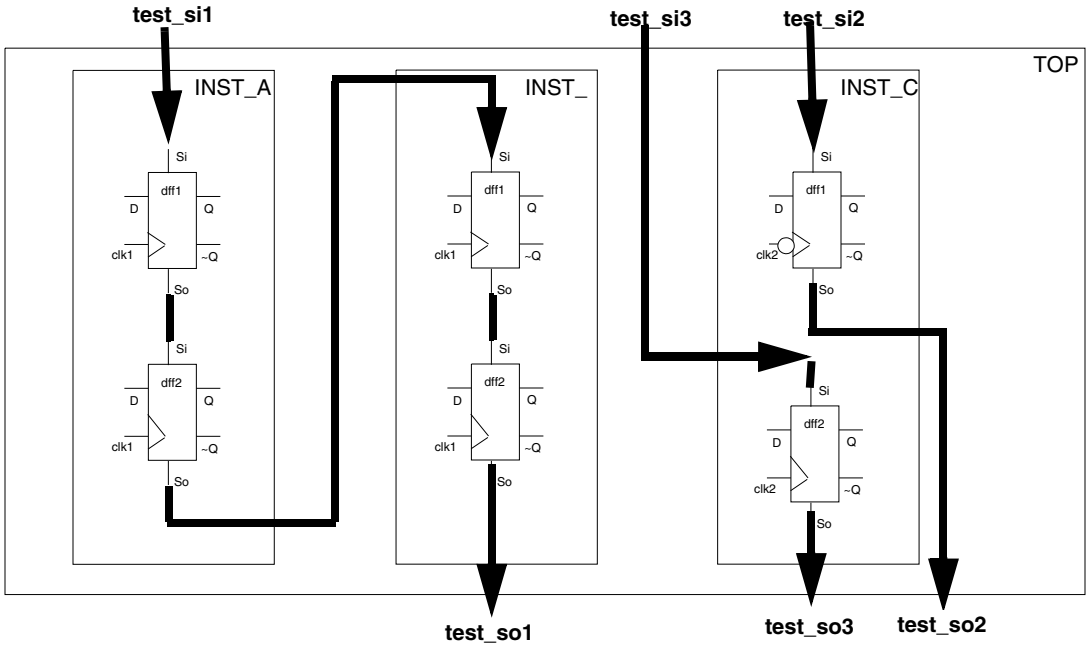
The *clock edge* of a scan cell represents both the clock identity and the active clock edge of the cell. For multiplexed flip-flop designs, DFT Compiler allocates cells to scan chains based on clock edges by default. You can override this default behavior by using the `set_scan_configuration -clock_mixing` command.

For example, assume that you have a design with three clock domains and your desired scan architecture contains two balanced scan chains.

```
dc_shell> set_dft_signal -view existing_dft \
            -type ScanClock -timing [list 45 55] \
            -port {clk1 clk2}

dc_shell> set_scan_configuration -chain_count 2
```

In the default case shown in Figure 6-3, DFT Compiler overrides your request for two chains and generates three scan chains, one for each clock edge (clk1, positive-edge clk2, negative-edge clk2). Because the clock domains contain unequal numbers of cells, DFT Compiler generates unbalanced scan chains.

Figure 6-3 Unbalanced Scan Chains Due to Multiple Clock Domains

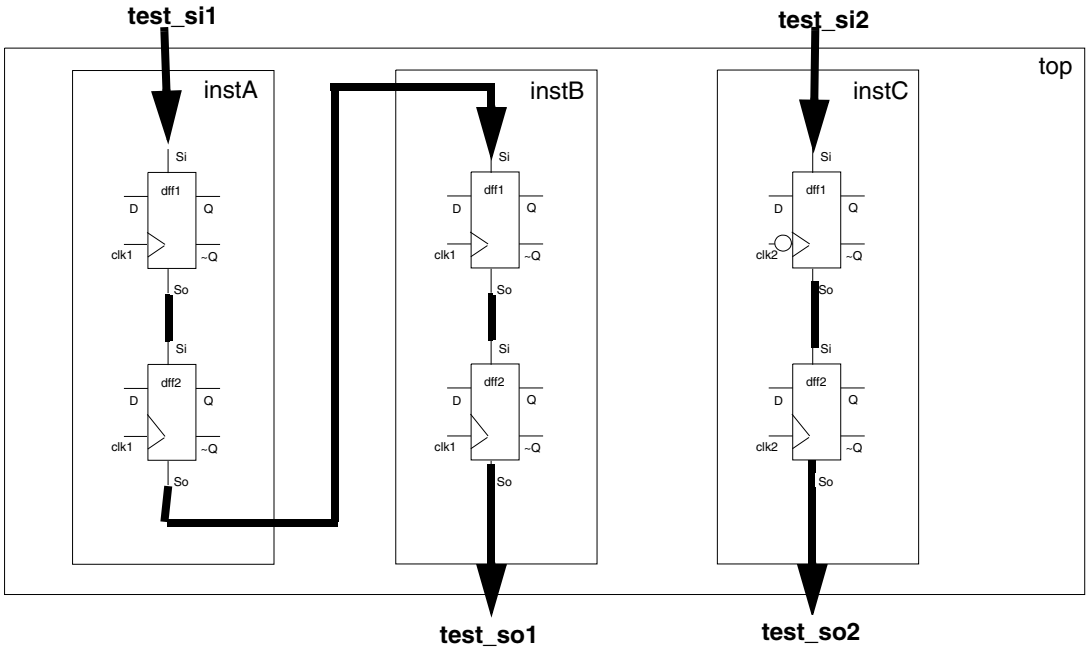


You can reduce the number of scan chains and achieve slightly better balancing by mixing clock edges within a single chain.

```
dc_shell> set_dft_signal -view existing_dft  \
              -type ScanClock -timing [list 45 55] \
              -port {clk1 clk2}

dc_shell> set_scan_configuration -chain_count 2

dc_shell> set_scan_configuration -clock_mixing mix_edges
```

Mixing clock edges in a single scan chain produces a small timing risk. DFT Compiler automatically orders the cells within the scan chain so the cells clocked later in the cycle appear earlier in the scan chain, resulting in a functional scan chain. Figure 6-4 shows the scan architecture when you allow edge mixing.

Figure 6-4 Better Balancing With Mixed Clock Edges

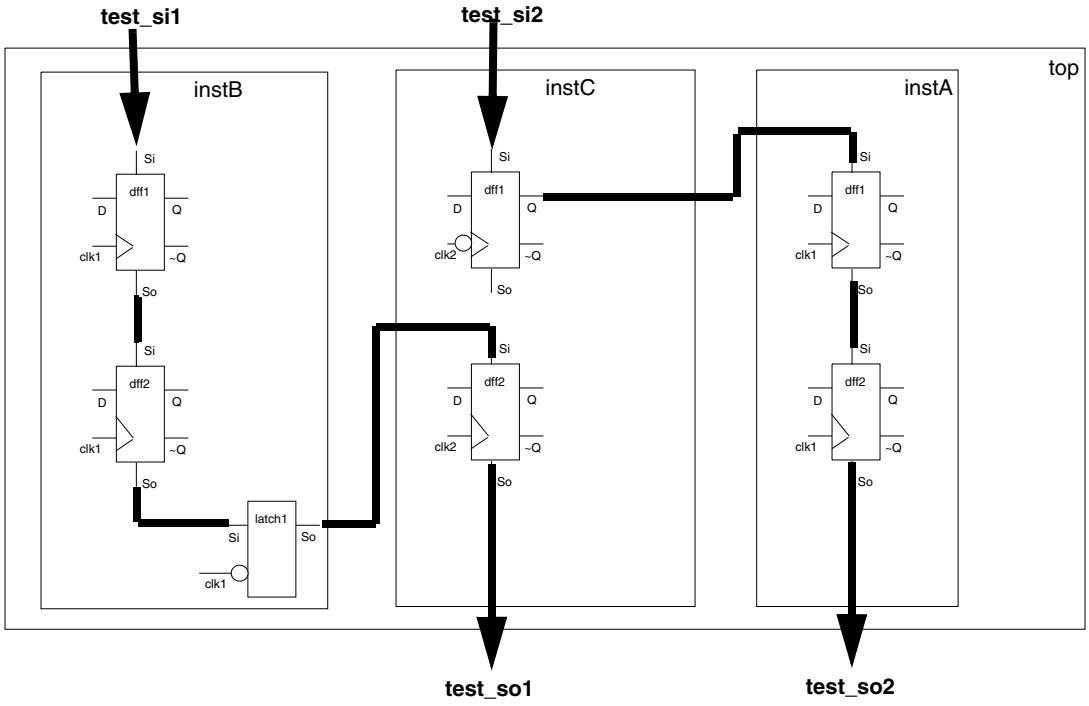


You can balance the scan chains by mixing clocks:

```
dc_shell> set_dft_signal -view existing_dft \
            -type ScanClock -timing [list 45 55] \
            -port {clk1 clk2}

dc_shell> set_scan_configuration -chain_count 2

dc_shell> set_scan_configuration -clock_mixing mix_clocks
```

Mixing clock edges in a single scan chain can produce a large timing risk. To reduce this risk, DFT Compiler adds lock-up latches to the scan path wherever clock skew issues might occur. Figure 6-5 shows the resulting scan architecture.

Figure 6-5 Balanced Scan Chains With Mixed Clocks



See "Architecting Test Clocks" on page 6-48 for details about scan lock-up latches and other clock-mixing considerations.

Multibit Components and Scan Chains

You can specify whether multibit components are implicitly treated as synthesizable segments by using the `set_scan_configuration -preserve_multibit_segment` command. This command applies globally to the current design and overrides any specification on subdesigns. By default, multibit components are implicitly treated as synthesizable segments. Note, for example, the command

```
dc_shell> set_scan_configuration \
              -preserve_multibit_segment false
```

This command ensures that each bit of all sequential multibit components is treated individually when the `insert_dft` command builds scan chains. Scan replacement is still performed homogeneously for each multibit component.

Controlling the Routing Order

Use the `set_scan_path` command to control the routing order explicitly. You can specify the routing order of nonscan as well as scanned sequential cells. Each `set_scan_path` command generates a scan chain; DFT Compiler uses the first command argument as the scan chain name. If you enter multiple `set_scan_path` commands with the same scan chain name, DFT Compiler uses only the last command entered.

You can provide partial or complete scan ordering specifications. Use the `-complete true` option to indicate that you have completely specified a scan chain. DFT Compiler does not add cells to a completely specified scan chain. If you provide a partial scan-ordering specification, DFT Compiler might add cells to the scan chain. DFT Compiler places the cells specified in a partial ordering at the end of the scan chain.

DFT Compiler validates the specified scan ordering. The checks performed by DFT Compiler include

- Cell assignment

  DFT Compiler verifies that you have not assigned a cell to more than one scan chain. A violation triggers the following error message during execution of the `set_scan_path` command:

  ```
  Error: Scan chains '%s' and '%s' have common elements. (TESTDB-256)
  Common elements are:
    %s
  ```

  DFT Compiler discards the second scan path specification, keeping the first scan path specification which contains the common element.

- Clock ordering

  DFT Compiler verifies that the active clock edge of the next scan cell occurs concurrently or before the active clock edge of the current scan cell or that the active edge can be synchronized with a scan lock-up latch.

  If your multiplexed flip-flop design violates this requirement, DFT Compiler reorders the invalid mixed-clock scan chains and triggers the following warning message during execution of the `preview_dft` command:

  ```
  Warning: User specification of chain '%s' has been reordered.
  (TEST-342)
  ```

- Clock mixing

  DFT Compiler verifies that all cells on a scan path have the same clock unless you have specifically requested clock mixing. A violation triggers the following warning message during execution of the `preview_dft` command:

  ```
  Warning: Chain '%s' has elements clocked by different clocks.
  (TEST-353)
  ```

DFT Compiler creates the requested scan chain. Unless you have disabled scan lock-up latch insertion, DFT Compiler inserts a scan lock-up latch between clock domains.

- Black-box cells

  DFT Compiler verifies that the specified cells are valid scan cells. If a sequential cell has a test design rule violation or has a `scan_element false` attribute, DFT Compiler considers it a black-box cell. A violation triggers the following warning message during execution of the `preview_dft` command:

  ```
  Warning: Cannot add '%s' to chain '%s'. The element is not being
  scanned. (TEST-376)
  ```

  DFT Compiler creates the requested scan chain without the violating cells.

Preventing Half-Cycle Paths Between Hierarchical Scan Chains

In some cases, hierarchical blocks can contain scan chains with a mix of positive edge-triggered and negative edge-triggered flip-flops. When these block-level scan chains are combined at a higher level in the design hierarchy to form longer scan chains, half-cycle paths across clock edges might be created between the scan chains. Meeting timing for these half-cycle scan chain paths can be challenging at higher frequencies, especially for chips with long top-level routes between blocks.

To avoid these half-cycle paths when block-level scan chains are combined, use the `-add_test_retiming_flops` option of the `set_scan_configuration` command. For example,

```
dc_shell> set_scan_configuration -add_test_retiming_flops begin_and_end
```

When this option is specified for block-level scan chain insertion, flip-flops triggering on the leading edge of the test clock are added as needed to any scan chains that begin or end with trailing-edge-triggered flip-flops. For return-to-zero clocks, rising-edge flip-flops are used to retime to the leading edge. For return-to-one clocks, falling-edge flip-flops are used to retime to the leading edge. The tool automatically chooses the edge-triggered retiming cell from the target library.

Valid keywords for the `-add_test_retiming_flops` option are `begin_and_end`, `begin_only`, `end_only`, and `none`. The default is `none`, which disables retiming register insertion.

Table 6-1 shows the retiming behaviors provided by the `-add_test_retiming_flops` option.

Table 6-1 Retiming Behaviors Provided by the -add_test_retiming_flops Option

If the scan chain	`begin_only`	`end_only`	`begin_and_end`
Begins with a leading-edge flip-flop			
Begins with a trailing-edge flip-flop	Adds retiming flip-flop to beginning of scan chain		Adds retiming flip-flop to beginning of scan chain
Ends with a leading-edge flip-flop			
Ends with a trailing-edge flip-flop		Adds retiming flip-flop to end of scan chain	Adds retiming flip-flop to end of scan chain

To report the locations where these retiming flip-flops are to be added, use the `preview_dft` command.

If you generate a SCANDEF file for a design with retiming flip-flops, the retiming flip-flops are not included between the START and STOP points in the SCANDEF file. Only the original design scan cells are included between the START and STOP points.

Routing Scan Chains and Global Signals

Most scan cells have both a scan output pin (test_scan_out) and an inverted scan output pin (test_scan_out_inverted) defined in the logic library. If the functional path through a sequential cell has timing constraints, DFT Compiler automatically selects the scan output pin with the most timing slack for use as the scan output. To disable this behavior, set the `test_disable_find_best_scan_out` variable to `true`.

Scan chain allocation and ordering might differ between a top-down implementation and a bottom-up implementation because

- DFT Compiler does not modify subdesign scan chains unless explicitly specified in your scan configuration.
- DFT Compiler overrides alphanumeric ordering to provide a shared scan output connection on the current design but not on subdesigns.

Rerouting Scan Chains

The scan specification process previously discussed enables both initial routing and rerouting of your design. However, the specify-preview loop runs faster than the specify-synthesize loop. Try to avoid rerouting by iterating through the specify-preview loop until the scan architecture meets your requirements.

To optimize the design during scan assembly, DFT Compiler

- Performs scan-specific optimizations to reduce the timing impact of scan routing.

  In many cases, the scan path uses the functional output as the scan output. The scan path routing increases the output load on the functional output. If you used test-ready compile for scan replacement, this additional loading is compensated for during optimization. If you used constraint-optimized scan insertion, DFT Compiler uses focused optimization techniques during scan assembly to minimize the impact of the additional load on the overall design performance.

- Replaces unrouted scan cells with their nonscan equivalents.

  If you used test-ready compile for scan replacement, your design might contain unrouted scan cells. These unrouted scan cells occur because the cell has a test design rule violation.

  DFT Compiler replaces these unrouted scan cells with their nonscan equivalents during execution of the `insert_dft` command.

  Your design might contain sequential cells that are defined in the logic library as scan cells but can also implement functional logic in your design. These cells have functional connections to both the data and scan inputs, and DFT Compiler does not modify these cells during scan assembly.

- Fixes hold time violations on the scan path if the clock net has the `fix_hold` attribute.

Stitching Scan Chains Without Optimization

In some circumstances, you might want to stitch your design's scan chains together but avoid the optimization step. This process is referred to as "rapid scan synthesis." Such circumstances might include

- Stitching completed subdesigns together
- Performing synthesis and scan insertion in the logic domain and optimizations in the physical domain
- Performing analysis on the design

Specifying a Stitch-Only Design

When DFT Compiler performs scan stitching without optimization, it still performs comprehensive logic DFT design rule checks, but it eliminates the runtime-intensive synthesis mapping, timing violation fixing, and design rule fixing steps.

Consequently, the design is only stitched and no further optimizations are performed on the design.

To enable scan stitching without optimization, use the following command:

```
dc_shell> set_scan_replacement
```

Mapping the Replacement of Nonscan Cells to Scan Cells

You might want to stitch a design that has not been scan-replaced. The `set_dft_insertion_configuration -synthesis_optimization none` command can perform scan replacement on designs of this sort.

If a simple one-to-one mapping of a nonscan to a scan cell is not available in the library, DFT Compiler performs a cell decomposition followed by a sequential mapping algorithm. You can avoid this step by using the following command:

```
dc_shell> set_scan_replacement  \
               -nonscan nonscan_cell_list \
               -multiplexed_flip_flop scan_cell
```

The options in this command should always be specified as a pair. If they are not, an error results. Many cells can be listed in the `-nonscan` option, but only one cell can be listed in the `-multiplexed_flip_flop` option. You can use the `-lssd` option in place of the `-multiplexed_flip_flop` option.

If you use this command and a scan cell definition exists in the ASIC library, the mapping you specified with the `set_scan_replacement` command overrides the library definition. This command is global in nature; it affects the entire design.

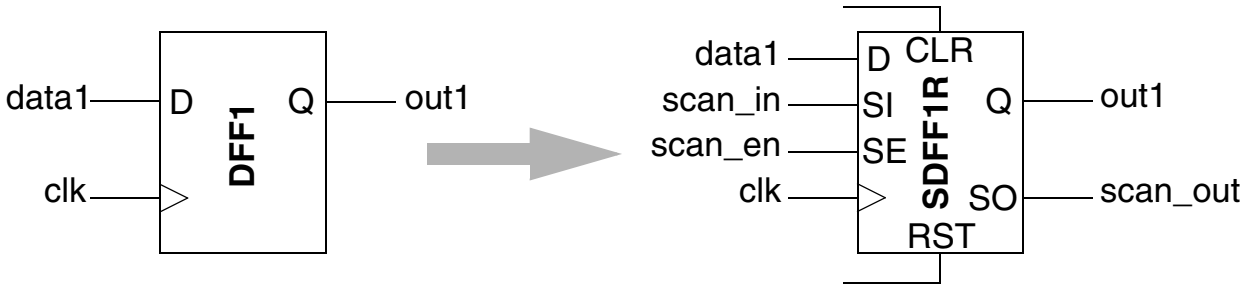
For example, the scan cell DFFS1 is a direct mapping of the nonscan cell DFFD1, but with scan pins. To specify the mapping of the DFFD1 nonscan cell to the DFFS1 scan cell, use the following command:

```
dc_shell> set_scan_replacement -nonscan DFFD1 \
               -multiplexed_flip_flop  DFFS1
```

Few-Pins-to-Many-Pins Scan Cell Replacement Situation

If you select a scan cell that has more pins than the nonscan cell it replaces, the extra pins are tied to the inactive state and a warning is issued. You can fix this problem by respecifying a more appropriate cell with the `set_scan_replacement` command.

Figure 6-6 Few-to-Many Scenario (Accepted)




```
set_scan_replacement  -nonscan DFF1\
   -multiplexed_flip_flop SDFF1R
```

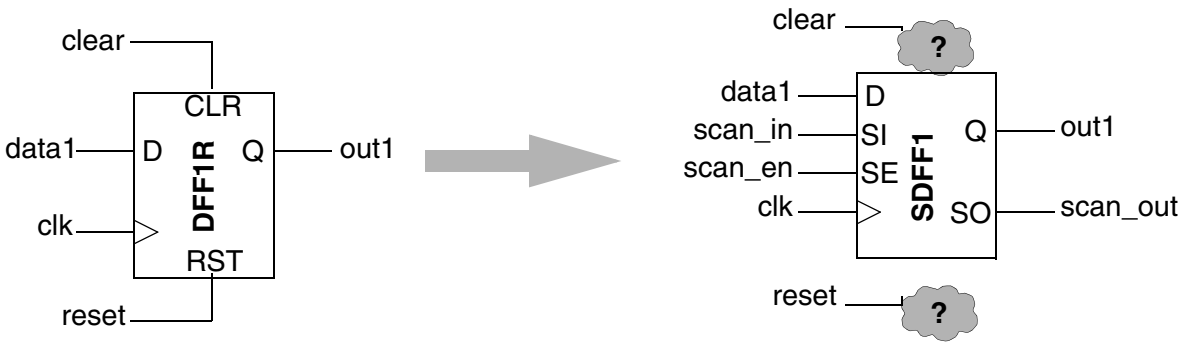
In Figure 6-6, the replacement cell has more inputs and outputs than required by the nonscan cell. The unused pins of the scan cell are left unconnected.

Many-Pins-to-Few-Pins Scan Cell Replacement Scenario

Alternatively, if you select a scan cell that has fewer pins than the nonscan cell it replaces, the extra pins are left unconnected. To avoid problems with incorrect logic, an error message is issued and the replacement does not occur. You can fix this problem by respecifying a more appropriate cell with the `set_scan_replacement` command.

In Figure 6-7, for example, the clear and reset pins do not exist on the scan cell. They are left unconnected, causing incorrect logic.

Figure 6-7 Many-to-Few Scenario (Rejected)




```
set_scan_replacement  -nonscan DFF1R\
   -multiplexed_flip_flop SDFF1
```

Criteria for Conversion Between Nonscan and Scan Cells

This section describes the conditions under which

- A sequential cell is excluded from the DRC violations
- A sequential cell is excluded from the scan chains
- A nonscan cell becomes a scan cell
- A scan cell is unscanned

DRC Violation Report (dft_drc)

A cell XYZ should be reported as a valid nonscan cell by DRC if the following command is used:

```
dc_shell> set_scan_element false XYZ
```

Scan Architect (insert_dft)

A cell XYZ will not be part of the scan chains if any of the following conditions are met:

- The following command is used:

  ```
  dc_shell> set_scan_element false XYZ
  ```

- The cell XYZ is DRC violated
- The following command is used:

  ```
  dc_shell> set_scan_configuration -exclude_elements XYZ
  ```

  Note:

  You use `set_scan_configuration -exclude_elements` to prevent flip-flops from being stitched into the scan chains. The difference between using `set_scan_configuration -exclude_elements` and `set_scan_element false` is that the former command does not unscan the specified flip-flops during `insert_dft` whereas the latter command does unscan the flip-flops.

Scan Replacement (insert_dft)

A nonscan flip-flop cell, FF, will become a scan cell in either of the two following cases:

- Both of the following conditions are met:
  - The nonscan flip-flop cell is not DRC violated
  - The following command is used:

    ```
    dc_shell> set_scan_element true FF
    ```

- Both of the following conditions are met:
  - The following command is used:

```
dc_shell> set_scan_element true FF
```

  - The following command is used:

```
dc_shell> set_scan_configuration -exclude_elements FF
```

A scan cell, SFF, will be converted to a nonscan cell in either of the two following cases:

- The following command is used:

```
dc_shell> set_scan_element false SFF
```

- Both of the following conditions are met:
  - The scan cell, SFF, is DRC violated
  - The following command is used:

```
dc_shell> set_dft_insertion_configuration \
              -unscan true
```

Using Existing Subdesign Scan Chains

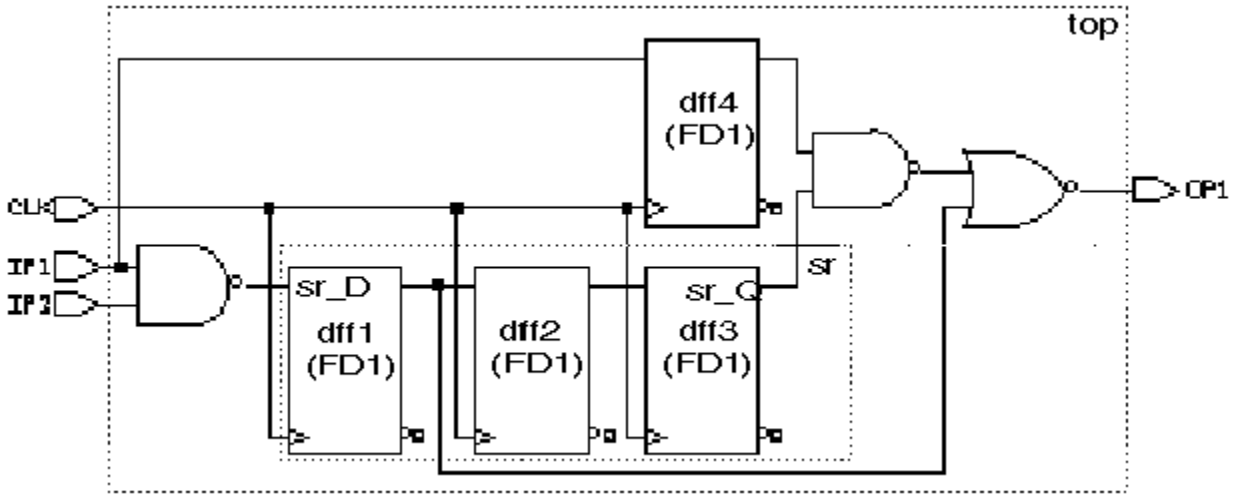
A subdesign scan chain uses subdesign ports for all test signals. DFT Compiler can infer subdesign scan chains during test design rule checking.

To reuse existing subdesign scan chains, follow these steps:

- Set the current design to the subdesign containing the existing scan chain.
- Use the `set_dft_signal` command to identify the existing scan ports.
- Create a test protocol by using the `create_test_protocol` command.
- Set the current design to the design where you are assembling the scan structures.
- Use the `set_scan_path` command to control the scan chain connections, if desired.

For example, subdesign sr in Figure 6-8 contains a shift register. The shift register performs a serial shift function, so DFT Compiler can use this existing structure in a scan chain. The scan input signal connects to subdesign port sr_D. The scan output signal connects to subdesign port sr_Q. The shift register always performs the serial shift function, so the shift register does not need a scan-enable signal.

Figure 6-8 Subdesign Scan Chain Example Before Scan Insertion



Use the following command sequence to infer the subdesign scan chain in module sr:

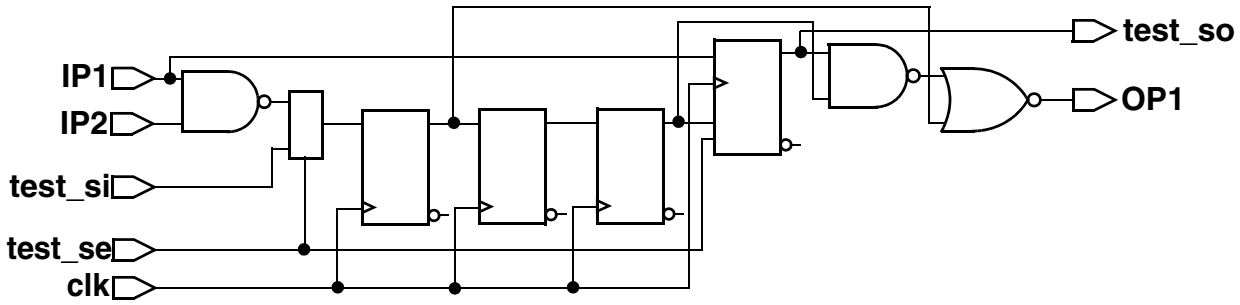
```
dc_shell> current_design sr
dc_shell> set_dft_signal -view spec -port sr_D -type ScanDataIn
dc_shell> set_dft_signal -view spec -port sr_Q -type ScanDataOut
dc_shell> create_test_protocol
dc_shell> dft_drc
```

Use the following command sequence to include this scan chain in a top-level scan chain:

```
dc_shell> current_design top
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> insert_dft
dc_shell> dft_drc
```

Figure 6-9 shows the top-level scan chain, which includes the subdesign scan chain. DFT Compiler added a multiplexer, controlled by the scan-enable signal, to select between the functional data input and the scan input. The hierarchical cell name determines the location of the subdesign scan chain in the top-level scan chain.

Figure 6-9 Subdesign Scan Chain Example After Scan Insertion



Uniquifying Your Design

When you run the `insert_dft` command, DFT Compiler automatically assigns a unique name to any subdesigns that changed during the scan insertion process. The default naming convention saves subdesign A as A_test_1. If two instances of subdesign A are different, they are saved as A_test_1 and A_test_2. The following scenarios illustrate examples in which unique names are assigned to instances of a subdesign:

- You specify a different scan ordering in each instance of the same reference design.

  For example, if you route and rebalance a design so that two instances of the subdesign have different scan chain ordering, the `insert_dft` command uniquifies the design.

- The `insert_dft` command identifies different solutions during constraint optimization and design rule fixing.

  Constraint optimization and design rule fixing are features of the `insert_dft` command. To eliminate unnecessary uniquification, turn off these features by entering the following commands:

  ```
  dc_shell> set_dft_insertion_configuration \
                -synthesis_optimization none
  ```

- There are scan violations in one instance but not in another instance, and `insert_dft` repairs one but not the other.

You can choose the suffix that gets appended to the design name to create the unique name. The naming convention for the suffix appended to the design name is controlled by the following command:

```
set insert_test_design_naming_style name
```

In the previous example, the default name is *design_name*_test_*counter*.

Note:

To prevent uniquification of your design, enter the command:

```
dc_shell> set_dft_insertion_configuration \
              -preserve_design_name true
```

Reporting Scan Path Information on the Current Design

Use the report_scan_path command to display scan path information for the current design.

Note:

To show changes caused by running the insert_dft command, you must run the dft_drc command before the report_scan_path command. Running an incremental compile or any other command that changes the database causes the dft_drc results to be discarded. In such a case, you need to run dft_drc again before you use report_scan_path.

Example 6-2 shows the type of information displayed by the report_scan_path -chain all command.

Example 6-2 Scan Path Information Displayed by the report_scan_path Command

```
========================================
AS BUILT BY insert_dft
========================================

Scan_path    Len   ScanDataIn  ScanDataOut ScanEnable  MasterClock SlaveClock
-----------  ---- ----------- ----------- ----------- ----------- -----------
I 1          22    test_si1    test_so1    test_se     CLK         -
I 2          21    test_si2    test_so2    test_se     CLK         -
I 3          21    test_si3    test_so3    test_se     CLK         -
```

For more information on the options of the report_scan_path command, see the man pages.

Architecting Scan Signals

For test design rule checking to recognize test ports in your design, your scan-inserted design must have appropriate signal_type attributes on the test ports. If you are using your own placeholder test ports, you must set these attributes with the set_dft_signal command. If the insert_dft command creates any needed ports, these attributes are automatically set.

The following topics discuss the process for architecting scan signals:

Specifying Scan Signals for the Current Design

Use the `set_dft_signal` command to specify one or more scan signals for the current design.

Table 6-2 provides a list of `signal_type` attribute values.

Table 6-2 signal_type Attribute Values for Test Signals

Test I/O port signal	signal_type value	Valid on input	Valid on output	Valid on three-state output	Valid on bidirectional input/output
Scan-in	`ScanDataIn`	Yes	No	No	Yes
Scan-out	`ScanDataOut`	No	Yes	Yes	Yes
Scan-enable	`ScanEnable`	Yes	No	No	Yes
Bidirectional enables	`InOutControl`	Yes	No	No	*
Asynchronous control ports	`Reset`	Yes	No	No	Yes

***Not recommended: complex methodologies required.**

The following is an example of the `set_dft_signal` command specifying a scan-in port. If you enter

```
dc_shell> set_dft_signal -view spec -port scan_in -type ScanDataIn
```

DFT Compiler responds with the following:

```
Accepted dft signal specification for modes: all_dft
```

In the preceding example, the `-view spec` option indicates that the specified ports are to be used during DFT scan insertion and that DFT Compiler is to perform the connections. In this example, `scan_in` is the name of the scan-in port that the `insert_dft` command uses. (The other value of the `-view` argument is `-existing_dft`, which directs the tool to use the specified ports as is because they are already connected.)

When the `insert_dft` command creates additional ports for scan test signals, it assigns a name to each new port. You can control the naming convention by using the port naming style variables shown in Table 6-3.

Table 6-3 Port Naming Style Variables

Name	Default
test_scan_in_port_naming_style	test_si%s%s
test_scan_out_port_naming_style	test_so%s%s
test_scan_enable_port_naming_style	test_se%s
test_scan_enable_inverted_port_naming_style	test_sei%s
test_clock_port_naming_style	test_c%s
test_scan_clock_port_naming_style	test_sc%s
test_scan_clock_a_port_naming_style	test_sca%s
test_scan_clock_b_port_naming_style	test_scb%s
test_mode	test_mode%s
test_point_clock	none

Follow these guidelines when using the `set_dft_signal` command:

- Use the `set_dft_signal` command for scan insertion and for design rule checking. The `set_dft_signal` command indicates I/O ports that are to be used as scan ports. After

the `insert_dft` command connects these ports, it places the necessary `signal_type` attributes on the ports for post-insertion design rule checking.

- Use the `set_dft_signal -view existing_dft` command if you read in an ASCII netlist and you need to perform design rule checking. Before you use the `set_dft_signal` command, the ASCII netlist does not contain the `signal_type` attributes annotated by scan insertion. Without these attributes, `dft_drc` does not know which ports are scan ports and therefore reports that the design is untestable.
- Use the `set_dft_signal -view existing_dft` command if the ports in your design are already connected and no connection is to be made by DFT Compiler.
- Use the `set_dft_signal -view spec` command if the connections do not exist in your design and you expect DFT Compiler to make the connections for you.

Using the -view spec and -view existing_dft Arguments

Unlike other tools used in the implementation flow, DFT Compiler changes the functionality of your design such that the design can operate in either functional (“mission”) mode or test mode.

To construct this dual modality, DFT Compiler needs to know what already exists in the design. You use the `-view existing_dft` option with the `set_dft_signal` command to provide such information. The tool then uses this information to perform pre-insertion design rule checking (DRC) to determine the elements that can be incorporated into scan chains.

Typical examples that use the `-view existing_dft` option include

- Clock signals:

```
set_dft_signal -view existing_dft -type ScanClock -port \
     clk -timing {45 55}
```

- Asynchronous set and reset signals:

```
set_dft_signal -view existing_dft -type  Reset -port rst \
     -active_state 0
```

By default, DFT Compiler creates new ports in the design if they are needed. You can specify which existing ports the tool uses to build the DFT structures by using the `-view spec` option with the `set_dft_signal` command.

Typically, the `-view spec` option is used to specify ports that are to function as scan-in and scan-out ports (either dedicated scan-in and scan-out ports or shared functional ports used also as scan-in and scan-out ports), such as

```
set_dft_signal -view spec -type ScanDataIn -port scan_in_1
set_dft_signal -view spec -type ScanDataOut -port scan_out_1
```

and

```
set_dft_signal -view spec -type ScanDataIn -port data_in_bus_2
set_dft_signal -view spec -type ScanDataIn -port data_out_bus_2
```

As a general rule,

- If the information is needed for pre-insertion DRC, then it should be specified by using the `-view existing_dft` option.
- If the information is needed to build DFT structures, then is should be specified by using the `-view spec` option.

Allocating Scan Ports

Ports that are defined to be scan-in and scan-out data ports are used in the order specified by the commands. For example, suppose that you identify three scan-in data ports and three scan-out data ports as follows:

```
set_dft_signal -type ScanDataIn -port [list SIN1 SIN2 SIN3]
set_dft_signal -type ScanDataOut -port [list SOUT1 SOUT2 SOUT3]
```

DFT Compiler allocates the listed ports to scan-in and scan-out functions as follows:

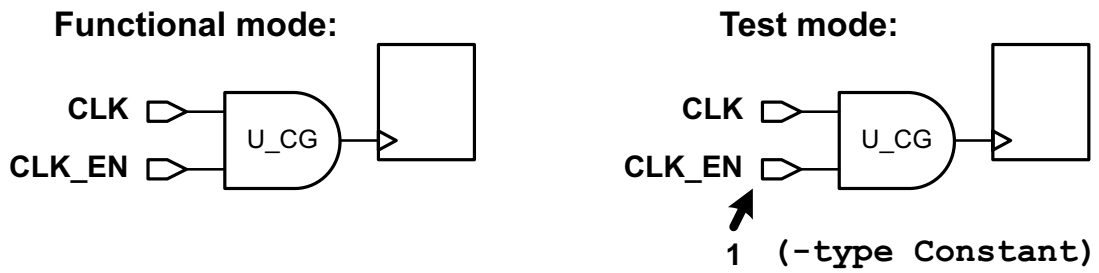
- If you specify two standard scan chains, the tool uses only the first two of the listed scan-in ports, SIN1 and SIN2, and only the first two listed scan-out ports, SOUT1 and SOUT2. SIN3 and SOUT3 are not used for scan chain connection purposes.
- If you specify three standard scan chains, the tool uses all of the listed scan-in and scan-out ports: SIN1, SIN2, SIN3, SOUT1, SOUT2, and SOUT3.
- If you specify four standard scan chains, the tool first uses the three designated scan-in and three designated scan-out ports. For the fourth chain, it creates an additional dedicated scan-in port and an additional scan-out port. The scan-out port can be an existing output port connected to the output of a flip-flop, which can be reused as a scan-out port, or it can be a new dedicated output port.

Using -type Constant versus Using -type TestMode

The difference between `Constant` and `TestMode` signal definitions is as follows:

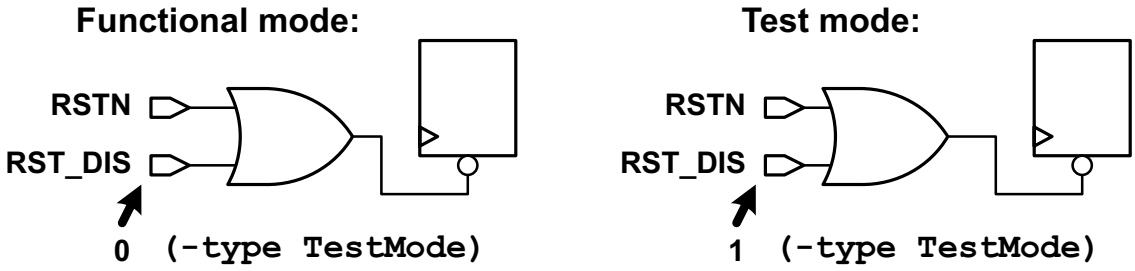
- `Constant` signal definitions specify the value of the signal in test mode, but they do not specify anything about the signal's value in functional mode. Figure 6-10 shows the behavior of a `Constant` signal in the functional and test modes.

Figure 6-10 Constant Signal Specification



- `TestMode` signal definitions specify the value of the signal in both test mode and functional mode. Figure 6-11 shows the behavior of a `Constant` signal in the functional and test modes.

Figure 6-11 Design With a Controlled Clock Signal



This difference affects the generation of verification setup files, which are specified with the `set_svf` command. Verification setup files contain information about the functional mode values for any defined DFT signal. This allows Formality equivalence checking to disable the test logic added by DFT Compiler when the `synopsys_auto_setup` variable is set to `true`.

This difference affects verification setup file generation for the preceding examples as follows:

- The `Constant` signal definition does not result in any verification setup file directives because it contains no information about the functional mode value of CLK_EN.
- The `TestMode` signal definition specifies that the functional mode value for the RST_DIS signal is 1. This results in the following verification setup file directive:

```
guide_scan_input \
  -design { top } \
  -disable_value 0 \
  -ports { RST_DIS }
```

Selecting Test Ports

By default, DFT Compiler creates dedicated test ports as needed, but it also minimizes the number of dedicated test ports by sharing scan outputs with functional ports when the design contains scannable cells that directly drive functional ports.

You can also share ports between test and normal operation, which minimizes the number of dedicated test ports required for internal scan. If your semiconductor vendor does not support this configuration, you can request dedicated scan output ports. Always use dedicated ports for scan-enable and test clock signals.

The following sections describe how to select and define existing ports in your design as test ports:

- Defining Existing Unconnected Ports as Scan Ports
- Sharing a Scan Input With a Functional Port
- Sharing a Scan Output With a Functional Port
- Controlling Subdesign Scan Output Ports

Defining Existing Unconnected Ports as Scan Ports

You can define existing unconnected ports in your RTL description for use as test ports. These are known as placeholder scan ports or dummy scan ports. This approach allows you to use the same testbench for the RTL and gate-level implementations of your design.

Use the `set_dft_signal` command to instruct DFT Compiler to use these ports:

```
dc_shell> set_dft_signal -type ScanDataIn -view spec \
            -port SI1

dc_shell> set_dft_signal -type ScanEnable -view spec \
            -port SE \
            -active_state 1

dc_shell> set_dft_signal -type ScanDataOut -view spec \
            -port SO
```

Sharing a Scan Input With a Functional Port

By default, DFT Compiler always creates a dedicated scan input port. To share a scan input port with a specified existing functional port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -type ScanDataIn -view spec \
            -port DATA_in[0]
```

If you select a bidirectional port as the scan input port, DFT Compiler automatically inserts the necessary bidirectional control logic to enable the input path during scan shift.

Sharing a Scan Output With a Functional Port

By default, if a scannable cell directly drives an output port in the current design, DFT Compiler automatically uses it as the last cell in the scan chain. DFT Compiler disrupts the ordering to place this cell at the end of the scan chain. If multiple scannable sequential cells directly drive output ports, DFT Compiler uses the cell that would have been stitched closest to the end of the scan chain. If the scan cell is the last cell in a scan segment, the entire scan segment is placed at the end of the scan chain. Use the `preview_dft` command to see if a cell or segment has been moved to the end of the scan chain to prevent a dedicated scan output port.

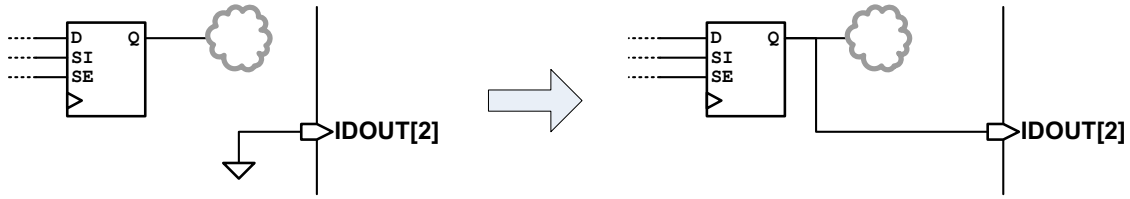
To select the functional port to be used as a scan output port, use the `set_dft_signal` command.

```
dc_shell> set_dft_signal -type ScanDataOut -view spec \
            -port DATA_out[0]
```

If a scannable sequential cell drives the specified output port, DFT Compiler places that cell last in the scan chain. Otherwise, DFT Compiler automatically adds the control or multiplexing logic required to share the scan output port with the functional output port. If you select a bidirectional or three-state port as the scan output port, DFT Compiler automatically inserts the necessary control logic to enable the output path during scan shift.

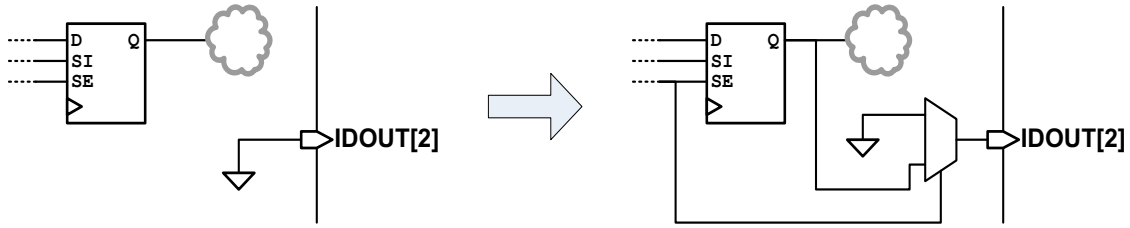
By default, if the specified port is tied to logic 0 or logic 1, DFT Compiler ignores the constant value during scan insertion and drives the port directly as a scan output, as shown in Figure 6-12. This behavior ensures that no additional logic is added at the port if it was undriven in the RTL and tied to logic 0 during synthesis.

Figure 6-12 Scan Output Port With Constant MUXing Disabled



If the existing constant value driving the port is required for proper operation in functional mode, set the `test_mux_constant_so` variable to `true`. In this case, DFT Compiler multiplexes the scan-out signal with the constant value, using the scan-enable signal to control the multiplexer, as shown in Figure 6-13.

Figure 6-13 Scan Output Port With Constant MUXing Enabled



If your semiconductor vendor requires dedicated top-level scan output ports or you prefer them, use the `set_scan_configuration` command to always use dedicated scan outputs:

```
dc_shell> set_scan_configuration \
                -create_dedicated_scan_out_ports true
```

Controlling Subdesign Scan Output Ports

By default, when DFT Compiler routes scan chains through subdesigns, it uses existing subdesign output ports driven by scan cells wherever possible. This minimizes the number of new output ports added to subdesigns, and it can reduce the amount of design uniquification required for multiply-instantiated designs. However, it can also add the loading of external scan routes outside the subdesign to functional nets inside the subdesign.

To always create dedicated scan-out ports on subdesigns, set the following variable:

```
dc_shell> set_app_var test_dedicated_subdesign_scan_outs true
```

Note that this variable setting alone does not guarantee isolation of the functional path from external scan path loading. You must also apply the `set_fix_multiple_port_nets` command to subdesigns where the functional subdesign outputs should be isolated from external scan path loading. For more information on this command, see the man page.

Controlling Scan-Enable Connections to DFT Logic

By default, DFT Compiler uses a global scan-enable signal for DFT logic connections. In some cases, you might want to create multiple scan-enable signals and control how they connect to the DFT logic.

The following sections describe how to control scan-enable connections to DFT logic:

- Associating Scan-Enable Ports With Specific Scan Chains
- Defining Dedicated Scan-Enable Signals for Scan Cells
- Connecting the Scan-Enable Signal in Hierarchical Flows
- Preserving Existing Scan-Enable Pin Connections

Associating Scan-Enable Ports With Specific Scan Chains

To associate a specific port with specific scan chains, use the `set_dft_signal` and `set_scan_path` commands, as follows:

```
dc_shell> set_dft_signal -type ScanEnable -view spec \
             -port port_name -active_state 1

dc_shell> set_scan_path {chain_names} -view spec \
            -scan_enable port_name
```

If the condition set with these commands cannot be met, a warning is issued during scan preview and scan insertion.

Defining Dedicated Scan-Enable Signals for Scan Cells

By default, DFT Compiler chooses an available ScanEnable signal to connect to the scan-enable pins of scan cells. However, you can also define a dedicated ScanEnable signal to use for these scan-enable pin connections.

Specifying a Global Scan-Enable Signal

You can define a global ScanEnable signal to use for the scan-enable pins of scan cells by using the `-usage scan` option when defining the signal with the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable
    -view spec
    -usage scan
    -port port_list
```

To define a signal with the `scan` usage, the `-view` option must be set to `spec`.

When you define a ScanEnable signal with the `scan` usage, the `insert_dft` command is limited to using only that signal to connect to the scan-enable pins of scan cells. If there are insufficient ScanEnable signals for other purposes, DFT Compiler creates additional ScanEnable signals as needed.

You can use the `report_dft_signal` and `remove_dft_signal` commands for reporting and removing the specification, respectively.

Specifying Object-Specific Scan-Enable Signals

You can also define dedicated ScanEnable signals for specific parts of the design by using the `-connect_to` option and associated options of the `set_dft_signal` command:

```
set_dft_signal
    -type ScanEnable
    -view spec
    -usage scan
    -port port_list
    [-connect_to object_list]
    [-connect_to_domain_rise clock_list]
    [-connect_to_domain_fall clock_list]
    [-exclude object_list]
```

The `-connect_to` option specifies a list of design objects that are to use the specified ScanEnable signal. The supported object types are

- Scan cells
- Hierarchical cells
- Designs
- Test clock ports

  This allows you to make clock-domain-based signal connections. It includes scan cells clocked by the specified test clocks. The functional clock behavior is not considered.
- Scan-enable pins of CTL-modeled cores

The `-connect_to_domain_rise` and `-connect_to_domain_fall` options accept a test clock port list and work the same as the `-connect_to` option, except that they apply only to rising-edge and falling-edge scan cells, respectively.

You can also use the `-exclude` option to specify a list of scan cells, hierarchical cells, or design names to exclude from the object-specific control signal.

The following example defines two ScanEnable signals, named SE_CLK1 and SE_CLK2, to connect to the scan-enable pins of test clock domains CLK1 and CLK2, respectively:

```
dc_shell> set_dft_signal -type ScanClock -view existing_dft \
            -port {CLK1 CLK2} -timing {45 55}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port SE_CLK1 -connect_to {CLK1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port SE_CLK2 -connect_to {CLK2}
```

Note the following limitation:

- You cannot specify object-specific scan-enable specifications for pipelined scan-enable signals.

Connecting the Scan-Enable Signal in Hierarchical Flows

When you insert DFT at a top level that contains cores, which are DFT-inserted blocks represented by CTL models, the cores already contain complete scan-enable networks. Instead of connecting the top-level ScanEnable signal to target pins inside the core, DFT Compiler must connect to ScanEnable signal pins at the core boundary.

When ScanEnable signals at the core and/or top level are defined with the `-usage` option of the `set_dft_signal` command, DFT Compiler attempts to determine which top-level signal should drive each core-level signal, using the priorities shown in Table 1-2 on page 1-33.

You can override the default connection behaviors for cores by using object-specific signal definitions at the top level, applied using the `set_dft_signal -connect_to` command and associated options:

```
set_dft_signal
    -type ScanEnable
    -view spec
    -usage scan | clock_gating
    -port port_list
    [-connect_to object_list]
    [-connect_to_domain_rise clock_list]
    [-connect_to_domain_fall clock_list]
    [-exclude object_list]
```

Object-specific specifications are described in "Specifying Object-Specific Scan-Enable Signals" on page 6-35. However, not all object types accepted by the *object_list* argument apply to cores. The object types that apply to cores are

- Test clock ports

  This allows you to make clock-domain-based signal connections to cores. It includes core-level scan-enable pins associated with the specified test clocks. The functional clock behavior is not considered.

- Scan-enable pins of CTL-modeled cores

  This allows you to make direct pin-to-pin connections from top-level signal sources to core-level pins.

Specifying Domain-Based Connections to Core Scan-Enable Pins

When you specify a top-level domain-based signal connection, DFT Compiler uses information inside a core's CTL model to determine the core scan-enable pins associated with each core clock. To ensure that this information is present in the model, the following requirements must be observed during core creation:

- Core-level ScanEnable signals must be defined using the `-usage` option of the `set_dft_signal` command. This ensures that the core's internal scan-enable signal is not used outside its intended usage.

- Core-level ScanEnable signals defined with a usage of `clock_gating` must also be defined as domain-specific signals using the `-connect_to` *`clock_list`* option of the `set_dft_signal` command. This ensures that clock-specific clock-gating annotations are included in the CTL model.

Example 6-3 shows part of a core-level ASCII CTL model that contains clock domain information for a scan-enable signal defined with a usage of `scan`.

Example 6-3 Scan Chain Clock Information in an ASCII CTL Model

```
CTL all_dft {
    ...
    Internal {
        "SE_SCAN" {
            CaptureClock "CLK" {
                LeadingEdge;
            }
            DataType User "ScanEnableForScan" {
                ActiveState ForceUp;
            }
        }
    }
}
```

Example 6-4 shows part of a core-level ASCII CTL model that contains clock domain information for a scan-enable signal defined with a usage of `clock_gating`. The domain-based signal specification causes the CaptureClock constructs to be included.

Example 6-4 Clock-Gating Clock Information in an ASCII CTL Model

```
CTL all_dft {
    ...
    Internal {
        "SE_CG" {
            CaptureClock "CLK" {
                LeadingEdge;
            }
            DataType User "ScanEnableForClockGating" {
                ActiveState ForceUp;
            }
        }
    }
}
```

The CTL model information for a signal can contain multiple DataType constructs (for signals defined with multiple usages) and multiple CaptureClock constructs (for signals associated with multiple clocks).

Consider a core with two scan-enable pins, where pin SE1 is associated with CLK1 and pin SE2 is associated with CLK2. The following top-level commands connect corresponding

top-level scan-enable signals TOP_SE1 and TOP_SE2 to these core-level pins indirectly using domain-based specifications:

```
dc_shell> set_dft_signal -type ScanClock -view existing_dft \
            -port {CLK1 CLK2} -timing {45 55}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port TOP_SE1 -connect_to {CLK1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port TOP_SE2 -connect_to {CLK2}
```

Specifying Connections Directly to Core Scan-Enable Pins

You can specify pin-based signal connection specifications that connect any top-level ScanEnable signal to any core-level ScanEnable pin. These pin-based connection specifications override the default connection behavior, and there is no requirement for the top-level and core-level signal usages to match.

Consider a core with two scan-enable pins, where pin SE1 is associated with CLK1 and pin SE2 is associated with CLK2. The following top-level commands connect top-level scan-enable signals to corresponding core-level pins using direct core pin specifications:

```
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port TOP_SE1 -connect_to {CORE/SE_SCAN1}
dc_shell> set_dft_signal -type ScanEnable -view spec -usage scan \
            -port TOP_SE2 -connect_to {CORE/SE_SCAN2}
```

Preserving Existing Scan-Enable Pin Connections

During DFT insertion, DFT Compiler identifies the scan-enable pins of scan cells and scan cores that should be connected to the global scan-enable signal. These are known as scan-enable target pins.

By default, if a scan-enable target pin already has a connection, DFT Compiler disconnects it to make the connection to a scan-enable signal. During DFT insertion, the `insert_dft` command issues a TEST-394 warning to note the disconnection:

```
Warning: Disconnecting pin 'memwrap/UMEM/SE' to route scan enable.
(TEST-394)
```

To preserve existing connections to scan-enable target pins during DFT insertion, set the following variable:

```
dc_shell> set test_keep_connected_scan_en true
```

In this case, the `insert_dft` command issues a TEST-410 warning to confirm that the existing connection is kept:

```
Warning: Not disconnecting pin 'memwrap/UMEM/SE' to route scan enable.
(TEST-410)
```

For more information, see the man page. For an example application, see SolvNet article 034774, "How To Connect DFT Signals to Hierarchical Pins of Verilog Wrappers."

Suppressing Replacement of Sequential Cells

Use the `set_scan_element` command to determine whether specific sequential cells are to be replaced by scan cells that become part of the scan path during the `insert_dft` command.

For full-scan designs, the `insert_dft` command replaces all nonviolated sequential cells with equivalent scan cells by default. Therefore, you do not need to set the `scan_element` attribute unless you want to suppress replacement of sequential cells with scan cells. To prevent such replacement for certain cells, set the `scan_element` attribute to `false` for those cells.

Note:

If you want to specify which scan cells are to be used for scan replacement, use the `set_scan_register_type` command.

You should not use the `set_scan_element true` command if you use the `compile -scan` command to replace elements.

In Logic Scan Synthesis

In logic scan synthesis, the `set_scan_element false` command unscans the cell on a design in which scan replacement has already occurred.

Changing the Scan State of a Design

In certain circumstances, you might find it necessary to manually set the scan state of a design. Use the `set_scan_state` command to do so. The `set_scan_state` command has three options: `unknown`, `test_ready`, and `scan_existing`.

If there are nonscan elements in the design, use the `set_scan_element false` command to properly identify them.

You can check the test state of the design by using the `report_scan_state` command.

One situation in which you would set the scan state is if you needed to write a netlist of a test-ready design and read it into a third-party tool. After making modifications, you can bring the design back into DFT Compiler as shown in Example 6-5.

Example 6-5 Changing the Scan State of a Design

```
dc_shell> read_file -format verilog my_design.v

dc_shell> report_scan_state

****************************************
Report : test
         -state
Design : MY_DESIGN
Version: 2002.05
Date   : Wed Jul 25 18:12:39 2001
****************************************

Scan state                   : unknown scan state

1
dc_shell> set_scan_state test_ready
Accepted scan state.
1
dc_shell> report_scan_state

****************************************
Report : test
         -state
Design : MY_DESIGN
Version: 2002.05
Date   : Wed Jul 25 18:14:47 2001
****************************************

Scan state                   : scan cells replaced with loops
```

Important:
You do not need to set the scan state if you are following the recommended design flow.

Removing Scan Configurations

The `reset_scan_configuration` command removes scan specifications from the current design. Note that this command deletes only those specifications you defined with the `set_scan_configuration` command.

Specifications defined using other commands are removed by issuing the corresponding remove command. For example, you use the `remove_scan_path` command to remove the path specifications you defined with the `set_scan_path` command.

Note that the `reset_scan_configuration` command does not change your design. It merely deletes specifications you have made.

You can use the `reset_scan_configuration` command to remove explicit specifications of synthesizable segments. When you remove an explicit specification, the multibit component inherits the current implicit specification.

Note:

The `reset_scan_configuration` command does not affect the settings made with the `set_scan_register_type` command. These settings must be removed with the `remove_scan_register_type` command.

Keeping Specifications Consistent

The set of user specifications contributing to the definition of the scan design must be consistent. User-supplied specification commands forming part of a consistent specification have the following characteristics:

- Each specification command is self-consistent. It cannot contain mutually exclusive requirements. For example, a command specifying the routing order of a scan chain cannot specify the same element in more than one place in the chain.
- All specification commands are mutually consistent. Two specification commands must not impose mutually exclusive conditions on the scan design. For example, two specification commands that place the same element in two different scan chains are mutually incompatible.
- All specification commands yield a functional scan design. You cannot impose a specification that leads to a nonfunctional scan design. For example, a specification that mandates fewer scan chains than the number of incompatible clock domains is not permitted.

The number of clock domains in your design, together with your clock-mixing specification, determines the minimum number of scan chains in your design. If you specify an exact number of scan chains smaller than this minimum, the `insert_dft` command issues a warning message and implements the minimum number of scan chains.

Synthesizing Three-State Disabling Logic

DFT Compiler can, by default, handle three-state nets. It does so with the following functionality:

- By default, it distinguishes between internal and external three-state nets.
- By default, it prevents bus contention by causing only one three-state driver to be active at one time.
- By default, it modifies internal three-state nets in bottom-up design methodology to make exactly one three-state driver active.

To prevent bus contention or bus float, internal three-state nets in your design must have a single active driver during scan shift. DFT Compiler automatically performs this task.

DFT Compiler determines if the internal three-state nets in your design meet this requirement.

By default, DFT Compiler adds disabling logic to internal three-state nets that do not meet this requirement. The scan-enable signal controls the disabling logic and forces a single driver to be active on the net throughout scan shift.

In some cases, DFT Compiler adds redundant disabling logic because the disabling logic checks for internal three-state nets are limited.

Figure 6-14 shows the simple internal three-state net used as an example throughout this section.

Figure 6-14 Internal Three-State Net Example

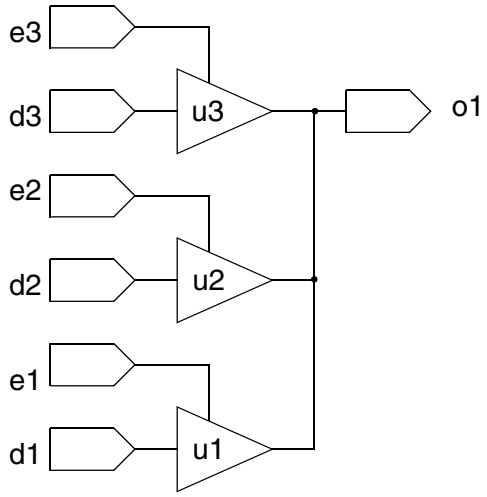
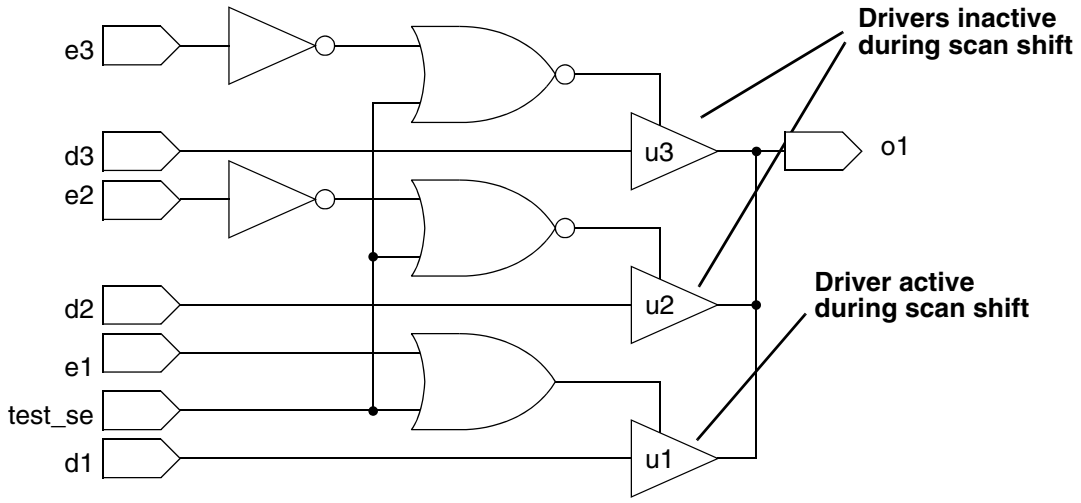


Figure 6-15 shows the disabling logic added by DFT Compiler during the `insert_dft` process.

Figure 6-15 Three-State Output With Disabling Logic



If the design already contains logic that prevents or can be configured to prevent the occurrence of bus contention and bus float during scan shift, you can use the `set_dft_configuration` command to prevent DFT Compiler from inserting the disabling logic:

```
dc_shell> set_dft_configuration -fix_bus disable
```

During scan shift, DFT Compiler does not check for bus contention or bus float conditions. If you do not add the three-state disabling logic, verify that no invalid conditions occur during scan shift.

If you want to perform bottom-up scan insertion, you must choose a strategy for handling the insertion of three-state enabling and disabling logic. If you use the `set_dft_configuration -fix_bus disable` command, your design will be free of bus float or bus contention during scan shift. However, during bottom-up scan insertion, the `insert_dft` command might be forced to modify modules that it has already processed.

This strategy is easy to implement in scripts but can result in repeated modifications to subblocks. Note that DFT Compiler does recognize three-state enabling and disabling logic that it has previously inserted in a submodule and so does not insert unnecessary or redundant enabling and disabling logic.

For example, consider a top-level design with two instances of a module of type sub_type_1. Both of these instances drive a three-state bus that, in turn, drives inputs on another module. If you perform scan insertion with default settings on the design sub_type_1, then in the top design, the three-state ports that drive this common bus will be turned off in scan shift, thus creating a float condition. In other words, when you run the `insert_dft` command at the top level with default options selected, the `insert_dft` command modifies one of the two

instances of sub_type_1. As a result, each net within the bus has a single enabled driver during scan shift.

You can consider two other, nondefault, strategies when you want to use bottom-up scan insertion.

You can synthesize three-state disabling logic at the top level only. Synthesis of disabling logic at the top level guarantees a consistent implementation across all subdesigns. Use the `set_dft_configuration -fix_bus disable` command to disable synthesis of three-state disabling logic in subdesigns.

```
/* subdesign command sequence */
dc_shell> current_design subdesign
dc_shell> set_dft_configuration -fix_bus disable
...
dc_shell> insert_dft

/* top-level command sequence */
dc_shell> current_design top
dc_shell> set_dft_configuration -fix_bus enable
...
dc_shell> insert_dft
```

A third option is to use the `preview_dft -show tristates` command before you run the `set_dft_configuration` command on each submodule to determine what enabling and disabling logic should be inserted on the external three-state nets for each module. This strategy is the most complex to use, and your scripts need to be specific to each design. However, if you implement this method correctly, you can assemble submodules into a complete testable design without further modification of a submodule by the `set_dft_configuration` command.

Configuring Three-State Buses

The `set_dft_configuration` command can configure three-state buses according to settings applied by the `set_autofix_configuration` command.

If the `-fix_bus` option of the `set_scan_configuration` command is set to `disable`, no changes to the three-state driver logic are made, regardless of any other three-state settings.

Configuring External Three-State Buses

On external three-state nets, the `-type external_bus` option of the `set_autofix_configuration` command controls three-state disabling behavior. If you want to make no changes to the external three-state nets, use the `-method no_disabling` option. If you want to allow exactly one three-state driver to be enabled on each external

three-state net, you can use the `-method enable_one` option. If you want to ensure that all external three-state nets are disabled, use the `-method disable_all` option, which is the default behavior for the `external_tristates` type.

You might have multiple modules that are stitched together at the top level, and you might want to be sure that one of those modules contains the active three-state drivers while the other modules are all off. You can do that by using a bottom-up scan insertion methodology and by setting the `set_autofix_configuration` command appropriately for each module before you run the `insert_dft` command on that module.

Configuring Internal Three-State Buses

The same rules apply for internal three-state nets as for external three-state nets. If you allow all your subdesigns to be set to the default behavior, `insert_dft` can choose a three-state driver on the net to make active and can disable all others.

Overriding Global Three-State Bus Configuration Settings

You can override these internal and external three-state net settings by using the `set_autofix_element` command, which can be applied to individual nets in your design.

This command applies only to the nets and not to individual three-state drivers.

You might have a situation in which multiple instances of the same design must have separate three-state configuration settings. You can achieve this by uniquifying the particular instances and then using the `set_autofix_element` command to define the type of enabling or disabling logic you want to see applied on that instance.

Disabling Three-State Buses and Bidirectional Ports

There are several different methods you can use to disable logic to ensure that three-state buses and bidirectional ports are properly configured during scan shift:

- To set the default behavior for top-level three-state specifications, use the following command:

```
set_dft_configuration \
   -fix_bus enable | disable
```

- To set the default behavior for top-level bidirectional port specifications, use the following command:

```
set_dft_configuration \
   -fix_bidirectional enable | disable
```

- To set global three-state specifications, use the following command:

```
set_autofix_configuration \
   -type internal_bus | external_bus \
   -method disable_all | enable_one | no_disabling
```

- To set global bidirectional port specifications, use the following command:

```
set_autofix_configuration \
   -type birectional \
   -method input | output | no_disabling
```

- To set local three-state specifications on a specific list of objects, use the following command:

```
set_autofix_element \
   -type internal_bus | external_bus \
   -method input | output | no_disabling \
   object_list
```

- To set local birectional port specifications on a specific list of objects, use the following command:

```
set_autofix_element \
   -type bidirectional \
   -method input | output | no_disabling \
   object_list
```

Handling Bidirectional Ports

Every semiconductor vendor has specific requirements regarding the treatment of bidirectional ports during scan shift. Some vendors require that bidirectional ports be held in input mode during scan shift, some require that bidirectional ports be held in output mode during scan shift, and some have no preference. DFT Compiler provides the ability to set the bidirectional mode both globally and individually.

Before you insert control logic for bidirectional ports, understand your vendor's requirements for these cells during scan shift.

If the `-fix_bidirectional disable` option of the `set_dft_configuration` command is set, no disabling logic is added to any bidirectional ports, regardless of any other bidirectional port settings.

Setting Individual Bidirectional Port Behavior

To specify bidirectional behavior on individual ports, use the `set_autofix_element` command.

Use the `reset_autofix_element` command to remove all `set_autofix_element` specifications for the current design.

Use the `preview_dft` command to see the bidirectional port conditioning that will be implemented for each bidirectional port in a design.

Fixed Direction Bidirectional Ports

Bidirectional ports that have enables connected to constant values and that are therefore always configured in either input mode or output mode are referred to as degenerated bidirectional ports. DFT Compiler does not add control logic for degenerated bidirectional ports.

DFT Compiler recognizes constant values on the enable pins of bidirectional ports for the following cases:

- Enable forced to a constant value by a tie-off cell in the circuit
- Enable forced to a constant value by a `set_dft_signal` command

Assigning Test Port Attributes

If you always save and read mapped designs in the .ddc format, you usually do not need to explicitly set `signal_type` attributes. If you do not save your design in .ddc format, you must use the `set_dft_signal` command.

Note:

Use the `set_dft_signal` command for scan-inserted, existing-scan, and test-ready designs.

When `insert_dft` sets attributes on test ports, for all scan styles, it creates the following values:

- It places either a `test_scan_enable` or a `test_scan_enable_inverted` attribute on scan-enable ports. The `test_scan_enable` attribute causes a logic 1 to be applied to the port for scan shift. The `test_scan_enable_inverted` attribute causes a logic 0 to be applied to the port for scan shift.
- Scan input ports are identified with the `test_scan_in` attribute.
- Scan output ports are identified with the `test_scan_out` or `test_scan_out_inverted` attribute.

Note that some scan styles require test clock ports on the scan cell.

Architecting Test Clocks

When DFT Compiler creates a test protocol, it uses defaults for the clock timing, based on the clock type, unless you explicitly specify clock timing.

This section shows you how to set test clocks and handle multiple clock designs. It include the following subsections:

- Defining Test Clocks
- Requirements for Valid Scan Chain Ordering
- Adding Lock-Up Latches Between Clock Domains
- Adding Lock-Up Latches Within Clock Domains
- Assigning Scan Chains to Specific Clocks
- Handling Multiple Clocks in LSSD Scan Styles

Defining Test Clocks

To explicitly define test clocks in your design, use the `set_dft_signal` command. For example,

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
            -port CLK -timing {45 55}
```

Specify the clock signal type with the `-type` option. For the multiplexed flip-flop style, use the `ScanClock` type. For other scan styles, see the man page.

Define the test clock waveform with the `-timing` option. The waveform definition is a pair of values that specifies the rising-edge arrival time followed by the falling-edge arrival time. Figure 6-16 shows a return-to-zero clock waveform definition.

Figure 6-16 Return-to-Zero Test Clock Waveform Definition

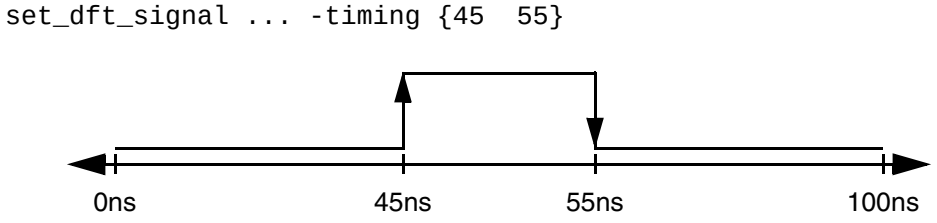
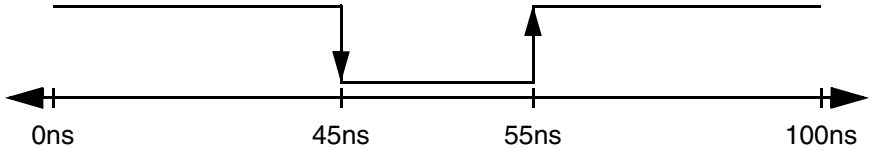


Figure 6-17 shows a return-to-one clock waveform definition.

Figure 6-17 Return-to-One Test Clock Waveform Definition

```
set_dft_signal ... -timing {55  45}
```



If you use the `-infer_clock` option of the `create_test_protocol` command to infer test clocks in your design, the tool uses the default clock waveforms shown in Table 6-4.

Table 6-4 Default Test Clock Waveform Timing

Scan clock type	First edge (ns)	Second edge (ns)
Edge-triggered (non-LSSD styles)	45.0	55.0
Master clock (LSSD styles)	30.0	40.0
Slave clock (LSSD styles)	60.0	70.0

For all test clocks, the clock period is the value defined by the `test_default_period` variable.

After defining or inferring your test clocks, you can verify their timing characteristics by using the `report_dft_signal` command.

The rise and fall clock waveform values are the same as the values specified in the statements that make up the STIL waveform section. The rise argument becomes the value of the rise argument in the waveform statement in the test protocol clock group. The fall argument becomes the value of the fall argument in the waveform statement in the test protocol clock group.

Specifying a Hookup Pin for DFT-Inserted Clock Connections

In some cases, DFT Compiler might need to make a connection to an existing scan clock network during DFT insertion. Some examples are

- Pipeline clock connections for automatically inserted pipelined scan data registers
- ATE clock connections for DFT-inserted OCC controllers
- Codec clock connections for serialized compressed scan

By default, DFT Compiler makes the clock connection at the source port specified in the `-view existing_dft` signal definition. However, if you want DFT Compiler to make the

clock connection at an internal pin, such as a pad cell or clock buffer output, you can specify it with the `-hookup_pin` option in a subsequent `-view spec` signal definition. For example,

```
dc_shell> set_dft_signal -view existing_dft -type ScanClock \
            -port CLK -timing {45 55}
dc_shell> set_dft_signal -view spec -type ScanClock \
            -port CLK -hookup_pin UCLKBUF/Z
```

You do not specify the clock waveform timing for the `-view spec` signal definition.

Requirements for Valid Scan Chain Ordering

This section describes the requirements for valid scan chain ordering in the multiplexed flip-flop scan style.

DFT Compiler generates valid mixed-clock scan chains based on the ideal test clock timing. Scan chain cells are ordered by the ideal test clock edge times, as defined with the `-timing` option of the `set_dft_signal` command. Cells clocked by later clock edges are placed before cells clocked by earlier clock edges. This guarantees that all cells in the scan chain get the expected data during scan shift.

Figure 6-18 shows the ideal test clock waveforms for two test clocks. The clock edges are numbered by their edge timing order, with the latest clock edge indicated by (1).

Figure 6-18 Ideal Test Clock Waveforms for Two Test Clocks

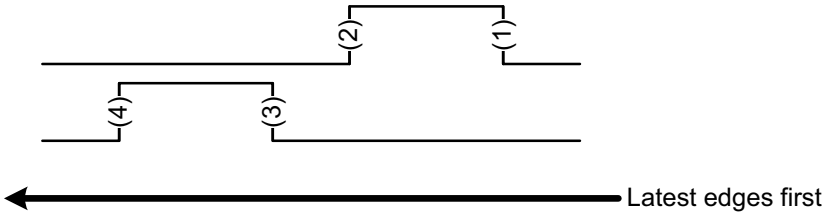
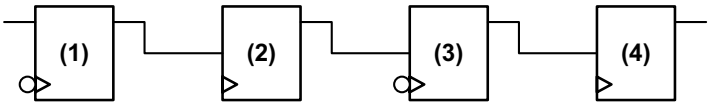


Figure 6-19 shows how DFT Compiler constructs a scan chain containing a scan cell clocked by each clock edge. The scan cells are ordered with the cells clocked by the latest clock edges coming first.

Figure 6-19 Scan Chain Cells for Two Test Clocks



To maintain the validity of your scan chains, do not change the test clock timing after assembling the scan structures.

Although DFT Compiler chooses an order that ensures correct shift function under ideal clock timing, it cannot guarantee that capture problems will not occur. Capture problems are

caused by your logic functionality; modify your design to correct capture problems. For more information, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

By default, when you request clock mixing within a multiplexed flip-flop scan chain, DFT Compiler inserts lock-up latches to prevent timing problems. For more information, see “Adding Lock-Up Latches Between Clock Domains” on page 6-51.

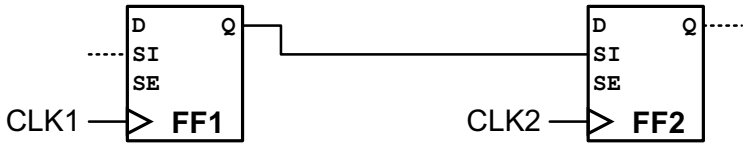
Adding Lock-Up Latches Between Clock Domains

This section describes how the tool adds lock-up latches between clock domains in the multiplexed flip-flop scan style.

A *scan lock-up latch* is a retiming sequential cell on a scan path that can address skew problems between adjacent scan cells when clock mixing or clock-edge mixing is enabled. DFT Compiler inserts them to prevent skew problems that might occur.

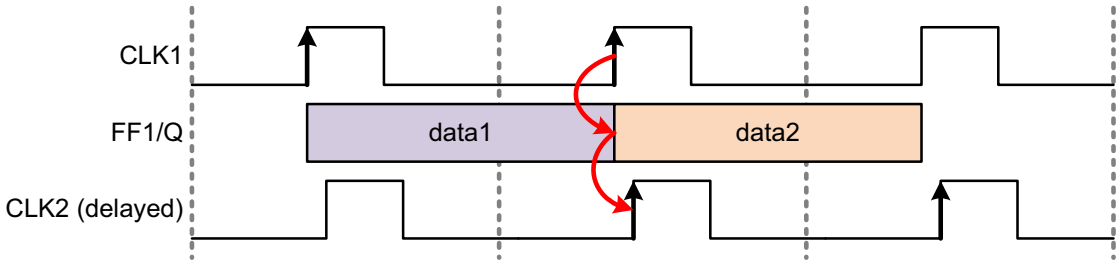
Consider the scan structure in Figure 6-20, where a scan cell clocked by CLK1 feeds a scan cell clocked by CLK2, and both clocks are defined with the same ideal waveform definition.

Figure 6-20 Two Scan Cells Clocked By Two Different Clocks



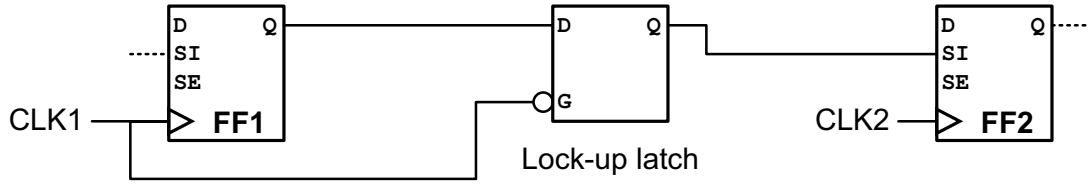
If both scan cells receive a clock edge at the same time, no timing violations occur. However, if the CLK2 waveform at FF2 is delayed, perhaps due to higher clock tree latency, a hold violation might result where FF2 incorrectly captures the current cycle’s data instead of the previous cycle’s data. Figure 6-21 shows this hold violation for leading-edge scan cells.

Figure 6-21 Timing for Two Leading-Edge Scan Cells Clocked By Two Different Clocks



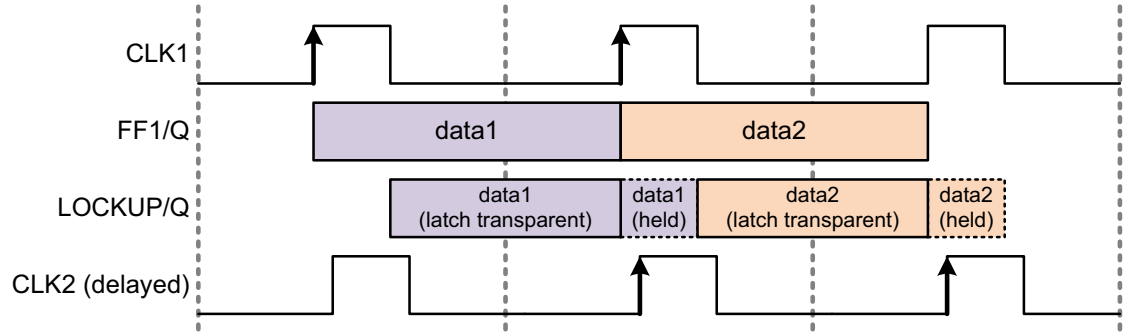
A lock-up latch prevents hold violations for scan cells that might capture data using a skewed clock edge. It is a latch cell that is inserted between two scan cells and clocked by the inversion of the previous scan cell’s clock. Figure 6-22 shows the same two scan cells with a lock-up latch added.

Figure 6-22 Two Scan Cells With a Lock-Up Latch



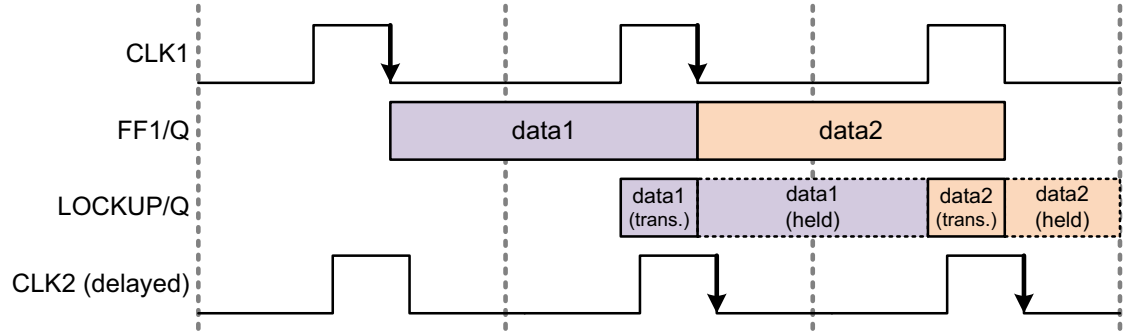
The lock-up latch cell works by holding the previous cycle's scan data while the current cycle's scan data is captured, effectively delaying the output data transition to the next edge of the source clock. Figure 6-23 shows the lock-up timing behavior for the example. Although this example uses return-to-zero clock waveforms, lock-up latch operation is similar for return-to-one clock waveforms.

Figure 6-23 Timing for Two Leading-Edge Scan Cells With a Lock-Up Latch



Lock-up latch operation for trailing-edge scan cells is similar to that of leading-edge scan cells, except that the data is held into the next clock cycle as shown in Figure 6-24.

Figure 6-24 Timing for Two Trailing-Edge Scan Cells With a Lock-Up Latch



By default, DFT Compiler adds scan lock-up latches as needed to multiplexed flip-flop scan chains. Scan chain cells are ordered by the ideal test clock edge times, as defined with the `-timing` option of the `set_dft_signal` command. Cells clocked by later clock edges are

placed before cells clocked by earlier clock edges. Adjacent scan chain cells clocked by different clock edges are handled as follows:

- When the scan cells are clocked by clock edges with different ideal clock edge timing, DFT Compiler does not insert a lock-up latch. The second scan cell captures data using an earlier clock edge, and DFT Compiler assumes this difference in the ideal clock edge timing is sufficient to avoid a hold time violation.
- When the scan cells are clocked by clock edges with identical ideal clock edge timing, DFT Compiler inserts a lock-up latch to avoid a potential hold violation due to clock skew.

Note:

DFT Compiler assumes that any clock skews are smaller than the edge differences in the ideal test clock edge times. In Figure 6-23, if CLK2 is skewed later than CLK1 by more than the active-high pulse width of CLK1, a hold violation can still occur.

Figure 6-25 shows a set of ideal test clock waveforms for a set of overlapping test clocks. The clock edges are numbered by their edge timing order, with the latest clock edge indicated by (1). Clock edges with identical ideal clock edge timing are highlighted.

Figure 6-25 Ideal Test Clock Waveforms for Overlapping Test Clocks

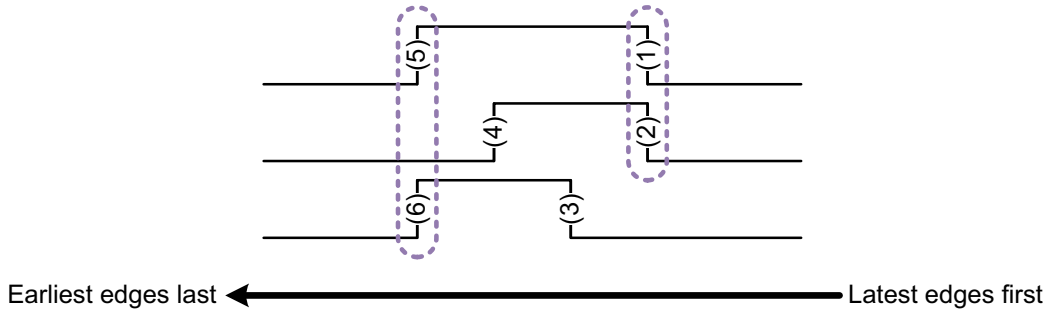
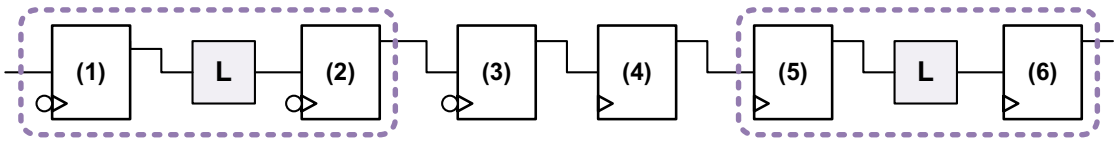


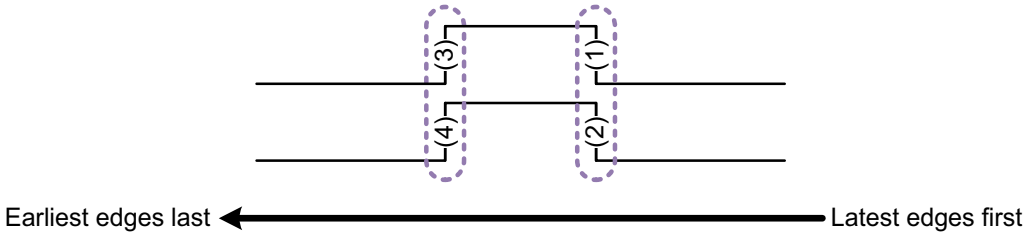
Figure 6-26 shows how DFT Compiler constructs a scan chain containing a scan cell clocked by each clock edge. The scan cells are ordered with the cells clocked by the latest clock edges coming first. Lock-up latches are inserted between scan cells clocked by the clock edges with identical ideal clock edge timing.

Figure 6-26 Scan Chain Lock-Up Latches for Overlapping Test Clocks



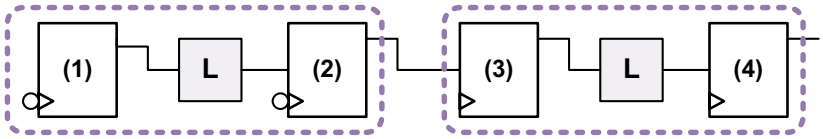
However, in most designs, all test clocks share identical return-to-zero test clock waveforms, as shown in Figure 6-27.

Figure 6-27 Ideal Test Clock Waveforms for Simple Test Clocks



In this case, the ordering behavior is simplified. The scan cells are ordered with all falling-edge scan cells first and all rising-edge scan cells last, as shown in Figure 6-28. Lock-up latches are inserted between differently-clocked scan cells within the rising-edge and falling-edge sections of the scan chain.

Figure 6-28 Scan Chain Lock-Up Latches for Simple Test Clocks



DFT Compiler inserts a lock-up latch at the same level of hierarchy as the scan output pin of the preceding scan element:

- If the preceding element is a CTL model or is located in a block containing CTL model information, the lock-up latch is inserted at the level of hierarchy where the CTL model exists.
- If the preceding element is a leaf scan cell (that does not exist in a CTL-modeled block), the lock-up latch is inserted at the level of hierarchy where the scan cell exists.

The `set_scan_configuration` command provides options to control lock-up latch insertion. By default, DFT Compiler performs automatic lock-up latch insertion for multiplexed flip-flop scan chains. To disable this feature, use the `-add_lockup` option of the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration -add_lockup false
```

To add lock-up latches at the end of each scan chain to assist with potential block-to-block timing issues during core integration, use the `-insert_terminal_lockup` option of the `set_scan_configuration` command:

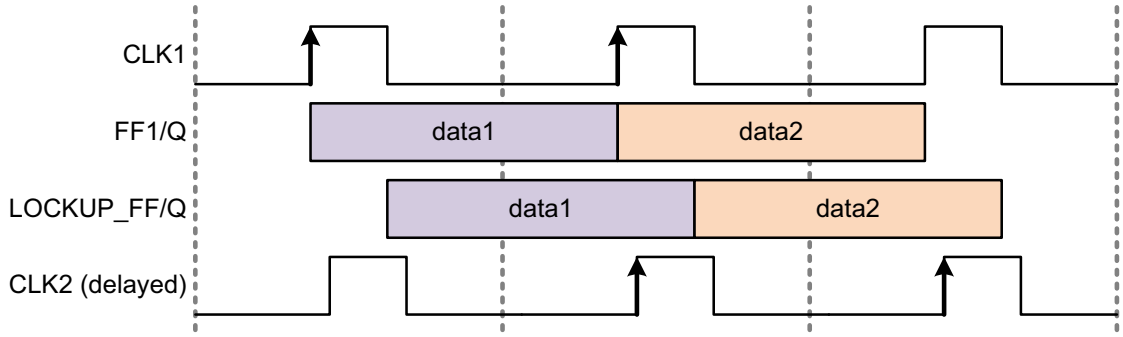
```
dc_shell> set_scan_configuration -insert_terminal_lockup true
```

The default lock-up element type is a level-sensitive lock-up latch. To use a lock-up flip-flop instead, use the `-lockup_type` option of the `set_scan_configuration` command:

```
dc_shell> set_scan_configuration -lockup_type flip_flop
```

When a lock-up flip-flop is used, the data is held as shown in Figure 6-29.

Figure 6-29 Timing for Two Scan Cells With a Lock-Up Flip-Flop



Regardless of your selected scan style or configuration, you can explicitly add scan lock-up elements to your scan chain by using the `set_scan_path` command.

For successful lock-up operation, the falling edge of the current scan cell must occur after or concurrent with the rising edge of the next scan cell. This requirement is always inherently met when DFT Compiler inserts lock-up elements between scan chain cells. However, when you are manually inserting lock-up elements with the `set_scan_path` command, you must ensure that this requirement is met.

Use the `preview_dft -show cells` command to see where the `insert_dft` command will insert scan lock-up elements in your scan chain:

```
dc_shell> preview_dft
...
Scan chain '1' (test_si --> Z[3]) contains 4 cells:

  Z_reg[0]                        (CLK1, 45.0, rising)
  Z_reg[1] (l)
  Z_reg[2]                        (CLK2, 45.0, rising)
  Z_reg[3]
```

You can also use the `scan_lockup` cell attribute to locate lock-up elements:

```
dc_shell> set lockup_cells \
            [get_cells -hierarchical * -filter {scan_lockup==true}]
```

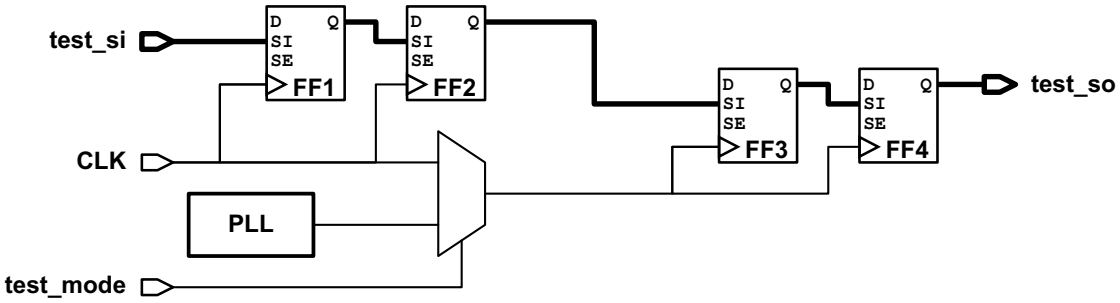
Adding Lock-Up Latches Within Clock Domains

This section describes how the tool adds lock-up latches within clock domains in the multiplexed flip-flop scan style.

For the purpose of building scan chains, the `insert_dft` command, by default, treats all internal clock signals driven by the same top-level clock port as the same clock signal.

Consider the netlist shown in Figure 6-30, which shows a clock network structure before clock tree synthesis. By default, the `insert_dft` command treats all four flip-flops as belonging to the same top-level clock, CLK.

Figure 6-30 Circuit With Same Top-Level Clock Driving Internal Clock Signals




```
set_scan_configuration \
  -internal_clocks none  ;# default setting
```

Note that the MUX cell introduces a delay in the clock network. If clock tree synthesis balances the test mode clock latency equally to all flip-flops, the MUX cell should not cause any timing problems. However, since clock tree synthesis might not consider the test mode clock tree latencies used for scan shift, a potential scan path hold violation could occur at FF3/SI.

In this case, you can avoid creating the potential hold time violation by treating the flip-flops clocked by the MUXed clock as if they belonged to a different clock domain. To do this, use the following command:

```
dc_shell> set_scan_configuration -internal_clocks multi
```

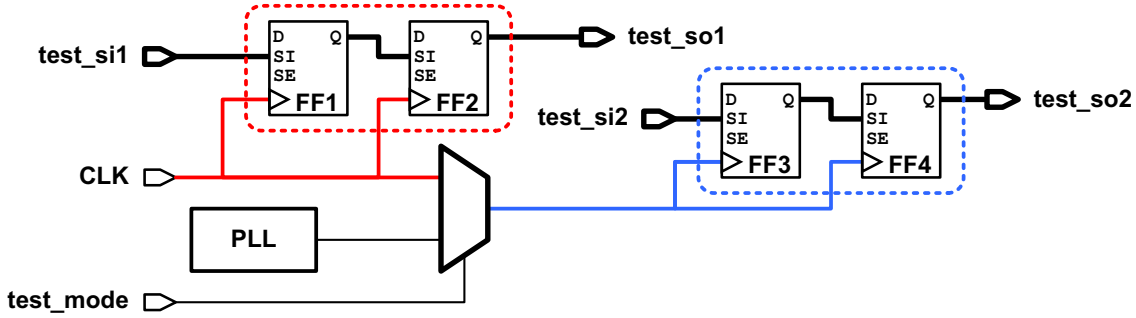
This command instructs the `insert_dft` command to treat internal clock network regions driven by multiple-input gates, such as MUX cells, as separate clocks for the purposes of scan chain architecture. This feature only affects scan chain architecture; the clock network is still a single test clock domain for all other DFT operations.

Note:

Integrated clock-gating cells, which are combinational or sequential cells that have the `clock_gating_integrated_cell` attribute defined, are not considered during internal clock analysis; they are transparent for the determination of internal clocks.

The resulting scan chains depend on the current clock-mixing setting, which is controlled by the `-clock_mixing` option of the `set_scan_configuration` command. If clock mixing is disabled by specifying the `no_mix` or `mix_edges` clock-mixing mode, the `insert_dft` command creates separate scan chains for each internal clock, as shown in Figure 6-31.

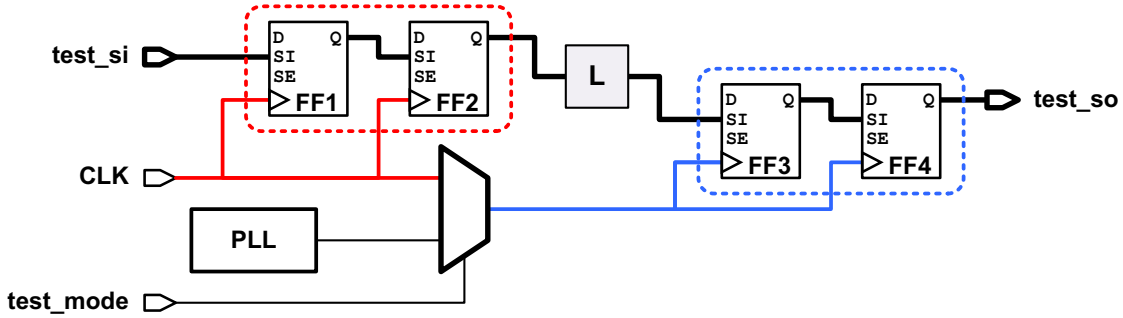
Figure 6-31 Circuit With Internal Clocks With Clock Mixing Disabled




```
set_scan_configuration \
  -internal_clocks multi  -clock_mixing no_mix | mix_edges
```

If clock mixing is enabled by specifying the `mix_clocks` or `mix_clocks_not_edges` clock-mixing mode, the `insert_dft` command can use lock-up latches to keep the scan cells from different internal clocks on the same scan chain, as shown in Figure 6-32.

Figure 6-32 Circuit With Internal Clocks With Clock Mixing Enabled




```
set_scan_configuration \
  -internal_clocks multi -clock_mixing mix_clocks | mix_clocks_not_edges
```

If clock mixing is disabled, you can still enable internal clock domain mixing with the `-mix_internal_clock_driver` option of the `set_scan_configuration` command, which enables clock mixing only for internal clocks:

```
dc_shell> set_scan_configuration \
            -internal_clocks multi \
            -clock_mixing no_mix | mix_edges \
            -mix_internal_clock_driver true
```

You can specify internal clock settings on individual clock domains by using the `-internal_clocks` option of the `set_dft_signal` command. The following command tells

the insert_dft command to treat only internal clocks driven by multiple-input cells as separate clocks in the CLK domain:

```
dc_shell> set_dft_signal -view existing_dft \
                -type ScanClock -timing [list 45 55] \
                -internal_clocks multi -port CLK
```

If you set different -internal_clocks values using the set_scan_configuration and set_dft_signal commands, the more specific setting applied with the set_dft_signal command takes precedence. For example, assume that you set the following opposing -internal_clocks values by using these two commands:

```
dc_shell> set_scan_configuration -internal_clocks none

dc_shell> set_dft_signal -view existing_dft \
                -type ScanClock -timing [list 45 55] \
                -internal_clocks multi -port CLK
```

Because the value set by the set_dft_signal command takes precedence, signals driven by CLK via MUX cells or other multiple-input gates are treated as separate clocks. All other clocks in the design are treated according to the default configuration.

Note:

The -internal_clocks option affects only scan chain building for the multiplexed flip-flop scan style.

When you use user-defined test points or test points inserted by AutoFix, testability logic might be inserted in the clock network. The preview_dft command does not see internal clocks created by test points, but the insert_dft command does. For more information on test points, see Chapter 7, “Advanced DFT Architecture Methodologies.

Assigning Scan Chains to Specific Clocks

You might want to allocate specific clocks to given scan chains. By doing this, you gain additional control over scan chain specifications. To associate a scan chain to a clock domain, use the set_scan_path command.

```
dc_shell> set_scan_path chain_name \
                -view spec -scan_master_clock clock_name
```

If you also use the -exact_length option to define the number of scan cells to be included in that scan chain, DFT Compiler includes additional scan cells clocked by other clocks if the clock-mixing requirements allow.

If you use the -edge option with the -scan_master_clock option when defining a scan path using the set_scan_path command, the tool includes only the elements controlled by the specified edge of the scan_master_clock in the scan chain. The valid arguments to the -edge option are rising and falling.

For example,

```
dc_shell> set_scan_path c1 \
               -scan_master_clock clk1 -edge rising -view spec
```

In this example, the scan chain c1 will contain elements that are triggered by the rising edge of clock clk1.

Note the following limitations to the `-edge` option:

- When there are no elements present for the defined scan chains, the scan path name is reused.
- This option is not supported with multivoltage designs.
- This option is not supported with the other `set_scan_path` options, such as the `-head`, `-tail`, `-ordered`, and `-length` options.

If the specifications in the `set_scan_path` command cannot be met, they are not applied.

Handling Multiple Clocks in LSSD Scan Styles

This section provides information on handling multiple clocks in level-sensitive scan designs (LSSD) scan styles.

Using Multiple Master Clocks

In LSSD scan designs, you need not allocate scan chains by clock for timing purposes; however, you might want to do so. Assume that you have a latch-based design with two system enables, en1 and en2, and you want a scan chain allocated for each enable. The command sequence given in Example 6-6 accomplishes this.

Example 6-6 Command Sequence for Multiple Master Clocks in LSSD

```
dc_shell> set_scan_configuration -style lssd

# create test A clock ports and assign to scan chains
dc_shell> create_port -direction in {A_CLK1 A_CLK2}
dc_shell> set_dft_signal -view spec -port A_CLK1 \
               -type ScanMasterClock
dc_shell> set_scan_path 1 -view spec \
               -scan_master_clock A_CLK1
dc_shell> set_dft_signal -view spec -port A_CLK2 \
               -type ScanMasterClock
dc_shell> set_scan_path 2 -view spec \
               -scan_master_clock A_CLK2

# explicitly allocate cells to scan chains by system enable
dc_shell> create_clock en1 -name cclk1 -period 100
dc_shell> set cclk1_cells [get_object_name [all_registers -clock cclk1]]
```

```
dc_shell> set_scan_path 1 -include_elements $cclk1_cells
dc_shell> create_clock en2 -name cclk2 -period 100
dc_shell> set cclk2_cells [get_object_name [all_registers -clock cclk2]]
dc_shell> set_scan_path 2 -include_elements $cclk2_cells

# preview scan configuration and implement
dc_shell> create_test_protocol
dc_shell> dft_drc
dc_shell> preview_dft -show all
dc_shell> insert_dft
```

Dedicated Test Clocks for Each Clock Domain

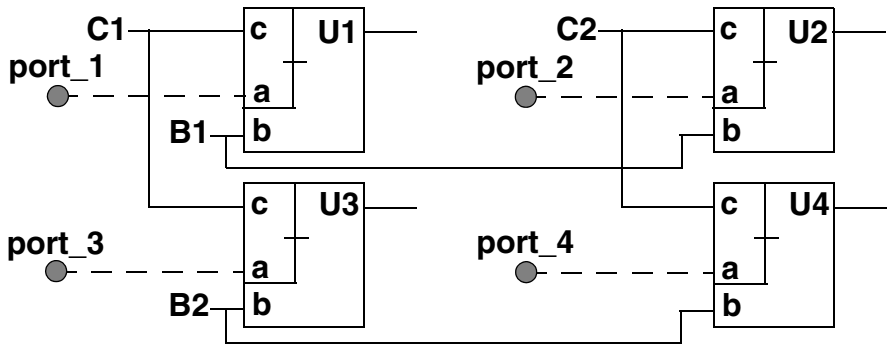
The `insert_dft` command creates clocks that are used only for test purposes when it routes scan chains by using the following scan styles:

- LSSD (which includes clocked LSSD)
- Scan-enabled LSSD

The test clocks are dedicated for each system clock domain. This makes clock trees and clock signal routing easier. The `insert_dft` command uses the following guidelines to determine how test clocks are added:

- For sequential cells with multiple test clocks, the `insert_dft` command adds a test clock for each unique set of master and slave system clocks. For example, in Figure 6-33, cell U1 is clocked by C1 (master) and B1 (slave), cell U2 is clocked by C2 and B1, cell U3 is clocked by C1 and B2, and cell U4 is clocked by C2 and B2, resulting in four unique clock sets. As a result, the `insert_dft` command adds four test clocks, one for each unique clock set.

Figure 6-33 Adding Test Clocks for Sequential Cells With Multiple Test Clocks



- For cells that are clocked by the same system clock, the `insert_dft` command adds the same test clock to these cells, even though they are clocked by different clock senses (rising edge, falling edge, active low, and active high). When a clock is distributed to pins

with mixed clock senses, the `insert_dft` command inserts inverters to ensure design functionality.

Controlling LSSD Slave Clock Routing

For designs using either LSSD scan style or clocked LSSD scan style, all single-latch and flip-flop elements have an unconnected slave clock pin after scan replacement.

If possible, the `insert_dft` command uses the slave clocks distributed to double-latch elements and does either of the following:

- Creates, at most, one new port per design when you want to use only the slave clocks distributed to the double-latch elements
- Creates one or more ports when you want test clocks created according to different system clocks

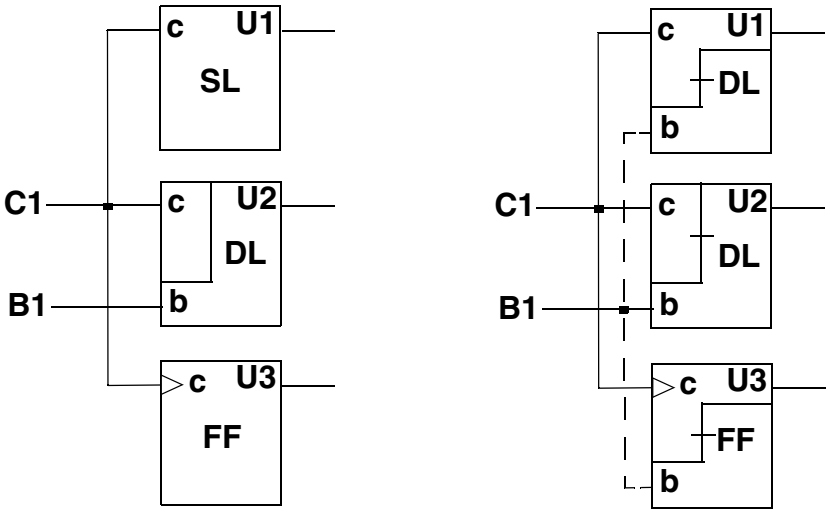
The `insert_dft` command uses the following guidelines when connecting slave clock pins of single-latch and flip-flop elements after scan replacement:

- Connect the unconnected slave clock pin of LSSD scan style single-latch or flip-flop elements to the slave clock pin of the double-latch that is clocked by the same system clock. See Figure 6-34.

  Note:

  For clarity, the A clock is omitted in Figure 6-34 through Figure 6-37 after scan replacement.

Figure 6-34 Single-Latch and Double-Latch Cells With the Same System Clock

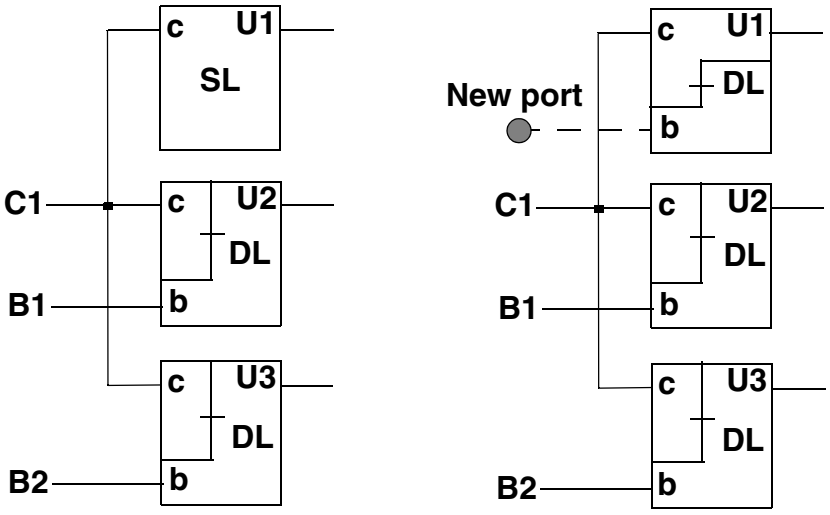


Before scan replacement

After scan replacement

- Connect to a new slave clock, creating a new one if necessary, if a system clock drives multiple cells with different slave clocks. See Figure 6-35.

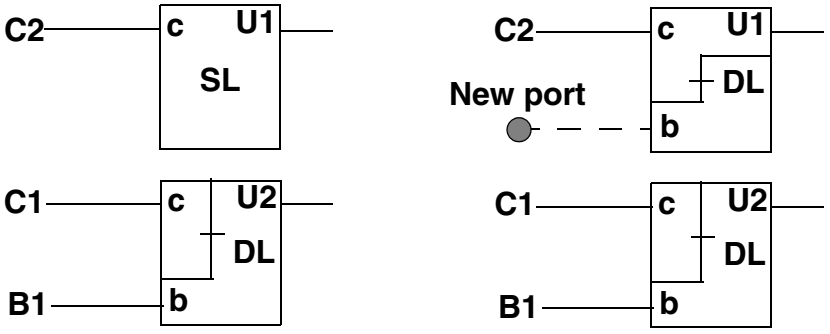
Figure 6-35 Single-Latch and Double-Latch Cells Clocked by the Same System Clock



Before scan replacement After scan replacement

- Connect to a new slave clock port, creating one if necessary, if double-latch cells are driven by different clocks. See Figure 6-36.

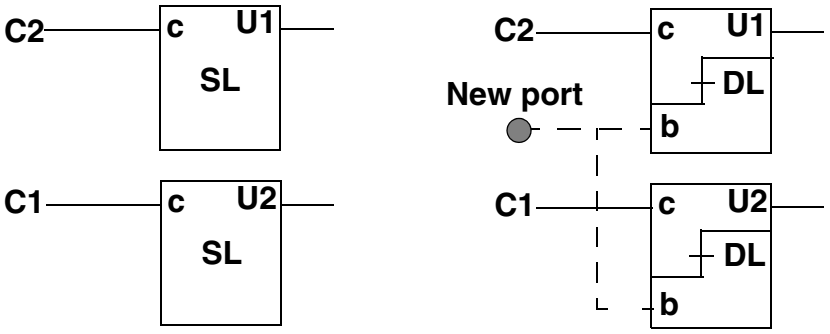
Figure 6-36 Single-Latch and Double-Latch Cells Clocked by Separate System Clocks



Before scan replacement After scan replacement

- Connect to a new slave clock, creating a new port if necessary, if there are no double-latch cells. See Figure 6-37.

Figure 6-37 Connecting Slave Clock Pin: No Double-Latch Cells



Modifying Your Scan Architecture

Unless conflicts occur, the `set_scan_configuration` commands are additive. You can enter multiple `set_scan_configuration` commands to define your scan configuration. If a conflict occurs, the latest `set_scan_configuration` command overrides the previous configuration.

To modify your scan configuration, you can rely on the override capability or remove the complete scan configuration and start over. Use the `reset_scan_configuration` command to remove the complete scan configuration. Do not use the `reset_design` command to remove the scan configuration. Configuring the scan chain does not place attributes on the design, so the `reset_design` command has no effect on the scan configuration and removes all other attributes from your design, including constraints necessary for optimization.

To make minor adjustments to the scan architecture, modify the scan specification script generated by the `preview_dft -script` command.

```
dc_shell> preview_dft -script > scan_arch.tcl
# manually modify scan_arch.tcl to reflect desired architecture */
dc_shell> source scan_arch.tcl
dc_shell> preview_dft
dc_shell> insert_dft
```

Post-Scan Test Design Rule Checking

After you perform scan insertion, you need to perform design rule checking again to ensure that no violations have been introduced into your design by the scan insertion process.

This section has the following subsections related to post-scan test DRC:

Preparing for Test Design Rule Checking After Scan Insertion

Before performing test design rule checking, you need to ensure that your timing parameters are set properly. For more information on timing parameters, see Chapter 5, “Pre-Scan Test Design Rule Checking”.

Check the following settings:

- Ensure that the following timing parameters are correctly set in your .synopsys_dc.setup file:
  - `test_default_delay`
  - `test_default_strobe`
  - `test_default_bidir_delay`
  - `test_default_period`

  Note:
  Consult your ASIC vendor for timing value requirements.
- If you bring your design in as a netlist and not in .ddc format, you need to reset all your `set_scan_configuration` and `set_dft_signal` attributes.
- Ensure that `signal_type` attributes exist on test ports.

  If you used the `insert_dft` command to build scan chains and you saved the design in .ddc format, DFT Compiler automatically establishes the signal type attributes.

  If you did not use the `insert_dft` command (for example, you read in an ASCII netlist) or if you have saved the `insert_dft` design as a netlist instead of as a .ddc file, you must reestablish the signal type attributes by using the `set_dft_signal` command.

  You can list test ports by using the `report_dft_signal` command.

- Use the `report_scan_state` command to verify that the `scan_existing` constraint is set. If the constraint is set, the report reads

  ```
  Existing Scan     : True
  ```

  If the constraint is not set, you cannot do a scan test design rule checking run and the report reads

  ```
  Existing Scan     : False
  ```

Checking for Topological Violations

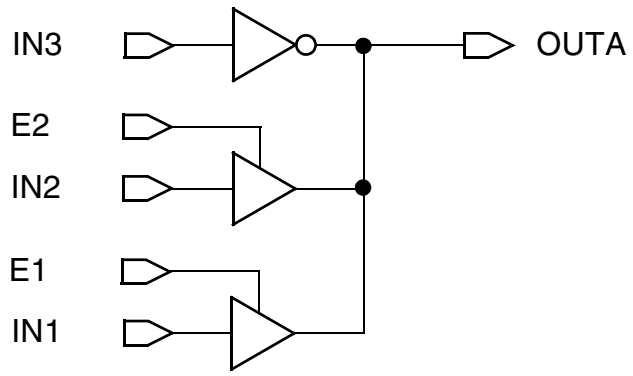
Topological checks are global connectivity checks that the `dft_drc` command performs in a structural manner.

If the `dft_drc` command cannot determine the logic function associated with a wired net, it issues the following warning message:

```
Warning: Type of wired net %s is unknown. (TEST-114)
```

The presence of a non-three-state driver on a three-state net (see Figure 6-38) results in contention on that net.

Figure 6-38 A Non-Three-State Driver



If the `dft_drc` command detects such a condition, it flags the violation with:

```
Warning: Three-state net %s is not properly driven. (TEST-115).
```

If the `dft_drc` command detects the presence of a pull-up driver or a pull-down driver on a non-three-state net, it flags the problem with

```
Error: Pullup/pulldown net %s has illegal driver(s).
(TEST-331)
```

Any violation on a net forces the net to the value X for the entire protocol simulation.

Checking for Scan Connectivity Violations

After the `dft_drc` command completes test protocol simulation, it analyzes the simulation results to determine the following:

- The architecture of the scan chains
- Whether the capture state and the state of the cell that is scanned are the same

The `report_scan_path` command reports the scan chain architecture determined by the `dft_drc` command.

Running an incremental compile or other command that affects the database can cause the information gathered by `dft_drc` to be invalidated. If you run a `report_scan_path` and get an error message saying that no scan path is defined, try running `dft_drc` again, immediately followed by a `report_scan_path` command.

Scan Chain Extraction

A scan chain is a group of sequential elements through which a uniquely identifiable bit of scan data travels. The `dft_drc` command extracts scan chains from a design by tracing scan data bits through the multiple time frames of the protocol simulation. Scan chains are protocol dependent: For a given design, specifying a different test protocol can result in different scan chains. As a corollary, scan-chain-related problems can be caused by an incorrect protocol, by incorrect `set_dft_signal` specifications, or even by incorrectly specified timing data.

Causes of Common Violations

During test design rule checking on scan designs, DFT Compiler simulates the test protocol to verify that the scan operation functions correctly. Protocol simulation verifies that scan cells predictably perform the following tasks:

- Receive data during scan input
- Capture data during parallel capture
- Shift data during scan output

The following sections describe the scan operation checks for each of these tasks and provide guidance in correcting the problems.

Ability to Load Data Into Scan Cells

To ensure that the scan shift process can successfully load data into the scan cells, DFT Compiler verifies that

- Data arrives at the scan input pin of each scan cell
- The test clock pulse arrives at the test clock pin of each scan cell
- Scan data is not corrupted during scan shift

If a scan cell does not meet these conditions, DFT Compiler cannot control the scan cell. Typical causes for uncontrollable scan cells include

- Incorrect or incomplete test configuration
- Invalid clock logic
- Incorrect timing relationships between clocks for two-phase clocking
- Nonscan sequential cells clocked by the test clock
- Invalid scan path

DFT Compiler generates this error message when it detects that it cannot shift through a scan chain:

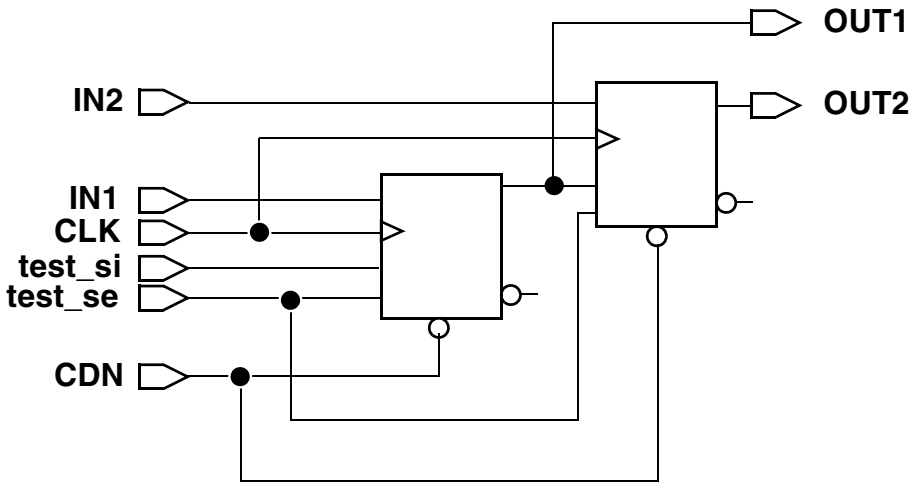
```
Begin Scan chain violations...

Error: Chain c1 blocked at DFF gate FF_A after tracing 2 cells. (S1-1)

Scan chain violations completed...
```

The following sections provide examples of the typical causes of uncontrollable scan cells.

Incomplete Test Configuration

Figure 6-39 shows a simple scan design with a scan chain.

Figure 6-39 Simple Scan Design



When reading a design from an ASCII netlist that contains existing scan chains, you must specify the test ports. If you do not identify the scan input port, DFT Compiler does not flag any violations during DRC, but it will not be able to extract scan chains.

If the scan input port information is not specified, the `dft_drc` command generates a pre-DFT DRC report even though the netlist contains scan chains:

```
dc_shell> dft_drc
In mode: all_dft...
  Pre-DFT DRC enabled

Information: Starting test design rule checking. (TEST-222)
...
```

Also, the `report_scan_path -chain all` command does not report any scan chains:

```
dc_shell> report_scan_path -chain all
...
=======================================
TEST MODE: Internal_scan
VIEW     : Existing DFT
=======================================

=======================================
AS SPECIFIED BY USER
=======================================

=======================================
AS BUILT BY insert_dft
=======================================

No scan path defined in this mode.
```

To resolve this, identify the scan input ports, scan output ports, test clocks, and asynchronous sets and resets, then rerun `dft_drc`. For example,

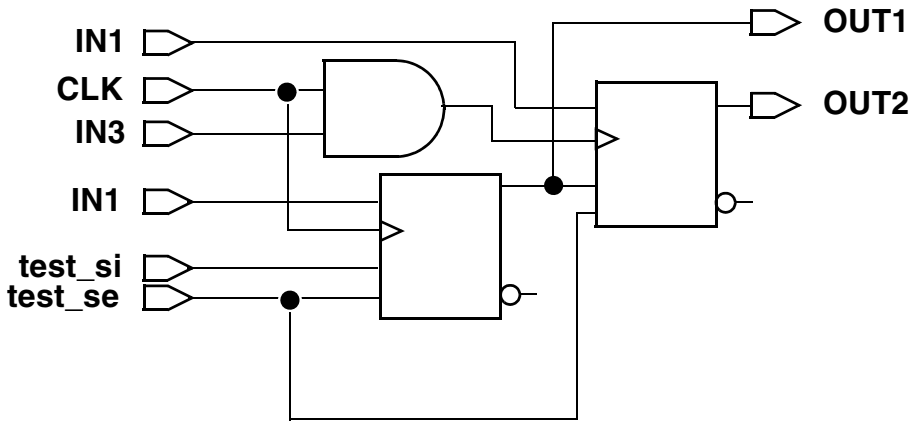
```
set_scan_state scan_existing

set_dft_signal -view existing -type ScanEnable -port test_se
set_dft_signal -view existing -type ScanDataIn -port test_si
set_dft_signal -view existing -type ScanDataOut -port OUT2

set_scan_path C1 -view existing \
   -scan_data_in test_si -scan_data_out OUT2
```

After the test ports are defined, the `dft_drc` command generates a post-DFT DRC report, and the scan chains are properly inferred.

Invalid Clock Logic

Figure 6-40 shows a design with a combinationally gated clock.

Figure 6-40 Combinationally Gated Clock

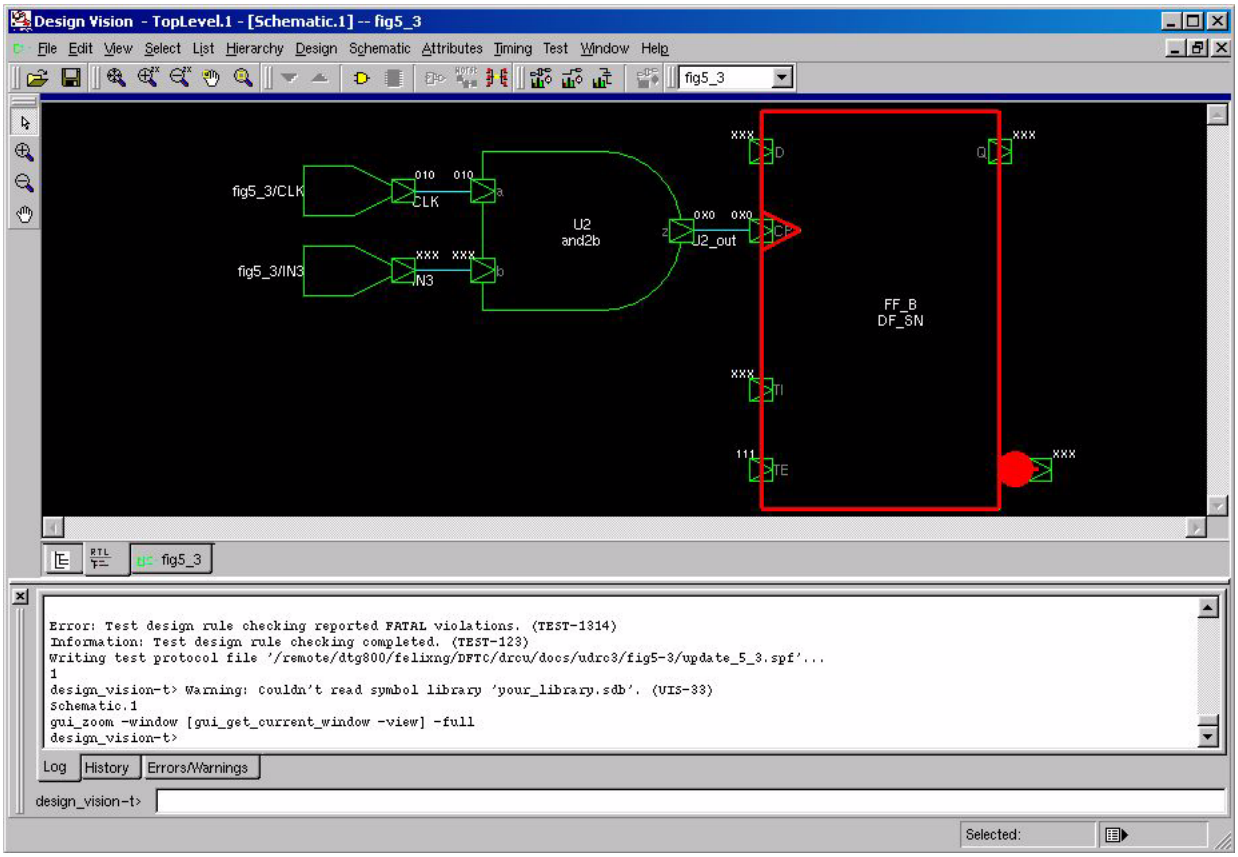


If you do not hold port IN3 at logic 1 during scan shift, pulses applied at clock port CLK might not reach the clock pin of cell FF_B; therefore, the clock input of cell FF_B violates the test clock requirements. DFT Compiler generates error messages such as these:

```
------------------------------------------------------------
Begin Scan chain violations...

Error: Chain c1 blocked at DFF gate U1 after tracing 0 cells. (S1-1)

Scan chain violations completed...
------------------------------------------------------------
```

Invoke the Design Vision Graphical Schematic Debugger, as shown in Figure 6-41.

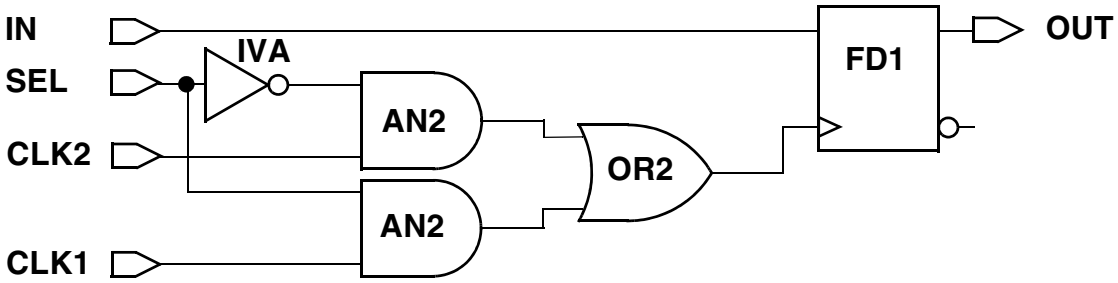
Figure 6-41 The Design Vision Graphical Schematic Debugger



The debugger shows that the clock input of the cell FF_B contains an X. This indicates that the clock was is completely controllable.

In Figure 6-42, if SEL = 1, the path from CLK1 is active, although the path from CLK2 is not. In general, you use the `set_dft_signal` command to specify constant logic values on ports, as explained later in this chapter.

Figure 6-42 A Clock Selector Network



In this example, if you specify `set_dft_signal-view existing_dft -type Constant -active_state 1` on the SEL port, you will see this violation:

```
----
Begin Pre-DFT violations...

Warning: Clock CLK2 cannot capture data with other clocks
off. (D8-1)

Pre-DFT violations completed...
---
```

A D8 violation indicates that a clock cannot capture data while others are off. Each clock must be capable of capturing data. This does not prevent scan insertion, but you might want to investigate the cause of the violation.

You can correct invalid clock-gating violations by inserting logic.

If a clock pin is driven by constant logic, the `dft_drc` command issues a warning:

```
Warning: Clock input CP of DFF FF_A couldn't capture data.
(D17-1)
```

The waveforms of the inferred clocks are taken either from a previous invocation of the `set_dft_signal` command or from the scan style-dependent default timing values.

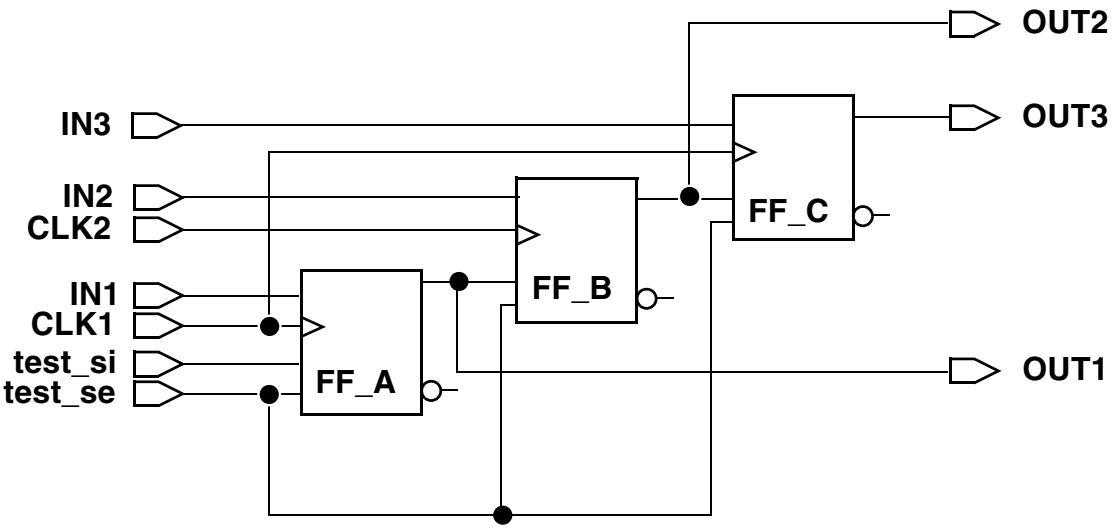
Incorrect Clock Timing Relationship

A structurally valid scan chain becomes invalid due to the clock timing definitions in the following cases:

- The cell ordering of the scan chain in a scan design with multiple clock domains has later cells triggered by later clocks (data flow-through).
- The active levels of the master clock and the slave clock overlap in designs with two-phase clocking.

Figure 6-43 shows a scan design with multiple clocks. Structurally this design meets the scan design rules. However, the ability to shift data through the scan chain depends on the relationship between the multiple clocks.

Figure 6-43 Existing Scan Design With Multiple Clocks



Unless CLK1 and CLK2 have identical timing, this design always results in an invalid scan path due to the clock timing relationship. CLK2 triggers cell FF_B, and CLK1 triggers both the cell driving it (FF_A) and the cell driven by it (FF_C).

If the clock timings are identical, design rule checker will report warning messages such as

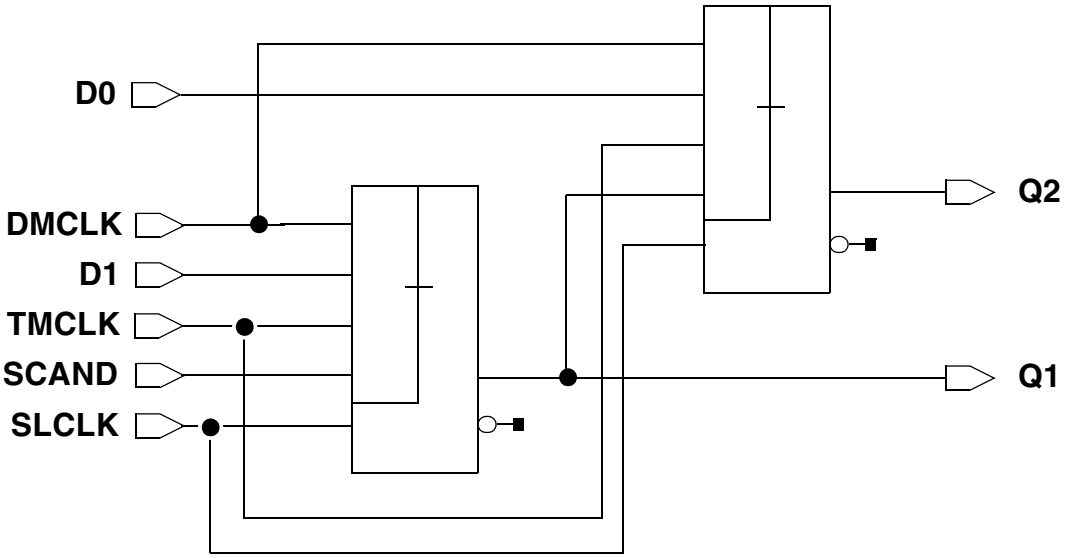
```
Warning: Multiple clocks (CLK1 CLK2) were used to shift scan chain c1.
(S22-1)
```

If the clock timings are different, design rule checker will report warning messages such as

```
Warning: Dependent slave FF_B may not hold same value as master FF_A.
(S29-1)
```

Figure 6-44 shows an LSSD design. Structurally, this design meets the scan design rules. However, the ability to shift data through the scan chain depends on the relationship between the master clock (TMCLK) and the slave clock (SLCLK).

Figure 6-44 Simple LSSD Design



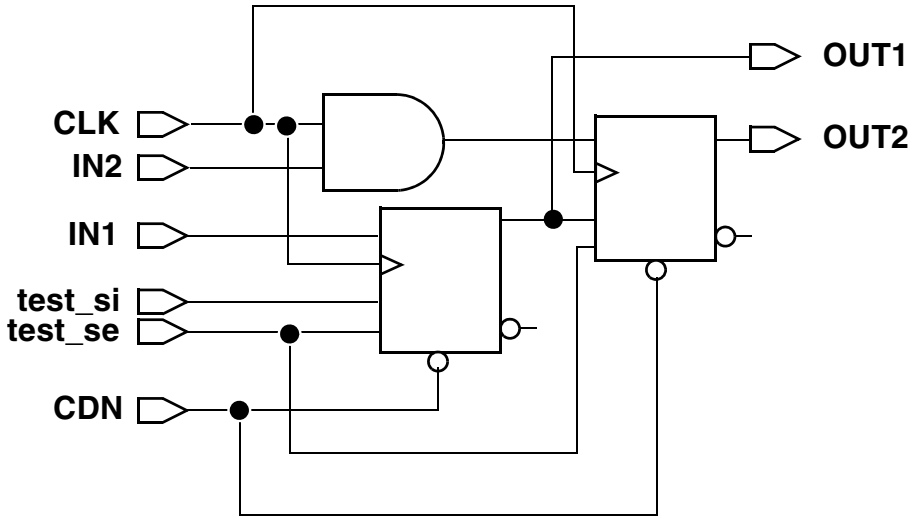
DFT Compiler uses zero-delay timing, so you cannot depend on delays in the clock nets to prevent overlapping master and slave clocks. Because DFT Compiler considers both the master and slave clocks active at 55 ns after the start of the vector, this command sequence defines an invalid timing relationship for the design in Figure 6-44:

```
dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock -timing [list 45 55] \
              -port TMCLK

dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock -timing [list 55 65] \
              -port SLCLK
```

Nonscan Sequential Cells

Figure 6-45 shows a scan design with a nonscan sequential cell.

Figure 6-45 Scan Design With Nonscan Sequential Cell



DFT Compiler supports this configuration but generates uncontrollable-scan-cell messages to indicate exclusion of the nonscan cell from the scan chain.

If the nonscan cell has a `scan_element false` attribute, DFT Compiler generates warning messages such as this:

```
Warning: Nonscan DFF U1 disturbed during time 45 of shift
procedure. (S19-1)
```

Ability to Capture Data Into Scan Cells

To ensure that the parallel capture cycle results in data that is successfully captured into the scan cells, DFT Compiler verifies that

- The capture data is valid

  Valid capture data depends only on the scanned-in state and primary input values. Modification of capture data by other capture data or the capture clock invalidates the capture data.

- The system clock pulse arrives at the system clock pin of each scan cell

If a scan cell does not meet these conditions, DFT Compiler cannot capture data into the scan cell. Typical causes of failed data capture include the following:

- A clock signal drives the data input to a scan cell.
- A functional path in the design has sequential endpoints clocked by different clock domains (untestable functional path).
- A bidirectional port drives the data input to a scan cell, and the data is released before the capture clock.
- A master-slave cell with an inferred behavior for the B clock pulse causes the cell capture state to be different from the cell scan-out state.
- A sequential element drives an asynchronous input to a scan cell.
- The test protocol does not include a capture clock.

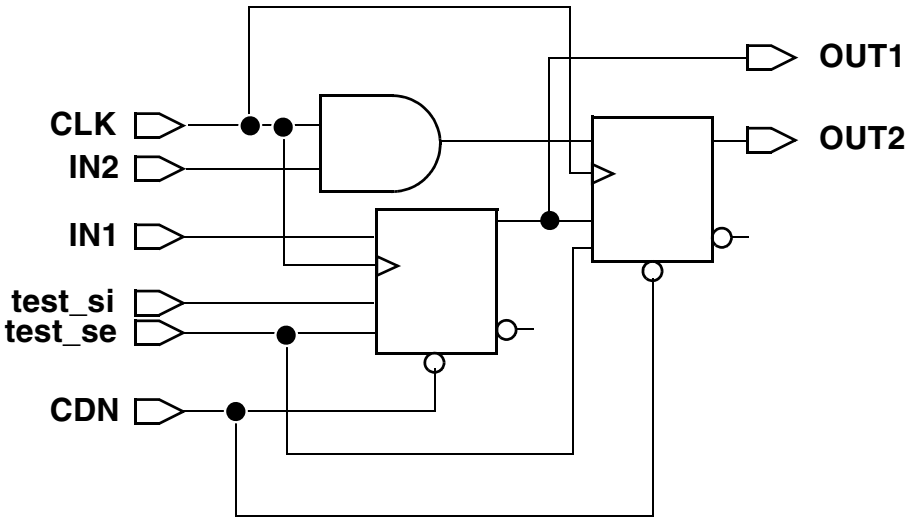
DFT Compiler generates diagnostic messages indicating the source of the violation.

The following sections provide examples of the typical causes of failed data capture.

Clock Driving Data

In the design shown in Figure 6-46, the clock signal CLK drives the data input to cell FF_B. Pulsing the clock signal during capture can cause the data input to cell FF_B to change.

Figure 6-46 Design With Clock Driving Data



DFT Compiler generates this warning message:

```
Warning: Clock CLK connects to LE clock/data inputs CP/D of DFF FF_B.
(C12-1)
```

Although the `dft_drc` output and the scan path report indicate that the affected cell is scannable, the cell is actually scan controllable only.

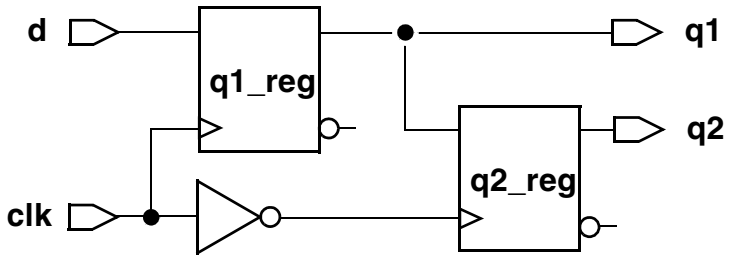
This violation usually has a minor impact on fault coverage, so make it one of the last violations you correct, if at all. Correcting this violation requires the addition of test-mode logic, which also has a minor fault coverage impact. Fixing the violation means trading one set of untested faults for another, possibly smaller, set of untested faults.

Use the Design Vision Graphical Schematic Debugger to locate and analyze the clock-driving data problem.

Untestable Functional Path

Figure 6-47 shows a design with an untestable functional path. A functional path exists between cells q1_reg and q2_reg. Using the default clock waveform of rising edge at 45 ns and falling edge at 55 ns, q2_reg receives the data captured in cell q1_reg.

Figure 6-47 Untestable Functional Path



Because the capture data in cell q2_reg depends on data other than the scanned-in state and the primary input values, DFT Compiler generates warning messages such as these:

```
Warning: Clock clk can capture new data on TE input CP of DFF q2_reg.
(D14-1)
         Source of violation: input CP of DFF q1_reg.
```

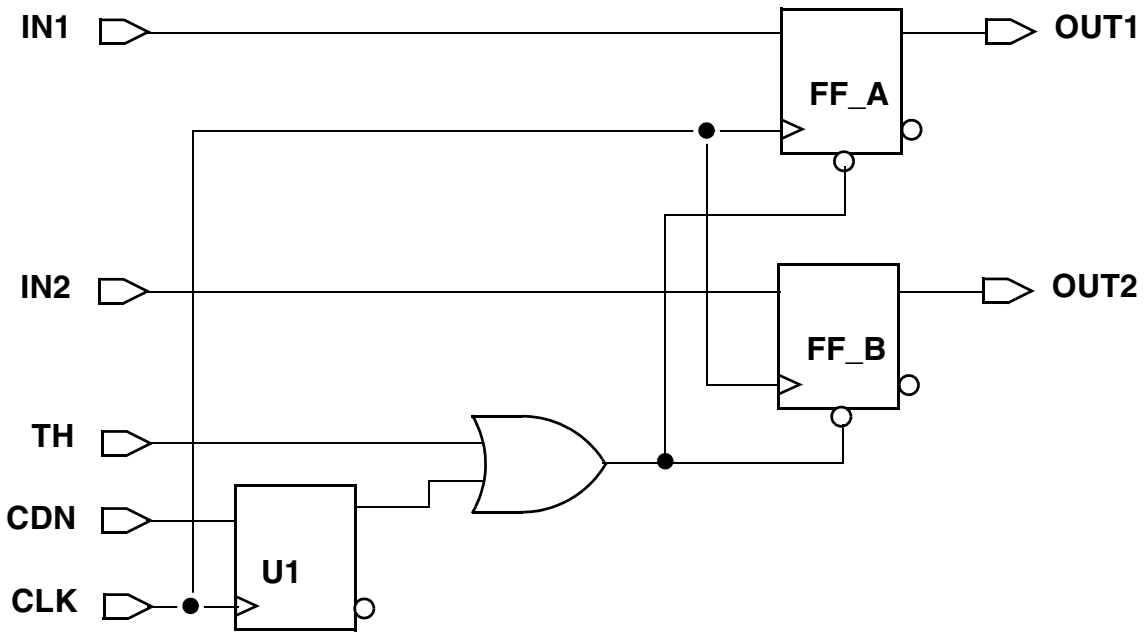
Use the Design Vision Graphical Schematic Debugger to locate and analyze the untestable functional path problem. Contact Synopsys support personnel for access to a script that loads the debugger.

In most cases, you must change the design to correct the problem.

Uncontrollable Asynchronous Pins

The asynchronous pins shown in Figure 6-48 are uncontrollable, because they are driven by sequential logic. If you hold the TM signal at logic 1 only during scan shift, the asynchronous resets on cells FF_A and FF_B can change as a result of the capture clock.

Figure 6-48 Uncontrollable Asynchronous Pins

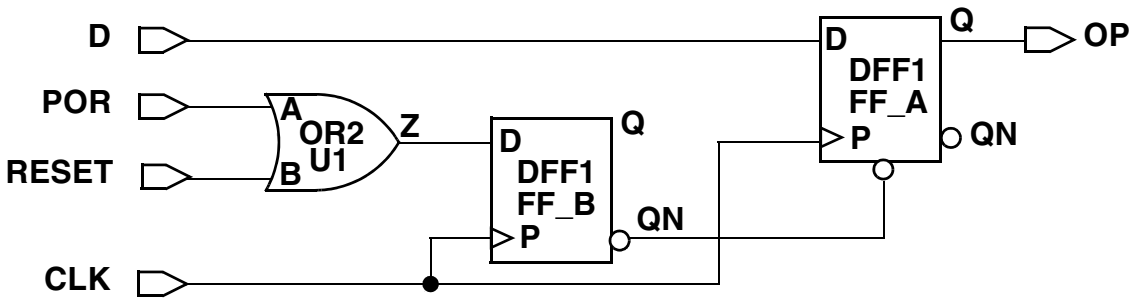


DFT Compiler will report the following:

```
Warning: Clock CDN cannot capture data with other clocks off. (D8-1)
```

Uncontrollable pins usually occur when the asynchronous signal is generated from the state of other sequential devices, as shown in Figure 6-49. You can correct this violation by inserting test-mode logic.

Figure 6-49 Circuit With Uncontrollable Asynchronous Clear



7

Advanced DFT Architecture Methodologies

This chapter describes advanced features that can be used while inserting scan circuitry into your design. These features can be used to improve design testability using manual and automatic techniques, improve the frequency of the scan testing logic, reduce power consumption during test, and provide improved integration of tool-inserted and user-defined test logic.

The chapter describes advanced DFT architecture-related methodologies and processes in the following sections:

Performing Scan Extraction

The scan chain extraction process extracts scan chains from a design by tracing scan data bits through the multiple time frames of the protocol simulation. For a given design, specifying a different test protocol can result in different scan chains. As a corollary, scan chain-related problems can be caused by an incorrect protocol, by incorrect `set_dft_signal` command specifications, or even by incorrectly specified timing data.

When performing scan extraction, you always use the descriptive view (`-view existing_dft`), because you are defining test structures that already exist in your design.

To perform scan extraction

1. Define the scan input and scan output for each scan chain. To define these relationships, first use the `set_scan_configuration` command to specify the scan style and then use the `-view existing_dft` option with the `set_scan_path` and `set_dft_signal` commands, as shown in the following examples:

```
dc_shell> set_scan_configuration \
              -style multiplexed_flip_flop

dc_shell> set_dft_signal -view existing_dft \
              -type ScanDataIn -port TEST_SI

dc_shell> set_dft_signal -view existing_dft \
              -type ScanDataOut -port TEST_SO

dc_shell> set_dft_signal -view existing_dft \
              -type ScanEnable -port TEST_SE

dc_shell> set_scan_path chain1 \
              -view existing_dft \
              -scan_data_in TEST_SI \
              -scan_data_out TEST_SO
```

2. Define the test clocks, reset, and test-mode signals by using the `set_dft_signal` command.

```
dc_shell> set_dft_signal -view existing_dft \
              -type ScanClock -port CLK \
              -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
              -type Reset -port RESETN \
              -active_state 0
```

3. Create the test protocol by using the `create_test_protocol` command.

```
dc_shell> create_test_protocol
```

4. Extract the scan chains by using the `dft_drc` and `report_scan_path` commands.

```
dc_shell> dft_drc

dc_shell> report_scan_path -view existing_dft \
              -chain all

dc_shell> report_scan_path -view existing_dft \
              -cell all
```

Inserting Test Points

Test points are points in the design where DFT Compiler inserts logic to improve the testability of the design. The tool can automatically determine where to insert test points to improve test coverage and reduce pattern count. You can also manually define where test points are to be inserted.

The test point capabilities are described in the following sections:

- Test Point Types
- Sharing Test Point Scan Cells
- Automatically Inserting Test Points
- Inserting User-Defined Test Points

Test Point Types

The *force* and *control* test point types allow signals within logic cones to be actively controlled during test mode to improve the controllability of the logic. These test point types require the insertion of a multiplexer to conditionally override the original signal value, resulting in a slight delay and area penalty.

The *observe* test point type passively captures the value of selected hard-to-observe signals to improve the observability of the logic. No additional levels of logic are inserted along the path of the observed signal, but the extra observation logic does slightly increase the capacitive loading of the observed signal.

Test points are described in the following subsections:

- Force Test Points
- Control Test Points
- Observe Test Points

- Test Point Signals
- Sharing Test Point Scan Cells

Note:

The test point schematics in these sections show the functional operation of the test points. During synthesis, constant logic is simplified, and the test point logic is optimized into the surrounding logic.

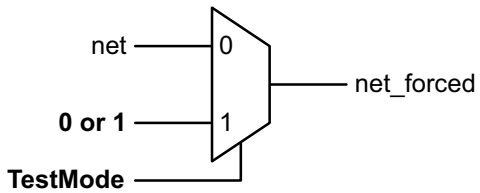
Force Test Points

Force test points are used when a value must be forced throughout the entire test session. The following force test point types are available:

- `force_0`
- `force_1`
- `force_01`
- `force_z0`
- `force_z1`
- `force_z01`

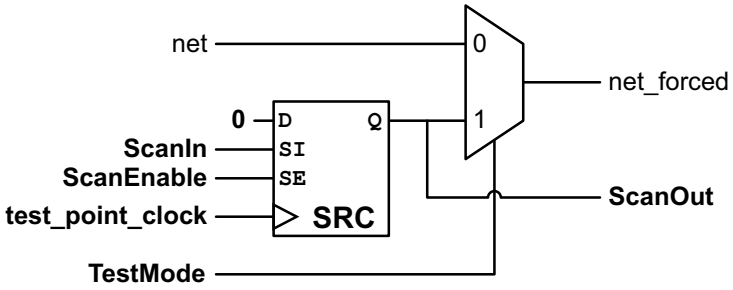
The `force_0` and `force_1` test point types allow a signal to be replaced with a constant 0 or constant 1 value throughout the entire test session. These test point types are useful when a particular signal must be forced to a known value for testability purposes. A multiplexer is used to replace the original signal with a fixed constant 0 or 1 value when the TestMode signal is asserted. See Figure 7-1.

Figure 7-1 Example of a force_0 or force_1 Test Point



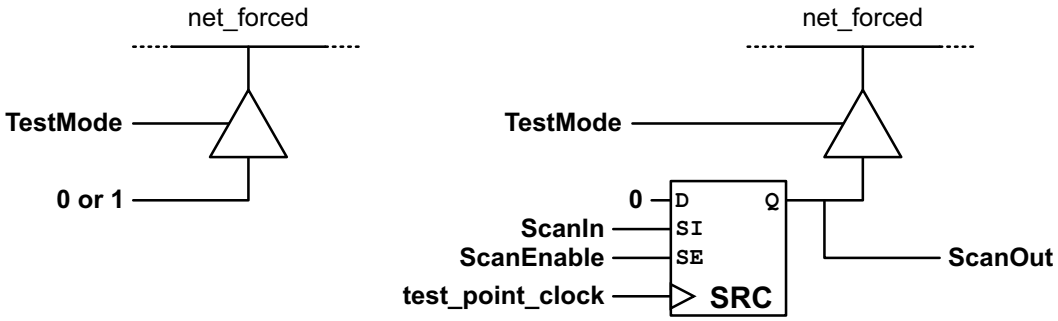
The `force_01` test point type allows a signal to be replaced with a scan-selected value throughout the entire test session. The scan-selected value comes from a source signal scan register, allowing the forced value to change for each test vector. A multiplexer is used to replace the original signal with the output of this scan register when the TestMode signal is asserted. See Figure 7-2.

Figure 7-2 Example of a force_01 Test Point



The `force_z0`, `force_z1`, and `force_z01` test point types allow either a constant value or a scan-selected source signal value to be driven onto a tristate bus that is guaranteed to have no other active drivers during test mode. See Figure 7-3.

Figure 7-3 Examples of force_z0, force_z1, and force_z01 Test Points



Note that the AutoFix feature of DFT Compiler uses `force_0` and `force_1` test points for asynchronous signal fixing and `force_01` test points for clock fixing and for fixing clock-as-data and X propagation.

Control Test Points

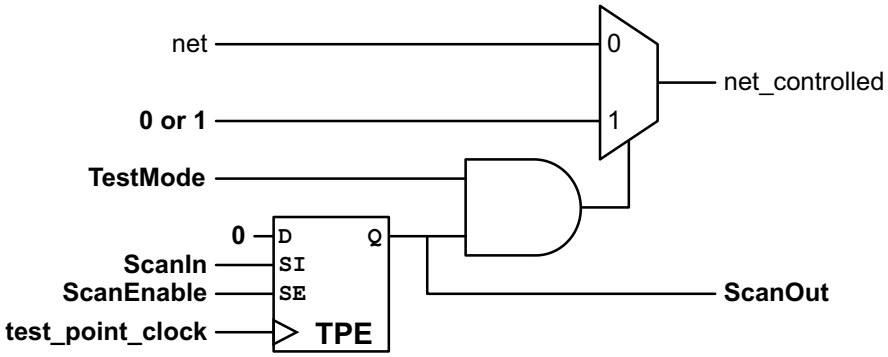
Control test points are used when a hard-to-control signal should be controllable (selectively forced) for some test vectors but left unaltered for others. Control test points are typically inserted to increase the fault coverage of the design. The following control test point types are available:

- `control_0`
- `control_1`
- `control_01`
- `control_z0`

- control_z1
- control_z01

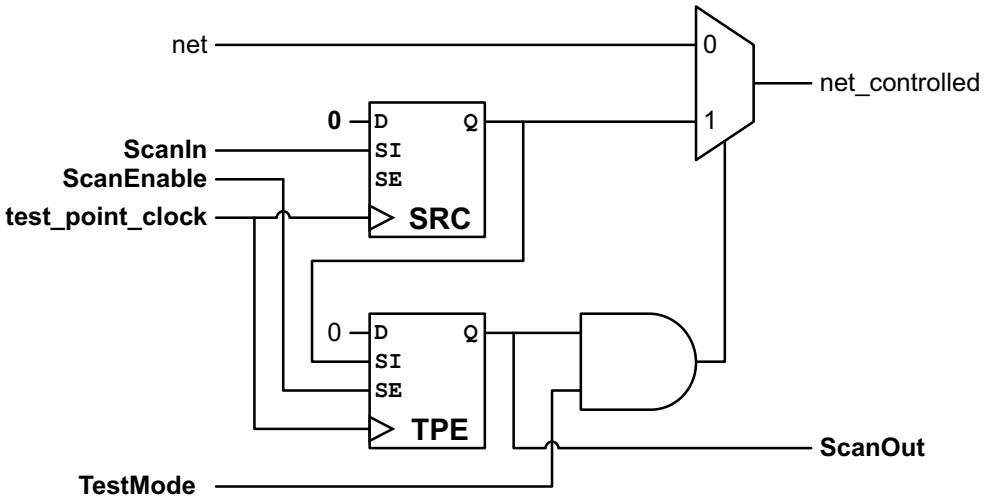
A control_0 or control_1 test point is built with a multiplexer, an AND gate, and a test-point enable scan register. When TestMode is not asserted, the signal always retains its original value. When TestMode is asserted, the signal is forced with a fixed constant 0 or 1 value only when the output of the test point enable scan register selects the constant value. This allows the test program to select either the original signal behavior, or the constant-forced behavior on a vector-by-vector basis. This has the advantage of being able to control the signal for some test vectors without losing the observability of the upstream logic for the remaining vectors. See Figure 7-4.

Figure 7-4 Example of a control_0 or control_1 Test Point



A control_01 test point is similar to the control_0 and control_1 test point types, except that a scan-selected source signal value from a scan register is selectively driven onto the net on a vector-by-vector basis. As a result, the control_01 test point requires two scan cells per control point, one for the source signal value and one for the enable register that specifies that the source signal should be driven. See Figure 7-5.

Figure 7-5 Example of a control_01 Test Point



The `control_Z0`, `control_Z1`, and `control_z01` test point types allow either a constant value or a scan-selected source signal value to be selectively driven onto a bus that might be three-stated for some vectors but not for others. See Figure 7-6 and Figure 7-7.

Figure 7-6 Example of a control_z0 or control_z1 Test Point

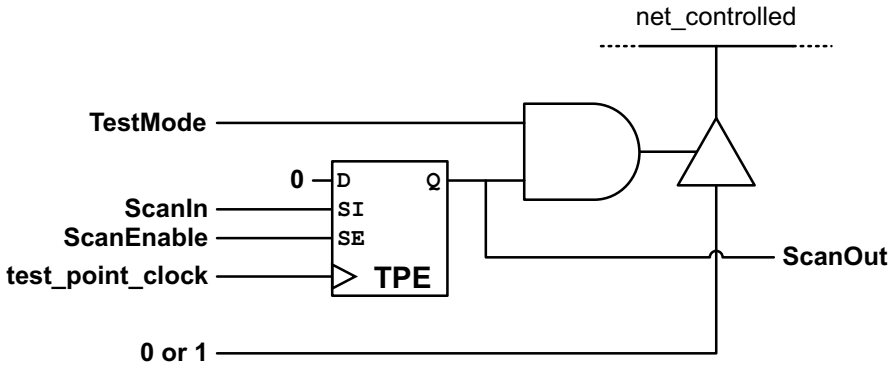
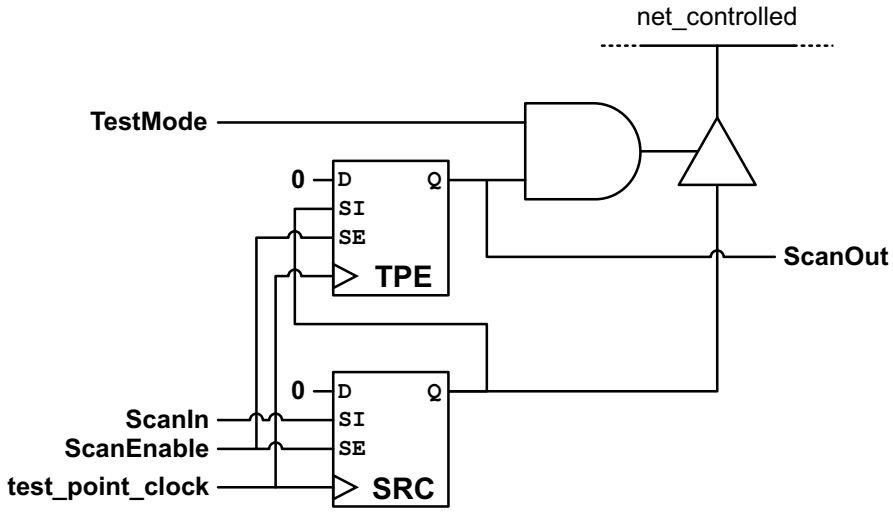


Figure 7-7 Example of a control_z01 Test Point

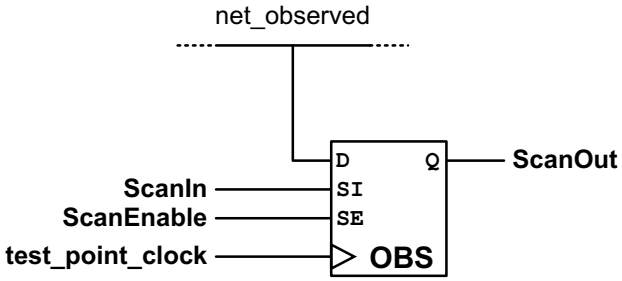


Observe Test Points

The `observe` test point type is typically inserted at hard-to-observe signals in a design to reduce test data volume or to increase the coverage.

An `observe` test point is a scan register with its data input connected to the sink signal to be observed. See Figure 7-8.

Figure 7-8 Example of an observe Test Point



Test Point Signals

Test points use source, sink, test point enable, and control signals as shown in Table 7-1.

Table 7-1 Test Point Signal Types

Test point type	Source signal	Enable signal	Sink signal	Control signal
`force_0`, `force_1`				X
`force_01`	X			X
`force_z0`, `force_z1`				X
`force_z01`	X			X
`control_0`, `control_1`		X		X
`control_01`	X	X		X
`control_z0`, `control_z1`		X		X
`control_z01`	X	X		X
`observe`			X	X[1]

1. *Test-mode signal used only if low-power XOR observability tree is enabled*

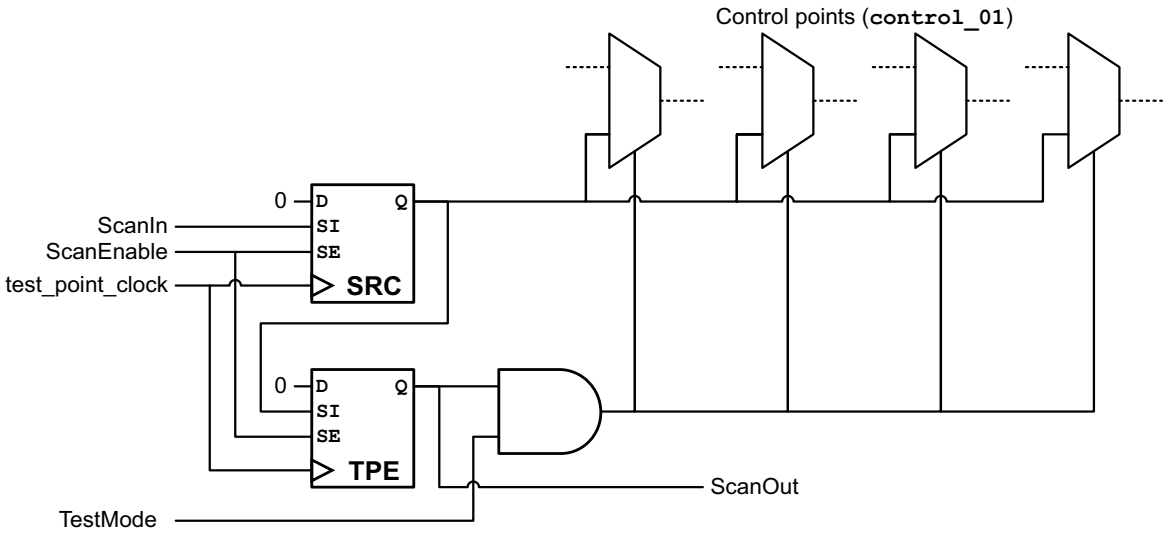
The control signal is the TestMode signal that activates the test point logic.

Sharing Test Point Scan Cells

To reduce the area requirements of test point logic, DFT Compiler allows you to share the test-point enable, source signal, and sink signal scan registers with multiple test points.

You can share the same test-point enable or source signal register with multiple control or force test points. No additional logic gates are required; the scan register outputs are tied to multiple test point logic gates. Figure 7-9 shows the logic for multiple `control_01` test points that share the same scan registers.

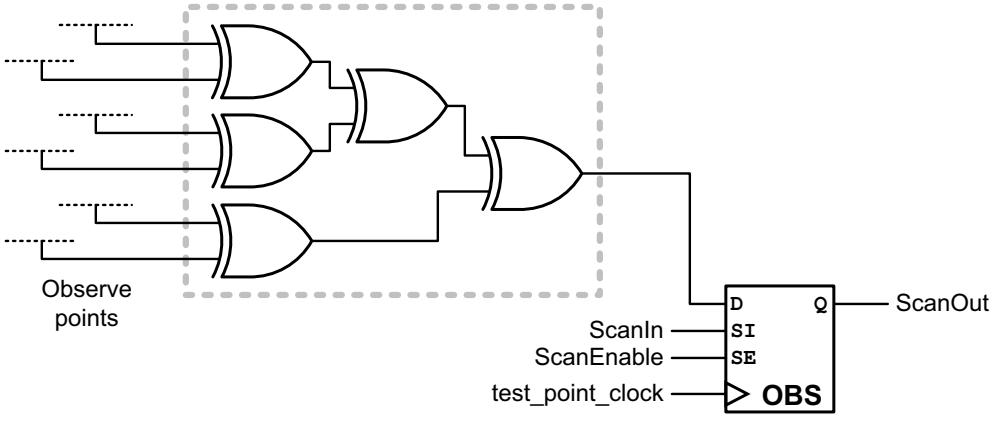
Figure 7-9 Shared Scan Registers for Multiple control_01 Test Points



Test-point enable and source signal scan registers are not shared between different test point types.

You can share the same observe sink signal scan register with more than one `observe` test point. DFT Compiler builds an observability XOR tree which collapses multiple observed signals down to a single sink signal connected to the data input of the shared sink signal scan register. See Figure 7-10.

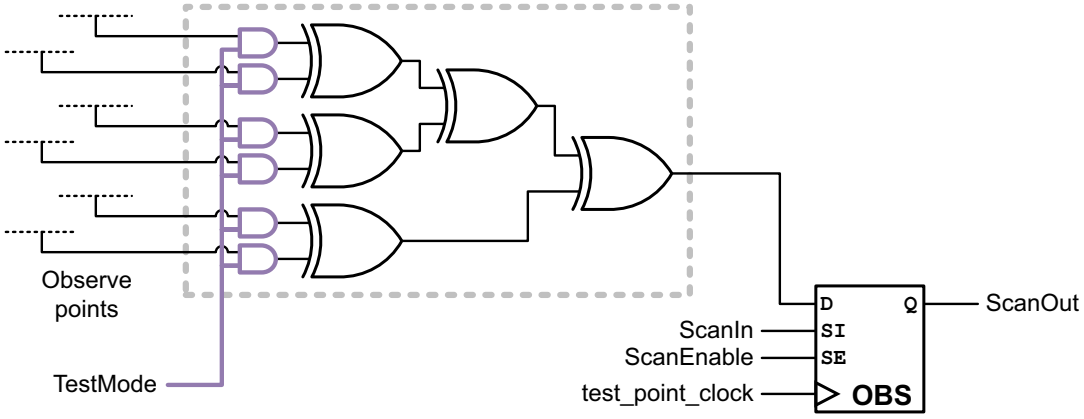
Figure 7-10 XOR Observability Tree For Multiple observe Test Points



When a device is in functional mode, every time the logic value on an observe node changes, either from 1 to 0 or from 0 to 1, the entire fanout path through the XOR observability tree toggles. This toggling results in unnecessary power losses.

To avoid such losses in power, you can create a low-power observability tree for a shared observe scan register. The observe point signals are gated with a 2-input AND gate, with a test-mode signal used as the gating signal. See Figure 7-11.

Figure 7-11 Low-Power XOR Observability Tree For Multiple observe Test Points



If a test-mode signal is defined, DFT Compiler uses it for the low-power gating signal. Otherwise, DFT Compiler creates a new test-mode signal that is used only for low-power observability gating.

Automatically Inserting Test Points

The tool can automatically insert test points to improve the testability of the design. You can optionally specify requirements for test point insertion, such as the maximum number of test points or the maximum additional area overhead. During DFT insertion, the tool inserts the optimal set of test points that meets the requirements.

The following sections describe how to configure automatic test point insertion:

- Enabling Automatic Test Point Insertion
- Configuring Pattern Reduction and Testability Test Point Insertion
- Previewing the Test Point Logic
- Inserting the Test Point Logic
- Script Example

Enabling Automatic Test Point Insertion

To enable automatic test point insertion, you must first issue the following command before pre-DFT DRC:

```
dc_shell> set_dft_configuration -test_points enable
```

Note:
A DFTMAX license is required to use the automatic test point insertion feature.

After you have enabled automatic test point insertion, you can enable and configure one or more automatic test point targets with the `set_test_point_configuration` command, as described in the following sections. To enable multiple test point targets, issue a separate configuration command for each target.

Configuring Pattern Reduction and Testability Test Point Insertion

You can use the pattern reduction and testability automatic test point insertion targets to improve the testability of hard-to-test logic in your design. They work as follows:

- `pattern_reduction` – Enables only observe points

  This mode reduces the pattern count needed to achieve a given amount of test coverage. Observe points increase the loading along the observed path, but do not directly increase the logic depth. This mode has less impact on timing.

- `testability` – Enables both control and observe points

  This mode improves the testability of the design by increasing the controllability of hard-to-test logic. Keep in mind that control points insert logic along the path being controlled. Although gate-level optimization can combine the control points with the surrounding logic, there might be some impact on timing.

During pre-DFT DRC, performed by the `dft_drc` command, the tool analyzes the design to determine the optimal set of test points. During DFT insertion, performed by the `insert_dft` command, the tool inserts the test points into the design. Any needed dedicated clock or test mode signals are created.

To enable and configure the testability or pattern reduction targets of automatic test point insertion, use the `set_test_point_configuration` command as follows:

```
set_test_point_configuration
    -target pattern_reduction | testability
    [-control_signal control_name]
    [-clock_signal clock_name]
    [-clock_type dominant | dedicated]
    [-max_control_points n]
    [-max_observe_points n]
    [-test_points_per_scan_cell n]
    [-power_saving enable | disable]
    [-max_additional_logic_area n]
```

The `-target` option specifies which automatic test point insertion target to enable, and it is a required option. The `pattern_reduction` or `testability` targets are mutually exclusive.

You can use the following additional options, along with the `-target` option, to configure pattern reduction or testability test point insertion:

- By default, control points use any available TestMode port previously defined with the `set_dft_signal` command. To specify the TestMode signal that should activate the test points, use the following option:

  ```
  dc_shell> set_test_point_configuration ... -control_signal pin_port
  ```

  The specified control signal must be defined as a `TestMode` signal type with the `set_dft_signal` command.

- By default, the `insert_dft` command uses the dominant clock, which is the clock that clocks the most sequential elements in the design, to clock the inserted test point scan registers. In an on-chip clocking (OCC) flow, it chooses an OCC clock instead.

  To specify that a dedicated test point clock signal should be used, use the following option:

  ```
  dc_shell> set_test_point_configuration ... -clock_type dedicated
  ```

  This option causes a new dedicated test point clock signal, tpclk, to be created.

  To specify the test clock signal that should clock the test point scan registers, use the `-clock_signal` option:

  ```
  dc_shell> set_test_point_configuration ... \
              -clock_type dedicated \
              -clock_signal clock_name
  ```

  DFT Compiler supports the following clock name specifications:

  - You can specify the name of a scan clock signal, defined as a `ScanClock` signal type with the `set_dft_signal` command.
  - In a DFT-inserted OCC controller flow, you can specify the name of a PLL output pin. In this case, DFT Compiler maps the test point clock to the output pin of the corresponding OCC controller during DFT insertion.
  - In a user-defined OCC controller flow, you can directly specify the name of an output pin of an existing OCC controller.

  For more information on OCC controller flows, see Chapter 9, “On-Chip Clocking Support.”

- By default, automatic test point insertion is limited to a maximum of 1000 control points. To specify a different limit, use the following option:

  ```
  dc_shell> set_test_point_configuration ... -max_control_points n
  ```

  This option is only valid in testability mode; it is ignored in pattern reduction mode.

- By default, automatic test point insertion is limited to a maximum of 1000 observe points. To specify a different limit, use the following option:

```
dc_shell> set_test_point_configuration ... -max_observe_points n
```

- By default, each source, sink, or enable scan register can be shared by up to eight test points. To specify the maximum number of test points that can share a single source, sink, or enable scan register, use the following option:

```
dc_shell> set_test_point_configuration ... \
            -test_points_per_scan_cell n
```

  For more information on sharing scan cells, see “Sharing Test Point Scan Cells” on page 7-10.

- To insert power-saving AND gates at the top of the XOR observability trees to avoid excess switching power consumption during scan shift, use the following option:

```
dc_shell> set_test_point_configuration ... -power_saving enable
```

  For more information on power-saving logic, see “Sharing Test Point Scan Cells” on page 7-10.

- To apply an area limit to the inserted test point logic, use the following option:

```
dc_shell> set_test_point_configuration ... \
            -max_additional_logic_area P
```

  The value *p* is the percentage of the total design area that can be consumed by the test point logic and must be a value between 1 and 50. Low-power observability logic is included in the test point area value. If specified, the area limit applies in addition to the test point limit.

Previewing the Test Point Logic

To preview the observe test logic that the tool will implement according to your specifications, use the following command:

```
dc_shell> preview_dft -test_points all
```

This command reports the following information:

- Control test-point locations
- Observe test-point locations
- Instance names of inserted test-point flip-flops
- Clocks used by inserted test-point flip-flops
- Low-power observability tree status

You can use other options of the `preview_dft` command with the `-test_points` option.

Inserting the Test Point Logic

After you define the test point insertion configuration, the `insert_dft` command inserts the test point logic. Test point scan registers are placed in the lowest level of hierarchy shared by all test points for that register.

The following additional signals are created, depending on the test configuration:

- A new test point clock signal, if a test point clock is not defined
- A new test-mode signal for force, control, and observe test points, if a test-mode signal is not defined

Script Example

The following script inserts testability test points, using a test-mode control signal named TM_TESTPOINTS.

```
# define DFT signals
set_dft_signal -view existing_dft -type ScanClock \
  -port CLK -timing [list 45 55]
set_dft_signal -view spec -type TestMode -port TM_TESTPOINTS

# enable automatic test point insertion
set_dft_configuration -test_points enable

# enable and configure testability test points
set_test_point_configuration \
  -target testability \
  -control_signal TM_TESTPOINTS

# preview test points
preview_dft -test_points all

# insert DFT logic
insert_dft
```

Inserting User-Defined Test Points

User-defined test points provide you with the flexibility to insert control and observe test points at user-specified locations in the design. User-defined test points can be used for a variety of purposes, including the ability to fix uncontrollable clocks and asynchronous signals, increase the coverage of the design, and reduce the pattern count.

The following sections describe how to implement user-defined test points:

- Defining User-Defined Test Points
- Previewing the Test Point Logic
- Inserting the Test Point Logic
- User-Defined Test Points Example

Defining User-Defined Test Points

You can use the `set_test_point_element` command to specify the location and type of user-defined test points to insert in the design during DFT insertion, as well as other aspects of test point construction. User-defined test points can be defined at leaf pins, hierarchy pins, and ports. These test points are then inserted during the `insert_dft` command.

To define a user-defined test point, specify the test point type and list of signal pins or ports to be forced, controlled, or observed:

```
dc_shell> set_test_point_element -type test_point_type signal_list
```

For a list of test point types and their descriptions, see "Test Point Types" on page 7-4.

By default, any needed source, sink, and enable signals are supplied by scan registers inserted by the `insert_dft` command. Each scan register is shared by up to eight source, sink, or enable test point signals. For shared source and enable signal registers, the same scan-selected signal value is used by all shared test points. For shared sink signal registers, an XOR observability tree is used to combine the observed signals for capture by the sink register.

You can use the following options to control user-defined test point insertion:

- By default, force and control points use any available TestMode port previously defined with the `set_dft_signal` command. To specify the TestMode or ScanEnable signal that should activate the test points, use the following option:

  ```
  dc_shell> set_test_point_element -control_signal pin_port ...
  ```

  The specified control signal must be defined as a `TestMode` or `ScanEnable` signal type with the `set_dft_signal` command.

- By default, the `insert_dft` command creates a new clock signal, tpclk, to clock any inserted test point scan registers, even when test clocks have been defined. To specify the test clock signal that should clock the scan registers, use the `-clock_signal` option:

  ```
  dc_shell> set_test_point_element -clock_signal clock_name ...
  ```

DFT Compiler supports the following clock name specifications:

- You can specify the name of a scan clock signal, defined as a `ScanClock` signal type with the `set_dft_signal` command.
- In a DFT-inserted OCC controller flow, you can specify the name of a PLL output pin. In this case, DFT Compiler maps the test point clock to the output pin of the corresponding OCC controller during DFT insertion.
- In a user-defined OCC controller flow, you can directly specify the name of an output pin of an existing OCC controller.

For more information on OCC controller flows, see Chapter 9, “On-Chip Clocking Support.

- To specify the maximum number of test points that can share a single source, sink, or enable register, use the following options:

```
dc_shell> set_test_point_element \
            -test_points_per_source_or_sink n ...

dc_shell> set_test_point_element \
            -test_points_per_test_point_enable n ...
```

Source, sink, or enable registers are created as needed, according to the specified sharing limit and the number of test point signal pins provided.

- To specify that the source, sink, or enable signals should come from primary input and output ports instead of scan registers, use the following options:

```
dc_shell> set_test_point_element -scan_source_or_sink false ...

dc_shell> set_test_point_element -scan_test_point_enable false ...
```

The same source, sink, and enable signal sharing is performed, except that primary input and output ports are created instead of scan registers.

- To specify that a specific user-supplied source, sink, or enable signal be used for a given test point definition, use the following options:

```
dc_shell> set_test_point_element \
            -source_or_sink source_or_sink_name ...

dc_shell> set_test_point_element \
            -test_point_enable test_point_enable_name ...
```

When a user-supplied source, sink, or enable signal is specified, it is used for all test points in that `set_test_point_element` command, and the previously described sharing limit and scan register options do not apply.

- To insert power-saving AND gates at the top of an XOR observability tree to avoid excess switching power consumption during scan shift, use the following option:

  ```
  dc_shell> set_test_point_element -power_saving enable ...
  ```

Each `set_test_point_element` command describes a unique test point element definition. Sharing is not performed between test point element definitions. If test points within a limited geographic region should share the same source, sink, or enable signals, they should all be provided in a single `set_test_point_element` command. If test points across a wide geographic region should not share signals to avoid routing congestion, they should be broken up into localized groups and specified with separate `set_test_point_element` commands.

After specifying test point definitions with the `set_test_point_element` command, you can report them with the `report_test_point_element` command, or remove them before DFT insertion with the `remove_test_point_element` command. For more information on these commands, see the man pages.

Previewing the Test Point Logic

To preview the test-point implementation you have defined, use the following command:

```
dc_shell> preview_dft -test_points all
```

This command reports the following information:

- User-defined test-point locations
- Instance names of inserted test-point flip-flops
- Clocks used by inserted test-point flip-flops
- Low-power observability tree status

You can use other options of the `preview_dft` command with the `-test_points` option.

Inserting the Test Point Logic

After you define the user-defined test points, the `insert_dft` command inserts the test point logic. Test point scan registers are placed in the lowest level of hierarchy shared by all test points for that register.

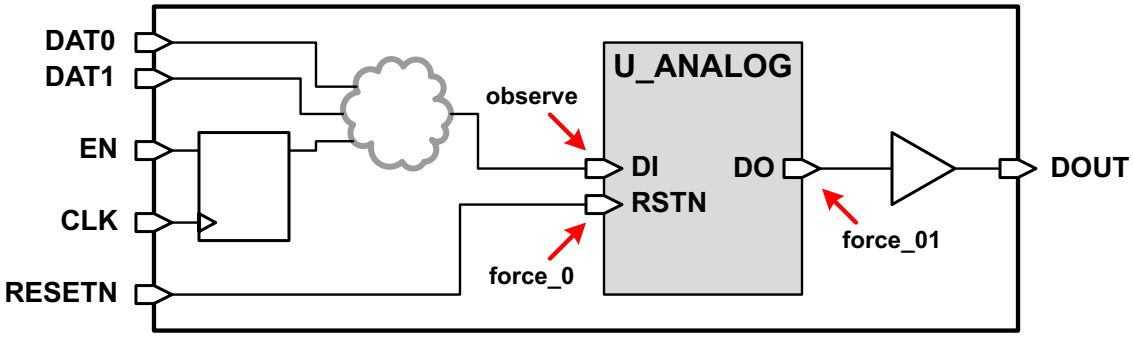
The following additional signals are created, depending on the test configuration:

- A new test point clock signal if a test point clock has not been defined
- A new test-mode signal for low-power observability trees if a test-mode signal has not been defined

User-Defined Test Points Example

Consider the simple design shown in Figure 7-12 and the corresponding example using the user-defined test points flow shown in Example 7-1 on page 7-21.

Figure 7-12 Design Example for User-Defined Test Points



In this design example, some control signals are combined using a cloud of combinational logic, then fed to the DI input of an analog block. The DO output of the analog passes through an output drive buffer so that it can drive a possible long route outside the block. Because this analog block is untestable, the logic fanin to the DI input cannot be observed, and the logic fanout from the DO output cannot be controlled.

To improve the testability of the logic around this analog block, the following user-defined test points can be specified:

- Insert an `observe` test point at the DI input of the analog block to provide observability of the data logic cone:

  ```
  set_test_point_element -type observe U_ANALOG/DI
  ```

- Insert a `force_0` test point at the RSTN pin of the analog block to hold the block in a quiet, low-power reset state during the test program:

  ```
  set_test_point_element -type force_0 U_ANALOG/RSTN
  ```

- Insert a `force_01` test point at the DO output of the analog block to provide controllability of the downstream logic:

  ```
  set_test_point_element -type force_01 U_ANALOG/DO
  ```

A force test point is used at the DO output pin instead of a control test point. The output of the analog block is always unknown, and there is no reason to selectively allow this unknown value to propagate downstream. This force test point is placed at the analog block DO output instead of the DOUT output port so that faults at the drive buffer can be detected. If the test point was placed at the DOUT output port instead, faults between the analog block and the output port could not be detected.

The existing clock CLK is used to clock the test point scan registers. A new TESTMODE port is created to enable the test points.

Example 7-1 Example of a User-Defined Test Point Flow

```
# Read in the design and synthesize it
read_file -format verilog ./rtl/design.v
current_design TEST
link
read_sdc TEST.sdc
compile -scan

# Define the clock, reset, test-mode ports
set_dft_signal -view existing_dft -type ScanClock \
  -port CLK -timing {45 55}
set_dft_signal -view existing_dft -type Reset \
  -port RST -active_state 0
set_dft_signal -view spec -type TestMode \
  -port TESTMODE -active_state 1
set_scan_configuration -chain_count 10

# Provide the UDTP specifications
set_test_point_element -type observe U_ANALOG/DI \
  -clock_signal CLK -control_signal TESTMODE

set_test_point_element -type force_0 U_ANALOG/RSTN \
  -clock_signal CLK -control_signal TESTMODE

set_test_point_element -type force_01 U_ANALOG/DO \
  -clock_signal CLK -control_signal TESTMODE

# Run pre-DFT DRC
create_test_protocol
dft_drc -verbose

# Preview and insert DFT
preview_dft -show all -test_points all
insert_dft

# Run post-DFT DRC
dft_drc -verbose
report_scan_path

# Write out the netlist
write -hierarchical -format ddc -output TEST_udtp_scan.ddc
change_names -rules verilog
write -hierarchical -format verilog -output TEST_udtp_scan.v
```

Using AutoFix

The AutoFix feature automatically fixes scan rule violations resulting from the following types of uncontrollable signals:

- Clock signals
- Asynchronous set signals
- Asynchronous reset signals
- Three-state bus enable signals
- Bidirectional enable signals

By default, only the three-state bus and bidirectional fixing capabilities are enabled. You can enable fixing for one or more additional signal types to fix testability problems in your design. You can specify AutoFix configurations at the top-level design, hierarchical cell, and leaf cell levels. AutoFix is supported in both the multiplexed flip-flop and LSSD scan styles.

When enabled, AutoFix automatically fixes all violations of the specified type(s) found by the `dft_drc` command. If there are no violations, AutoFix makes no changes to the design.

This section contains the following subsections:

When to Use AutoFix

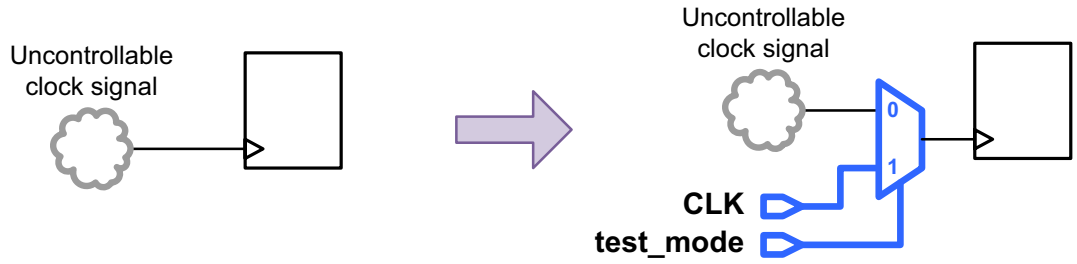
Use AutoFix to resolve testability problems caused by uncontrollable signals, as described in the following subsections:

Uncontrollable Clock Signals

Each scan flip-flop in a design must be clocked by a signal that can be controlled by a primary input port. Otherwise, the clocking of data into the flip-flop cannot be controlled during test. Uncontrollable clock signals are flagged by the `dft_drc` command as design rule violations. If you do not fix these violations, the associated flip-flops are not included in scan chains and faults downstream from the flip-flop outputs might not be detectable.

When AutoFix is enabled for uncontrollable clock signals, it inserts a multiplexer test point to select a controllable clock signal during test, as shown in Figure 7-13. The multiplexer is controlled by a test-mode signal. For mission-mode operation, the test-mode signal is inactive and the circuit operation is unchanged. During test, the signal is asserted and the flip-flop is clocked by the controllable primary input signal.

Figure 7-13 AutoFix Controllability Logic for an Uncontrollable Clock Signal

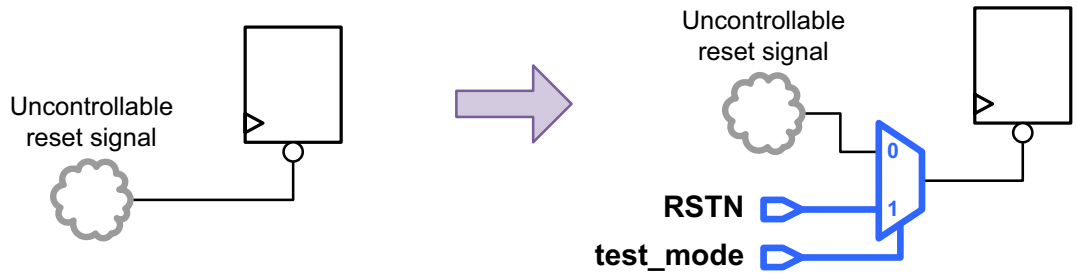


Uncontrollable Asynchronous Set and Reset Signals

The asynchronous set and reset inputs of each flip-flop must be inactive during test. Otherwise, the data in the flip-flop can be set or cleared at any time, leaving unknown data in the flip-flop.

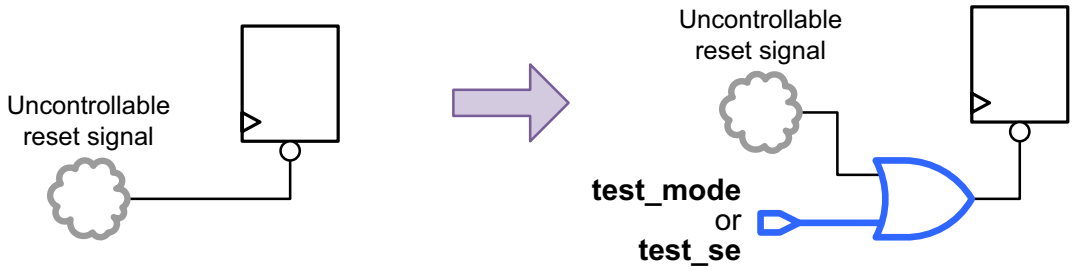
When AutoFix is enabled for uncontrollable asynchronous set or reset signals, by default it inserts a multiplexer test point to select a controllable set or reset signal during test, as shown in Figure 7-14. The multiplexer is controlled by a test-mode signal. For mission-mode operation, the test-mode signal is inactive and the circuit operation is unchanged. During test, the asynchronous signal is driven by the controllable primary input signal.

Figure 7-14 AutoFix MUX-Based Controllability Logic for an Uncontrollable Reset Signal



AutoFix can also insert gating logic to de-assert the uncontrollable asynchronous set or reset signal, as shown in Figure 7-15. The gating logic is controlled by a test-mode or scan-enable control signal. For mission-mode operation, the control signal is inactive and the circuit operation is unchanged. When controlled by a test-mode signal, the asynchronous signal is held inactive throughout the entire test program. When controlled by a scan-enable signal, the asynchronous signal is held inactive during scan shift but remains controlled by the functional logic during scan capture.

Figure 7-15 AutoFix Gating-Based Controllability Logic for an Uncontrollable Reset Signal

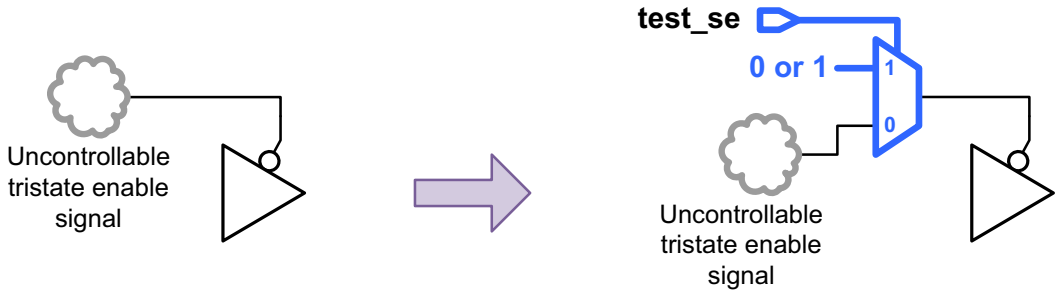


Uncontrollable Three-State Bus Enable Signals

Three-state bus enable signals must be controllable during scan shift. Otherwise, the three-state buses might float or be driven to contention as controlling scan cells shift values through the scan chains.

When AutoFix is enabled for an uncontrollable three-state enable signal, it inserts a multiplexer test point to select a constant enable signal during test, as shown in Figure 7-16. The multiplexer is controlled by a scan-enable signal. For mission-mode and scan capture operation, the test-mode signal is inactive and the circuit operation is unchanged. During scan shift, the signal is asserted during scan shift and the three-state driver is controlled by the constant value. Scan capture operation is unchanged.

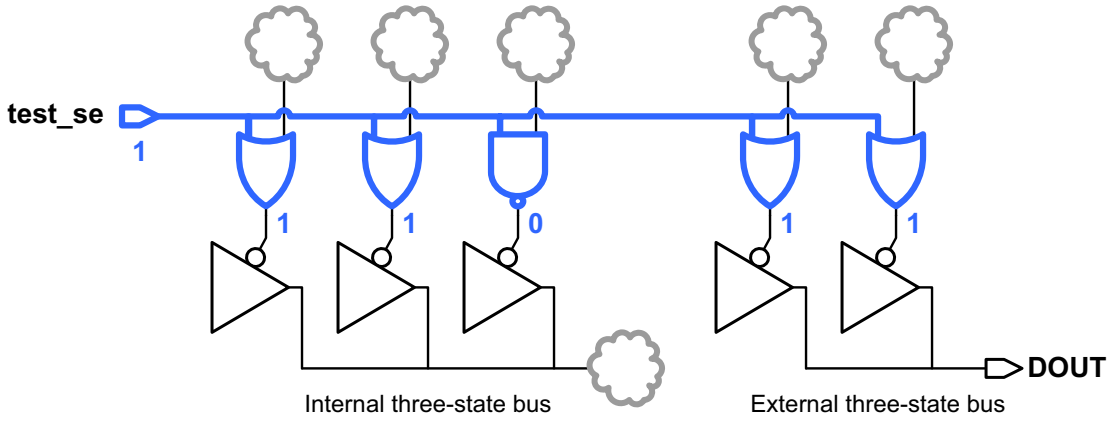
Figure 7-16 AutoFix Controllability Logic for an Uncontrollable Tristate Signal



The constant value applied to the three-state driver depends on the type of three-state bus, shown in Figure 7-17. For an internal bus that exists entirely within the current design, only a single tristate driver is active on each tristate net during scan shift; the rest are held

inactive. For an external bus that has a driver outside the current design, none of the drivers are active on each tristate net during scan shift.

Figure 7-17 Three-State Bus Types



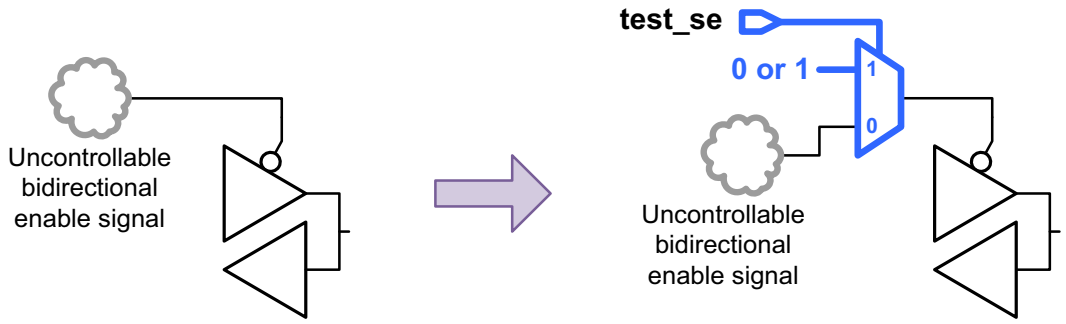
If you have a three-state bus that spans multiple blocks that are AutoFixed separately, at least one block should be fixed as an internal bus so that the bus is driven during scan shift.

Uncontrollable Bidirectional Enable Signals

Bidirectional enable signals must be controllable during scan shift. Otherwise, the bidirectional ports might float or be driven to contention as controlling scan cells shift values through the scan chains.

When AutoFix is enabled for an uncontrollable bidirectional enable signal, it inserts a multiplexer test point to select a constant enable signal during test, as shown in Figure 7-18. By default, the constant value is chosen so that the bidirectional driver is in input mode during scan shift; scan capture operation is unchanged.

Figure 7-18 AutoFix Controllability Logic for an Uncontrollable Bidirectional Signal



Scan-out ports driven by bidirectional drivers are always forced to the output direction during scan shift, regardless of whether AutoFix bidirectional fixing is enabled or how it is

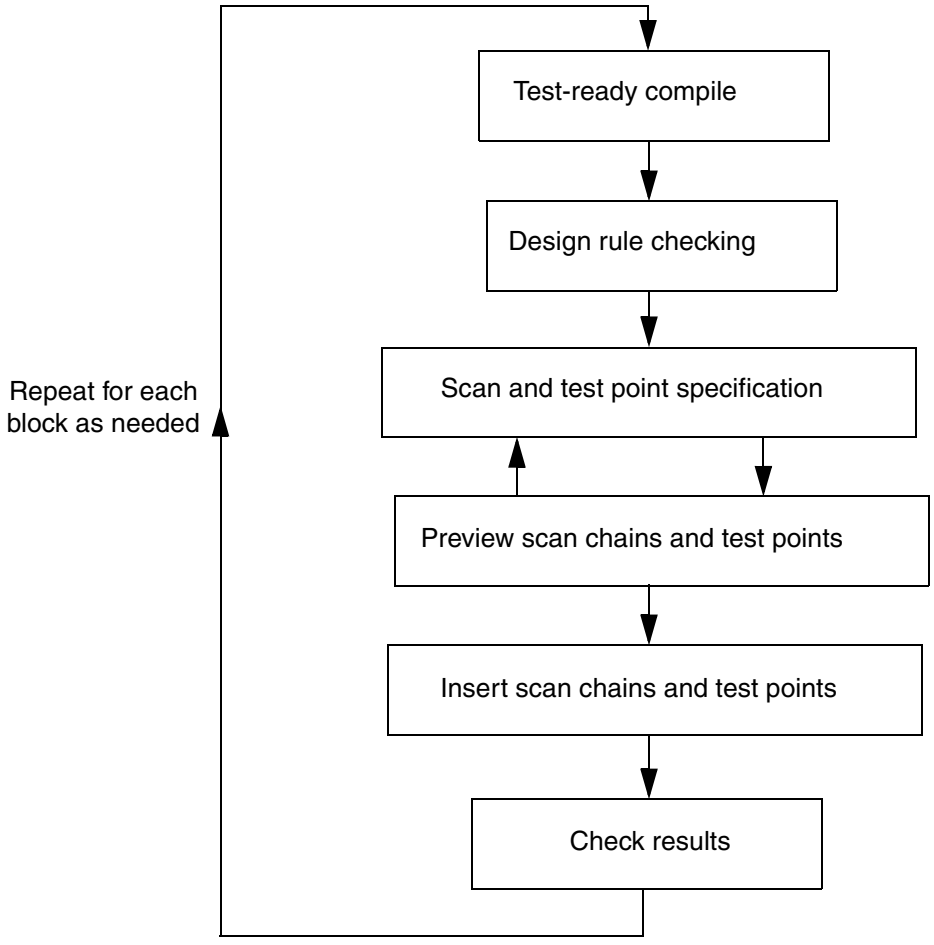
configured. For more information, see “Sharing a Scan Output With a Functional Port” on page 6-32.

If a bidirectional driver cell drives an output port instead of an inout port, AutoFix classifies the driver as a three-state bus driver instead of a bidirectional driver because data values cannot propagate in the input direction.

The AutoFix Flow

The AutoFix design flow is very similar to the ordinary scan synthesis design flow. The general steps in the design flow are illustrated in Figure 7-19.

Figure 7-19 Scan Synthesis Design Flow With AutoFix



You start with the `compile -scan` and `dft_drc` commands. Then you specify the parameters for scan insertion and AutoFix. After you set these parameters, you run the `preview_dft` command to get a preview of the scan chains and AutoFix test points. If necessary, you repeat the setup steps to obtain the desired configuration of scan chains and test points.

When this configuration is satisfactory, you perform scan chain routing and test point insertion with the `insert_dft` command. Finally, you check the results with the `dft_drc` and `report_scan_path` commands.

Using AutoFix

The following sections explain how to use AutoFix:

- Enabling AutoFix Capabilities
- Configuring Clock AutoFixing
- Configuring Set and Reset AutoFixing
- Configuring Three-State Bus AutoFixing
- Configuring Bidirectional AutoFixing
- Applying Hierarchical AutoFix Specifications
- Previewing the AutoFix Implementation

Enabling AutoFix Capabilities

You can enable or disable individual AutoFix capabilities using the options of the `set_dft_configuration` command shown in Table 7-2.

Table 7-2 Options of the set_dft_configuration Command to Enable AutoFix

Signal type to AutoFix	**Enabling option of the `set_dft_configuration` command**	**Default**
Clock signals	`-fix_clock disable | enable`	`disable`
Asynchronous set signals	`-fix_set disable | enable`	`disable`
Asynchronous reset signals	`-fix_reset disable | enable`	`disable`
Three-state bus enable signals	`-fix_bus disable | enable`	`enable`
Bidirectional enable signals	`-fix_bidirectional disable | enable`	`enable`

By default, only the three-state bus and bidirectional fixing capabilities are enabled. To show the current `set_dft_configuration` settings, use the `report_dft_configuration` command. To remove the current settings, use the `reset_dft_configuration` command.

After enabling an AutoFix capability, configure it using the `set_autofix_configuration` command.

Configuring Clock AutoFixing

Use the following options of the `set_autofix_configuration` command to configure clock AutoFixing:

```
set_autofix_configuration
  -type clock
  [-test_data clock_signal]
  [-control_signal test_mode_signal]
```

To specify an existing scan clock signal to use, define it as a `TestData` signal as well as a `ScanClock` signal, then specify it with the `-test_data` option of the `set_autofix_configuration` command:

```
set_dft_signal -view existing_dft -type ScanClock -port CLK \
  -timing [list 45 55]
set_dft_signal -view spec -type TestData -port CLK

set_autofix_configuration -type clock -test_data CLK
```

If no clock signal is specified, AutoFix chooses an available scan clock signal that is also defined as a `TestData` signal. If no such signal exists, AutoFix creates a dedicated scan clock signal to use for fixing uncontrollable asynchronous clock signals.

To specify an existing test-mode signal to use for AutoFixed clock test points, use the `-control_signal` option of the `set_autofix_configuration` command:

```
set_dft_signal -view existing_dft -type TestMode -port AUTOFIX_TM

set_autofix_configuration -type clock -control_signal AUTOFIX_TM
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

Configuring Set and Reset AutoFixing

Use the following options of the `set_autofix_configuration` command to configure set and reset AutoFixing:

```
set_autofix_configuration
  -type set | reset
  [-method mux | gate]
  [-test_data set_reset_signal]
  [-control_signal control_signal]
  [-fix_latch disable | enable]
```

By default, AutoFix uses MUXs to fix uncontrollable asynchronous set or reset signals. To specify an existing controllable set or reset signal to be selected by the MUXs, define it as a `TestData` signal as well as a `Set` or `Reset` signal, then specify it with the `-test_data` option of the `set_autofix_configuration` command:

```
set_dft_signal -view existing_dft -type Reset -port RSTN -active_state 0
set_dft_signal -view spec -type TestData -port RSTN -active_state 0

set_autofix_configuration -type set -test_data RSTN
set_autofix_configuration -type reset -test_data RSTN
```

If no such signal is specified, AutoFix chooses a set or reset signal that is also defined as a `TestData` signal. If no such signal exists, AutoFix creates a dedicated asynchronous reset signal to use for fixing uncontrollable asynchronous set and reset signals.

Use the `-type set` or `-type reset` option of the `set_autofix_configuration` command to configure set or reset AutoFixing, respectively. You can use a single existing asynchronous set or reset signal to AutoFix both uncontrollable set and reset signals. However, the same asynchronous signal cannot be used to AutoFix both the set and reset pins of the same cell. If this occurs in your design, you must specify separate set and reset signals.

To specify an existing test-mode signal to control the MUXs, use the `-control_signal` option of the `set_autofix_configuration` command:

```
set_dft_signal -view spec -type TestMode -port AUTOFIX_TM

set_autofix_configuration -type set -control_signal AUTOFIX_TM
set_autofix_configuration -type reset -control_signal AUTOFIX_TM
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

To fix uncontrollable sets and resets using gating logic instead of a MUX, use the `-method` option to specify the `gate` fixing method:

```
set_autofix_configuration -type set -method gate
set_autofix_configuration -type reset -method gate
```

To specify a scan-enable or test-mode signal to control the gating logic, use the `-control_signal` option of the `set_autofix_configuration` command:

```
set_autofix_configuration -type set -method gate -control_signal TM_FIX
set_autofix_configuration -type reset -method gate -control_signal TM_FIX
```

If no control signal is specified, AutoFix chooses an available test-mode signal that is not already used for another purpose. If no such signal exists, AutoFix creates a dedicated test mode signal.

By default, AutoFix does not consider set or reset pins of latch cells. To consider latch set and reset pins, use the `-fix_latch enable` option of the `set_autofix_configuration` command.

Configuring Three-State Bus AutoFixing

Use the following options of the `set_autofix_configuration` command to configure three-state bus AutoFixing:

```
set_autofix_configuration
  -type internal_bus | external_bus
  [-method no_disabling | enable_one | disable_all]
```

The `internal_bus` type configures three-state buses that do not drive ports of the current design. The `external_bus` type configures three-state buses that drive ports of the current design.

Use the `-method` option to control how the tristate drivers are to be enabled during scan shift. The values are

- `no_disabling` – do not insert controlling logic
- `enable_one` – enable one driver; disable all other drivers
- `disable_all` – disable all drivers

The default methods for the `internal_bus` and `external_bus` types are `enable_one` and `disable_all`, respectively. If you have a three-state bus that spans multiple blocks that are AutoFixed separately, configure one block to use the `enable_one` method to avoid bus float during scan shift.

AutoFix combines all scan-enable signals defined without the `-usage` option into a single merged signal to enable all three-state bus AutoFix test points. This ensures that no three-state contention occurs when data is scanned though the shift chains. You cannot use the `-control_signal` option to specify a control signal for three-state bus AutoFixing.

Configuring Bidirectional AutoFixing

Use the following options of the set_autofix_configuration command to configure bidirectional AutoFixing:

```
set_autofix_configuration
  -type bidirectional
  [-method input | output | no_disabling]
```

Use the -method option to control how the bidirectional drivers are to be enabled during scan shift. The values are

- input – force bidirectional drivers to input direction
- enable_one – force bidirectional drivers to output direction
- no_disabling – do not insert controlling logic

The default method for the bidirectional type is input.

AutoFix combines all scan-enable signals defined without the -usage option into a single merged signal to enable all bidirectional AutoFix test points. This ensures that no bidirectional contention occurs when data is scanned through the shift chains. You cannot use the -control_signal option to specify a control signal for bidirectional AutoFixing.

Applying Hierarchical AutoFix Specifications

By default, AutoFix specifications applied with the set_autofix_configuration command apply to the entire design. You can fix only certain hierarchical cells or leaf cells by specifying them with the -include_elements option of the set_autofix_configuration command. For example,

```
set_autofix_configuration -type set -include_elements {MY_CORE}
set_autofix_configuration -type reset -include_elements {MY_CORE}
```

You can also exclude certain hierarchical cells or leaf cells by specifying them with the -exclude_elements option of the set_autofix_configuration command. For example,

```
set_autofix_configuration -type set -exclude_elements {U_IP_CORE}
set_autofix_configuration -type reset -exclude_elements {U_IP_CORE}
```

You can specify both the -include_elements and -exclude_elements options to exclude cells within an included cell, but not vice versa.

The set_autofix_configuration command applies a global configuration; subsequent specifications for a capability take precedence over previous specifications. If you need to specify different fixing configurations for different areas of the design, use the

set_autofix_element command, which differs from the set_autofix_configuration command as follows:

- The set_autofix_element command applies a local fixing configuration to the specified list of leaf cells, hierarchical cells, nets, or designs.
- Multiple set_autofix_element command specifications can be applied to the design.

You can mix global and local specifications. The global fixing configuration is used where no local configuration applies. For example,

```
set_autofix_configuration -type clock -test_data CLK
set_autofix_element -type clock -test_data RXCLK {U_RX_BLK}
set_autofix_element -type clock -test_data TXCLK {U_TX_BLK}
```

Note the following precedence rules:

- Local specifications, applied with the set_autofix_element command, take precedence over global specifications, applied with the set_autofix_configuration command.
- Local specifications at lower hierarchy levels take precedence over local specifications at higher levels.
- If multiple specifications apply to the same object, later specifications take precedence over earlier specifications.
- Specifications for different fixing types are independent and do not affect each other.

Previewing the AutoFix Implementation

After you enable and configure the AutoFix capabilities, you can preview the scan architecture with the fixes included. To do this, use the preview_dft -test_points all -show {cells} command, which provides the following information:

- The number of test points implemented by AutoFix
- The scan chain configuration with AutoFix considered

The test point section of the preview report shows the test points to be implemented. It does not include three-state bus or bidirectional test points.

The scan cells section of the preview report shows only the sequential cells included in scan chains; it does not show the cells omitted due to DRC violations. Check to see if the DRC-violating cells to be AutoFixed exist in the report. If needed, you can revise the AutoFix configuration and rerun the preview_dft command.

Note:

Before DFT insertion, the dft_drc command always reports the DRC violations without AutoFix considered; use the preview_dft command to assess the effects of AutoFix.

When you are satisfied with the scan chains and test points reported by the `preview_dft` command, insert DFT using the `insert_dft` command. You should always run the `dft_drc` command after DFT insertion to check for any remaining design rule violations.

AutoFix Script Example

The script in Example 7-2 fixes uncontrollable clock and reset signals using AutoFix.

Example 7-2 Scan Synthesis With Test Point Insertion

```
current_design MY_DESIGN
compile -scan
create_test_protocol
dft_drc

# configure scan
set_scan_configuration -clock_mixing mix_edges ...

# configure DFT signals
set_dft_signal -view spec -type TestMode -port TEST_MODE

set_dft_signal -view existing_dft -type ScanClock \
  -port {CLK RXCLK TXCLK} -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port RSTN -active_state 0

# configure signals for AutoFix
set_dft_signal -view spec -type TestData -port {CLK RXCLK TXCLK RSTN}

# configure clock AutoFix
set_autofix_configuration -type clock -test_data CLK
set_autofix_element -type clock -test_data RXCLK {U_RX_BLK}
set_autofix_element -type clock -test_data TXCLK {U_TX_BLK}

# configure reset AutoFix
set_dft_configuration -fix_reset enable
set_autofix_configuration -type reset -test_data RSTN \
  -exclude_elements {U_IP_CORE}

preview_dft -test_points all
insert_dft
dft_drc
report_scan_path -chain all
```

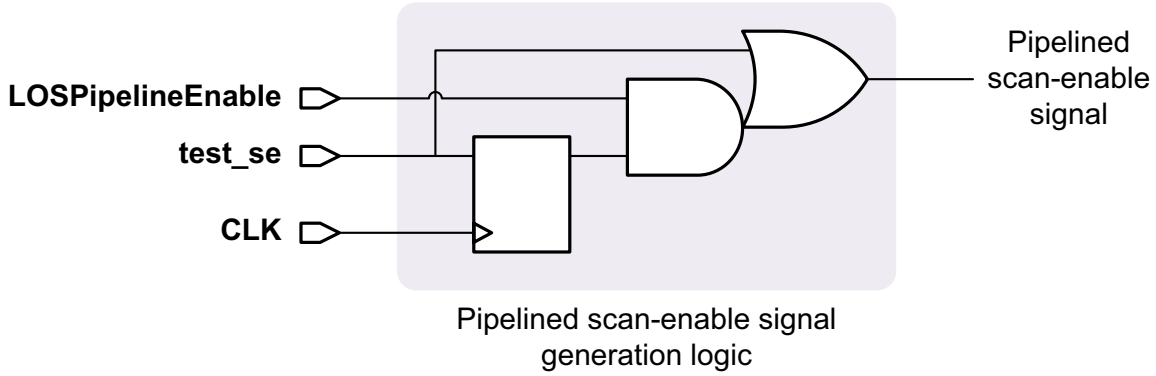
Pipelined Scan-Enable Architecture

Pipelined scan-enable signals are used for transition delay testing by making use of launch-on-shift (LOS). This method of transition delay testing requires additional circuitry to manipulate the scan-enable signals. Also, the pipelined scan-enable signals must be treated as single-cycle paths at the capture clock frequency.

Pipelined scan-enable signals are never used with launch-on-capture (LOC), which does not require additional scan-enable circuitry and can use relaxed timing on the scan-enable signals.

Figure 7-20 shows a representation of the scan-enable pipelining logic.

Figure 7-20 Pipelined Scan-Enable Architecture



To enable pipelined scan-enable signals, use the following command:

```
set_scan_configuration
    -pipeline_scan_enable true
    [-pipeline_fanout_limit integer]
```

The `-pipeline_fanout_limit` option specifies the maximum number of scan flip-flops that can be driven by each pipelined scan-enable logic construct. The default is to not apply a limit. For more information, see the man page.

Note:

If you combine clock mixing with pipelined scan-enable signals, lock-up latches might be ineffective because the clock timing in the extra shift uses the launch waveform table rather than shift timing. As a result, the first flip-flop following the clock domain crossing is marked as disturbed by ATPG. You can avoid this problem by disabling clock mixing with the `-clock_mixing no_mix` option of the `set_scan_configuration` command.

The pipelined scan-enable architecture requires a pipeline enable signal. This signal determines the launch mode: launch-on-shift (LOS) when it is active and launch-on-capture

(LOC) when it is inactive. To explicitly define a pipeline enable signal, use the `set_dft_signal` command as follows:

```
dc_shell> set_dft_signal -view spec -type LOSPipelineEnable \
            -port PSE_EN -active_state 1 -test_mode all
```

If no such signal exists, the tool creates a signal named `global_pipe_se`.

Post-DFT DRC uses the LOC protocol. The protocol generated by TetraMAX ATPG also uses the LOC protocol. The LOS protocol must be created by copying and modifying this protocol.

Pipelined Scan-Enable Signals in Hierarchical Flows

The following two hierarchical flows are supported for pipelined scan-enable architecture:

- Pipelined scan-enable registers inserted at the subdesign level

  Use the following setting at both the subdesign level and the top level:

  ```
  set_scan_configuration -pipeline_scan_enable true
  ```

  The DFT architecture recognizes when a test model already contains pipeline scan-enable logic and inserts this logic only where it does not already exist. The result is that all scan cells will have exactly one level of scan-enable pipelining.

- Pipelined scan-enable registers inserted only at the top level

  Use the following settings at the subdesign level:

  ```
  set_scan_configuration -domain_based_scan_enable true \
      -clock_mixing no_mix
  ```

  Use the following setting at the top level:

  ```
  set_scan_configuration -pipeline_scan_enable true
  ```

You can use a mixture of the two types of subdesigns (those with pipeline scan-enable logic inserted in the block and those without). Top-level integration adds or omits pipeline scan-enable logic as required for each subdesign.

Clock mixing should not be used at the subdesign level. It can be used at the top level, but there are potential side effects. For more information, see the clock-mixing note in “Pipelined Scan-Enable Architecture” on page 7-34.

Limitations

Note the following requirements and limitations:

- All scan-enable signals must have the same pipelined scan-enable depth after DFT insertion. Valid depths are zero (not pipelined) and one (pipelined).
- You cannot define scan-enable signals with the `-usage` option of the `set_dft_signal` command when using the pipelined scan-enable architecture.
- The pipelined scan-enable architecture does not work correctly with DFT-inserted OCC controllers.
- If you use power-aware functional output gating with pipelined scan-enable signals, the critical path for transition delay testing becomes the path from the scan-enable pipeline register through the toggle suppression gate.

Multiple Test Modes

A design's scan interface must accommodate a variety of structural tests. Scan test, burn-in, and other tests performed at various production steps might require different types of access to scan elements of a design. To accommodate these different test requirements, multiple scan architectures can be provided on the same scan design. This approach is also useful for complex test schemes such as DFTMAX compression adaptive scan and core wrapping, which target tests in different parts of a design at different times.

This section explains the process for setting up architectures to perform multiple test-mode scan insertion. It includes the following subsections:

- Introduction to Multiple Test Modes
- Defining Test Modes
- Applying Test Specifications to a Test Mode
- Using Multiple Test Modes in Hierarchical Flows
- Supported Test Specification Commands for Test Modes
- Multiple Test-Mode Scan Insertion Script Examples

Introduction to Multiple Test Modes

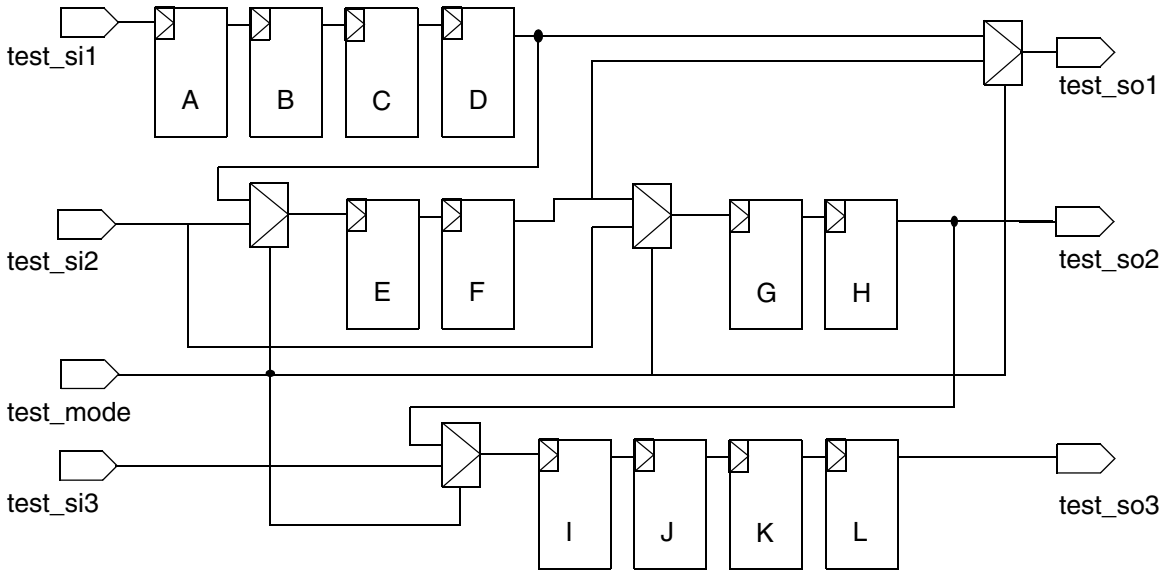
You can reconfigure the scan chains in your design to suit various tester requirements by defining different modes of operation, called *test modes*. For example, suppose you have a simple design with 12 scan cells that must operate in two different scan modes. In one

mode, scan data is shifted through two chains (six cells each), and in the other mode, scan data is shifted through three chains (four cells each). Figure 7-21 shows how DFT Compiler inserts MUXs to support these two scan chain configurations. These MUXs are known as *reconfiguration MUXs*.

Note:

Reconfiguration MUXs might appear at any level in your design. This is usually dependent on the location of the scan elements where the MUXing takes place.

Figure 7-21 Configuring Scan Chains With Test-Mode Logic



Each test mode is activated by asserting one or more test-mode signals according to a particular test-mode encoding. Different test modes can have different scan-in and scan-out pin counts, and can even have independent sets of scan-in and scan-out pins altogether.

Defining Test Modes

To define a test mode, use the `define_test_mode` command:

```
define_test_mode test_mode_name
```

Each test mode must have a unique name that is used to refer to the test mode in subsequent DFT commands or reports. The name of the default standard scan test mode is

`Internal_scan`. You can configure this default mode and define additional modes, or you can create an entirely new set of named modes without using the default test mode.

Example 7-3 defines three newly-named test modes.

Example 7-3 Defining Three New User-Defined Test Modes

```
define_test_mode LONG_CHAINS
define_test_mode MEDIUM_CHAINS
define_test_mode SHORT_CHAINS
```

Defining the Usage of a Test Mode

By default, a test mode represents a standard scan test mode of operation. To specify how a test mode is to be used, use the `-usage` option of the `define_test_mode` command. The valid keywords for this option are:

- `scan` – This is the traditional standard scan mode operation, which is the default if the `-usage` option is not specified. The scan chains are driven directly by top-level scan-in ports, and they drive, in turn, top-level scan-out ports. This mode is used for testing all logic internal to the core.
- `scan_compression` – This is the compressed scan mode of operation provided by the DFTMAX tool. In this mode, the internal scan chains are driven by scan data decompressors, and the scan chains drive the scan data compressors. This mode is used for testing all logic internal to the core with reduced test data volume and test application time.
- `wrp_if` – This is the inward facing, or INTEST, mode of wrapper operation. This mode is used for testing all logic internal to the core. In this mode, wrappers are enabled and configured to drive and capture data within the design, in conjunction with the internal scan chains.
- `wrp_of` – This is the outward facing, or EXTEST, mode of wrapper operation. This mode is used for testing all logic external to the design. Wrappers are enabled and configured to drive and capture data outside of the design. In this mode the internal chains are disabled.
- `wrp_safe` – This is the safe wrapper mode. In this mode, the internal chains are disabled, and the internal core is protected from any toggle activity. This mode is optional and provides isolation of the core while other cores are being tested. When active, safe mode enables driving steady states into or out of the design.

Example 7-4 defines three standard scan test modes plus a DFTMAX compressed scan mode.

Example 7-4 Providing Test-Mode Usage Information With the -usage Option

```
define_test_mode LONG_CHAINS
define_test_mode MEDIUM_CHAINS
define_test_mode SHORT_CHAINS
define_test_mode COMPRESSED_CHAINS -usage scan_compression
```

After test modes have been defined, you can use the `list_test_modes` command to report the currently defined test-mode names. Example 7-5 shows the report for the three standard scan test modes defined in Example 7-3.

Example 7-5 Test Modes Reported by the list_test_modes Command

```
Test Modes
==========

        Name: LONG_CHAINS
        Type: InternalTest
        Focus:

        Name: MEDIUM_CHAINS
        Type: InternalTest
        Focus:

        Name: SHORT_CHAINS
        Type: InternalTest
        Focus:

        Name: Mission_mode
        Type: Normal
```

Defining the Encoding of a Test Mode

Each test mode is activated by asserting one or more test-mode signals according to a particular encoding associated with that test mode. To declare these test-mode signals, use the `set_dft_signal -type TestMode` command. If no test-mode signals are available, or not enough test-mode signals are available to satisfy the test-mode encodings, DFT Compiler creates new test-mode ports as needed.

By default, DFT Compiler assigns a binary encoding to the test modes. Binary encoding requires the fewest number of test-mode signals. With binary encoding, *n* test-mode signals can provide test-mode encodings for up to 2^n-1 test modes, allowing for an encoding that deactivates all test modes and activates mission mode.

To report the test-mode signals and encodings associated with each test mode, use the `preview_dft` command. Example 7-6 shows the preview report for the three standard scan test modes defined in Example 7-3.

Example 7-6 Test-Mode Encodings Reported by the preview_dft Command

```
================================
Test Mode Controller Information
================================

Test Mode Controller Ports
--------------------------
test_mode: test_mode2
test_mode: test_mode1

Test Mode Controller Index (MSB --> LSB)
----------------------------------------
test_mode2, test_mode1

Control signal value - Test Mode
--------------------------------
00 LONG_CHAINS - InternalTest

01 MEDIUM_CHAINS - InternalTest

10 SHORT_CHAINS - InternalTest
```

You can also specify one-hot test-mode encoding by using the following command:

```
set_dft_configuration -mode_decoding_style one_hot
```

This command causes one-hot test-mode encodings to be used, and simplified test-mode decoding logic to be built that takes advantage of the properties of one-hot encodings. One-hot encodings provide simplified decoding logic, at the expense of a greater number of required test-mode signals. With one-hot encoding, *n* test-mode signals can provide test-mode encodings for up to *n* test modes, with mission mode being activated by an encoding where none of the test-mode signals are asserted.

Note:

One-hot test-mode encodings take the active state of each test-mode signal into account. If you define all test-mode signals using the `-active_state 0` option of the `set_dft_signal` command, each one-hot encoding contains a single asserted `0` value.

You can also specify your own test-mode encodings with the `-encoding` option of the `define_test_mode` command. The syntax of the encoding argument consists of one or more test-mode signal names and binary value pairs. These pairs are separated by a space when multiple ports are specified.

Example 7-7 shows how to use a binary encoding where mission mode is activated by the `00` encoding and the three test-mode encodings have at least one test-mode signal asserted.

Example 7-7 Specifying User-Defined Binary Test-Mode Encodings

```
set_dft_signal -view spec -port {TM1 TM0} -type TestMode

define_test_mode LONG_CHAINS   -encoding {TM1 0 TM0 1}
define_test_mode MEDIUM_CHAINS -encoding {TM1 1 TM0 0}
define_test_mode SHORT_CHAINS  -encoding {TM1 1 TM0 1}
```

If you are providing your own one-hot encodings, configure the test-mode decoding to `one_hot` to build simplified one-hot decoding logic. Example 7-8 shows how to configure one-hot encodings for the three example test modes.

Example 7-8 Specifying User-Defined One-Hot Test-Mode Encodings

```
set_dft_signal -view spec -port {TM2 TM1 TM0} -type TestMode

set_dft_configuration -mode_decoding_style one_hot

define_test_mode LONG_CHAINS   -encoding {TM2 0 TM1 0 TM0 1}
define_test_mode MEDIUM_CHAINS -encoding {TM2 0 TM1 1 TM0 0}
define_test_mode SHORT_CHAINS  -encoding {TM2 1 TM1 0 TM0 0}
```

If your test-mode control signals come from internal design pins, such as the outputs of test control registers, use the `-hookup_pin` option from the internal pins flow to make the connections. Example 7-9 shows how to hook up two test-mode signals to corresponding control register output pins.

Example 7-9 Using Internal Design Pins for Test-Mode Signals

```
set_dft_signal -view spec -type TestMode \
    -port TM1 -hookup_pin TESTCTL_reg[1]/Q
set_dft_signal -view spec -type TestMode \
    -port TM0 -hookup_pin TESTCTL_reg[0]/Q

define_test_mode LONG_CHAINS   -encoding {TM1 0 TM0 1}
define_test_mode MEDIUM_CHAINS -encoding {TM1 1 TM0 0}
define_test_mode SHORT_CHAINS  -encoding {TM1 1 TM0 1}
```

If the internal pins do not directly correspond to top-level ports, you must use the internal pins flow. For more information, see “Internal Pins Flow” on page 7-92.

Applying Test Specifications to a Test Mode

After you have defined test modes, you can apply mode-specific test specifications to each test mode. Not all commands and options can be used to apply mode-specific test specifications. For more information on the available commands and options, see "Supported Test Specification Commands for Test Modes" on page 7-46.

When you apply test specifications before defining any test modes, they are applied to the default test mode. As new test modes are defined, they inherit the test specification settings from this default mode. You can use this behavior to predefine global test specifications shared by all modes.

After you define a test mode with the `define_test_mode` command, that test mode becomes the *current test mode*. Subsequent scan specification commands apply only to that test mode. You can use this behavior to implicitly apply mode-specific test specifications after defining a test mode, as shown in Example 7-10.

Example 7-10 Applying Scan Specifications Using Implicitly Defined Current Test Mode

```
define_test_mode LONG_CHAINS
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks

define_test_mode MEDIUM_CHAINS
set_scan_configuration -chain_count 3 -clock_mixing mix_clocks

define_test_mode SHORT_CHAINS
set_scan_configuration -chain_count 5
```

You can also use the `current_test_mode` command to change the current test mode at any time, as shown in Example 7-11.

Example 7-11 Applying Scan Specifications Using Explicitly Defined Current Test Mode

```
define_test_mode LONG_CHAINS
define_test_mode MEDIUM_CHAINS
define_test_mode SHORT_CHAINS

current_test_mode LONG_CHAINS
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks

current_test_mode MEDIUM_CHAINS
set_scan_configuration -chain_count 3 -clock_mixing mix_clocks

current_test_mode SHORT_CHAINS
set_scan_configuration -chain_count 5
```

You can use the -test_mode option to directly apply a scan specification to a particular test mode at any time, regardless of the current test mode, as shown in Example 7-12.

Example 7-12 Applying Scan Specifications Using the -test_mode Option

```
set_scan_configuration -test_mode LONG_CHAINS \
    -chain_count 2 -clock_mixing mix_clocks
set_scan_configuration -test_mode MEDIUM_CHAINS \
    -chain_count 3 -clock_mixing mix_clocks
set_scan_configuration -test_mode SHORT_CHAINS \
    -chain_count 5
```

To apply a scan specification to all test modes after previously defining some test modes, use the -test_mode all option, as shown in Example 7-13.

Example 7-13 Applying Scan Specification to All Test Modes

```
set_dft_signal -view spec -test_mode all \
    -type ScanDataIn -port {SI1 SI2 SI3}
```

Note:

The -test_mode option also accepts the all_dft keyword, which is equivalent to all.

When multiple scan specification commands are applied to a test mode, they are applied cumulatively. A new scan specification command overwrites a previous scan specification according to the same precedence rules used in a single test-mode flow.

Example 7-14 shows how to apply clock mixing to all test modes except for the SHORT_CHAINS test mode:

Example 7-14 Overwriting a Previous Scan Specification Setting for a Test Mode

```
set_scan_configuration -test_mode all \
    -clock_mixing mix_clocks
set_scan_configuration -test_mode LONG_CHAINS \
    -chain_count 2
set_scan_configuration -test_mode MEDIUM_CHAINS \
    -chain_count 3
set_scan_configuration -test_mode SHORT_CHAINS \
    -chain_count 5 -clock_mixing no_mix  ;# overwrites mix_clocks
```

The default for the -clock_mixing option is no_mix. In this example, the first command applies the -clock_mixing no_clocks option to all test modes. The subsequent two commands configure the chain counts for the first two test modes. Since the -chain_count option does not overwrite the -clock_mixing option, the mix_clocks specification remains in place for the LONG_CHAINS and MEDIUM_CHAINS test modes. In the last command, the -clock_mixing option reapplies the default clock mixing behavior of no_mix to the SHORT_CHAINS test mode to overwrite the mix_clocks behavior previously applied to all test modes.

When applying scan specifications in a multiple test-mode environment, perform these steps:

1. Define TestMode signals with the `set_dft_signal -test_mode all` command.
2. Define all test modes, their usage, and optional encodings with the `define_test_mode` command.
3. Define clock and asynchronous DFT signals and constants that are common to all test modes with the `set_dft_signal -test_mode all` command.
4. Define any mode-specific DFT signals with the `set_dft_signal -test_mode` *`test_mode_name`* command.
5. Specify any scan specifications to be used in all test modes using the `-test_mode all` option of the `set_scan_configuration` and `set_scan_path` commands.
6. Specify any scan specifications to be used in specific test modes using the `-test_mode` *`test_mode_name`* option of the `set_scan_configuration` and `set_scan_path` commands.

Note:

If you reference any test modes using the `-test_mode` option of any DFT configuration commands, you must define those test modes with the `define_test_mode` command before referencing them.

Using Multiple Test Modes in Hierarchical Flows

In hierarchical flows, different cores might have different test modes defined. In this case, DFT Compiler creates as many top-level test modes as needed to accommodate all of the core-level test modes.

By default, the relationship between core-level and top-level test modes is determined by test mode name according to the following rules:

- For each core-level test mode, a top-level test mode of the same name is created.
- If multiple cores share a test mode with the same name, those core-level test modes are included in a top-level test mode of the same name.
- If a core does not have a test mode that matches a top-level test mode name, it is excluded from that top-level test mode.

The `preview_dft` and `insert_dft` commands report the core-level test modes used in each of the top-level test modes.

Consider an example where coreA has two test modes named SHORT_CHAINS and MEDIUM_CHAINS, and coreB has two test modes named MEDIUM_CHAINS and

LONG_CHAINS. At the top level, DFT Compiler creates all three test modes. The `preview_dft` and `insert_dft` commands report the core-level test modes as shown in Example 7-15.

Example 7-15 Top-Level Test Mode Report for Default Name-Based Relationships

```
Control signal value - Integration Test Mode
  Core Instance - Test Mode
-----------------------------------------
00 SHORT_CHAINS - InternalTest
  coreA - SHORT_CHAINS: InternalTest

01 MEDIUM_CHAINS - InternalTest
  coreA - MEDIUM_CHAINS: InternalTest
  coreB - MEDIUM_CHAINS: InternalTest

10 LONG_CHAINS - InternalTest
  coreB - LONG_CHAINS: InternalTest
```

Note that coreB does not participate in top-level test mode SHORT_CHAINS, and coreA does not participate in top-level test mode LONG_CHAINS.

At the top level, you can override the default name-based association of core-level test modes by using the `define_test_mode -target` command. The `-target` option takes a list of core and test mode pairs that should be included in that top-level test mode.

The previous example can be modified to use the closest matches for the missing core-level test modes, as shown in Example 7-16.

Example 7-16 Specifying User-Defined Core-Level Test Mode Relationships

```
# top-level test mode definitions
define_test_mode SHORT_CHAINS \
  -target {coreA SHORT_CHAINS    coreB MEDIUM_CHAINS}
define_test_mode MEDIUM_CHAINS \
  -target {coreA MEDIUM_CHAINS   coreB LONG_CHAINS}
define_test_mode LONG_CHAINS \
  -target {coreA MEDIUM_CHAINS   coreB LONG_CHAINS}
```

The `preview_dft` and `insert_dft` commands report the core-level test modes as shown in Example 7-17.

Example 7-17 Top-Level Test Mode Report For User-Defined Test Mode Relationships

```
Control signal value - Integration Test Mode
  Core Instance - Test Mode
-----------------------------------------
00 SHORT_CHAINS - InternalTest
  coreA - SHORT_CHAINS: InternalTest
  coreB - MEDIUM_CHAINS: InternalTest
```

```
01 MEDIUM_CHAINS - InternalTest
  coreA - MEDIUM_CHAINS: InternalTest
  coreB - MEDIUM_CHAINS: InternalTest

10 LONG_CHAINS - InternalTest
  coreA - MEDIUM_CHAINS: InternalTest
  coreB - LONG_CHAINS: InternalTest
```

Note:

If you use the `-target` option of the `define_test_mode` command, you must completely define the core test mode relationships for all cores and test modes. When the `-target` option is used, name-based test mode association is no longer performed for any core or test mode.

Supported Test Specification Commands for Test Modes

This section lists the commands and options you can use to configure DFT insertion for specific test modes. These commands and options honor the `-test_mode` option or the current test-mode focus. Other DFT configuration commands and options apply to all test modes.

set_dft_signal

You can use the `set_dft_signal -test_mode` command to declare different DFT signals for different test modes. For example, each test mode can have different scan-in and scan-out ports.

Keep in mind that pre-DFT DRC only analyzes the global `all` test mode; it does not consider mode-specific signals applied to other modes. As a result,

- Mode-specific signals defined with the `-view spec` option, such as scan-in and scan-out signals, can be safely made, as these signals do not yet exist during pre-DFT DRC.
- Mode-specific signals defined with the `-view existing` option, such as constants and reset signals, must be made with care, as these signals are not considered during pre-DFT DRC. However, they are incorporated into the mode-specific test protocols used by post-DFT DRC.

  Note that you can apply a baseline set of signals to the `all` test mode to be used by pre-DFT DRC, along with mode-specific signals to be used by post-DFT DRC.

All test modes must share the same scan-enable signals. You cannot specify different scan-enable signals for different test modes.

If the `-test_mode` option is not specified, this command applies to the current test mode.

set_scan_configuration

The following `set_scan_configuration` options can be applied to specific test modes:

- `-chain_count`
- `-max_length`
- `-clock_mixing`

If the `-test_mode` option is not specified, these options apply to the current test mode. Other options of the `set_scan_configuration` command apply to all test modes.

set_scan_path

The following `set_scan_path` options can be applied to specific test modes:

- `-scan_master_clock`
- `-exact_length`

Use the `set_scan_path -test_mode` command to provide scan chain specifications for the test modes in your design.

The scan path specification can be given for any chains in any defined test mode. It can include pin access information provided with the `-scan_data_in`, `-scan_data_out`, `-scan_enable`, `-scan_master_clock`, and `-scan_slave_clock` options. The specification can also specify a list of design elements to be included. When specifying pins with the `-scan_data_in` or `-scan_data_out` options, the signals must be previously defined with the `set_dft_signal` command using the `ScanDataIn` or `ScanDataOut` signal types.

If the scan path specification applies to a test mode which has the usage specified as `scan`, both the port and hookup arguments of the `set_dft_signal` command can be specified for the `ScanDataIn` and `ScanDataOut` signals.

If the scan path specification applies to a test mode which has the usage specified as `scan_compression`, then only the `-hookup` option of the `set_dft_signal` command can be specified for the `ScanDataIn` and `ScanDataOut` signals. A port argument must not be used for compressed scan mode scan path definitions, as these codec-connected compressed chains have no direct access from the ports.

If the `-test_mode` option is not specified, the specification applies to the current test mode.

Multiple Test-Mode Scan Insertion Script Examples

This section provides examples of basic scan, DFTMAX compressed scan, and core wrapping scripts in a multiple test-mode environment.

Example 7-18 shows a basic scan script that defines four scan test modes. The scan1 mode has one chain, the scan2 mode has two chains, the scan3 mode has four chains, and the scan 4 mode has eight chains. Each set of chains uses separate scan-in and scan-out pins.

Example 7-18 Basic Scan Multiple Test-Mode Script

```
## Define the pins for use in any test mode with "-test_mode all"
    for {set i 1} {$i <= 15 } { incr i 1} {
     create_port -direction in test_si[$i]
     create_port -direction out test_so[$i]
     set_dft_signal -type ScanDataIn -view spec -port test_si[$i] \
          -test_mode all
     set_dft_signal -type ScanDataOut -view spec -port test_so[$i] \
          -test_mode all
    }

#Define Test Clocks
set_dft_signal -view existing_dft -type TestClock -timing {45 55} \
     -port clk_st

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port \
     [list i_trdy_de i_trdy_dd i_cs]

#Define the test modes
define_test_mode scan1 -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
define_test_mode scan2 -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0}
define_test_mode scan3 -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1}
define_test_mode scan4 -usage scan \
     -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0}

#Configure the basic scan modes
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
     -test_mode scan1
set_scan_configuration -chain_count 2 -clock_mixing mix_clocks \
     -test_mode scan2
set_scan_configuration -chain_count 4 -clock_mixing mix_clocks \
     -test_mode scan3
set_scan_configuration -chain_count 8 -clock_mixing mix_clocks \
     -test_mode scan4

## Give a chain spec to be applied in each of the modes
set_scan_path chain1 -view spec -scan_data_in test_si[1] \
     -scan_data_out test_so[1] -test_mode scan1

set_scan_path chain2 -view spec -scan_data_in test_si[2] \
     -scan_data_out test_so[2]  -test_mode scan2

set_scan_path chain3 -view spec -scan_data_in test_si[3] \
     -scan_data_out test_so[3]  -test_mode scan2
```

```
set_scan_path chain4 -view spec -scan_data_in test_si[4] \
     -scan_data_out test_so[4]  -test_mode scan3

set_scan_path chain5 -view spec -scan_data_in test_si[5] \
     -scan_data_out test_so[5]  -test_mode scan3

set_scan_path chain6 -view spec -scan_data_in test_si[6] \
     -scan_data_out test_so[6]  -test_mode scan3

set_scan_path chain7 -view spec -scan_data_in test_si[7] \
     -scan_data_out test_so[7]  -test_mode scan3

set_scan_path chain8 -view spec -scan_data_in test_si[8] \
     -scan_data_out test_so[8]  -test_mode scan4

set_scan_path chain9 -view spec -scan_data_in test_si[9] \
     -scan_data_out test_so[9]  -test_mode scan4

set_scan_path chain10 -view spec -scan_data_in test_si[10] \
     -scan_data_out test_so[10] -test_mode scan4

set_scan_path chain11 -view spec -scan_data_in test_si[11] \
     -scan_data_out test_so[11] -test_mode scan4

set_scan_path chain12 -view spec -scan_data_in test_si[12] \
     -scan_data_out test_so[12] -test_mode scan4

set_scan_path chain13 -view spec -scan_data_in test_si[13] \
     -scan_data_out test_so[13] -test_mode scan4

set_scan_path chain14 -view spec -scan_data_in test_si[14] \
     -scan_data_out test_so[14] -test_mode scan4

set_scan_path chain15 -view spec -scan_data_in test_si[15] \
     -scan_data_out test_so[15] -test_mode scan4

set_dft_insertion_configuration -synthesis_optimization none

create_test_protocol
dft_drc
preview_dft -show all

insert_dft

current_test_mode scan1
dft_drc -verbose

current_test_mode scan2
dft_drc -verbose

current_test_mode scan3
dft_drc -verbose
```

```
current_test_mode scan4
dft_drc -verbose

list_test_modes

list_licenses
change_names -rules verilog -hierarchy
write -format verilog -hierarchical -output vg/top_scan.v
write_test_protocol -test_mode scan1 \
  -output stil/scan1.stil -names verilog
write_test_protocol -test_mode scan2 \
  -output stil/scan2.stil -names verilog
write_test_protocol -test_mode scan3 \
  -output stil/scan3.stil -names verilog
write_test_protocol -test_mode scan4 \
  -output stil/scan4.stil -names verilog

exit
```

Example 7-19 shows a DFTMAX compression script. In this script, three test modes are created. One mode is used for compression testing, a second mode is used for basic scan test and has eight chains, and a third mode uses a single basic scan chain.

Example 7-19 Basic DFTMAX Compressed Scan Multiple Test-Mode Script

```
## Define the pins for compression/base_mode using "-test_mode all"
## These modes are my_comp and my_scan1
    for {set i 1} {$i <= 13 } { incr i 1} {
     create_port -direction in test_si[$i]
     create_port -direction out test_so[$i]
     set_dft_signal -type ScanDataIn -view spec -port test_si[$i] \
          -test_mode all
     set_dft_signal -type ScanDataOut -view spec -port test_so[$i] \
          -test_mode all
    }
#Define Test Clocks
set_dft_signal -view existing_dft -type TestClock -timing {45 55} \
     -port clk_st

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port \
     [list i_trdy_de i_trdy_dd i_cs]

#Define the test modes and usage
define_test_mode my_base1 -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1}
define_test_mode my_base2 -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0}
define_test_mode burn_in  -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1}
define_test_mode scan_compression1 -usage scan_compression \
```

```
     -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0}
define_test_mode scan_compression2 -usage scan_compression \
     -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 1}

#Enable adaptive scan
set_dft_configuration -scan_compression enable

#Configure adaptive scan
set_scan_compression_configuration -base_mode my_base1 -chain_count 32 \
     -test_mode scan_compression1 -xtolerance high
set_scan_compression_configuration -base_mode my_base2 -chain_count 256 \
     -test_mode scan_compression2 -xtolerance high

#Configure the basic scan modes
set_scan_configuration -chain_count 4 -test_mode my_base1
set_scan_configuration -chain_count 8 -test_mode my_base2
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
     -test_mode burn_in

set_dft_insertion_configuration -synthesis_optimization none

## Give a chain spec to be applied in my_base1
## This will also define the scan ports for scan_compression1
set_scan_path chain1 -view spec -scan_data_in test_si[1] \
     -scan_data_out test_so[1] \
     -test_mode my_base1
set_scan_path chain2 -view spec -scan_data_in test_si[2] \
     -scan_data_out test_so[2] \
     -test_mode my_base1
set_scan_path chain3 -view spec -scan_data_in test_si[3] \
     -scan_data_out test_so[3] \
     -test_mode my_base1
set_scan_path chain4 -view spec -scan_data_in test_si[4] \
     -scan_data_out test_so[4] \
     -test_mode my_base1

## Give a chain spec to be applied in my_base2
## This will also define the scan ports for scan_compression2
set_scan_path chain5 -view spec -scan_data_in test_si[5] \
     -scan_data_out test_so[5]  -test_mode my_base2
set_scan_path chain6 -view spec -scan_data_in test_si[6] \
     -scan_data_out test_so[6]  -test_mode my_base2
set_scan_path chain7 -view spec -scan_data_in test_si[7] \
     -scan_data_out test_so[7]  -test_mode my_base2
set_scan_path chain8 -view spec -scan_data_in test_si[8] \
     -scan_data_out test_so[8]  -test_mode my_base2
set_scan_path chain9 -view spec -scan_data_in test_si[9] \
     -scan_data_out test_so[9]  -test_mode my_base2
set_scan_path chain10 -view spec -scan_data_in test_si[10] \
     -scan_data_out test_so[10] -test_mode my_base2
set_scan_path chain11 -view spec -scan_data_in test_si[11] \
     -scan_data_out test_so[11] -test_mode my_base2
set_scan_path chain12 -view spec -scan_data_in test_si[12] \
```

```
     -scan_data_out test_so[12] -test_mode my_base2

## Give a chain spec to be applied in burn_in
set_scan_path chain4 -view spec -scan_data_in test_si[13] \
     -scan_data_out test_so[13] -test_mode burn_in

create_test_protocol
dft_drc
preview_dft -show all

insert_dft

list_test_modes

current_test_mode scan_compression1
report_dft_signal
dft_drc -verbose

current_test_mode scan_compression2
report_dft_signal
dft_drc -verbose

current_test_mode my_base1
report_dft_signal
dft_drc -verbose

current_test_mode my_base2
report_dft_signal
dft_drc -verbose

current_test_mode burn_in
report_dft_signal
dft_drc -verbose

change_names -rules verilog -hierarchy
write -format verilog -hierarchical \
  -output vg/10x_xtol_moxie_top_scan_mm.v
write_test_protocol -test_mode scan_compression1 \
     -output stil/scan_compression1.stil -names verilog
write_test_protocol -test_mode scan_compression2 \
     -output stil/scan_compression2.stil -names verilog
write_test_protocol -test_mode my_base1 \
     -output stil/my_base1.stil -names verilog
write_test_protocol -test_mode my_base2 \
     -output stil/my_base2.stil -names verilog
write_test_protocol -test_mode burn_in \
     -output stil/10x_xtol_moxie.burn_in.stil -names verilog

exit
```

Example 7-20 shows a core wrapper script. This script defines all the modes that are created in a wrapper insertion process, which supports at-speed test and shared wrapper cells.

Example 7-20 Basic Core Wrapper Multiple Test-Mode Script

```
#Define Test Clocks
set_dft_signal -view existing_dft -type TestClock -timing {45 55} \
     -port clk_s

#Define TestMode signals to be used
set_dft_signal -view spec -type TestMode -port \
     [list i_trdy_de i_trdy_dd i_cs i_wr]

#Define the test modes and usage
define_test_mode burn_in -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 0 i_wr 1}
define_test_mode domain  -usage scan \
     -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1 i_wr 0}
define_test_mode my_wrp_if -usage wrp_if \
     -encoding {i_trdy_de 0 i_trdy_dd 0 i_cs 1 i_wr 1}
define_test_mode my_wrp_if_delay -usage wrp_if \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0 i_wr 0}
define_test_mode my_wrp_if_scl_delay -usage wrp_if \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 0 i_wr 1}
define_test_mode my_wrp_of -usage wrp_of \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1 i_wr 0}
define_test_mode my_wrp_of_delay -usage wrp_of \
     -encoding {i_trdy_de 0 i_trdy_dd 1 i_cs 1 i_wr 1}
define_test_mode my_wrp_of_scl_delay -usage wrp_of \
     -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0 i_wr 0}
define_test_mode my_wrp_safe -usage wrp_safe \
     -encoding {i_trdy_de 1 i_trdy_dd 0 i_cs 0 i_wr 1}

#Set scan chain count as desired
set_scan_configuration -chain_count 1 -clock_mixing mix_clocks \
     -test_mode burn_in
set_scan_configuration -chain_count 5 -test_mode domain
set_scan_configuration -chain_count 8 -test_mode wrp_if
set_scan_configuration -chain_count 8 -test_mode wrp_of

# Enable and configure wrapper client
set_dft_configuration -wrapper enable

#Configure for shared wrappers, using existing cells and \
                create glue logic around existing cells
set_wrapper_configuration -class core_wrapper \
     -style shared \
     -shared_cell_type WC_S1 \
     -use_dedicated_wrapper_clock true  \
     -safe_state 1 \
     -register_io_implementation in_place \
```

```
     -delay_test true

#Create the test protocol and run pre-drc
create_test_protocol
dft_drc -verbose

#Report the configuration of the wrapper utility, optional
report_wrapper_configuration

#Preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all

report_dft_configuration

#Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none
insert_dft

list_test_modes

current_test_mode burn_in
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_burn_in.rpt

current_test_mode domain
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_domain.rpt

current_test_mode my_wrp_of
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_my_wrp_of.rpt

current_test_mode my_wrp_of_delay
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_my_wrp_of_delay.rpt

current_test_mode my_wrp_of_scl_delay
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_my_wrp_of_scl_delay.rpt

current_test_mode my_wrp_if
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_my_wrp_if.rpt

current_test_mode my_wrp_if_delay
report_scan_path -view existing_dft -cell all > \
     reports/xg_wrap_dedicated_delay_path_my_wrp_if_delay.rpt

current_test_mode my_wrp_if_scl_delay
report_scan_path -view existing_dft -cell all > \
     reports xg_wrap_dedicated_delay_my_wrp_if_scl_delay.rpt
```

```
report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchical -output db/scan.ddc
write -format verilog -hierarchical -output vg/scan_wrap.vg
write_test_model -output db/des_unit.scan.ctldb
write_test_protocol -test_mode burn_in -output stil/burn_in.spf
write_test_protocol -test_mode domain -output stil/domain.spf
write_test_protocol -test_mode my_wrp_if_delay \
     -output stil/my_wrp_if_delay.spf
write_test_protocol -test_mode my_wrp_if_scl_delay \
     -output stil/my_wrp_if_scl_delay.spf
write_test_protocol -test_mode my_wrp_if -output stil/my_wrp_if.spf
write_test_protocol -test_mode my_wrp_of -output stil/my_wrp_of.spf
write_test_protocol -test_mode my_wrp_of_delay \
     -output stil/my_wrp_of_delay.spf
write_test_protocol -test_mode my_wrp_of_scl_delay \
     -output stil/my_wrp_of_scl_delay.spf
write_test_protocol -test_mode wrp_if -output stil/extra_wrp_if.spf

exit
```

Note that you can also run the `report_scan_path -test_mode tms` command, which displays a report containing test-related information about the current design, as shown in Example 7-21.

Example 7-21 report_scan_path Output Example

```
dc_shell> report_scan_path -test_mode tms
Number of chains: 3
Scan methodology: full_scan
Scan style: multiplexed_flip_flop

Clock domain: mix_clocks
Chn Scn Prts (si --> so) #Cell Inst/Chain Clck (prt, tm,edge)
--- ------------------- ----- ---------- -------------------
S 1 test_si1 --> test_so1 2 U2/1 (s) CK3, 45.0,rising)
S 2 test_si2 --> test_so2 2 U2/2 (s) (CK2, 45.0,rising)
W 3 test_si3 --> test_so3 8 U2/WrapperChain_0 (s) (WCLK,45.0, rising)
```

Multivoltage Support

The increasing presence of multiple voltages in designs has resulted in the need for DFT insertion to build working scan chains with minimal voltage crossings and minimal level shifters. This section describes the methodology for running DFT insertion in designs containing multiple voltages.

The following subsections describe multivoltage support:

Configuring Scan Insertion for Multivoltage Designs

The following command instructs the DFT insertion process to build scan chains that can cross different voltage regions:

```
dc_shell> set_scan_configuration -voltage_mixing true
```

When set to `true`, DFT insertion attempts to minimize voltage crossings to reduce the number of level shifters added. The default of this option is `false`.

If you use any commands, such as the `set_scan_path` command, that violate the voltage mixing specification, the `preview_dft` and `insert_dft` commands issue the following warning message and continue with scan insertion:

```
Warning: elements are supplied by different voltages. (TEST-1026)
```

Configuring Scan Insertion for Multiple Power Domains

The following command instructs the DFT insertion process to build scan chains that can cross different power domains:

```
dc_shell> set_scan_configuration \
              -power_domain_mixing true
```

When set to `true`, DFT insertion attempts to minimize power domain crossings to reduce the number of isolation cells added. DFT insertion does not check or remove existing isolation cells. The default of this option is `false`.

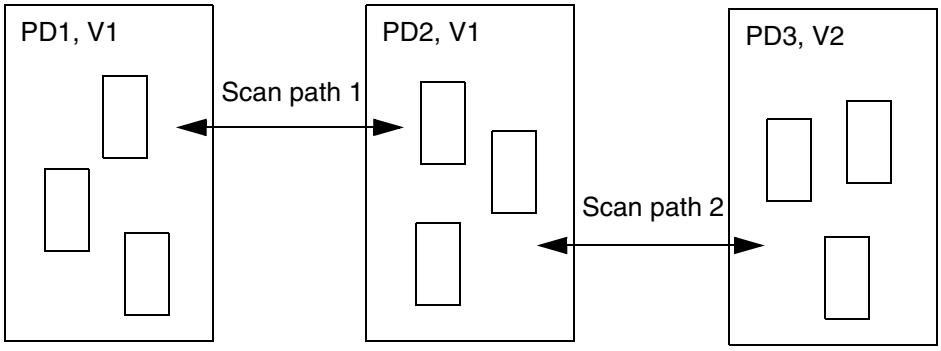
If you use any commands, such as the `set_scan_path` command, that violate the power domain mixing specification, the `preview_dft` and `insert_dft` commands issue the following warning message and continue with scan insertion:

```
Warning: elements are supplied by different power domains. (TEST-1029)
```

Mixture of Multivoltage and Multiple Power Domain Specifications

The interaction between the `-voltage_mixing` and `-power_domain_mixing` options is as shown in Figure 7-22.

Figure 7-22 Interaction Between Voltage Mixing and Power Domain Mixing



Here the scan cells are contained within three blocks as follows:

- Power domain PD1, Voltage V1
- Power domain PD2, Voltage V1
- Power domain PD3, Voltage V2

By default, DFT Compiler does not allow voltage mixing or power domain mixing within the same scan path. Table 7-3 shows the allowable scan paths with the various combinations of `-voltage_mixing` and `-power_domain_mixing`.

Table 7-3 Interactions Between the -voltage_mixing and -power_domain_mixing Options

If -voltage_mixing is:	And -power_domain_mixing is:	Allowable scan paths
False	False	None
False	True	Scan path 1 only
True	False	None
True	True	Scan paths 1 and 2

Note:

The behavior of DFT insertion in an always-on synthesis environment is such that you must run an incremental compile after running the `insert_dft` command to insert any missing multivoltage cells that might be needed.

Reusing Multivoltage Cells

By default, the `insert_dft` command reuses existing level shifters and isolation cells if they are on the scan path. It reuses only combinational multivoltage cells and does not reuse sequential multivoltage cells, such as latch-based isolation cells. If you do not want the `insert_dft` command to reuse existing multivoltage cells (level shifters and isolation cells), use the following command:

```
dc_shell> set_scan_configuration \
                -reuse_mv_cells false
```

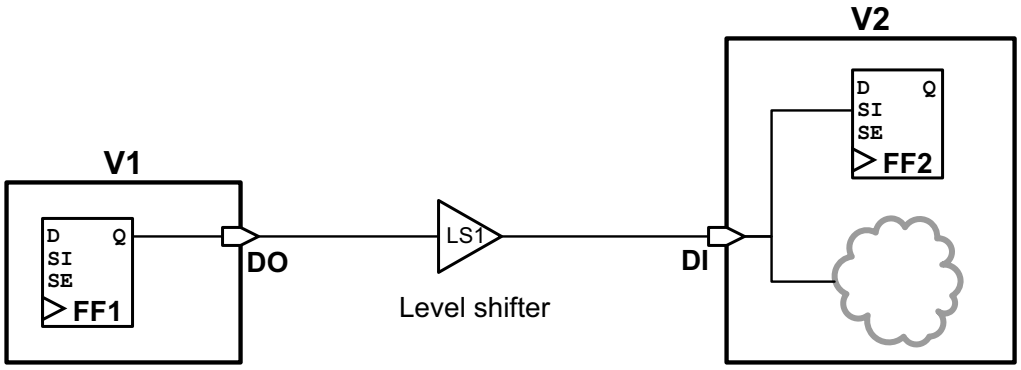
This section includes the following subsections:

- Reusing Level Shifters in Scan Paths
- Reusing Isolation Cells in Scan Paths

Reusing Level Shifters in Scan Paths

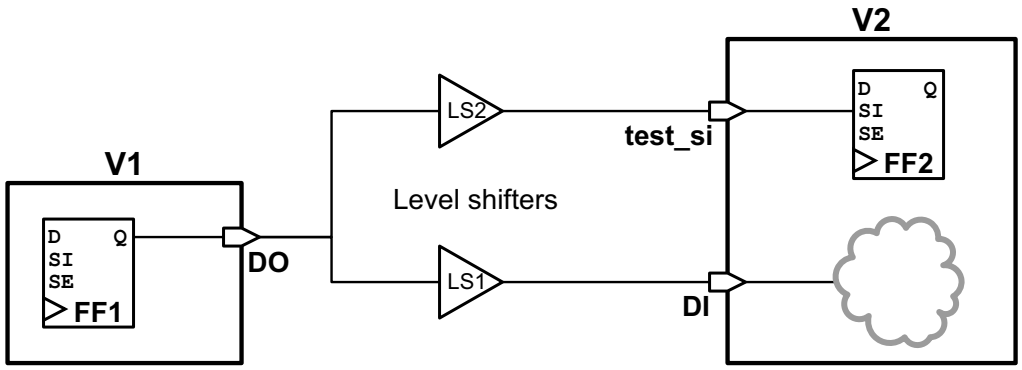
If a scan path goes through a level shifter and you enable multivoltage cell reuse, scan insertion connects the scan chain at the output side of the level shifter, as shown in Figure 7-23.

Figure 7-23 Shared Level Shifter Along Scan Path



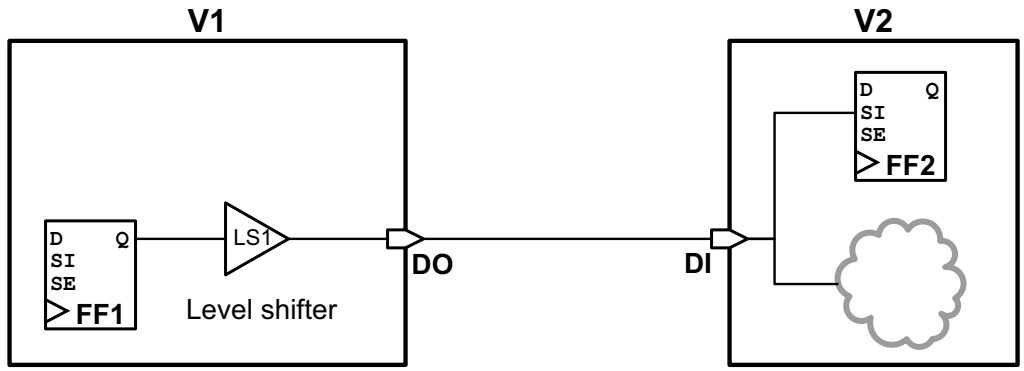
If the scan path goes through a level shifter and you disable multivoltage cell reuse, a new level shifter is added to connect to the scan path. See Figure 7-24.

Figure 7-24 Separate Level Shifters Along Scan Path



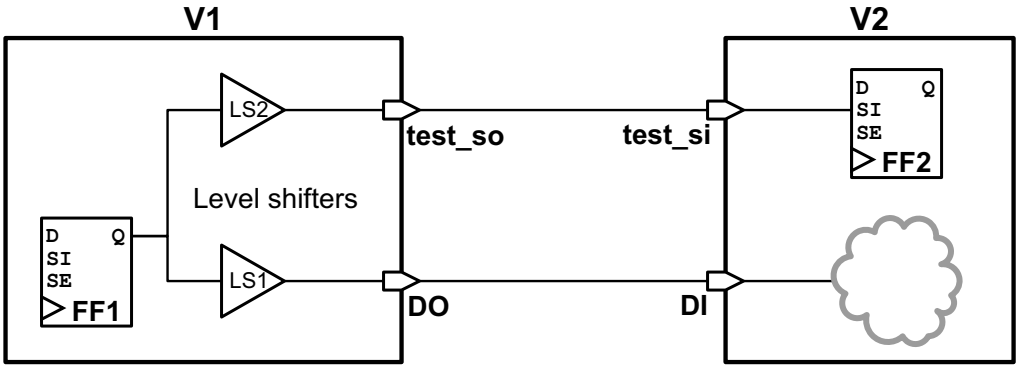
If the level shifter is within a block and you enable multivoltage cell reuse, then scan insertion reuses the existing level shifter and hierarchical port, as shown in Figure 7-25.

Figure 7-25 Shared Level Shifter in a Block Along Scan Path



If you disable multivoltage cell reuse and the existing level shifter is within a block, then the new level shifter is added within the block and a new hierarchical port is added. See Figure 7-26.

Figure 7-26 Separate Level Shifters in a Block Along Scan Path

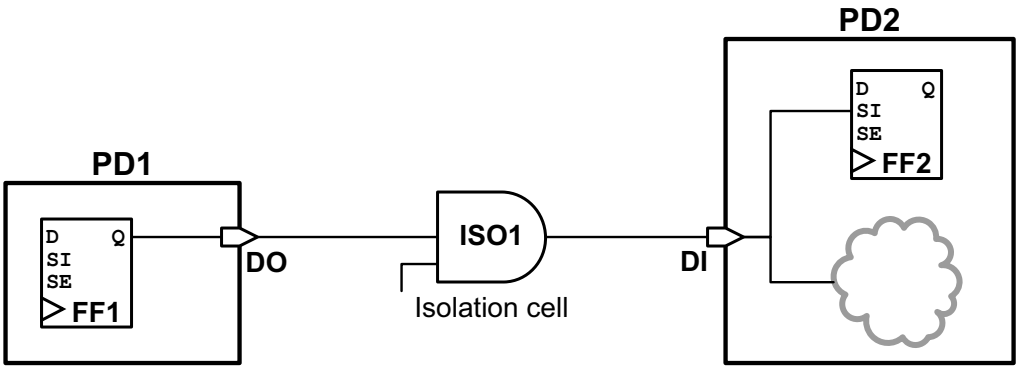


Reusing Isolation Cells in Scan Paths

In the following examples, the scan path and functional path for a given signal route to the same downstream power domain, which is different from the source power domain.

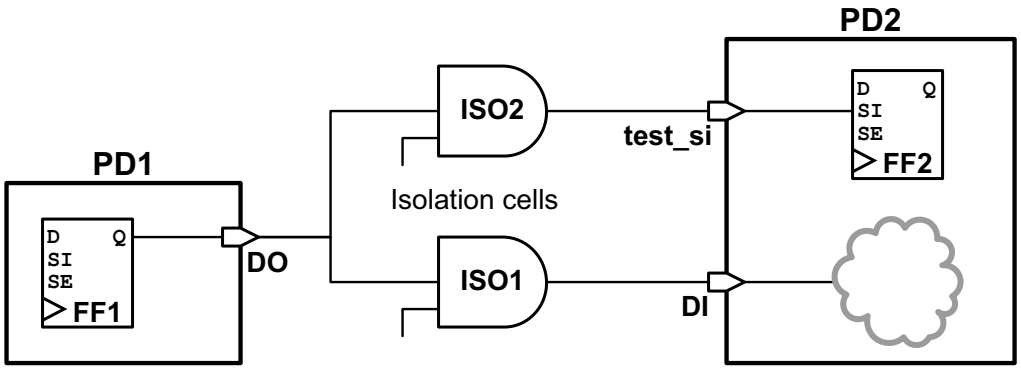
If a scan path goes through an isolation cell in a parent power domain, and you enable multivoltage cell reuse, then scan insertion connects the scan chain at the output side of the isolation cell, as shown in Figure 7-27.

Figure 7-27 Shared Isolation Cell Along Scan Path in Parent Domain



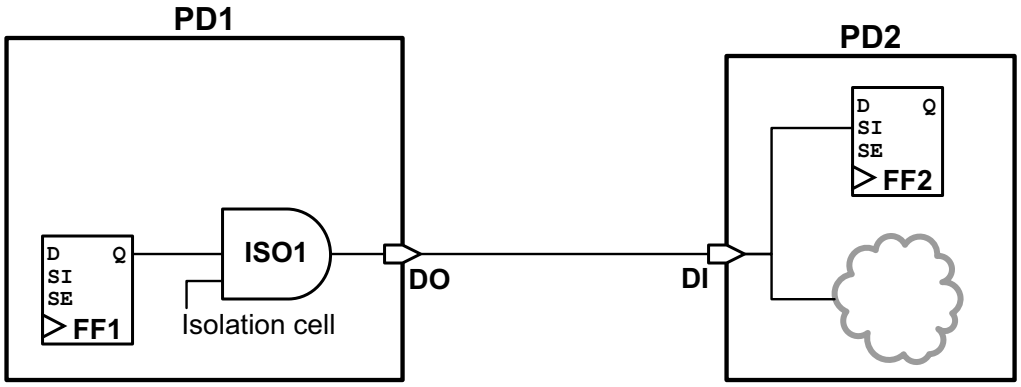
If you disable multivoltage cell reuse, then the scan path is connected to the net before the isolation cell and a new hierarchical port and isolation cell are added, as shown in Figure 7-28.

Figure 7-28 Separate Isolation Cells Along Scan Path in Parent Domain



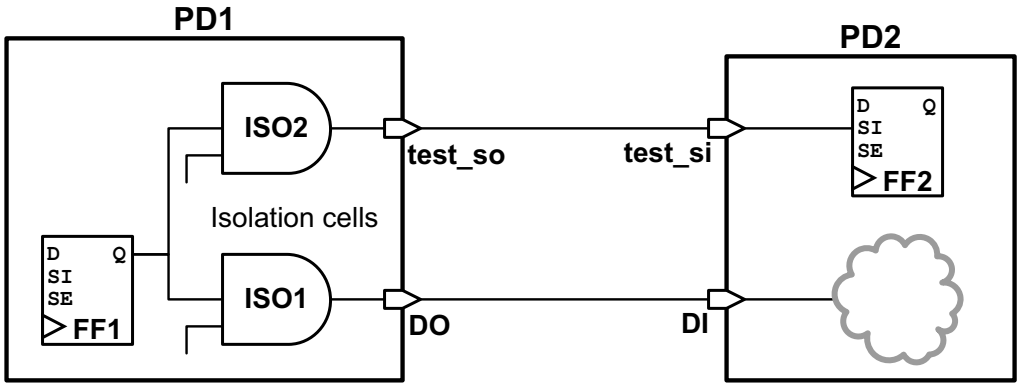
If the isolation cell is within the source power domain, and you enable multivoltage cell reuse, then scan insertion reuses the existing hierarchical port and isolation cell, as shown in Figure 7-29.

Figure 7-29 Shared Isolation Cell Along Scan Path in Source Domain



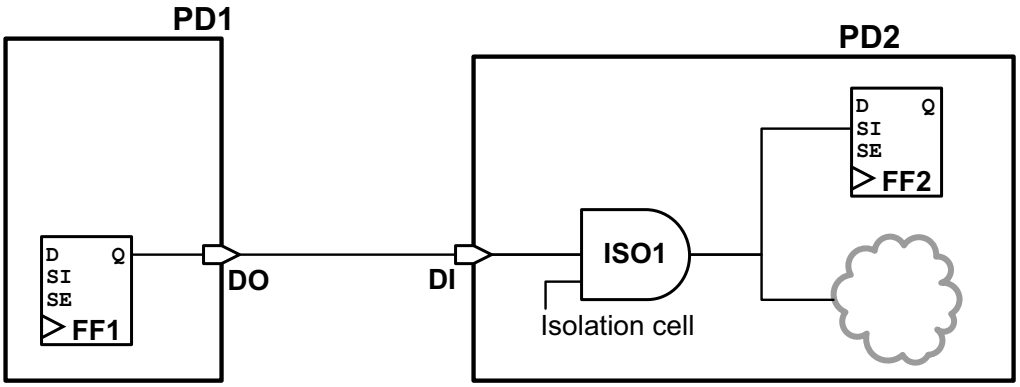
If you disable multivoltage cell reuse, and the existing isolation cell is within the source power domain, then a new hierarchical port and isolation cell are added, as shown in Figure 7-30.

Figure 7-30 Separate Isolation Cells Along Scan Chain in Source Domain



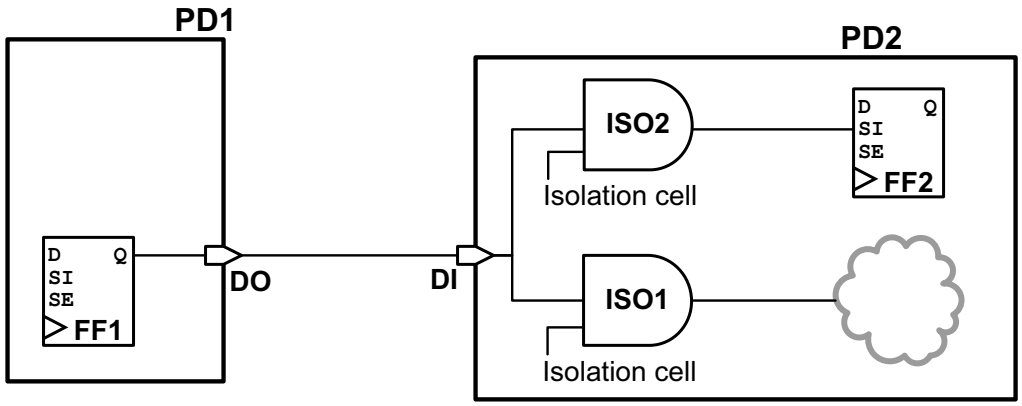
If the isolation cell is within the fanout power domain, and you enable multivoltage cell reuse, then scan insertion reuses the existing hierarchical port and isolation cell, as shown in Figure 7-31.

Figure 7-31 Shared Isolation Cell Along Scan Path in Fanout Domain



If you disable multivoltage cell reuse, and the existing isolation cell is within the fanout power domain, then a new hierarchical port and isolation cell are added, as shown in Figure 7-32.

Figure 7-32 Separate Isolation Cells Along Scan Path in Fanout Domain



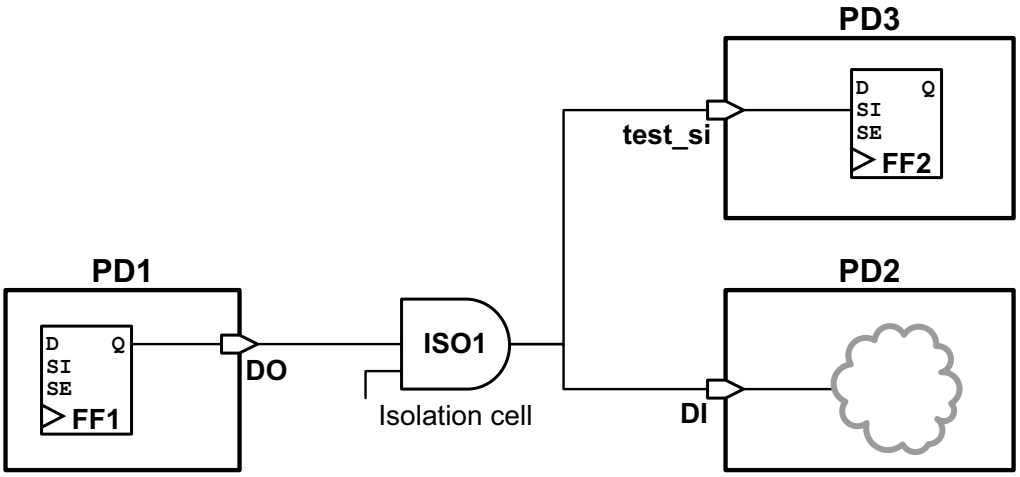
If the scan path routes to a different downstream power domain than the functional path, an existing isolation cell can be reused if the isolation strategy allows both the original power domain connection and the new scan path power domain connection to be driven by the isolation cell.

Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added to the parent power domain of PD1 with the `-location parent` option:

```
set_isolation iso_PD1 \
  -domain PD1 \
  -diff_supply_only true \
  -applies_to outputs
set_isolation_control iso_PD1 \
  -domain PD1 \
  -location parent
```

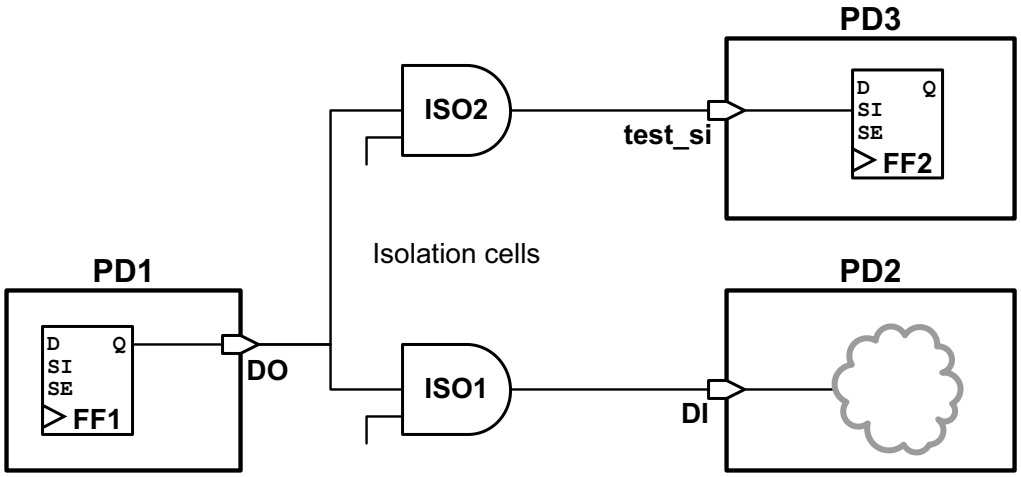
The `-diff_supply_only` strategy allows an isolation cell in the source domain to isolate multiple fanout power domains, if they differ from the source domain. If you enable multivoltage cell reuse, then scan insertion reuses the existing isolation cell and hierarchical port, as shown in Figure 7-33.

Figure 7-33 Shared Isolation Cell in Parent Domain with Multiple Sink Domains



If you disable multivoltage cell reuse, then a new isolation cell is added, as shown in Figure 7-34:

Figure 7-34 Separate Isolation Cells in Parent Domain with Multiple Sink Domains



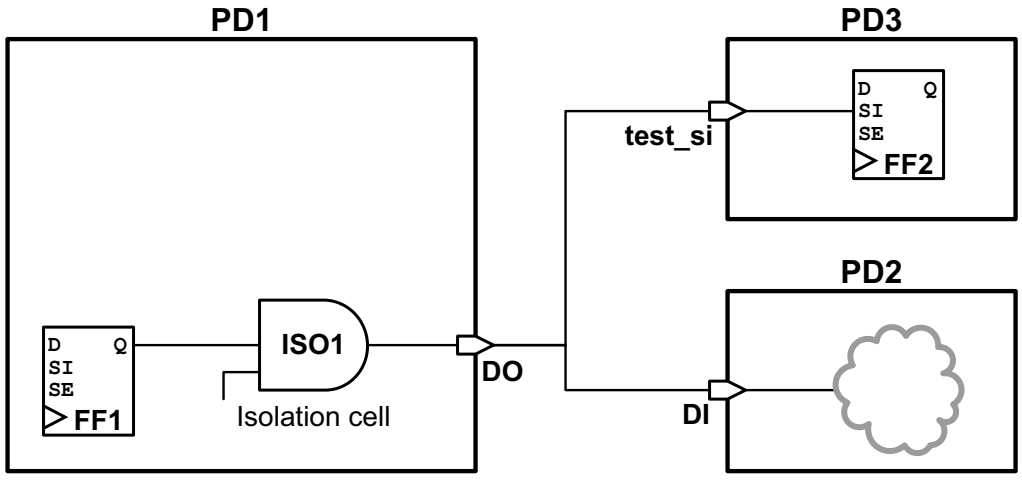
Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added within the power domain PD1 with the `-location self` option:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
```

```
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location self
```

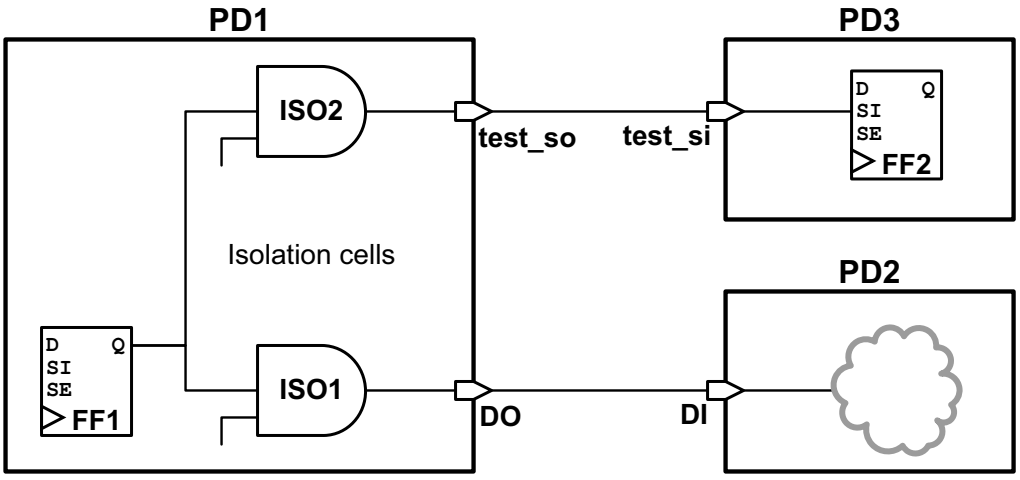
If you enable multivoltage cell reuse, then scan insertion reuses the existing isolation cell and hierarchical port, as shown in Figure 7-35:

Figure 7-35 Shared Block Isolation Cell with Multiple Sink Domains



If you disable multivoltage cell reuse, and the existing isolation cell is within a block, then a new isolation cell and hierarchical port are added, as shown in Figure 7-36:

Figure 7-36 Separate Parent Isolation Cells with Multiple Sink Domains



Scan Path Routing and Isolation Strategy Requirements

The isolation strategy applied to a cross-domain net might restrict the power domain connections allowed for that net:

- A `set_isolation -diff_supply_only` isolation strategy allows multiple fanout power domains to be driven by the same isolated net, if they differ from the source domain.
- A `set_isolation -source -sink` isolation strategy requires that any specified source and sink power domains connected to an isolated net match the isolation strategy.

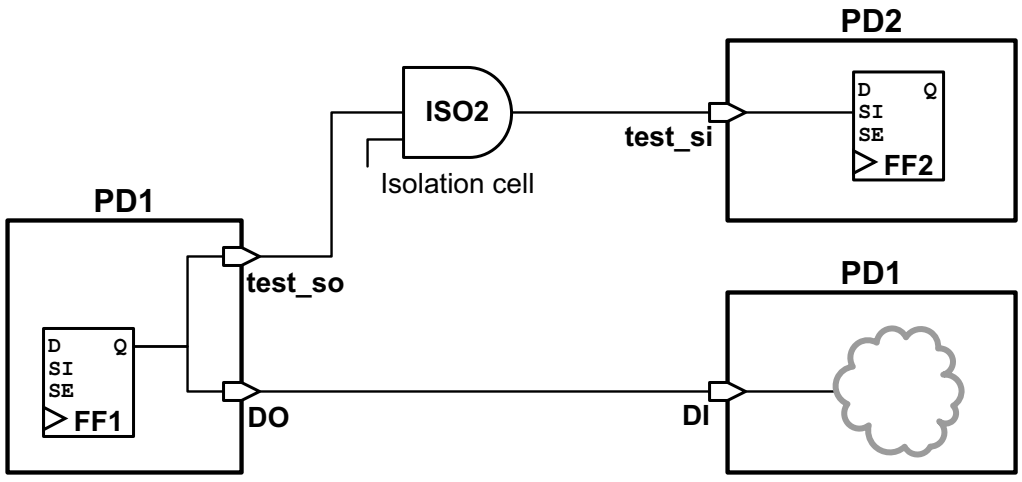
When the `insert_dft` command routes a scan chain along an existing hierarchical output port to a different downstream power domain, the isolation strategy requirements of the existing net might require new isolation cells and hierarchical ports to be added along the scan path.

Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added to the parent power domain of PD1:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location parent
```

If the existing functional path is routed through the same power domain, but the scan path is routed to a different power domain, an isolation cell is added within the parent power domain as shown in Figure 7-37.

Figure 7-37 Isolation Cell in Parent Domain with Differing Isolation Requirements

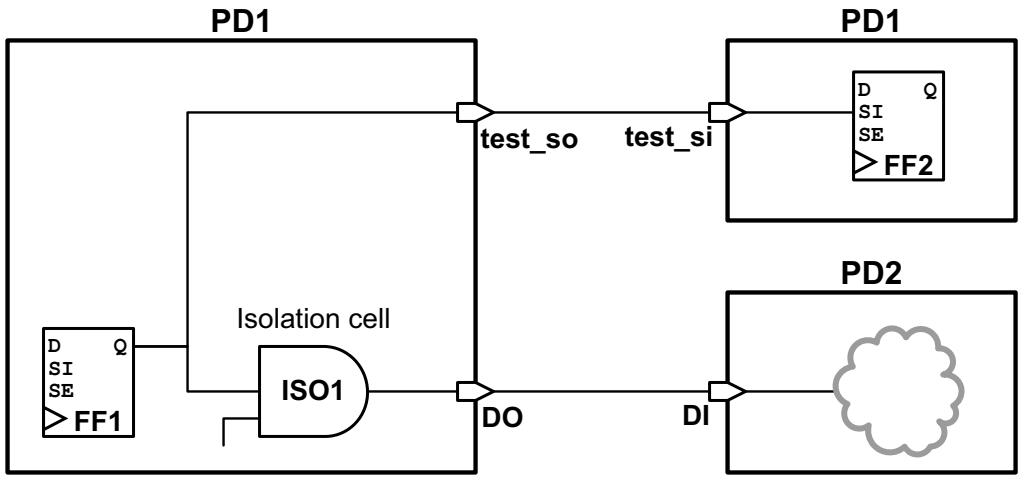


Consider the `-diff_supply_only` isolation strategy defined in the following example, which specifies that isolation cells should be added within the source power domain PD1:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location self
```

If the existing functional path is routed to a different power domain but the scan path is routed through the same power domain, a hierarchical port is added to bypass the isolation cell as shown in Figure 7-38.

Figure 7-38 Isolation Cell in Source Domain with Differing Isolation Requirements

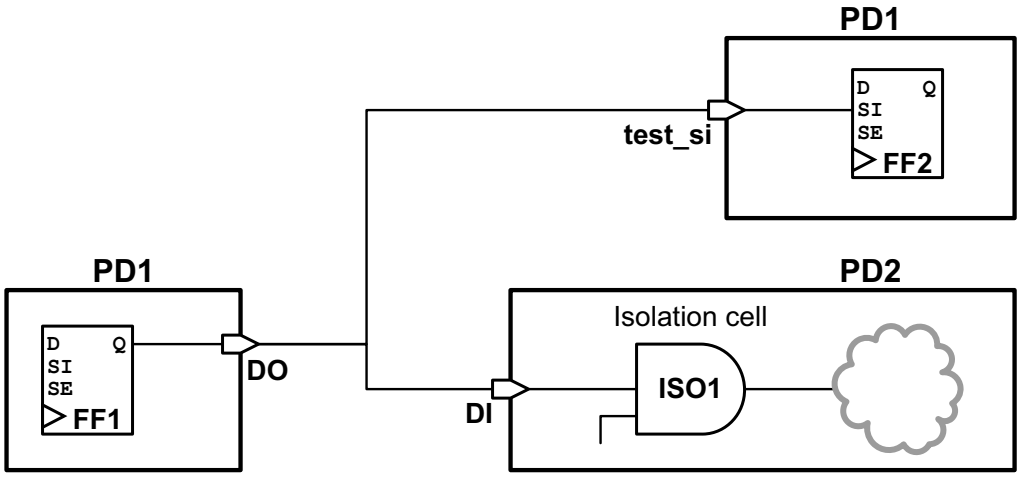


You can configure the isolation cell strategy to place the isolation cells in the fanout domains to avoid the creation of additional hierarchical scan pins. Consider the following example, modified to use the `-location fanout` option:

```
set_isolation iso_PD1 \
    -domain PD1 \
    -diff_supply_only true \
    -applies_to outputs
set_isolation_control iso_PD1 \
    -domain PD1 \
    -location fanout
```

In this modified example, the isolation cells and resulting isolated nets are now located in the fanout power domain, as shown in Figure 7-39. No additional hierarchical scan pins are needed to meet the isolation strategy requirements.

Figure 7-39 Isolation Cell in Fanout Domain with Differing Isolation Requirements

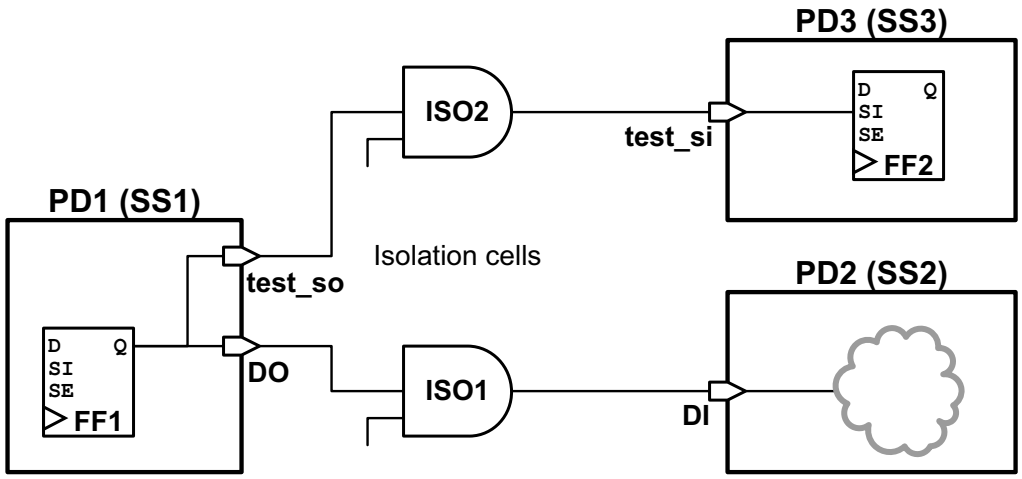


Consider the `-source -sink` isolation strategy defined in the following example, which specifies two isolation strategies: one that requires isolation cells for nets that fan out to sink power domain SS2, and one that requires isolation cells for nets that fan out to sink power domain SS3.

```
set_isolation iso1_PD1 \
    -domain PD1 \
    -source SS1 -sink SS2
set_isolation_control iso1_PD1 \
    -domain PD1 \
    -location parent

set_isolation iso2_PD1 \
    -domain PD1 \
    -source SS1 -sink SS3
set_isolation_control iso2_PD1 \
    -domain PD1 \
    -location parent
```

If you enable multivoltage cell reuse, the `insert_dft` command is unable to reuse the existing isolation cell ISO1 for the scan path connection, because the existing cross-domain isolated net cannot drive two different power domains. Instead, a new isolation cell and hierarchical port are added, as shown in Figure 7-40.

Figure 7-40 Separate Isolation Cells in Parent Domain with Differing Isolation Requirements



Using Domain-Based Strategies for DFT Insertion

For the `insert_dft` command to properly insert level shifters and isolation cells, you must specify the level shifter and isolation cell strategies on a power-domain basis, even if you have specified similar strategies on the blocks and individual ports. If you only specify the strategies on the blocks and ports, the `insert_dft` command might not be able to automatically insert level shifters and isolation cells on any new ports that it creates.

For example, if your power intent specification is applied to all outputs with the `-applies_to outputs` option:

```
create_power_domain PD1 -elements U_block

set_isolation iso_PD1 \
     -domain PD1 \
     -isolation_power_net VDD -isolation_ground_net VSS \
     -clamp_value 1 \
     -applies_to outputs
```

and if the `insert_dft` command creates new output pins on the blk_a block that requires isolation, then isolation cells are automatically added where they are needed.

However, if your power intent specification is applied to selected existing design elements with the `-elements` option:

```
create_power_domain PD1 -elements U_block
```

```
set_isolation PD1 \
     -domain BLOCK \
     -isolation_power_net VDD -isolation_ground_net VSS \
     -clamp_value 1 \
     -elements {Z}
```

The isolation strategy applies only to existing output pin Z. Any new output pins that are created by the `insert_dft` command are not isolated.

The same behavior applies to level shifter insertion strategies specified by the `set_level_shifter` command.

DFT Considerations for Low-Power Design Flows

Low-power designs often use the following special cells:

- Isolation cells
- Retention registers
- Power switches

You must configure these special cells such that the data shifts through the scan chains during test operations. After DFT insertion using the `insert_dft` command, the `dft_drc` command can identify design rule violations on isolation cells and retention registers, if any, that would prevent scan shifting through such cells. However, the command cannot identify design rule violations on isolation cells and retention registers before the `insert_dft` command is run.

The control signals for these special cells are typically driven by a power controller. If the power controller is located off-chip, you can constrain the control signals at the ports for correct shift operation. If the power controller is located on-chip and does not include testability logic, the tool can insert power controller override logic for you. For more information, see “Inserting Power Controller Override Logic” on page 7-74.

You must keep any DFT constraints on special cells in place up to the `write_scan_def` command when you generate a SCANDEF file for scan chain reordering.

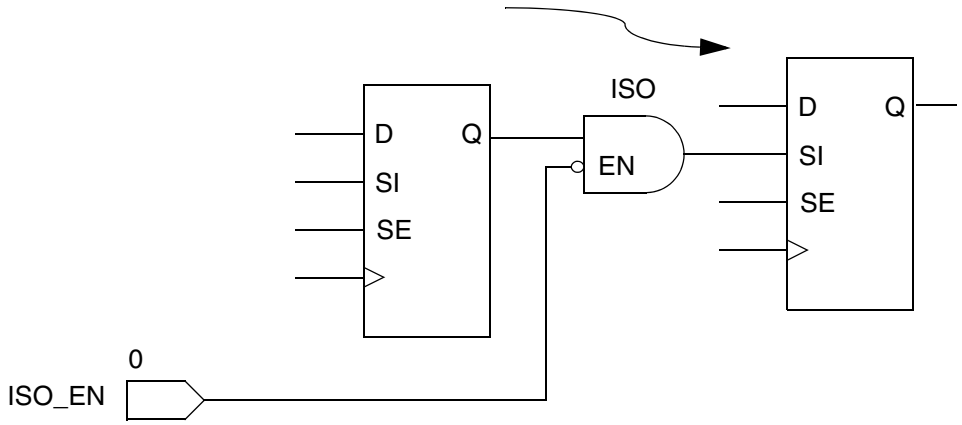
Also note that the `dft_drc` command cannot detect design rule violations on power switches in pre- or post-DFT insertion.

A multivoltage-aware verification tool such as MVSIM or MVRC can detect such violations if there are any. For further information, see the MVSIM and MVRC documentation.

Isolation Cells

For example, if a scan chain traverses an isolation cell, you must ensure that the isolation cell passes the scan data to the flip-flop driven by the isolation cell during the test operation, as shown in Figure 7-41.

Figure 7-41 Proper Configuration of a Scan Chain That Includes an Isolation Cell



Retention Registers

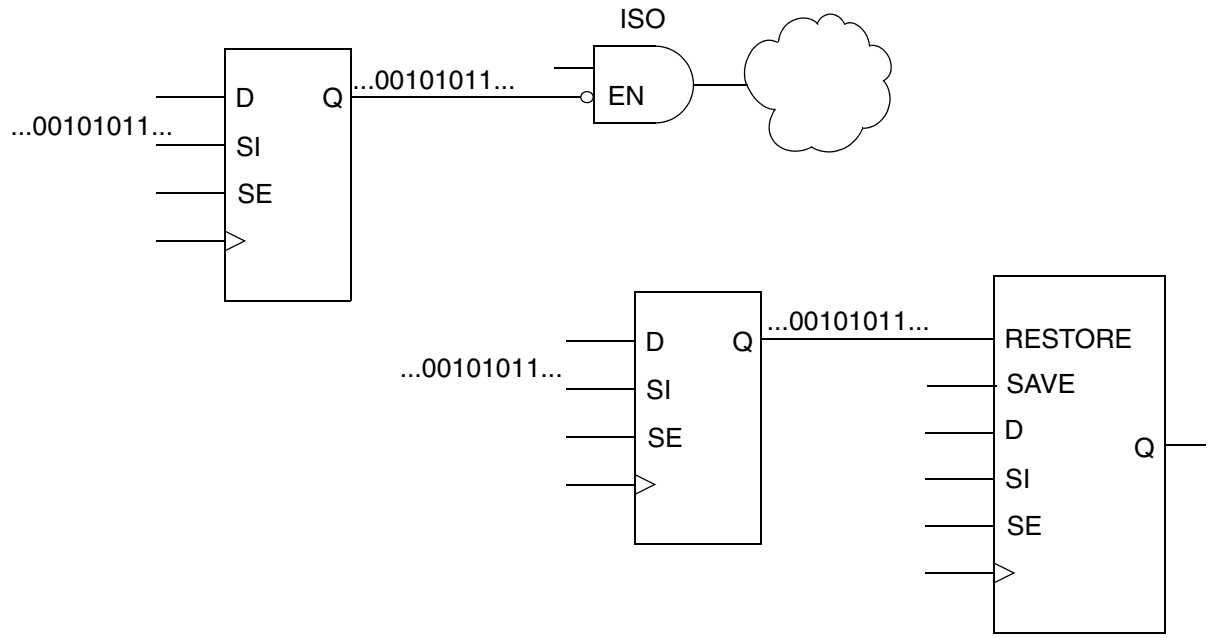
The design's retention registers must be in normal or power-up mode during the test operation. You should check that any save or restore signals are at their correct states.

Registers That Drive Low-Power Control Signals

If the design contains registers that drive low-power control signals, such as the enable signal of the isolation cells or the save/restore signals of the retention registers, you must not put these registers onto the scan chains. Otherwise, this could cause these control signals to switch during test operations. Figure 7-42 shows the consequences of putting such registers on the scan chains.

These registers must also drive the low power control signals to a constant state so that the controls cannot be toggled during test operations. You can check for this during the `dft_drc` command by ensuring that these registers are included in the TEST-504 and TEST-505 violations.

Figure 7-42 Consequences of Putting Registers That Drive Low-Power Control Signals on the Scan Chains



Previewing a Multivoltage Scan Chain

The `preview_dft -show all` command reports the operating condition and the power domain of a scan cell whenever a scan path crosses a voltage or power domain. It also indicates whether a scan cell is driving a level shifter or an isolation cell.

In Example 7-22, (v) indicates that the scan cell drives a level shifter and (i) indicates that the scan cell drives an isolation cell.

Example 7-22 Preview Report With Voltage and Power Domains

```
****************************************
Preview_dft report
For    : 'Insert_dft' command
Design : seqmap_test
Version: Z-2007.03
Date   : Tue Jan 30 17:02:47 2007
****************************************
Number of chains: 1
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: mix_clocks
Voltage Mixing: True
Power Domain Mixing: True
```

```
   (i) shows cell scan-out drives an isolation cell
   (l) shows cell scan-out drives a lockup latch
   (s) shows cell is a scan segment
   (t) shows cell has a true scan attribute
   (v) shows cell scan-out drives a level shifter cell
   (w) shows cell scan-out drives a wire

Scan chain '1' (test_si --> test_so) contains 56 cells:


  u2/q_reg[3]   (voltage 1.08) (pwr domain 'pd_2') (clk, 55.0, falling)
  u2/q_reg[4]
  ...
  u2/q_reg[26]
  u2/q_reg[27]
  u1/q_reg[3] (v)(i)  (voltage 0.80) (pwr domain 'pd_1')
  u1/q_reg[4]
  ...
  u1/q_reg[26]
  u1/q_reg[27]
  u2/q_reg[0] (v)(i)  (voltage 1.08) (pwr domain 'pd_2') (clk, 45.0,
rising)
  u2/q_reg[1]
  ...
  u2/q_reg[22]
  u2/q_reg[23]
  u1/q_reg[0] (v)(i)  (voltage 0.80) (pwr domain 'pd_1')
  u1/q_reg[1]
  ...
```

Scan Extraction Flows in the Presence of Isolation Cells

If you need to run the scan extraction flow on a netlist for a design that is already scan-inserted and contains isolation cells, you must specify any constraints that are needed at the enable pin of isolation cells in addition to the constraints that might be required for DFT signals. If this is not done, you might fail to extract scan chains. DFT Compiler does not check the validity of isolation logic.

Limitations

The following limitations apply to multivoltage and multipower domains:

- Multivoltage and multipower domains are supported only in multiplexed flip-flop scan style.
- Multivoltage and multipower domains are supported only in the following flows:
  - Basic scan, including AutoFix, observe point insertion, user-defined test point insertion, and HSS
  - DFTMAX compressed scan
- Multivoltage and multipower domains are not supported in the following flows:
  - BSD insertion
  - DFTMAX DBIST
  - Core integration

Controlling Power Modes During Test

Power-sensitive designs often contain multiple power domains. This allows power supplies for inactive logic to be switched off, reducing power consumption during operation. These designs typically have a *power controller* block, which supplies the necessary power supply control signals to power switches, isolation cells, and retention registers.

When the device is being tested, the power control signals must be controlled to ensure that the design is testable. This section describes the features provided by DFT Compiler to automate the control of power modes during test.

Inserting Power Controller Override Logic

The power controller block generates control signals for the power supply control logic that exists throughout the design. The power control signals at the block outputs are defined using commands that are part of the IEEE 1801 specification, also known as the Unified Power Format (UPF) specification.

DFT Compiler does not create the power controller block, but it can insert testability logic at the block outputs to override the power control signals during test mode. This logic is known as the *power controller override* logic. This logic allows the signals to be controlled during test mode without manually-created power controller initialization vectors. It also provides observability of the power controller outputs for improved test coverage.

To use this feature, enable the power controller override feature, and specify the power controller hierarchical cell instance with the following commands:

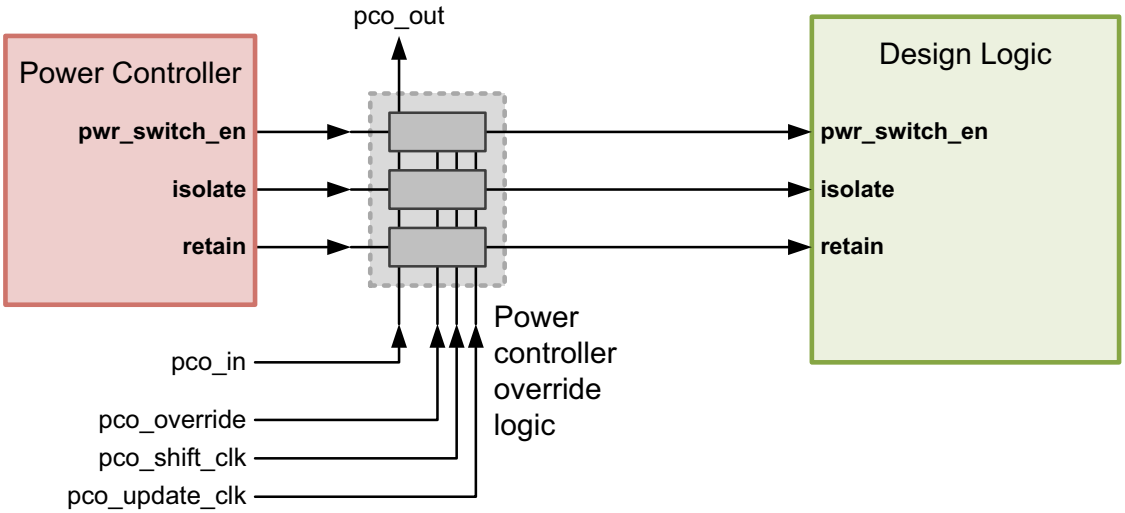
```
set_dft_configuration -power_control enable
set_dft_power_control power_controller_hier_cell
```

The control signal outputs of the power controller block must be defined using the signal specification options of the applicable UPF commands:

- `create_power_switch -control_port`
- `set_isolation_control -isolation_signal`
- `set_retention_control -save_signal`

When a power controller block is configured, the `insert_dft` command inserts wrapper chain override logic along the control signal outputs of the specified power controller block, inside the level of hierarchy that contains the specified power controller block. Figure 7-43 shows the structure of the power controller override logic.

Figure 7-43 Power Controller Override Logic



The power controller override wrapper chain is composed of control-observe cells along the power controller outputs. The wrapper chain is a separate chain that cannot be combined with other scan chains. For designs with scan compression, the power controller override wrapper chain exists outside the compressor-decompressor logic.

The following top-level ports are added during power controller override logic insertion:

- `pco_in` and `pco_out`

  These ports are the scan-in and scan-out ports for the wrapper chain cells.

- `pco_override`

  This port is asserted in test mode to override the power controller's control signals with the wrapper chain override values.

- `pco_shift_clk`

  When clocked with ScanEnable de-asserted, this port captures the current output values from the power controller to improve observability. When clocked with ScanEnable asserted, this port shifts data through the shadow registers of the wrapper chain. When the override is asserted, the current override state is not affected when data is shifted through the wrapper chain shadow registers.

- `pco_update_clk`

  This port is clocked to transfer the control signal values from the wrapper cell shadow registers to the wrapper cell output registers.

During power controller override logic insertion, the design's UPF specification is updated to reflect the new power switch control, isolation, and retention control points. The test_setup procedure is also updated with test vectors that place the power control signals into a stable state. If multiple test modes have been defined, they are all updated.

This stable state has the following characteristics:

- All controllable power supply switches are enabled.
- All controllable isolation cells are set to a pass-through state.
- All controllable retention cells are placed into save mode.

If you have defined all power controller signals in the UPF specification, the resulting test protocol can be used directly in the TetraMAX tool with no editing. If any power controller signals have not been captured in the UPF specification, you must add the required signal values to the test_setup procedure.

Limitations

Note the following limitations of the power controller override feature:

- Only one power controller instance can be specified.
- If the power controller logic is distributed across several blocks, you must first group it into a single block.
- It is not possible to specify that any particular ports be used as scan-in and scan-out for the power controller override wrapper chain.
- It is not possible to use an existing scan segment as the power controller override wrapper chain.

Power-Aware Functional Output Gating

During scan testing, while scan data shifts through scan chains, the functional logic driven by the scan flip-flops also toggles. This can cause increased power dissipation during testing, which could damage the design under test.

You can use the `set_scan_suppress_toggling` command to enable functional output gating. When this command is used, the tool inserts gating logic to suppress toggling on the functional output of scan flip-flops that either you specify or the tool automatically selects. The tool uses AND-gating or OR-gating logic, depending on which constant value most reduces toggling from other ungated signals entering the fanout logic cone.

Note:

This feature only works with designs that use the multiplexed flip-flop scan style.

Figure 7-44 shows the design after DFT insertion but without any gating logic inserted on the functional output.

Figure 7-44 Design After DFT Insertion Without Gating Logic on the Functional Output

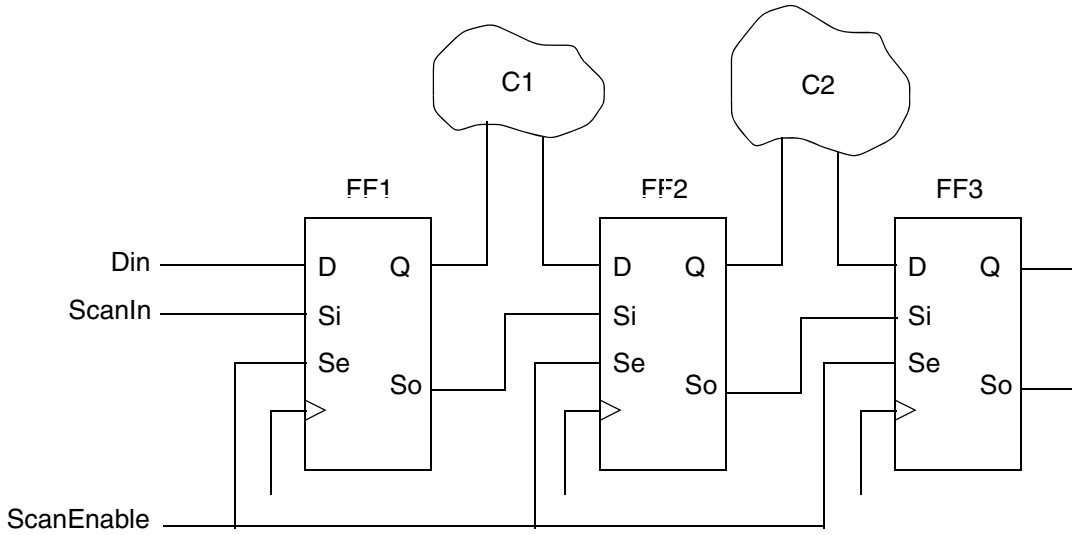
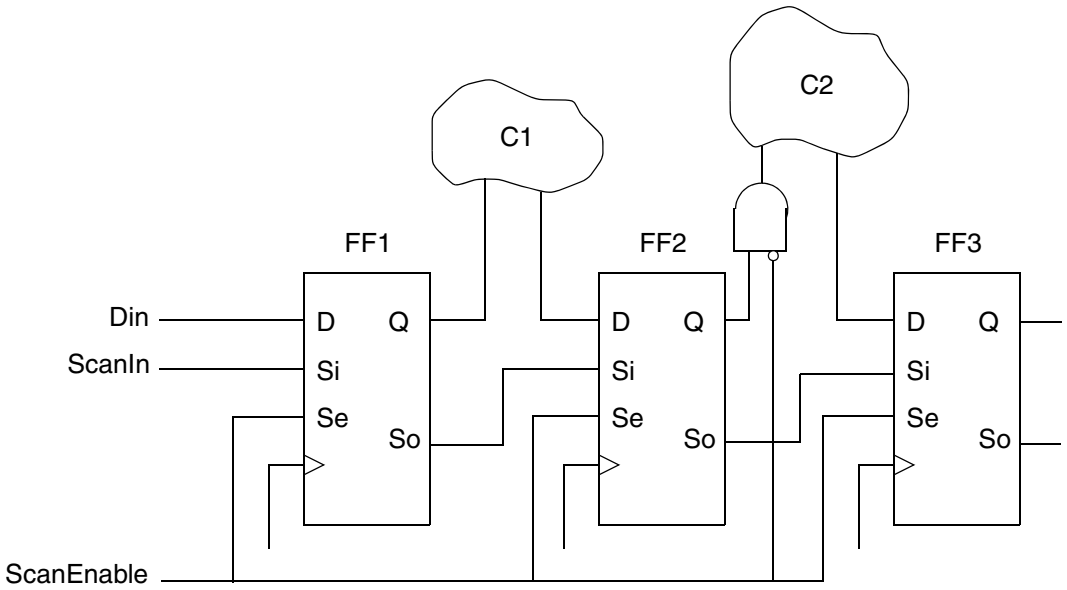


Figure 7-45 shows the design after the tool has inserted AND-gating logic on the output pin of the FF2 flip-flop.

Figure 7-45 Design After DFT Insertion With AND Clock Gating Inserted on the FF2 Functional Output



In the Figure 7-45 example, the logic cone C2 does not toggle during scan shifting because the Q output of FF2 is gated by the extra AND gate, which is disabled by the scan-enable signal.

If your scan flip-flops have only a single output pin that is shared between functional and scan output, the logic path going from the flip-flop output pin to the functional logic is gated. Figure 7-46 and Figure 7-47 show this case for both AND-gating logic and OR-gating logic, respectively.

Figure 7-46 Design With AND-Gating Logic Inserted on the Functional Output of FF2, Where FF2 Has a Single Output Pin Shared Between Functional and Scan Output

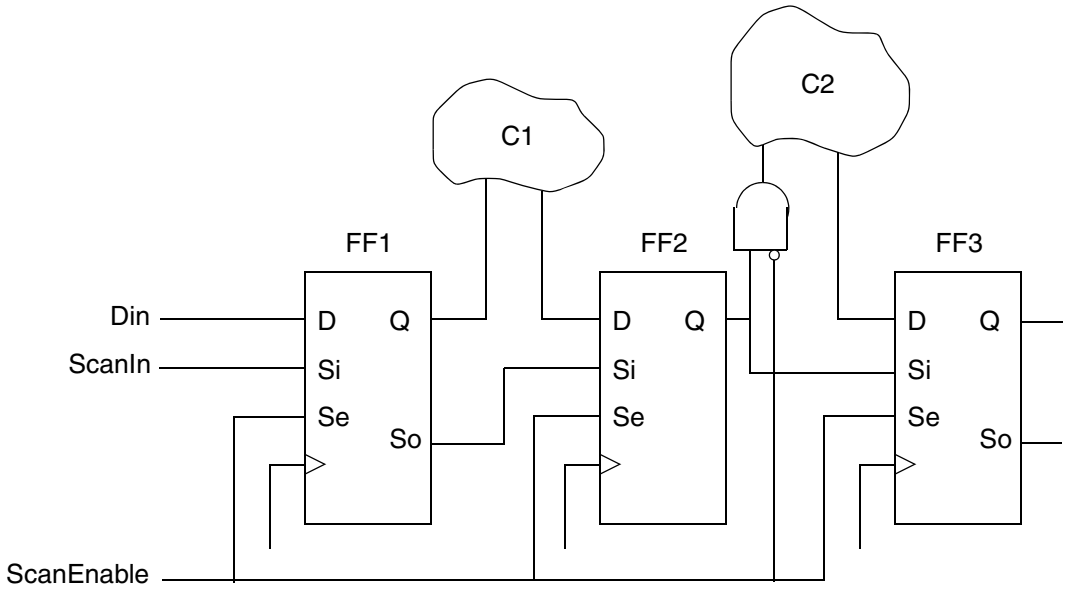
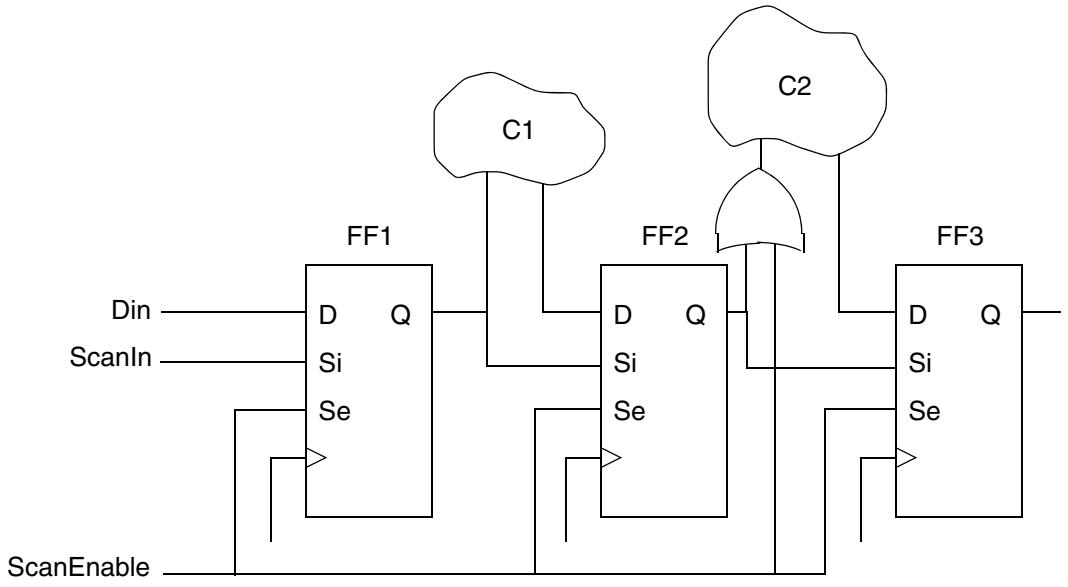


Figure 7-47 Design With OR-Gating Logic Inserted on the Functional Output of FF2, Where FF2 Has a Single Output Pin Shared Between Functional and Scan Output



To use this feature, run the `set_scan_suppress_toggling` command as part of your DFT specifications before you run the `insert_dft` command.

The `set_scan_suppress_toggling` command has the following syntax:

```
set_scan_suppress_toggling
     -selection_method [manual|auto|mixed]
     -include_elements collection_of_design_objects
     -exclude_elements collection_of_design_objects
     -ignore_timing_impact [true|false]
     -min_slack [0_to_1000]
     -total_percentage_gating [0.001_to_100]
```

Table 7-4 set_scan_suppress_toggling Command Syntax

Argument	Description
`-selection_method manual | auto | mixed`	This option defaults to `manual`, which applies gating only for objects manually specified with the `-include_elements` option. The `auto` value enables automatic selection of flip-flops for gating using power-based heuristics. The `mixed` value enables a combined approach, allowing the manual specification to be supplemented with automatic selection by the tool.
`-include_elements` *collection_of_design_objects*	This option is used to manually specify a collection of design objects for gating. Flip-flop instance names, hierarchical cell names, and flip-flop and design references can be specified. A hierarchical cell specification includes all scan flip-flops in the cell.
`-exclude_elements` *collection_of_design_objects*	This option is used to manually specify a collection of design objects that should be excluded from gating. Flip-flop instance names, hierarchical cell names, and flip-flop and design references can be specified. A hierarchical cell specification includes all scan flip-flops in the cell.
`-ignore_timing_impact true | false`	This option is `false` by default. When this option is set to `true`, the minimum slack requirement specified by the `-min_slack` option is ignored. When set to `true`, the `-exclude_elements` option might be useful for guiding automatic selection.
`-min_slack` *num_0_to_1000*	This option specifies the required minimum slack value after gating has been added. The default is 0, which means the slack should be nonnegative after gating.

Table 7-4 set_scan_suppress_toggling Command Syntax (Continued)

Argument	Description
`-total_percentage_gating` *num_0.001_to_100*	This option specifies the target percentage of scan flip-flops to be automatically selected for gating when the `-selection_method` option is set to `auto` or `mixed`. The default for this option is 5.

Functional gating logic can be inserted if the scan flip-flop is part of a scan chain. A flip-flop is excluded from gating consideration if it meets any of the following criteria:

- The flip-flop is manually excluded with the `-exclude_elements` option.
- The flip-flop is part of a shift register segment identified by the `compile_ultra` command.
- In a bottom-up flow, the flip-flop is within a block where test models are used or you have specified a `set_dont_touch` attribute on the block.
- The Q or QN pin of the flip-flop connects only to the scan-in signal. (They can be gated if they are functionally connected.)
- The logic does not meet the minimum slack limit or would not meet the minimum slack limit if gating is inserted.

When the `-selection_method` option is set to `auto` or `mixed`, flip-flops are automatically selected for gating using propagated switching power analysis. A percentage value between 5 and 30 percent is typically specified with the `-total_percentage_gating` option. Consider the tradeoffs between test-mode power consumption, functional power consumption, and functional timing when choosing a gating percentage value. As the gating percentage value is increased, leakage power increases, and the potential for layout congestion and timing closure difficulty increases. While considering timing and power tradeoffs, you should also consider that TetraMAX ATPG has several power-aware algorithms that seek to reduce shift and capture flip-flop toggle rates during the ATPG process.

When the `-selection_method` option is set to `manual` or `mixed`, all flip-flops specified with the `-include_elements` option are gated, except those meeting the exclusion criteria defined earlier.

When the `-selection_method` option is set to `mixed`, all flip-flops specified with the `-include_elements` option are gated, even if the `-total_percentage_gating` target is exceeded. If the `-total_percentage_gating` target is not met by the manually specified flip-flops, additional flip-flops are selected automatically to meet the goal.

Use the `report_scan_suppress_toggling` command to confirm the option settings you specified with the `set_scan_suppress_toggling` command. Use the

remove_scan_suppress_toggling command to remove the previous toggling suppression settings applied.

Before DFT insertion, use the preview_dft command with the -show qgates option to see which scan cells will be gated. This generates a report similar to the -show cells option, except the gated scan cells are annotated with the (g) attribute to indicate gating. For example:

```
dc_shell> preview_dft -show {qgates}

Number of chains: 8
Scan methodology: full_scan
Scan style: multiplexed_flip_flop
Clock domain: no_mix

  (l) shows cell scan-out drives a lockup latch
  (s) shows cell is a scan segment
  (t) shows cell has a true scan attribute
  (w) shows cell scan-out drives a wire
  (g) shows cell scan-out drives a toggle suppressing gate

Scan chain '1' (SI1 --> SO1) contains 48 cells
  CORE/Dstrobe_reg              (CLK, 45.0, rising)
  CORE/R1_reg[0] (g)
  CORE/R1_reg[1] (g)
  ...
```

When the -selection_method option is set to auto or mixed, the preview_dft command reports the number of scan flip-flops automatically selected for gating, the number considered for gating, the number that you manually included, and the number rejected due to timing considerations.

During DFT insertion, the combinational gating cell instances are named using the compile_instance_name_prefix variable, by default. To specify a different instance name prefix for the gating cells, set the test_suppress_toggling_instance_name_prefix variable to the desired prefix. When this variable is set, the scan cell leaf name and output pin name are also appended to the specified prefix. For example, if you set the variable to QGATE, a gating cell inserted at ENAB_reg/Q is named QGATE_ENAB_reg_Q.

After DFT insertion completes, you can use the test_scan_suppress_toggling cell attribute to find the gating cells. For example,

```
dc_shell> set cells [get_cells -hierarchical * \
            -filter {test_scan_suppress_toggling == true}]
{CORE/U181 CORE/U182 U224 U225 U226}
```

Controlling Clock-Gating Cell Test Pin Connections

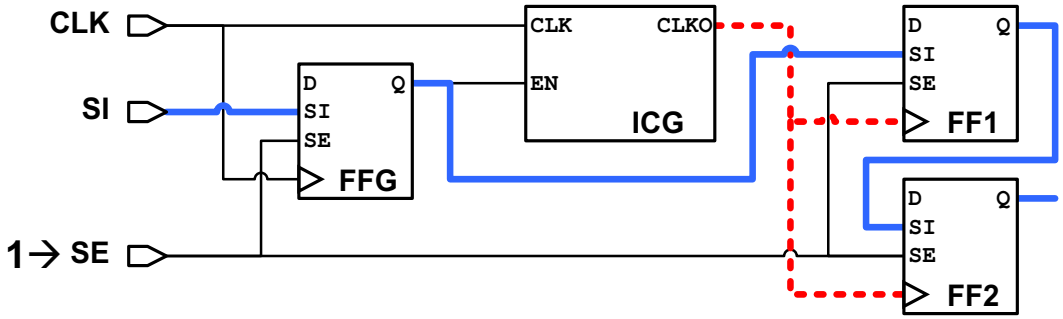
To insert clock-gating cells in a design, you can use the following methods:

- Automatic insertion of clock-gating cells by Power Compiler, using the `-gate_clock` option of the `compile` or `compile_ultra` commands
- Manual instantiation or insertion of clock-gating cells

A clock-gating cell propagates the clock signal to downstream logic only when the enable signal is asserted. During scan shift, if the enable signal is controlled by one or more scan flip-flops, the shifting test data values cause the clock-gating signal to toggle during scan shift. As a result, the clock signal does not reliably propagate through the clock-gating cell to downstream scan flip-flops during scan shift, resulting in scan shift violations.

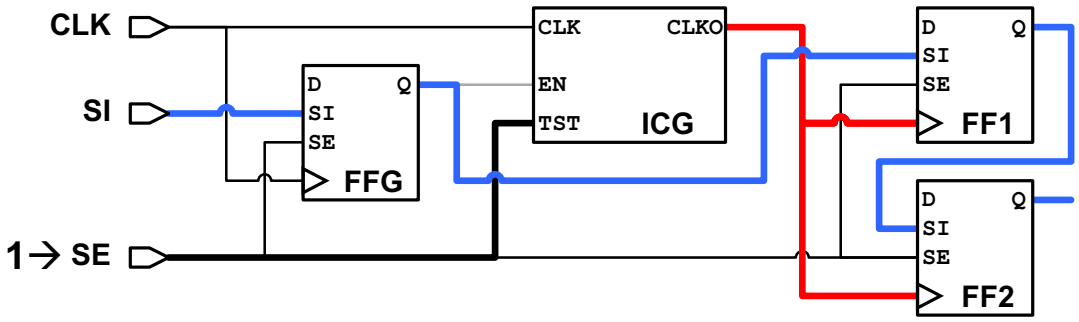
Figure 7-48 shows an example where the enable signal of an integrated clock-gating cell is driven by a scan flip-flop. The scan chain path is highlighted in blue. During scan shift, the clock-gating enable signal driven by FFG toggles, interrupting the scan shift clocks needed for FF1 and FF2.

Figure 7-48 Clock-Gating Cell Enabled by Scan Flip-Flop Output



To remedy this, most clock-gating cells have test pins that force the clock signal to pass through when the test pin is asserted. This ensures that downstream scan cells successfully receive the clock signal and shift values along the scan chain. Figure 7-49 shows a test-aware clock-gating cell with its test pin hooked up to the global scan-enable signal. During scan shift, FF1 and FF2 receive the clock signal and successfully shift data.

Figure 7-49 Clock-Gating Cell Enabled by Scan Flip-Flop Output



This section describes the methods provided by DFT Compiler to control clock-gating cell test pin connections.

Connecting User-Instantiated Clock-Gating Cells

You can use the `insert_dft` command to connect user-instantiated clock-gating cells, that is, to connect to the clock-gating cells that have not been inserted by Power Compiler. You use the `set_dft_clock_gating_pin` command to specify the unconnected test pin of the clock-gating cells in your design. Then you run the `insert_dft` command to connect these pins to the test ports.

Connecting user-instantiated clock-gating cells with the `set_dft_clock_gating_pin` and `insert_dft` commands has the following advantages:

- Provides flexibility
- Does not require setting Power Compiler attributes
- Has no dependency on the `identify_clock_gating` command and the `power_cg_auto_identify` variable
- Works with multiple ScanEnable and TestMode signal connectivity

Note:

When clock-domain-based connections are specified, using the `set_dft_signal -connect_to` command, user-instantiated clock-gating pins are not connected by domain. For this feature, only clock-gating cells recognized and inserted by Power Compiler are supported.

The syntax for the `set_dft_clock_gating_pin` command is

```
set_dft_clock_gating_pin object_list -pin_name instance_pin_name
   [-control_signal ScanEnable | TestMode | scan_enable | test_mode]
   [-active_state 1 | 0]
```

object_list

List of clock-gating cell instances for which test pins are specified. The argument is mandatory.

-pin_name

Name of the test pin on the specified instances. This pin name must be common to all specified instances. The argument is mandatory.

-control_signal

Specifies the type of control signal required by the test pin. The argument is optional. The default is ScanEnable.

-active_state

Specifies the active state of the test pin. The argument is optional. The default is 1.

The command is cumulative.

The specified cells and test pin are not checked. Verify that you have specified the actual clock-gating cells and test pin in the design and that they were not identified by Power Compiler. Specifying cells that are not clock-gating cells can cause undesired results when you run the dft_drc and insert_dft commands.

Using the set_dft_clock_gating_pin Commands: Examples

- To specify the test enable (TE) pin on instances U1 and U2 as the clock-gating pin, using the default signal type ScanEnable:

```
set_dft_clock_gating_pin [list U1 U2] -pin_name TE
```

- To specify the test_mode pin as the clock-gating pin of control signal type TestMode on the hierarchical design des_a:

```
set_dft_clock_gating_pin \
  [get_cells * -hierarchical -filter "@ref_name == des_a"] \
  -control_signal TestMode -pin_name test_mode
```

- To specify pin A as a clock-gating pin of control signal type ScanEnable with active state 1 for all instances of the unique library cell CGC1:

```
set_dft_clock_gating_pin \
   [get_cells * -hierarchical -filter "@reference == CGC1"] \
   -control_signal scan_enable -pin_name A
```

Use the report_dft_clock_gating_pin command to report the specifications you made with the set_dft_clock_gating_pin command. To remove the DFT clock-gating pin specifications, use the remove_dft_clock_gating_pin command.

Interaction With Other Commands

Clock-gating cells identified with the `set_dft_clock_gating_pin` command can be used or specified in the following commands:

- In the hook-up-only flow, using the following command:

  ```
  set_dft_configuration -scan disable -connect_clock_gating enable
  ```

- Using the `set_dft_signal -connect_to` command

  The connection is not made when doing clock-domain-based connections.

- Using the `set_dft_clock_gating_configuration -exclude_elements` command.

Script Example

Example 7-23 and Example 7-24 show a hookup-clock-gating only flow and a complete DFT insertion flow, respectively.

Example 7-23 Hookup-Clock-Gating Only Flow

```
read verilog test.v
link
set_dft_signal -view existing_dft -type ScanClock -port clk \
  -timing {45 55}
set_dft_signal -type ScanEnable -port SE1 -view spec
set_dft_clock_gating_pin {clk_gate_out1_reg sub1/clk_gate_out1_reg} \
  -pin_name TE
set_dft_configuration -scan disable -connect_clock_gating enable
report_dft_clock_gating_pin
insert_dft
```

Example 7-24 Complete DFT Insertion Flow

```
read verilog test.v
link
set_dft_signal -view existing_dft -type ScanClock -port clk \
  -timing {45 55}
set_dft_signal -type ScanEnable -port SE1 -view spec
set_dft_clock_gating_pin {clk_gate_out1_reg sub1/clk_gate_out1_reg} \
  -pin_name TE
set_dft_configuration -scan enable -connect_clock_gating enable
report_dft_clock_gating_pin
create_test_protocol
dft_drc -verbose
insert_dft
```

Limitations

Note the following limitations:

- You cannot connect the clock-gating pins by using the `set_dft_signal -connect_to` command when making clock-domain-based connections.
- The feature is not supported for pipelined ScanEnable.

Excluding Clock-Gating Cells From Test-Pin Connection

You might have a portion of the design that is excluded from scan testing, and you do not want DFT Compiler to connect the test pins of those clock-gating cells to test-mode or scan-enable signals. To prevent the `insert_dft` command from connecting the test pins of some clock-gating cells, use the `-exclude_elements` option of the `set_dft_clock_gating_configuration` command:

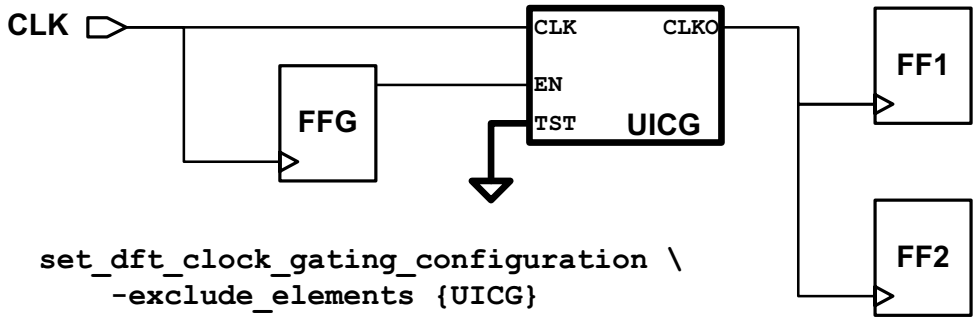
```
set_dft_clock_gating_configuration -exclude_elements object_list
```

The *`object_list`* can include the following object types:

- Clock-gating cell leaf instances
- Clock-gating observation cell leaf instances
- Hierarchical cell instances – All clock-gating cells within the instances are included in the specification.
- Clock-gating library cell – All instances of that cell are included in the specification.
- Clocks – All clock-gating cells in the clock domains are included in the specification.

Instead of connecting the test pins of excluded clock-gating cells to test-mode or scan-enable signals, DFT Compiler leaves the existing connections in place, which are typically constant drivers that de-assert the test-mode bypass. Figure 7-50 shows an example of an excluded clock-gating cell.

Figure 7-50 Directly Specified Excluded Clock-Gating Cell



The `dft_drc` command does not report any TEST-130 unconnected test-pin messages for excluded clock-gating cells. However, any downstream scan cells driven by the excluded clock-gating cells will result in D1 or D9 violations if their clock is uncontrolled.

If you do not know the clock-gating cell instances, but you do know the nonscanned flip-flops whose upstream clock-gating cells should not be connected, you can use the `-dont_connect_cgs_of` option of the `set_dft_clock_gating_configuration` command:

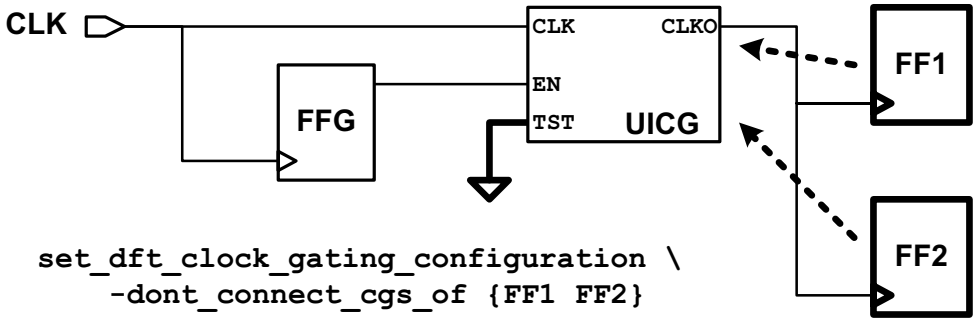
```
set_dft_clock_gating_configuration -dont_connect_cgs_of object_list
```

The *`object_list`* can include the following object types:

- Flip-flop leaf instances
- Hierarchical cell instances – All flip-flops within the instances are included in the specification.

With the `-dont_connect_cgs_of` option, DFT Compiler identifies any upstream clock-gating cells from these registers, and prevents their test pin connections. Figure 7-51 shows an example of an excluded clock-gating cell.

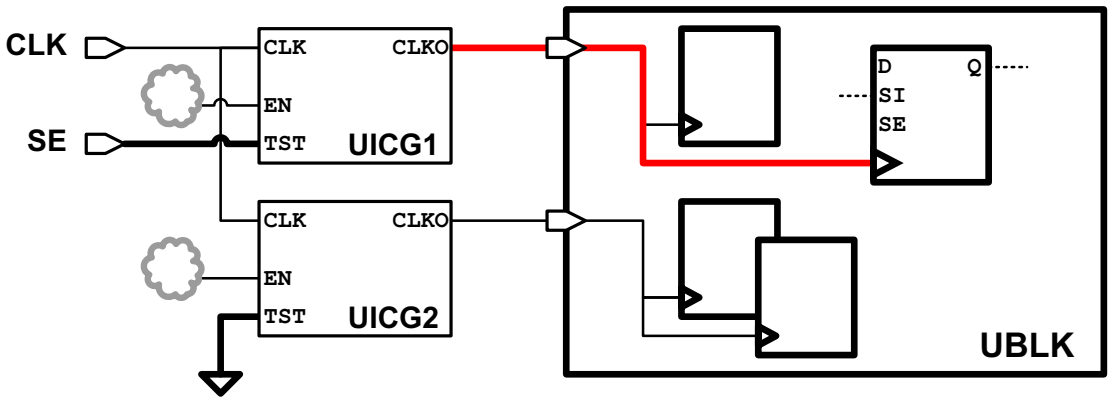
Figure 7-51 Upstream Excluded Clock-Gating Cell



The upstream clock-gating cell can only drive nonscan cells or scan-replaced cells that are excluded from scan stitching. If the clock-gating cell drives any valid scan cells that are incorporated into scan chains, the test pin is connected to ensure proper scan shift clocking.

Figure 7-52 shows an example where hierarchical block UBLK is specified with the `-dont_connect_cgs_of` option. The four flip-flops inside the block are included in the specification. However, one of the flip-flops is a valid scan flip-flop that will be included on a scan chain. As a result, the test pin of upstream clock-gating cell UICG1 is tied to a test signal. The test pin of clock-gating cell UICG2 is left tied to its de-asserted constant value.

Figure 7-52 Upstream Excluded Clock-Gating Cell Due to Downstream Scan Flip-Flops




```
set_dft_clock_gating_configuration \
    -dont_connect_cgs_of {UBLK}
```

If DFT Compiler cannot find the upstream clock-gating cells for flip-flops during the `preview_dft` or `insert_dft` commands, it issues TEST-154 information messages for those flip-flops.

For both exclusion methods, use the `report_dft_clock_gating_configuration` command to report the specifications that you previously set. Use the `reset_dft_clock_gating_configuration` command to remove the specifications.

Exclusion works by leaving the default de-asserted constant value connected to the test pins of excluded clock-gating cells. If an exclusion applies to a clock-gating cell with an existing test pin connection other than the default de-asserted constant value, that existing test pin connection remains in place.

The clock-gating cell test pin connection control methods have the following precedence, in order of highest to lowest priority:

- `set_dft_signal -connect_to` (highest priority)
- `set_dft_clock_gating_configuration -exclude_elements`
- `set_dft_clock_gating_configuration -dont_connect_cgs_of` (lowest priority)

When the `-dont_connect_cgs_of` option is used, upstream clock-gating cells of the specified registers are excluded from clock-gating cell test pin connection only if none of the downstream registers are valid scan cells. When the `-exclude_elements` option is used, the test pin connections of the specified clock-gating cells are suppressed even if there are downstream valid scan cells.

Note:

These `set_dft_clock_gating_configuration` options work with, but are not intended to be used for, user-instantiated clock-gating cells. If the test pin of a user-instantiated clock-gating cell should not be hooked up, do not include it in a

set_dft_clock_gating_pin specification. Registers driven by user-defined clock gating cells are traced through simple buffer and inverter logic only.

Connecting Clock-Gating Cell Test Pins Without Scan Stitching

You can use the insert_dft command to connect the test pins of clock-gating cells to scan-enable or test-mode signals without also performing scan insertion or scan stitching. Note, however, that only clock-gating cells with Power Compiler attributes are considered.

To use this feature, you must disable scan insertion and enable the clock-gating connection before running the insert_dft command. This is accomplished by issuing the following command:

```
set_dft_configuration -scan disable -connect_clock_gating enable
```

When using this flow, do not run the create_test_protocol or dft_drc commands. If you do, it will prevent the insert_dft command from making the clock-gating cell test pin connections.

Example 7-25 shows how to implement this feature.

Example 7-25 Using the insert_dft Command to Connect the Test Pins of Clock-Gating Cells

```
read ddc design.ddc
set_clock_gating_style -control_signal scan_enable
create_clock clk -name clk
compile_ultra -scan -gate_clock

set_dft_signal -type ScanEnable -view spec -port ICG_SE

# Disable scan insertion, enable only clock-gating cell
# test pin connections
set_dft_configuration -scan disable -connect_clock_gating enable

# Run insert_dft to connect the clock-gating cell test pins only
insert_dft
```

The insert_dft command issues an information message indicating that clock-gating cell test pins are being connected to the specified test signal:

```
   Routing clock-gating cells
Information: Routing clock-gating cell test pins with no specified
driver to scan enable 'ICG_SE'
1
```

Note:

This capability is not meant to be used as part of a scan-stitching flow. The feature is solely intended to allow you to connect the test pins of clock-gating cells, separate from any scan synthesis run. To connect clock-gating test pins to a different scan-enable signal during scan stitching, use the -usage option of the set_dft_signal command.

For more information, see “Connecting Test Pins to Clock-Gating Cells Using the insert_dft Command” on page 1-59.

The following scenarios show how to use the `insert_dft` command to connect clock-gating cell test pins to various types of test signal ports and pins:

- To connect to the default test_se port:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
insert_dft
```

- To connect to an existing scan-enable port:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type ScanEnable -port SE
insert_dft
```

- To connect to an existing test-mode port:

```
set_clock_gating_style -control_point after -control_signal TestMode
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type TestMode -port TM
insert_dft
```

- To connecting to an input pad cell hookup pin:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
set_dft_signal -view spec -type ScanEnable -port SE \
     -hookup_pin UPAD_SE/IO_Q
insert_dft
```

- To connect to an internal pin—for example, to a black-box output:

```
compile_ultra -gate_clock -scan
set_dft_configuration -scan disable -connect_clock_gating enable
# Enable internal pins flow
set_dft_drc_configuration -internal_pins enable
set_dft_signal -view spec -type ScanEnable -port SE \
     hookup_pin IP_CORE/SE_OUT
insert_dft
```

The following features apply to automatic connection of clock-gating cell test pins:

- Connections are made only to the test pins of valid clock-gating cells that have been correctly identified and have the required Power Compiler attributes.
- Only connections to clock-gating cells are made unless boundary scan insertion is enabled, in which case boundary scan insertion takes precedence.

- Connections specified with the `set_dft_signal -connect_to` command are honored.
- The internal pins flow is supported.

The following features or capabilities do not work or have limited capability:

- If test models need to be connected, you must specify the core-level ScanEnable pin to be connected to the top-level ScanEnable signal, using the `set_dft_signal -connect_to` command.
- Partially incomplete flows (in which you have performed the clock-gating cell connections to the blocks but not performed the rest of the DFT flow) should not be used for top-level DFT insertion.

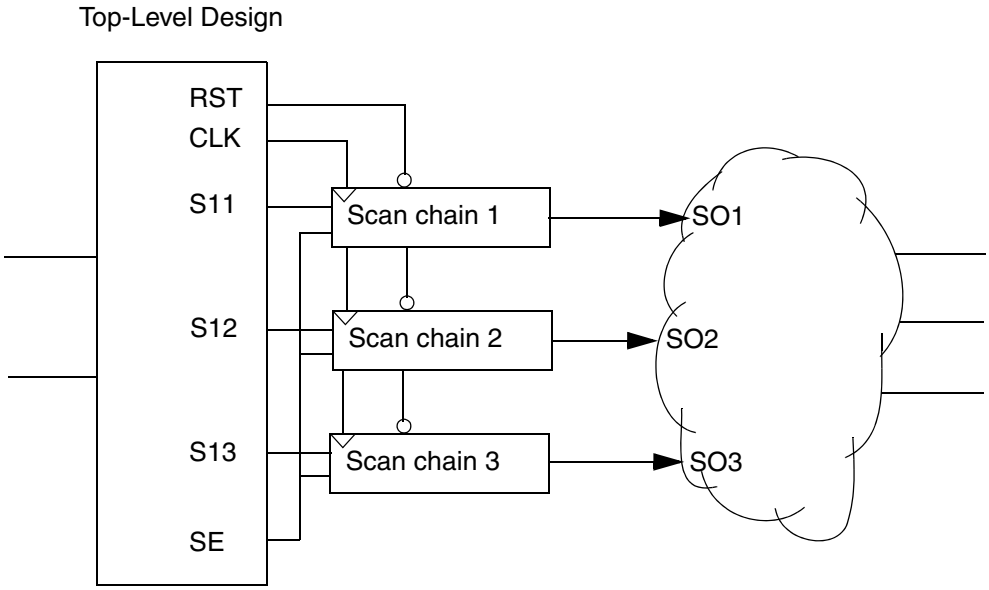
Internal Pins Flow

The DFT Compiler internal pins flow is a test methodology that provides a way to use internal pins, such as scan-in, scan-out, and clocks ports, as top-level ports. This section has the following subsections:

Understanding the Architecture

In the traditional scan architecture approach, the scan chain clock, set/reset, scan-in, scan-out, and scan-enable signals are top-level ports. The internal pins flow enables the use of internal pins instead of top-level ports. The diagram in Figure 7-53 shows the architecture of a typical top-level design with internal pins used for scan inputs, scan outputs, clocks, and resets.

Figure 7-53 Top-Level Design With Internal Signal Connections



In the internal pins flow, the original design is modified in the internal database so that the internal pins are used as top-level ports. These changes are transparent and do not affect the netlist and protocol that is written out.

DFT Commands

The internal pins flow is controlled primarily by two commands:

- `set_dft_drc_configuration` – enables the internal pins flow
- `set_dft_signal` – specifies all hookup pins

Enabling the Internal Pins Flow

The `set_dft_drc_configuration` command enables the internal pins flow. See the following example:

```
dc_shell> set_dft_drc_configuration \
                -internal_pins enable
```

The default of the `-internal_pins` switch is `disable`.

Specifying Hookup Pins

All hookup pins are specified by use of the `-hookup_pin` switch in the `set_dft_signal` command. The following examples show how to use the `-hookup_pin` switch for various data types.

Clocks:

```
dc_shell> set_dft_signal -view existing_dft \
               -type ScanClock \
               -hookup_pin {pinName} -timing [list 45 55]
```

Reset:

```
dc_shell> set_dft_signal -view existing_dft -type Reset \
               -hookup_pin {pinName} -active_state 0|1
```

Constant:

```
dc_shell> set_dft_signal -view existing_dft -type Constant\
               -hookup_pin {pinName} -active_state 0|1
```

TestMode:

```
dc_shell> set_dft_signal -view spec -type TestMode \
               -hookup_pin {pinName} -active_state 0|1
```

Scan_in:

```
dc_shell> set_dft_signal -view spec -type ScanDataIn \
               -hookup_pin {pinName}
```

Scan_out:

```
dc_shell> set_dft_signal -view spec -type ScanDataOut \
               -hookup_pin {pinName}
```

Scan_enable:

```
dc_shell> set_dft_signal -view spec -type ScanEnable \
               -hookup_pin {pinName}
```

Scan path definition for using hookup pins:

```
dc_shell> set_scan_path {scanChainName} -scan_data_in \
               {pinName} -scan_data_out {pinName}
```

Note:

The internal pins flow does not work when inputs are used for the `-hookup_pin` option.

Note:

For all scan signals without an as-yet existing path between the port and hookup pin, only the `-hookup_pin` option should be used. That is, do not use `-port` for the internal pins flow.

Scan Insertion Flow

This section describes the scan insertion flow, which is currently the only flow that supports internal pins modification.

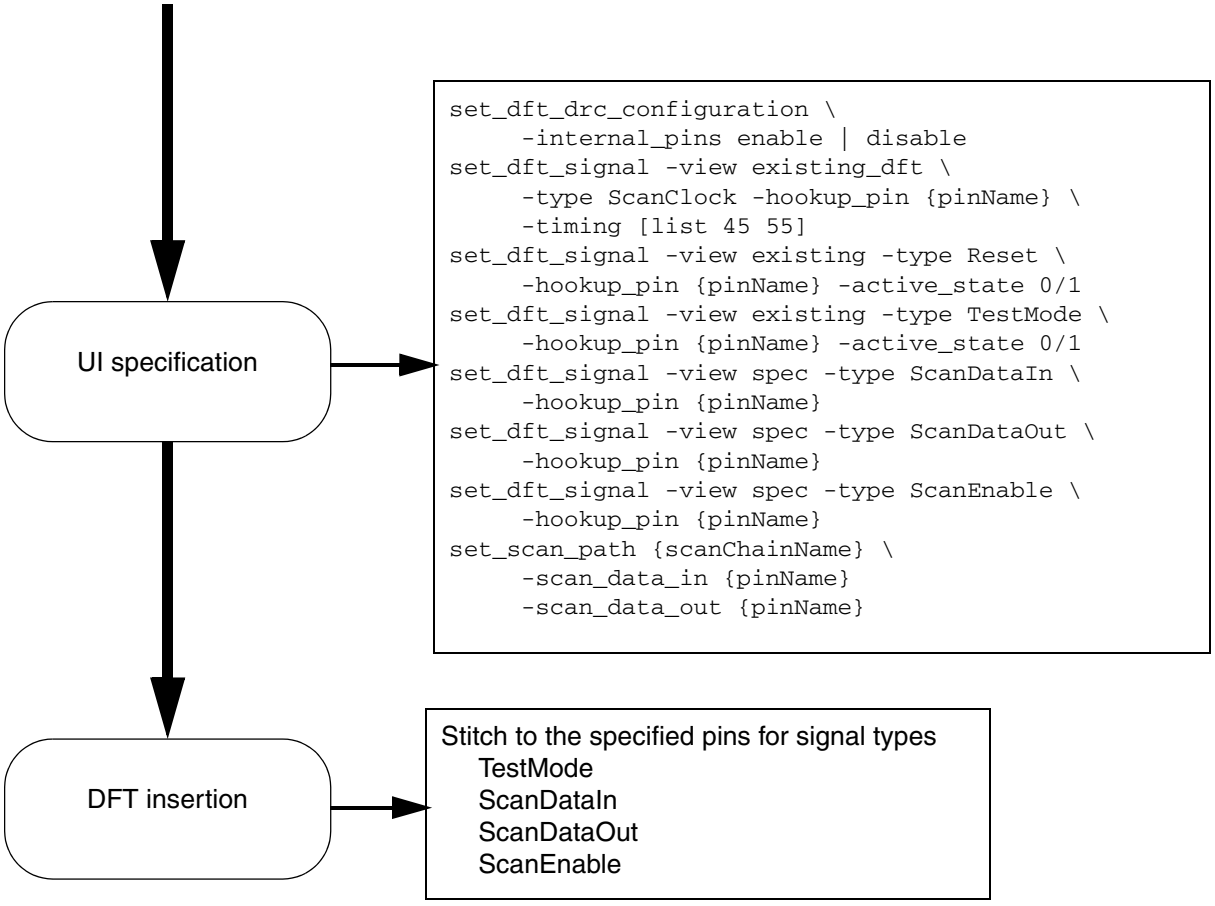
The following data types are supported in this flow as internal signals:

- ScanMasterClock
- MasterClock
- ScanClock
- Reset
- Constant
- TestMode
- ScanEnable
- ScanDataIn
- ScanDataOut

Internal pins are stitched during DFT insertion, based on what you specify with the `set_dft_signal` command. These specifications should precede the running of `dft_drc` command.

Figure 7-54 shows a typical internal pins flow for scan insertion.

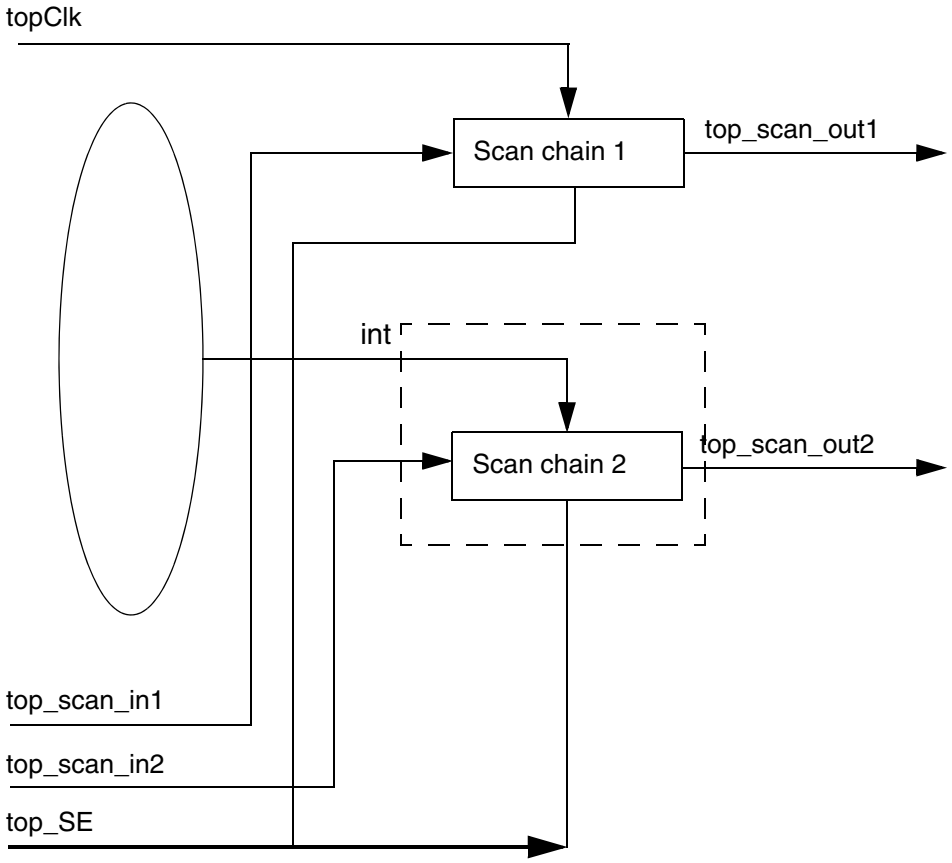
Figure 7-54 Simple Scan Insertion Flow



Mixing Ports and Internal Pins

You can mix the specification of top-level ports with user-specified internal pins for all the signal types. For example, you can specify both a top-level port and an internal pin as a clock, as shown in Figure 7-55.

Figure 7-55 Mixing Specifications of Top-Level Ports and Internal Pins




```
set_dft_signal -port topClk -view existing_dft \
               -type ScanClock -timing {45 55}

set_dft_signal -view existing_dft -type ScanClock \
               -hookup_pin blackBox/int_clk -timing {45 55}
```

Specifying Equivalency Between External Clock Ports and Internal Pins

Specifying an equivalency between external clock ports and internal pins allows you both to designate internal pins for clocks and to write out a usable top-level STIL procedure file. You use the `-associated_internal_clocks` option of the `set_dft_signal` command to

specify that a particular set of internal hierarchical pins has the same timing values and active state as an externally applied clock signal. The set of hierarchical pins is specified as a list.

These pins are considered internal clock sources by the `preview_dft` and `insert_dft` commands when the clock domains for the design are being determined. The top-level clock, to which the `set_dft_signal` command applies, is used in the test model to describe the scan clock of the chain that contains the scan elements triggered by these internal clocks.

The command syntax is

```
set_dft_signal -view existing_dft -port clk_port_name -type ScanClock \
    -associated_internal_clocks [list internal_pin_name]
    -timing timing_values
```

The following conditions apply:

- This feature is enabled only for the `-view existing_dft` specification.
- The association is valid only when `-type` is MasterClock, ScanMasterClock, ScanSlaveClock, or ScanClock.
- The `-associated_internal_clocks` argument only accepts a list, which may contain a single pin or multiple pins.
- To remove the internal pin associations, you must use the `remove_dft_signal` command to remove both the DFT signal and the association.
- The specification of a list of internal pins causes the creation of a list of internal clock pins. Scan flip-flops are associated with these pins and not with the top-level clock ports. This means there are, at a minimum, as many clock domains as the number of pins in the list. The way scan flip-flops are allocated to chains follows the same architecting rules. The behavior of `set_scan_configuration` options remains the same.

Examples of Associating Hierarchical Pins With an External Clock Port

The following examples demonstrate how to associate hierarchical pins with an external port for various cases.

- The following command associates hierarchical pin blkA/pinA with clockA.

```
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -timing {45 55} -associated_internal_clocks [list blkA/pinA]
```

- The following command associates hierarchical pins blkA/pinA and blkB/pinB with clockA.

```
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -associated_internal_clocks [list blkA/pinA blkB/pinB] \
   -timing {45 55}
```

- The result of the following two commands is that only hierarchical pin blkB/pinB is associated with clockA. Pin blkA/pinA is no longer associated with clockA.

```
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -associated_internal_clocks [list blkA/pinA] \
   -timing {45 55}
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -associated_internal_clocks [list blkB/pinB] \
   -timing {45 55}
```

- The result of the following two commands is that the timing specification of clockA is redefined as {55 45} and only blkB/pinB is associated with clockA. Pin blkA/pinA will not be associated with clockA.

```
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -associated_internal_clocks [list blkA/pinA] \
   -timing {45 55}
set_dft_signal -view existing_dft -type ScanClock -port clockA \
   -associated_internal_clocks [list blkB/pinB] \
   -timing {55 45} -hookup_sense inverted
```

Limitations

Note the following limitations:

- The `set_dft_signal -internal_clocks` command has no effect on the associated internal clocks.
- The `-hookup_sense` option has no effect. You can only associate the same clock edge of a list of pins to a top-level clock edge.
- The `report_dft_signal` command does not show the hookup signals.

Limitations to the Internal Pins Flow

The following limitations currently apply to the internal pins flow:

- In most cases, the output protocol is not accurate and cannot be used in the TetraMAX tool. The exception to this limitation occurs when you have only specified internal pins for the clocks and have also specified an association to a top-level clock pin.
- DBIST, phase-locked loop (PLL), core wrapper, boundary scan, and scan extraction flows do not support the internal pin flow.
- You cannot run Hierarchical Scan Synthesis on blocks that were created using the internal pin flow.

Creating Scan Groups

DFT Compiler offers a methodology that enables you to define certain scan chain portions so that they can be efficiently grouped with other scan chains. This is done without the need to insert scan at the submodule levels.

This section includes the following subsections:

Configuring Scan Grouping

DFT Compiler includes the following set of commands that enable you to configure scan grouping:

- `set_scan_group` – creates scan groups
- `remove_scan_group` – removes scan groups
- `report_scan_group` – reports scan groups

Creating Scan Groups

The `set_scan_group` command enables you to create a set of sequential cells, scan segments, or design instances that should be grouped together within a scan chain. If a design instance is specified, all sequential cells and segments within it are treated as a group and is logically ordered.

The syntax for this command is as follows:

```
set_scan_group scan_group_name
     -include_elements {list_of_cells_or_segments}
     [-access {list_of_access_pins}]
     [-serial_routed true|false]
```

scan_group_name

Specifies a unique group name.

-include_elements {*list_of_cells_or_segments*}

Specifies a list of cell names or segment names that are included in the group.

-access {*list_of_access_pins*}

Specifies a list of all access pins. Note that these access pins represent only a serially routed scan group specification. If the -serial_routed option is set to false, all specified access pins are ignored.

-serial_routed [true | false]

Specifies whether the scan group is serially routed (true) or not (false). The default is false.

Note the following:

- There is no -view option to the set_scan_group command, because the options only works in the specification view.
- All scan group specifications are applied across all test modes. You cannot specify a scan group to be applied on a particular test mode.
- An element specified as part of a scan group cannot be specified as part of a scan path, and vice versa.
- Scan group specifications are not cumulative. If you specify the same group name, the last scan group specification before the insert_dft command is used.
- Grouping elements implies that they must be adjacent in a scan chain.

Example

```
dc_shell> set_scan_group G1 -include_elements \
               [list ff1 ff3] -access \
               [list ScanDataIn ff1/TI ScanDataOut \
               ff3/QN] -serial_routed true

dc_shell> set_scan_group G2 -include_elements \
               [list ff2 a/ff1]

dc_shell> set_scan_group G3 -include_elements \
               [list U1/1]

dc_shell> set_scan_group G4 -include_elements \
               [list U1/3]
```

Removing Scan Groups

The remove_scan_group command removes all specified scan groups. The syntax of this command is as follows:

remove_scan_group *scan_group_name*

scan_group_name

This option specifies the name of the scan group to be removed.

Example

```
dc_shell> remove_scan_group G1
```

Integrating an Existing Scan Chain Into a Scan Group

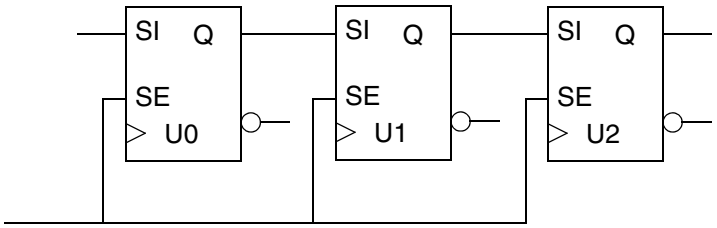
If you have existing serial routed segments at the current design level that you want to incorporate as part of a longer scan chain, you can use the `set_scan_group -serial_routed true` command to accomplish this. When you run the `insert_dft` command, it will then connect to this segment while keeping the segment in tact.

You need to provide the following information to the `set_scan_group -serial_routed true` command:

- Use the `-include_elements` option to specify the names of the elements within the segment.
- Use the `-access` option to specify how the `insert_dft` command should connect to this segment.

Consider the existing scan chain shown in Figure 7-56.

Figure 7-56 Integrating an Existing Scan Chain



In this example, the scan chain connects the flip-flops U0 through U1 to U2. The `insert_dft` command treats U0 through U2 as a subchain or group and connects through the scan-in pin of U0, the output pin of U2, and the scan-enable pin of U0. You can then use the following command to specify the existing scan segment:

```
set_scan_group group1 -include_elements [list U0 U1 U2] \
     -access [list ScanDataIn U0/SI ScanDataOut U2/Q ScanEnable U0/SE]
```

If you do not know the names of the individual elements within the scan group, you can try extracting the element names by using the `dft_drc` command. Note, however, that this strategy works only if the scan segment starts and ends at the top level of the current design. See “Performing Scan Extraction” on page 7-3 for details on how to extract pre-existing scan

chains in your design. After you have extracted the names of the elements of the scan chain, you can specify them using the `-include_elements` option. You might need to disconnect the net connecting the top-level scan-in port to the scan-in pin of the first flip-flop of the chain, as well as the net connecting the data-out or scan-out pin of the last flip-flop of the chain to the scan-out port.

Reporting Scan Groups

The `report_scan_group` command reports the names of the sequential cells or scan segments associated with a particular scan group, as specified by the `set_scan_group` command).

The syntax of this command is as follows:

```
report_scan_group [scan_group_name]
```

scan_group_name

This option specifies the name of the scan group used for reporting purposes. If a group name is not specified, all serially routed and unrouted groups are reported.

Scan Group Flows

You can specify scan groups in DFT Compiler in either the standard flat flow or the Hierarchical Scan Synthesis (HSS) flow.

In the standard flat flow, you can specify valid scan cells as input to scan groups. In the logic domain, these cells get ordered logically and placed as a group in a scan chain.

In the HSS flow, you can specify core segment names as part of a scan group and then reuse them in a scan path specification.

Known Limitations

The following known limitations apply when you specify scan groups in DFT Compiler:

- A scan group can contain only a set of elements that belong to one clock domain.
- You cannot specify a collection as a scan group element.
- You cannot specify a group as part of another group.
- You cannot specify a previously defined scan path in a scan group.

Identification of Shift Registers

When you perform a test-ready compile with the `compile_ultra -scan` command, DC Ultra can identify shift registers and perform scan replacement only on the first register of each identified shift register. Information about the identified shift registers is stored in the design database. DFT Compiler uses this information during scan stitching to efficiently incorporate the shift registers into the scan chain. This flow reduces the area overhead for scan replacement.

DC Ultra can recognize the following types of shift registers during a test-ready compile:

- Simple shift registers

  Each flip-flop directly captures the output of the previous flip-flop.

- Synchronous-logic shift registers

  Each flip-flop captures a combinational logic function which includes the output of the previous flip-flop.

For more information on how to configure DC Ultra to identify shift registers during a test-ready compile, see the “Sequential Mapping” chapter in the *Design Compiler Optimization Reference Manual*.

The `insert_dft` command uses the stored shift register information from DC Ultra to optimize the scan path stitching process. For each shift register, the first scan-replaced cell provides scan controllability for the entire shift register. Simple shift registers are used directly in the scan path. Synchronous-logic shift registers are updated by the `insert_dft` command so that the shift register logically degenerates to a simple shift register when the scan-enable signal is asserted. If needed, identified shift registers are broken up during DFT insertion to meet scan chain balancing or maximum scan chain length requirements.

Use the following methods to determine the shift registers that were identified by the `compile_ultra -scan` command:

- Use the `preview_dft -show segments` command to report the identified shift registers, which are treated as scan segments after being identified.
- Use the `shift_register_head` and `shift_register_flop` cell attributes to identify the leading shift register scan cells and subsequent nonscan cells, respectively:

```
dc_shell> get_cells -hierarchical * -filter \
            {shift_register_head==true || shift_register_flop==true}
```

For best results, write out the design in .ddc format to preserve the identified shift register attributes.

Shift Register Identification in an ASCII Netlist Flow

If you are using the ASCII netlist flow, the attributes for identified shift registers are not stored in the netlist file. Instead, you must perform shift register recognition when the netlist is read in, even if the shift registers were previously recognized at the time the netlist file was written.

When you read in an ASCII netlist (Verilog or VHDL) on which test-ready scan replacement has previously been performed, execute the `set_scan_state test_ready` command to indicate that the netlist is scan-replaced. This command identifies any previously-identified simple shift registers that have their leading cell scan-replaced by DC Ultra. This simple shift-register recognition does not require any additional licenses.

To perform synchronous-logic shift-register identification, set the following variable to `true` before running the `set_scan_state test_ready` command:

```
# the default of this variable is false
set_app_var \
  compile_seqmap_identify_shift_registers_with_synchronous_logic_ascii \
  true
```

This variable enables the `set_scan_state test_ready` command to use the DC Ultra shift register identification engine. In addition, you must ensure that the following two variables remain set to their default of `true` so that the DC Ultra shift register identification engine itself is enabled:

```
# shown for completeness; the default of these variables is true
set_app_var compile_seqmap_identify_shift_registers true
set_app_var \
   compile_seqmap_identify_shift_registers_with_synchronous_logic \
  true
```

When you enable the identification of shift registers with synchronous logic in the ASCII flow, DC Ultra issues the following information message:

```
Information: Performing full identification of complex shift registers.
(TEST-1190)
```

Synchronous-logic shift register identification requires a DC Ultra license.

8

Wrapping Cores

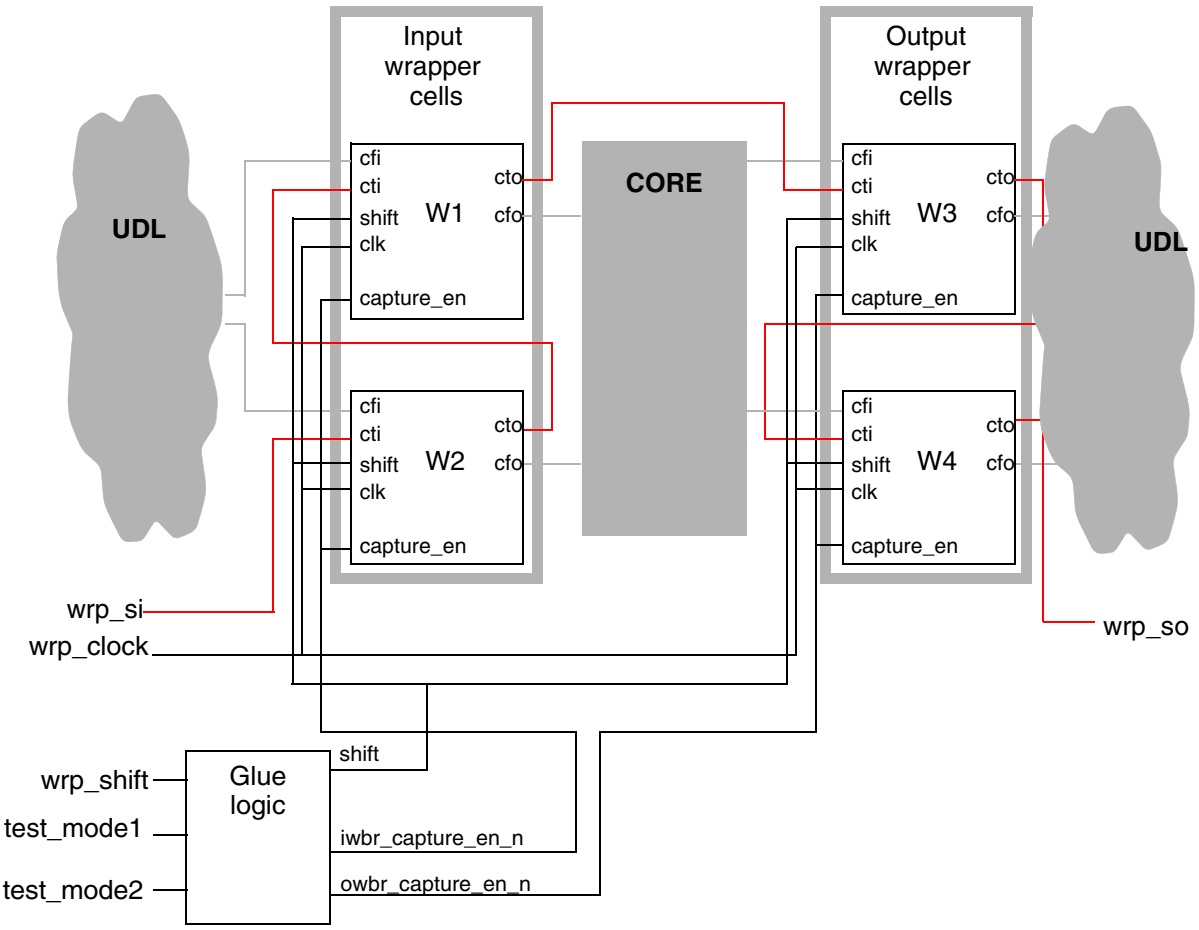
This chapter shows you how to add a test wrapper to a core design. A test wrapper provides both test access and test isolation during scan pattern application. The commands necessary to implement a test wrapper around a core are described in the following sections:

- Wrapper Chain Concepts
- Core Wrapping Flows
- Creating User-Defined Core Wrapping Test Modes
- Wrapping Cores With Existing Scan Chains
- Core Wrapping Scripts

Wrapper Chain Concepts

A test wrapper chain is a chain of wrapper cells placed along the I/O boundary of a core between the I/O ports and core logic. Wrapper cells provide both controllability and observability at functional core inputs and outputs. Figure 8-1 illustrates a core with wrapper cells inserted between the core logic and the surrounding top-level logic.

Figure 8-1 Test Wrapper



An input wrapper cell is associated with an input port; an output wrapper cell is associated with an output port. Wrapper chains can contain only input wrapper cells, only output wrapper cells, or a mix of both types of wrapper cells, depending on whether selective access to input and output wrapper cells is needed.

The input wrapper cells can control the inputs to the core, observe responses from the surrounding logic, or both. The output wrapper cells can control inputs to the surrounding logic, observe core responses, or both. Different core wrapper modes use different combinations of control and observe capabilities.

Core wrapping is primarily intended to wrap core data ports. The following ports are excluded from wrapping:

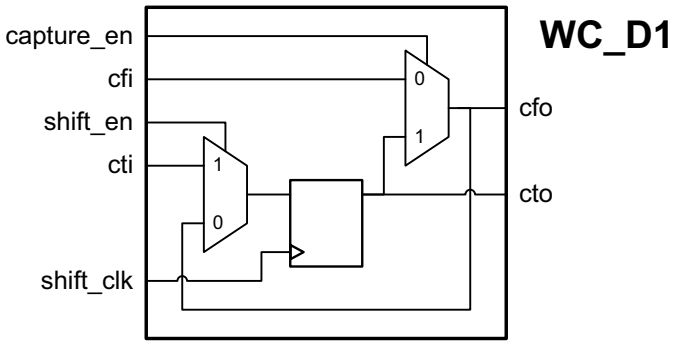
- Functional and test clock ports
- Any asynchronous set or reset signal
- Scan-input, scan-output, scan-enable, and other global test signal ports
- Wrapper shift signal ports
- Any port with a constant signal value defined

Wrapper Cells

Wrapper cells are inserted in a core-level synthesis run. Each wrapper cell is inserted at the core's top level, connected to the port being wrapped.

By default, DFT Compiler uses the WC_D1 *dedicated wrapper cell* for core wrapping. A dedicated wrapper cell is a wrapper cell that uses its own internal dedicated flip-flop to provide controllability, observability, and shift capabilities. Figure 8-2 shows the internal logic of the WC_D1 dedicated wrapper cell.

Figure 8-2 WC_D1 Wrapper Cell



The interface to the WC_D1 wrapper cell consists of the following signals:

cti – Core test input

This is the test input to the wrapper cell. It can come from either a primary input (if the cell is the first cell in the wrapper chain) or the cto signal of the previous wrapper cell in the chain.

cto – Core test output

This is the test output of the wrapper cell. It can drive either a primary output (if the cell is the last cell in the wrapper chain) or the cti signal of the next wrapper cell in the chain.

cfi – Core functional input

For input wrapper cells, this input is fed from the logic surrounding the core. For output wrapper cells, this input is fed from the core.

cfo – Core functional output

For input wrapper cells, this input drives the core. For output wrapper cells, this output drives the logic surrounding the core.

shift_clk – Wrapper clock

This is usually driven by the `wrp_clock` signal in the core. It clocks the flip-flop in the wrapper cell.

shift_en – Shift enable

This is like a scan-enable signal for wrapper cells. When the signal is high, the wrapper clock shifts data through the cti and cto scan data pins. When the signal is low, the wrapper clock captures the functional input value or holds the current state, depending on the value of the capture_en signal. The shift enable signal can be controlled differently for input and output wrapper cells.

capture_en – Capture enable

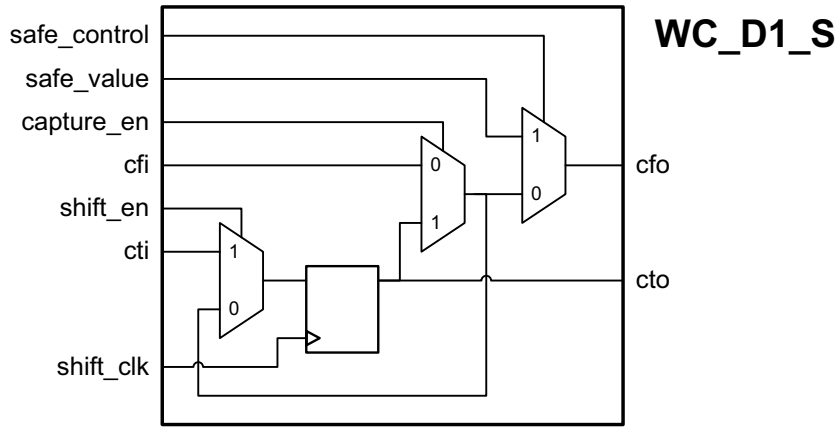
This signal controls what is captured when the wrapper cell is not shifting. When the signal is low, the wrapper clock captures the functional input value. When the signal is high, the wrapper clock holds the current wrapper cell state.

Safe-State Wrapper Cells

A wrapper cell provides observability at its input, and controllability at its output. The same WC_D1 wrapper cell is used for both core inputs and core outputs. However, the controlled output of the wrapper cell can toggle as data is shifted through the wrapper chain. In some cases, if edge-triggered or level-sensitive logic exists in the fanout of the wrapper cell, unintended circuit operation can occur.

To avoid this, you can specify a *safe value* for a wrapper cell. DFT Compiler uses the WC_D1_S wrapper cell to implement the safe value capability. It contains an additional multiplexer at its output to drive a static safe logic value, enabled by a safe value control signal. Figure 8-3 shows the internal logic of the WC_D1_S wrapper cell.

Figure 8-3 WC_D1_S Wrapper Cell



The interface to the WC_D1_S wrapper cell consists of the same signals as the WC_D1 wrapper cell, plus the following additional signals:

safe_control – Control signal

This signal determines when the safe state value is driven at the output.

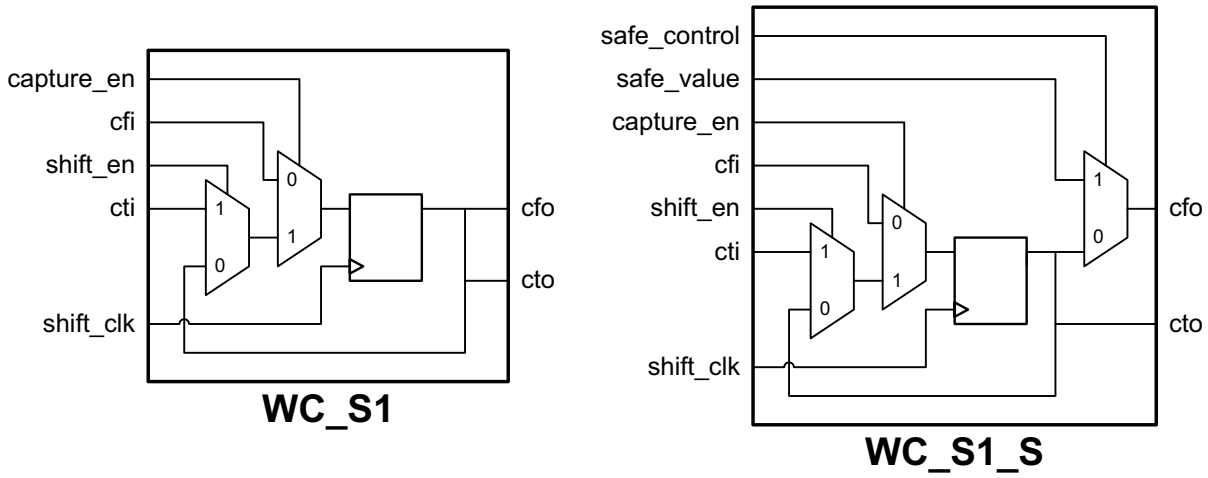
safe_value – Logic value

This signal specifies the safe state logic value.

Shared-Register Wrapper Cells

If your design has existing boundary I/O registers by the ports, you can share these functional registers with the wrapper cell logic to reduce the core wrapping area overhead. Shared wrapper cells replace the existing functional register, providing equivalent functionality in functional mode. Figure 8-4 shows the internal logic for the WC_S1 and WC_S1_S shared wrapper cells. The signals are identical to the signals used for dedicated wrapper cells.

Figure 8-4 WC_S1 and WC_S1_S Wrapper Cells



In order for the core wrapping feature to share the existing functional register, the I/O register and the port must meet the following conditions:

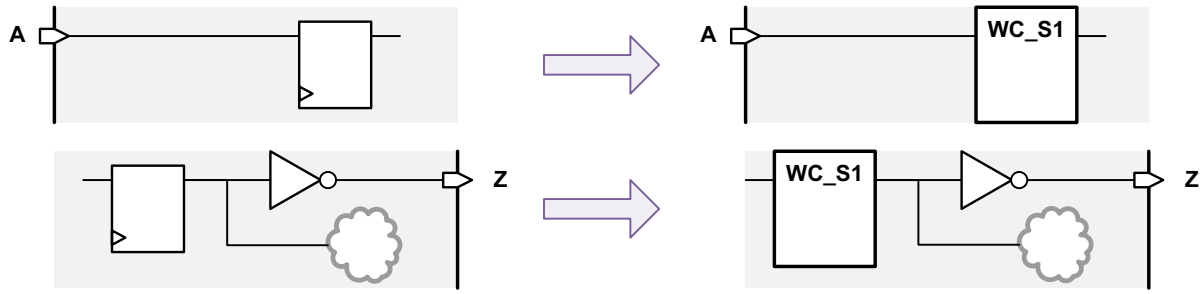
- The register's data input or output must be connected to a boundary port with a wire or logic path. The logic path must be sensitized to produce a buffering or inverting effect, and the sensitization must be controlled by a constant signal type (static value of 0 or 1) on a primary input or output.
- The shared registers must be clocked by a functional clock.

  Note:

  If the functional clock for the shared registers also clocks other functional registers internal to the core, it can disturb the internal core logic during boundary cell operation. You should either provide a separate functional clock for shared wrapper cells, or use the `-use_dedicated_wrapper_clock` option. For more information, see "Simple Core Wrapping Flow" on page 8-11

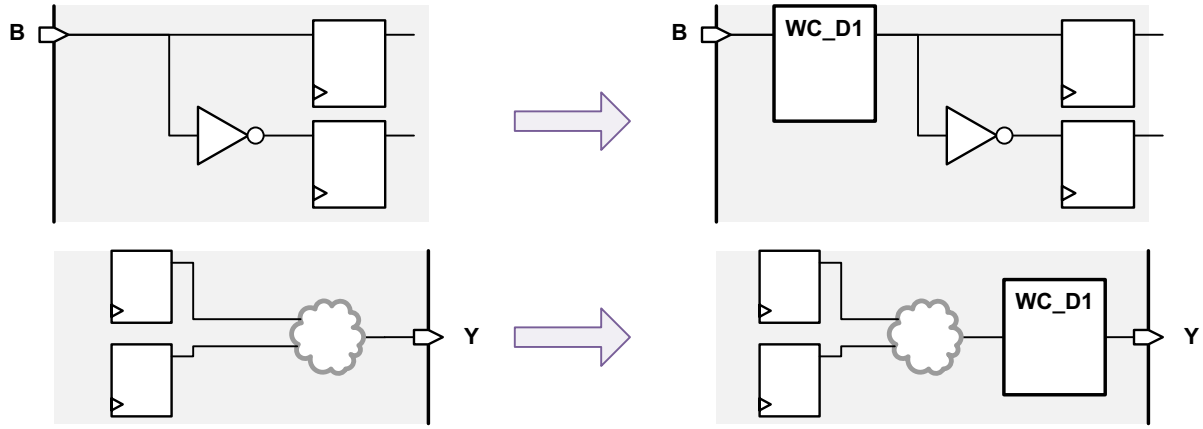
If you enable the shared wrapper cell style, DFT Compiler inserts shared wrapper cells wherever possible. Figure 8-5 illustrates some cases where shared wrapper cells can be used. The shared wrapper cell is placed at the same hierarchical location as the existing design register.

Figure 8-5 Examples of Supported Design Register Sharing



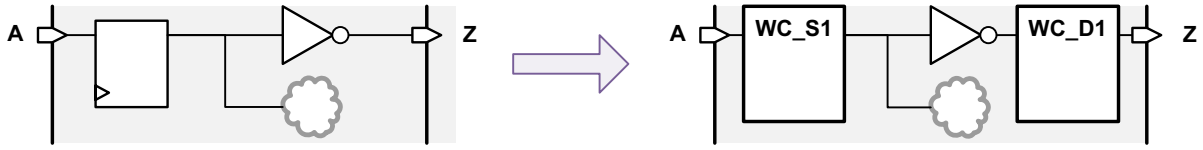
However, if a register does not meet the requirements, a dedicated wrapper cell is used instead. Figure 8-6 illustrates some cases where shared wrapper cells cannot be used.

Figure 8-6 Examples of Unsupported Design Register Sharing



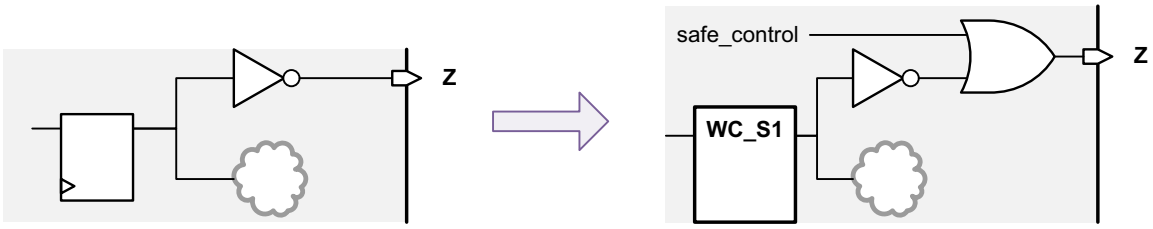
If a register qualifies as both an input shared register and an output shared register, the register becomes an input shared register, and a dedicated wrapper cell is placed at the output port. See Figure 8-7.

Figure 8-7 Shared Input and Output Register Example



If an output shared register has a safe state specified, a WC_S1_S wrapper cell is normally used. However, if the register's output drives internal logic in addition to the output port, the safe state logic inside a WC_S1_S wrapper cell would prevent the register from reliably driving the internal logic. When DFT Compiler detects this situation, it uses a WC_S1 shared wrapper cell and moves the safe state logic to the output port. See Figure 8-8.

Figure 8-8 Output Shared Register With Safe State and Internal Fanout Connections



Wrapper Operation Modes

When a wrapper chain is inserted, test-mode signals control the wrapper chain operating mode. Depending on the value of the test-mode signals, the test wrapper can operate in one of four modes of operation—inward-facing, outward-facing, normal, or safe. The four modes operate as follows:

- INTEST: inward-facing mode

  This mode is used when input vectors need to be applied to the core and the core response needs to be observed at the output. The wrapper cells on the core inputs provide controllability, and the wrapper cells on the core outputs provide observability. This mode is used to test the core and isolate the surrounding logic. If safe values are specified to protect the surrounding fanout logic from the core output response, they are driven at the core outputs. Figure 8-9 shows operation of the design in INTEST mode, and Figure 8-10 shows the timing in INTEST mode.

Figure 8-9 INTEST Mode of Operation of Wrapper Cells

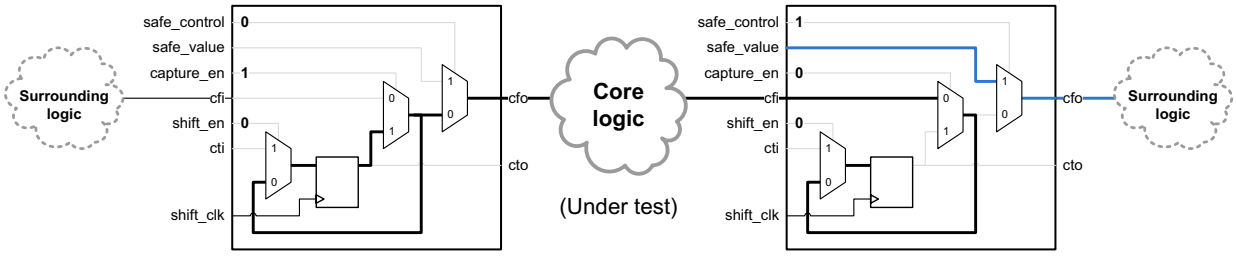
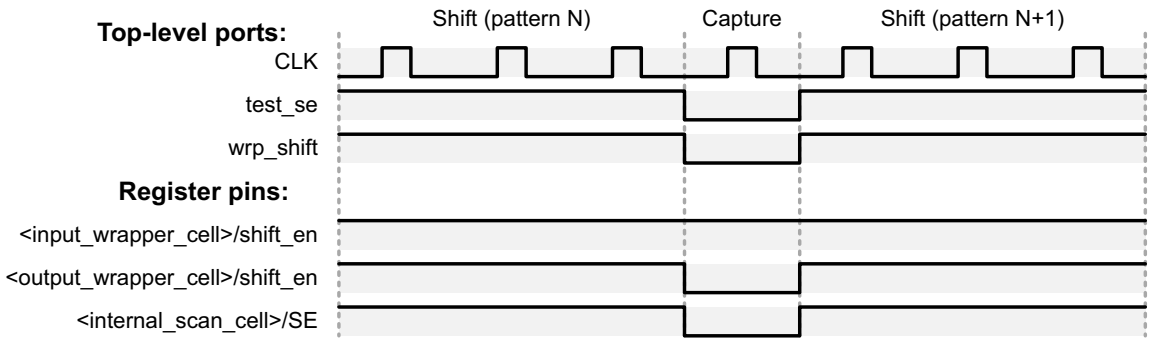


Figure 8-10 INTEST Mode Timing



- EXTEST: outward-facing mode

  This mode is used to test the logic surrounding the core and isolate the core itself. The wrapper cells on the core inputs provide observability, and the wrapper cells on the core outputs provide controllability. If safe values are specified to protect the core inputs from the surrounding fanin logic responses, they are driven at the core inputs. Figure 8-11 shows operation of the design in EXTEST mode, and Figure 8-12 shows the timing in EXTEST mode.

Figure 8-11 EXTEST Mode of Operation of Wrapper Cells

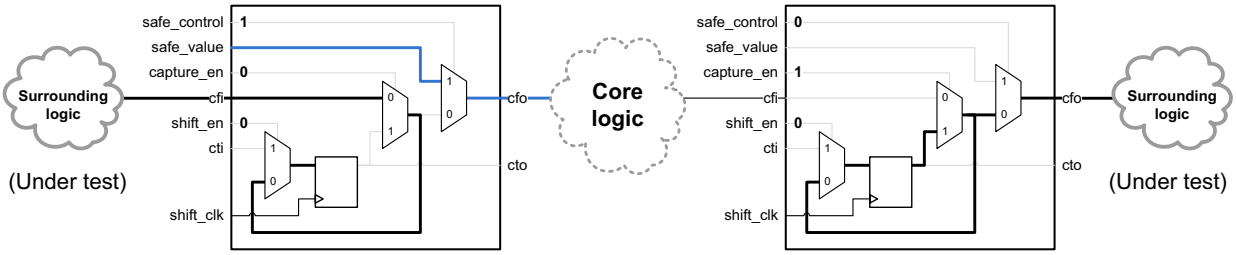
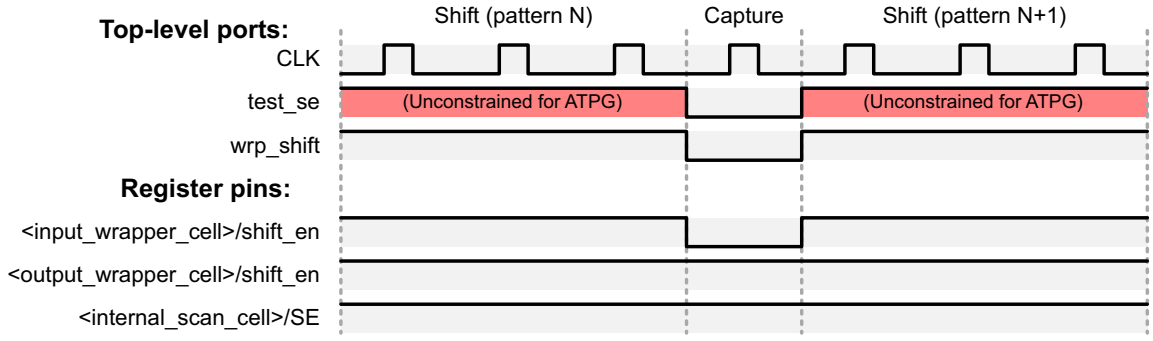


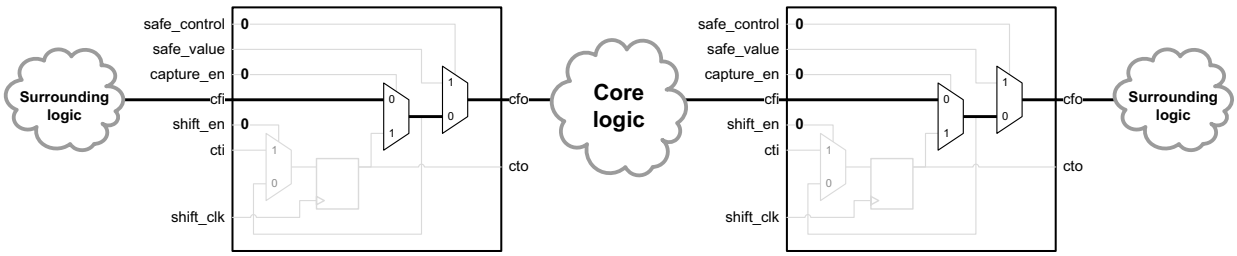
Figure 8-12 EXTEST Mode Timing



- NORMAL: normal mode of operation

  This is the functional mode of operation. The test wrapper is transparent. Figure 8-13 shows operation of the design in NORMAL mode.

Figure 8-13 Normal Mode of Operation of Wrapper Cells

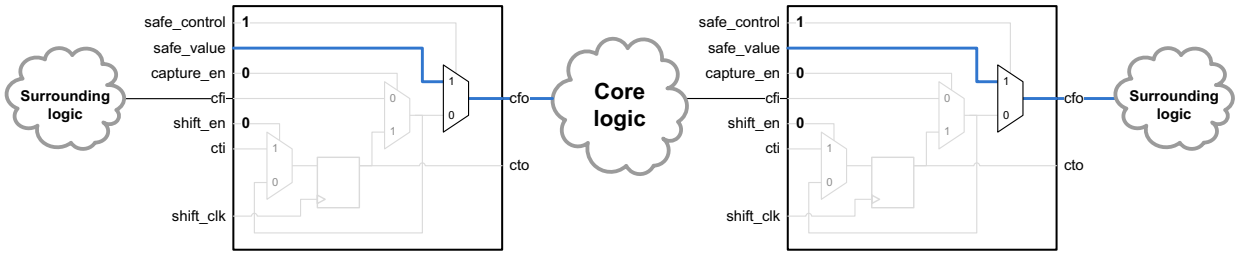


When the test-mode signals activate a standard scan mode, the wrapper chain is also configured for this transparent state for scan operation.

- SAFE: safe mode

  This mode drives the safe value from all wrapper cells that have a safe value specified. Safe values are driven into core inputs by any such input wrapper cells, and safe values are driven into the surrounding fanout logic by any such output wrapper cells. Figure 8-14 shows operation of the design in SAFE mode.

Figure 8-14 SAFE Mode of Operation of Wrapper Cells



This mode might be useful when another core is being tested, and this core and its surrounding logic should remain quiet. If no wrapper cells have a safe value specified, this mode is not created.

Core Wrapping Test Modes

When core wrapping is enabled, the `insert_dft` command creates the following test modes by default:

- wrp_if

  This is an inward-facing uncompressed scan mode. The wrapper chain is placed in the INTEST mode of operation. Both wrapper chains and core internal chains are active.

- ScanCompression_mode

  This is an inward-facing compressed scan mode. The wrapper chain is placed in the INTEST mode of operation. Both wrapper chains and core internal chains are active and

compressed by the scan compression codec. This mode is created if scan compression is also enabled.

- wrp_of

  This is an outward-facing uncompressed scan mode. Only the wrapper chain is active; it is placed in the EXTEST mode of operation.

- wrp_safe

  This mode drives safe values in both the inward and outward directions according to the safe value specifications. This mode is created if a safe value has been defined for any port.

When core wrapping is enabled, DFT Compiler does not create the Internal_scan mode by default because it does not provide inward-facing or outward-facing hierarchical test capabilities.

You can also create user-defined core wrapping test modes. For more information, see "Creating User-Defined Core Wrapping Test Modes" on page 8-42.

Core Wrapping Flows

DFT Compiler provides multiple core wrapping flows. These flows differ in how the wrapper cells are assembled into wrapper chains, and in how the control signals activate the different capabilities of the core-wrapping logic.

The simple core wrapping flow provides basic core wrapping capabilities. The maximized reuse core wrapping flow minimizes the area and timing impact of core wrapping by reusing more existing functional registers. The delay test core wrapping is a legacy flow that is documented for completeness.

This section includes the following subsections:

- Simple Core Wrapping Flow
- Maximized Reuse Core Wrapping Flow
- Delay Test Core Wrapping Flow

Simple Core Wrapping Flow

To use the simple core wrapping flow, you must include the DesignWare dw_foundation.sldb synthetic library in your link library list. This synthetic library contains the wrapper cell designs.

```
dc_shell> set link_library {* my_tech_lib.db dw_foundation.sldb}
```

To enable core wrapping, use the following command:

```
dc_shell> set_dft_configuration -wrapper enable
```

By default, DFT Compiler does not create safe state wrapper cells. To specify a safe state value for all wrapper cells in the wrapper chain, use the `-safe_state` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -safe_state 1
```

The `set_wrapper_configuration` command allows you to specify global configuration parameters that apply to the entire wrapper chain.

By default, DFT Compiler inserts dedicated wrapper cells to wrap input and output ports. If you have existing boundary I/O registers that you want to share with the wrapper cells, you can enable the shared wrapper cell style by using the `-style shared` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -style shared
```

When the shared wrapper cell style is enabled, by default, DFT Compiler uses dedicated wrapper cells as a fallback for any ports that do not meet the sharing criteria. To prevent the fallback insertion of dedicated wrapper cells, use the `-dedicated_cell_type none` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -style shared -dedicated_cell_type none
```

DFT Compiler automatically selects the type of wrapper cell (WC_D1, WC_D1_S, WC_S1, or WC_S1_S) based on the capabilities needed for each wrapper cell.

The `set_wrapper_configuration` command applies to all ports in the design. To specify the wrapper cell characteristics of specific ports, you can use the `set_boundary_cell` command.

To specify safe state values for specific ports, use the `-safe_state` option of the `set_boundary_cell` command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_D1_S -safe_state 0 | 1
```

Note:

When using the `set_boundary_cell` command, you must explicitly provide the wrapper cell type in the specification. The specified type should match the safe state and shared register characteristics for that port. In the preceding example, a dedicated wrapper cell is used.

To prevent the insertion of wrapper cells for a specific list of ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type none
```

This might be needed in cases where an output port drives downstream clock pins or asynchronous set or reset signals. If the output port is wrapped, toggle activity in the wrapper cell might cause unintended activity in the downstream logic. Since excluding ports from the wrapper chain reduces test coverage, you should use this capability only when necessary.

To specify a dedicated wrapper cell for ports that would otherwise use a shared wrapper cell, specify a WC_D1 or WC_D1_S wrapper cell:

```
dc_shell> # no safe state:
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_D1

dc_shell> # safe state:
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_D1_S -safe_state safe_value
```

Note:

You cannot use the `set_boundary_cell` command to force a shared wrapper cell type to be used for a port if the I/O register does not meet the requirements for a shared wrapper cell or if sharing has not been enabled with the `-style shared` option.

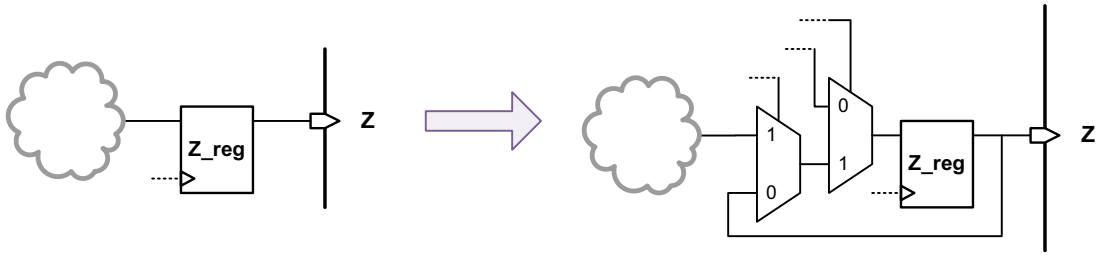
When a shared wrapper cell is used for a port, DFT Compiler replaces, or swaps, the entire I/O register for that port with a shared wrapper cell, as shown in Figure 8-5 on page 8-7. This process introduces a level of hierarchy around the register and renames the register cell itself.

To preserve the original hierarchical instance path of the register, use the `-register_io_implementation in_place` option of the `set_wrapper_configuration` or `set_boundary_cell` command:

```
dc_shell> # global:
dc_shell> set_wrapper_configuration -class core_wrapper \
            -style shared \
            -register_io_implementation in_place

dc_shell> # per-port:
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_S1 \
            -register_io_implementation in_place
```

This option implements the shared wrapper cell functionality by using discrete logic gates around the existing I/O register, as shown in Figure 8-15. The location of the original I/O register is not disturbed.

Figure 8-15 Shared Wrapper Cell Using In-Place Register Implementation

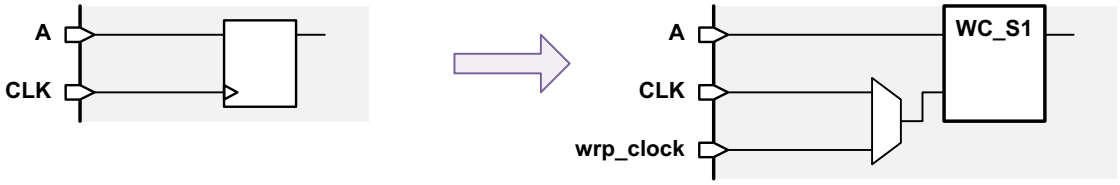


For dedicated wrapper cells, DFT Compiler uses the dedicated wrapper clock signal. For shared wrapper cells, it keeps the register's existing functional clock signal. This can disturb the internal core logic during boundary cell operation. You should either provide a separate functional clock for shared wrapper cells, or use the `-use_dedicated_wrapper_clock` option of the `set_wrapper_configuration` or `set_boundary_cell` command:

```
dc_shell> # global:
dc_shell> set_wrapper_configuration -class core_wrapper \
            -style shared \
            -use_dedicated_wrapper_clock true

dc_shell> # per-port:
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_S1 \
            -use_dedicated_wrapper_clock true
```

When enabled, this option uses the dedicated wrapper clock signal when wrapper test modes are active, but retains the original functional clock signal for other modes. See Figure 8-16.

Figure 8-16 MUXing Dedicated Wrapper Clocks For Shared Wrapper Cells



Wrapper chains follow the same clock mixing requirements as normal scan chains. By default, if the wrapper cells use multiple clock signals for wrapper operation, the cells will be placed in separate wrapper chains, one chain for each clock used.

The `set_scan_configuration -clock_mixing mix_clocks` command allows differently-clocked wrapper cells to be mixed in the same wrapper chain. Use the `-clock_mixing` option of the `set_scan_configuration` command to mix wrapper cells belonging to different clock domains on a wrapper chain. For example,

```
dc_shell> set_scan_configuration -clock_mixing mix_clocks
```

The default for the -clock_mixing option is no_mix. In this case, DFT Compiler creates separate wrapper chains for each of the wrapper clock domains. Lock-up latches are inserted between wrapper cells that do not use the same clock.

By default, DFT Compiler creates two new ports, wrp_shift and wrp_clock, for the wrapper clock and shift signals. If you have existing ports to use for these signals, you can use the set_dft_signal command to define them:

```
dc_shell> set_dft_signal -view spec -type wrp_clock \
            -port port_name

dc_shell> set_dft_signal -view spec -type wrp_shift \
            -port port_name
```

DFT Compiler also creates any test-mode ports needed to provide the test-mode encodings for the functional, scan, and wrapper modes. If you have existing ports to use for these test-mode signals, you can use the set_dft_signal to define them as TestMode signals.

Reporting the Wrapper Cells

After you have configured your wrapper chain and wrapper cells, use the preview_dft command with the -test_wrappers all option to preview the scan chain and wrapper chain characteristics. Example 8-1 shows a wrapper chain preview report for a design.

Example 8-1 Preview Report of Shared I/O Registers

```
dc_shell> preview_dft -test_wrappers all
...
****************************************
Test wrapper plan report
Design : coreJF
Version: 2004.12
Date : Wed Feb 2 12:00:12 2005
****************************************

Number of designs to be wrapped : 1
MY_core

Number of Wrapper Interface ports : 5

port type   port name
---------   ---------
WRP_CLOCK   wrp_clock
WRP_CLOCK   ck1
WRP_CLOCK   ck2
WRP_SHIFT   wrp_shift
```

```
Note: Dedicated wrapper cells are grouped into a hierarchical instance
named:
"coreJF_Wrapper_inst" (Module name: "coreJF_Wrapper_inst_design")

Wrapper Length: 7

              Wrapper   Wrapper    Control    Safe   Wrapper
Index  Port   Function  Cell Type  Cell Impl  Value  Clock      Cell Name
-----  ----   --------  ---------  ---- ----  -----  -------    ---------
6      A1     input     WC_S1       -  SWP     -     ck1        I1_reg
5      A2     input     WC_D1       -  SWP     -     wrp_clock  A2_wrp0_8
4      A3     input     WC_D1       -  SWP     -     wrp_clock  A3_wrp0_7
3      A4     input     WC_D1       -  SWP     -     wrp_clock  A4_wrp0_6
2      Q1     output    WC_S1       -  SWP     -     ck1        Q1_reg
1      Q2     output    WC_S1       -  SWP     -     ck2        iQ2_reg
0      Q3     output    WC_S1       -  SWP     -     ck2        iQ3_reg

Number of ports not wrapped     : 3
MAINT_PORT
ck1
ck2
```

Input and output wrapper chains as well as core scan chains are shown in the `preview_dft` report. To see this information, run the following command:

```
dc_shell> preview_dft -show all -test_wrappers all
```

For all ports that have wrapper cells, the following information is reported:

- Index number of shared wrapper cells
- Port name
- Wrapper cell type: dedicated or shared
- Function of wrapper cell: input, output, three-state, or control
- Wrapper cell clock
- Wrapper cell instance names

After DFT is inserted with the `insert_dft` command, you can report both normal scan chains and wrapper chains with the `report_scan_path` command. To report wrapper chains, use the `-test_mode` option of the `report_scan_path` command to provide the wrapper test-mode name. To list the available test modes, use the `list_test_modes` command.

Wrapper Signal Behavior

Table 8-1 describes the shift and capture behavior of the wrapper cells for the various test modes.

Table 8-1 Test-Mode Behavior of Wrapper Cells in the Simple Core Wrapping Flow

	Shift	Capture (inputs)	Capture (outputs)	Safe control (inputs)	Safe control (outputs)
Mission mode	0	0	0	0	0
Standard scan	0	0	0	0	0
Compressed scan	S[a]	1	0	0	1
wrp_if	S	1	0	0	1
wrp_of	S	0	1	1	0
wrp_safe	0	0	0	1	1

a. The default name for this wrapper signal is wrp_shift.

Compressed scan modes, defined with the `-usage scan_compression` option, configure the wrapper chain cells to operate in an inward-facing mode, just as the wrp_if mode does.

Standard scan modes, defined with the `-usage scan` option, configure dedicated wrapper cells to be transparent and shared wrapper cells to be in functional mode. These modes are not used for core-level testing; define and configure them only if core-level scan segments are needed by a top-level test mode.

Separating Input and Output Wrapper Cells

By default, the simple wrapper mode inserts a wrapper chain with the following characteristics:

- Input and output wrapper cells can be mixed on the same scan chain.
- Input and output wrapper cells share the same wrapper shift signal.

To prevent input and output wrapper cells from being mixed on the same scan chains, use the `-mix_cells false` option of the `set_wrapper_configuration` command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -mix_cells false
```

This command places the input and output wrapper cells in separate chains, but the wrapper cells still share a common wrapper shift signal. To use separate shift signals for input and

output wrapper cells, define two signals as `wrp_shift` signals with the `set_dft_signal` command, then specify them with the `-input_shift_enable` and `-output_shift_enable` options of the `set_wrapper_configuration` command:

```
dc_shell> set_dft_signal -view spec \
            -type wrp_shift -port {wrp_ishift wrp_oshift}

dc_shell> set_wrapper_configuration -class core_wrapper \
            -mix_cells false \
            -input_shift_enable wrp_ishift \
            -output_shift_enable wrp_oshift
```

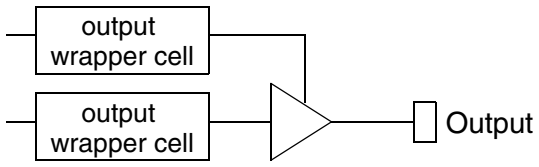
The `wrp_ishift` signal will be used as the wrapper shift signal for the input wrapper chain, and the `wrp_oshift` signal will be used as the wrapper shift signal for the output wrapper chain.

Wrapping Three-State and Bidirectional Ports

To prevent contention during core integration, three-state and bidirectional ports are wrapped differently from regular input and output ports. The wrapper cells are inserted in different locations.

For a three-state port, as shown in Figure 8-17, wrapper cells are added to the data-out and enable paths of the tristate driver or pad cell connected to the port. When the enable path wrapper cell asserts the tristate driver signal, the data-out path wrapper cell controls the output port value; otherwise, the data-out path wrapper cell has no effect.

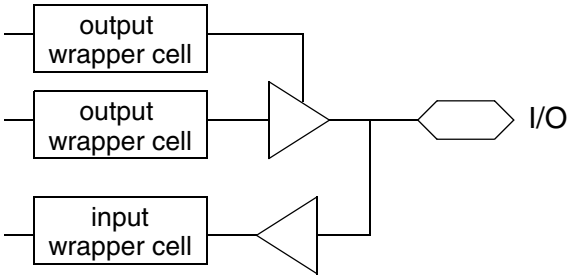
Figure 8-17 Three-State Port Wrapping



Specifying a safe state for a three-state port has no effect, because DFT Compiler automatically inserts safe-value logic on the enable pin of the three-state driver to assert a high-impedance safe state. The safe value applied to the tristate driver enable pin depends on the sense (inverted or noninverted) of the enable pin.

For a bidirectional port, as shown in Figure 8-18, wrapper cells are added to the data-out and enable paths as well as the data-in path of the bidirectional pad connected to the port. When the enable path wrapper cell asserts the output enable value for the pad, the data-out path controls the port; otherwise, the data-in path wrapper cell is controlled by the port.

Figure 8-18 Bidirectional Port Wrapping



DFT Compiler applies a safe value specified for a bidirectional port to the output pin of the three-state driver.

Degenerate three-state and bidirectional ports, in which the driver cell functionality is simplified using constant values, are not wrapped.

Controlling Wrapper Chain Count and Length

You can use the `set_scan_configuration -chain_count` command to control the number of scan chains and wrapper chains in the design:

```
dc_shell> set_scan_configuration -chain_count N
```

This command affects the chain counts as follows:

- In scan test modes, it specifies the number of scan chains.
- In wrapper test modes, it specifies the total number of chains, including both scan chains and wrapper chains.

In each test mode, DFT Compiler balances the chains within the constraints of chain count, clock mixing requirements, and other test constraints. In wrapper modes, DFT Compiler determines the number of scan chains and wrapper chains that satisfies the total chain count while achieving optimal chain balancing. Scan cells and wrapper cells cannot be mixed on the same chain.

In wrapper modes, DFT Compiler chooses how many wrapper chains and scan chains to create to satisfy the total chain count *N*. You can use the `-chain_count` option of the `set_wrapper_configuration` command to directly specify the number of wrapper chains created in wrapper modes:

```
dc_shell> set_scan_configuration -chain_count N ;# total chain count

dc_shell> set_wrapper_configuration -class core_wrapper \
            -chain_count W ;# wrapper chain count
```

In this case, DFT Compiler creates *W* wrapper chains, then allocates the remaining *(N – W)* chains to scan chains.

If no chain counts are specified, DFT Compiler builds the minimal number of scan chains in each mode.

If clock mixing is not enabled with the `set_scan_configuration -clock_mixing` option, separate wrapper chains are created for each wrapper clock used in the design, including the dedicated wrapper clock used for any dedicated wrapper cells. To avoid making so many wrapper chains, use one of the following techniques:

- Enable clock mixing with the `set_scan_configuration -clock_mixing` command.
- Use the dedicated wrapper clock for shared wrapper cells with the `-use_dedicated_wrapper_clock` option of the `set_wrapper_configuration` or `set_boundary_cell` command.

You can use the `set_scan_configuration -max_length` command to specify the maximum length requirement of scan chains and wrapper chains in the design:

```
dc_shell> set_scan_configuration -max_length L
```

DFT Compiler then creates as many wrapper chains as necessary to keep the length of any chain at or below the specified limit.

You can also use the `set_wrapper_configuration -max_length` command to apply a maximum length requirement for only wrapper chains:

```
dc_shell> set_wrapper_configuration -class core_wrapper -max_length X
```

A wrapper-specific chain count or length specification, applied with the `set_wrapper_configuration` command, takes precedence over a general chain count or length specification, applied with the `set_scan_configuration` command. If a chain count and chain length specification are both applied with the same command, the length requirement is used and the count requirement is ignored.

Using the set_scan_path Command With Wrapper Chains

You can use the `set_scan_path` command to control detailed aspects of wrapper chain construction. The following options are supported for wrapper chains:

```
set_scan_path
    -class wrapper scan_chain_name
    [-ordered_elements ordered_port_list]
    [-complete true | false]
    [-input_wrapper_cells_only enable | disable]
    [-output_wrapper_cells_only enable | disable]
    [-scan_enable se_port]
    [-test_mode test_mode_name]
    [-scan_data_in si_port]
    [-scan_data_out ordered_port_list]
```

You must always specify the `-class wrapper` option when using `set_scan_path` to configure wrapper chains.

You can use the `-ordered_elements` option to control the ordering of wrapper cells in the wrapper chain. Provide an ordered list of ports, and DFT Compiler uses it to order the corresponding wrapper cells according to the specification:

```
dc_shell> set_scan_path -class wrapper WC1 \
            -ordered_elements {C B A Z Y}
```

By default, DFT Compiler can add wrapper cells to the beginning of the specified chain. To prevent this, specify the `-complete true` option. You cannot use multiple ordered list specifications to add more wrapper cells to an already specified wrapper chain because the last specification for a chain overwrites any previous specification. Wrapper cells cannot belong to more than one chain, so if you specify a cell as belonging to more than one chain, the last specification takes precedence.

As shown in Figure 8-18 on page 8-19, multiple wrapper cells are inserted for three-state and bidirectional ports. You can reference these wrapper cells in the ordered list of ports by appending /out, /en, or /in to the name of the bidirectional or three-state port. For example, the following command specifies an ordering for ports named a, b, c, and d, where a is an input port, b is a three-state port, and c and d are bidirectional ports.

```
dc_shell> set_scan_path Wchain0 -class wrapper \
            -ordered_elements [list A B/in B/out C/out \
            D/out B/en C/en D/en] -complete true \
            -test_mode test_mode_name
```

Note:

All of the wrapper cells for an individual three-state or bidirectional port must be in the same wrapper chain. For example, you cannot specify that the enable wrapper cell and output wrapper cell belong to different chains for a given port.

You can use the `-scan_data_in` and `-scan_data_out` options to specify the wrapper scan-in or wrapper scan-out signals for a chain:

```
dc_shell> set_dft_signal -type ScanInData -port wsi
dc_shell> set_dft_signal -type ScanDataOut -port wso
dc_shell> set_scan_path -class wrapper WC2 \
            -ordered_elements [list A B C] \
            -scan_data_in wsi -scan_data_out wso
```

The following commands implement separate input and output wrapper chains with separate wrapper shift signals using the `set_scan_path` command:

```
dc_shell> set_dft_signal -view spec \
            -type wrp_shift -port {wrp_ishift wrp_oshift}

dc_shell> set_scan_path -class wrapper WC_inputs \
            -input_wrapper_cells_only enable \
            -scan_enable wrp_ishift
```

```
dc_shell> set_scan_path -class wrapper WC_outputs \
            -output_wrapper_cells_only enable \
            -scan_enable wrp_oshift
```

A `set_scan_path` specification applied with the `-class wrapper` option and without the `-test_mode` option applies to all wrapper modes. You can also explicitly specify the `-test_mode all` option. To apply a `set_scan_path` specification to a specific wrapper mode, you must predefine the wrapper mode before referencing it. For more information, see "Creating User-Defined Wrapper Modes" on page 8-22.

For more information on the `set_scan_path` command, see the man page.

Creating User-Defined Wrapper Modes

DFT Compiler supports the definition of multiple user-defined wrapper modes. Use the `-usage` option of the `define_test_mode` command to specify the wrapper mode for each user-defined test mode:

```
dc_shell> define_test_mode MY_INTEST1 -usage wrp_if

dc_shell> define_test_mode MY_INTEST2 -usage wrp_if

dc_shell> define_test_mode MY_EXTEST1 -usage wrp_of

dc_shell> define_test_mode ...
```

You can then configure the characteristics of each wrapper mode by using the `-test_mode` option of test configuration commands such as the `set_scan_configuration` and `set_wrapper_configuration` commands. The `set_boundary_cell` command does not support the `-test_mode` option.

To apply a test specification command to a specific wrapper mode using the `-test_mode` option, you must define the wrapper mode before referencing the wrapper mode names:

```
dc_shell> define_test_mode wrp_if -usage wrp_if

dc_shell> define_test_mode wrp_of -usage wrp_of

dc_shell> set_wrapper_configuration -class core_wrapper \
            -chain_count 8 -test_mode wrp_if

dc_shell> set_wrapper_configuration -class core_wrapper \
            -max_length 40 -test_mode wrp_of
```

Note that only the following options of the `set_wrapper_configuration` command can be used with the `-test_mode` option:

- `-max_length`
- `-chain_count`

- -mix_cells
- -input_shift_enable
- -output_shift_enable
- -shift_enable
- -no_dedicated_wrapper_cells

For more information on defining test modes, see “Multiple Test Modes” on page 7-36.

DRC Rule Checks

You can perform DRC for the wrapper modes. To verify that the input wrapper chains shift in wrp_if (INTEST) mode, use the following commands:

```
dc_shell> current_test_mode wrp_if

dc_shell> dft_drc
```

To verify that the output wrapper chains shift in wrp_of (EXTEST) mode, run the following commands:

```
dc_shell> current_test_mode wrp_of

dc_shell> dft_drc
```

Multiple Voltage Domains

By default, dedicated wrapper cells are added to the top-level voltage domain. To have dedicated wrapper cells added to voltage domains other than the top-level voltage domain, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -add_wrapper_cells_to_power_domains enable
```

When this option is enabled, no new wrapper chain hierarchical block is created to enclose the dedicated wrapper cells. Wrapper cells are added according to the following rules:

- If the port is connected to an I/O register, the dedicated wrapper cell is added to the top-level power-domain hierarchy of the I/O register.
- If the port is connected to a CTL model, the top-level power-domain hierarchy of the instantiated CTL model is used as the location for the dedicated wrapper cell.
- If the port is connected to a gate, the top-level power-domain hierarchy of the gate is used as the location for the dedicated wrapper cell.
- If none of these conditions apply, the cell is added to the top-level hierarchy.

SCANDEF Generation

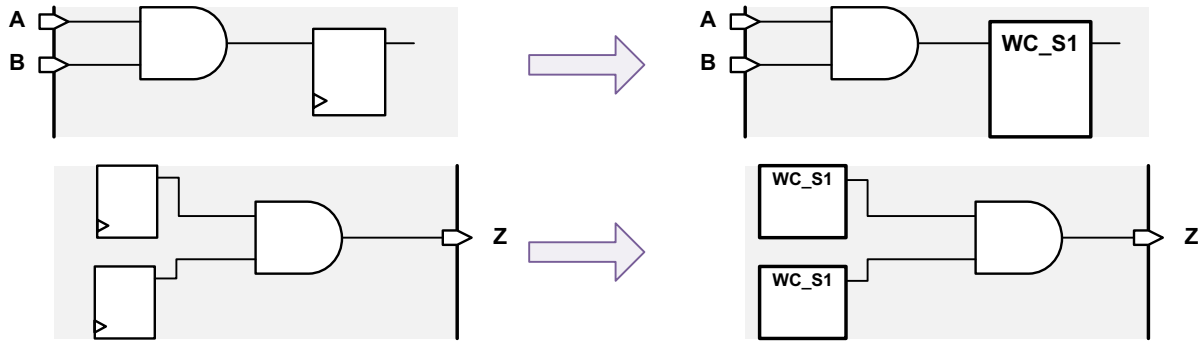
SCANDEF generation is supported for input and output wrapper chains. Cells from input wrapper chains are not allowed to be mixed with cells from other types of chains. Similarly, cells from output wrapper chains are not allowed to be mixed with cells from other types of chains.

Maximized Reuse Core Wrapping Flow

The simple core wrapping flow can add a number of dedicated wrapper cells when the design ports are not directly connected to registers through simple buffering or inverting logic. To reduce the number of added cells, the DFTMAX tool provides a *maximized reuse* mode that can share existing functional registers that are accessible from the top-level ports through combinational logic.

Figure 8-19 shows design registers, separated from their input or output ports by combinational logic, that are replaced by shared wrapper cells when maximized reuse is enabled.

Figure 8-19 Maximized Reuse Examples



Note:

The diagrams in this section show WC_S1 shared wrapper cell instances for clarity. However, maximized reuse uses the in-place register implementation to ensure that the names and locations of the wrapped functional registers are not disturbed.

Using existing functional registers as shared wrapper cells can reduce the area requirements for core wrapping. However, the potential cost of using a shared wrapper cell implementation is core testability; the combinational logic between the shared wrapper cell and the ports is effectively placed outside the block as far as core wrapping logic is concerned. This logic must be tested using the EXTEST wrapper mode that exercises the surrounding logic.

The maximized reuse feature considers how many registers exist in the fanout from an input port or the fanin to an output port. A single port might have many registers in its fanin or fanout, which would require many shared wrapper cells to fully wrap the port. If the number

of fanin or fanout registers for a port exceeds a *reuse threshold* value, a single dedicated wrapper cell is used for that port.

Note:
A DFTMAX license is required to use the maximized reuse feature.

To enable the maximized reuse feature, use the following command:

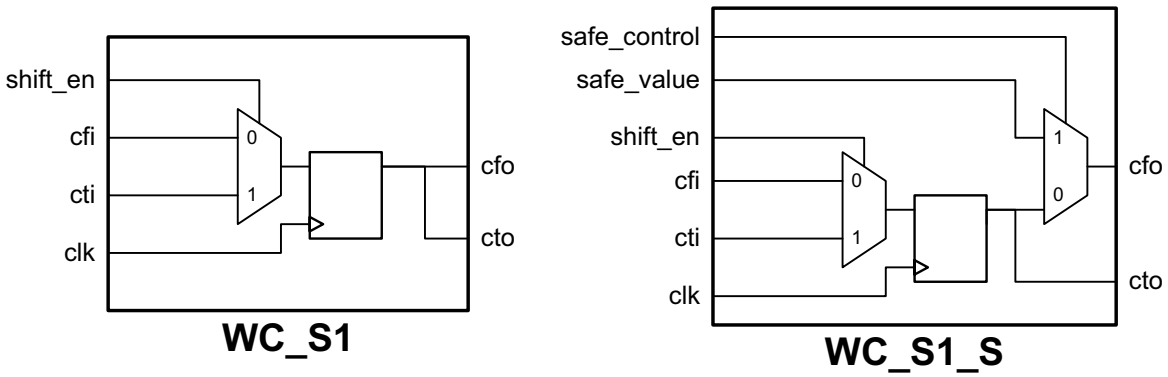
```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable -reuse_threshold N
```

where the value of *N* defines the reuse threshold. The default of *N* is 1.

With the `-maximize_reuse` option enabled, when the number of registers encountered from an input or to an output exceeds the reuse threshold, a dedicated wrapper cell is added to the port. If the number of registers is less than this threshold, the tool can use these registers for wrapping the core. If a reuse threshold value of zero is specified, the tool converts all I/O registers to shared wrapper cells. No dedicated wrapper cells are used unless specified by the user. For more information on how the reuse threshold is computed, see "Computing Reuse Threshold Values" on page 8-26.

The maximized reuse flow might result in a large number of shared wrapper cells, depending on the number of combinationally connected I/O registers and the specified reuse threshold value. To minimize area overhead, the maximized reuse flow uses an optimized shared wrapper cell design without a feedback loop, as shown in Figure 8-20. Most of this wrapper cell functionality is implemented by the internal data multiplexer in a scan-equivalent flip-flop. The tool creates the wrapper test protocols to use the optimized wrapper cell structures.

Figure 8-20 WC_S1 and WC_S1_S Wrapper Cells in Maximized Reuse Flow



When the `-maximize_reuse` option is enabled, the tool sets the following options automatically:

- `-style shared`
- `-register_io_implementation in_place`
- `-mix_cells false`

Also, you should not use the following options of the `set_wrapper_configuration` command with the `-maximize_reuse` option:

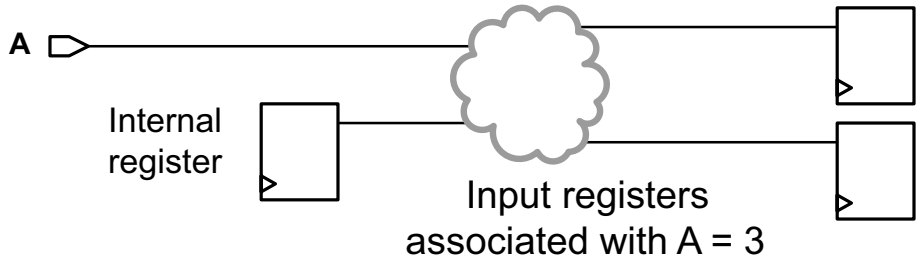
- `-delay_test`
- `-core`
- `-shared_cell_type`
- `-shared_design_name`
- `-use_dedicated_wrapper_clock`

Computing Reuse Threshold Values

When applying the reuse threshold value to a port, the tool determines the number of registers associated with that port. If the number of registers for a port does not exceed the reuse threshold, the registers are replaced with shared wrapper cells. If the number of registers exceeds the reuse threshold, a dedicated wrapper cell is placed at the port.

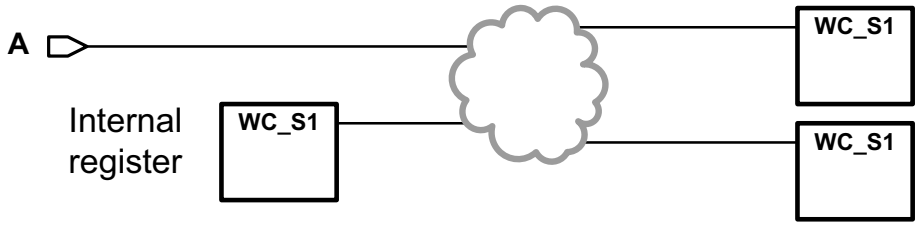
For an input port, the tool determines the number of registers in the fanout of the port. In addition, it includes any internal registers that feed data pins or synchronous set/reset pins of the fanout registers. Figure 8-21 shows an input port with an associated register count of three.

Figure 8-21 Input Port Register Count Computation Example



In this example, if the reuse threshold is set to a value of three or higher, the tool replaces the registers with shared wrapper cells, as shown in Figure 8-22.

Figure 8-22 Input Port Registers After Wrapper Cell Replacement



The tool places the shared wrapper cells for the fanout registers in the input wrapper chain. Because any internal registers feeding these fanout registers capture values from the core

logic instead of input ports, the tool places the shared wrapper cells for these internal registers in the output wrapper chain.

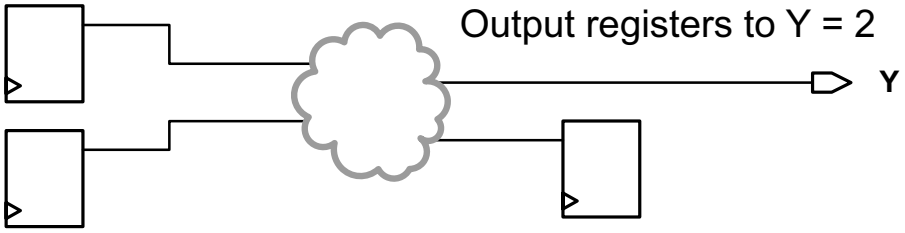
In the maximized reuse flow, dedicated wrapper cells are added to input ports according to the following rules:

- If the sum of the input fanout registers and the internal registers feeding them exceeds the reuse threshold, then a dedicated wrapper cell is added to the input port.
- When the input register cells exceed the reuse threshold for a bidirectional port, dedicated wrapper cells are added to the data-out, enable, and data-in paths.
- If an input port fans out to a CTL model and no netlist exists for that model, a dedicated wrapper cell is added to the port.
- If an input port fans out to a CTL model, a netlist exists for that model, and the pin of the CTL model connects internally to a register, a dedicated wrapper cell is added to the port.

A warning is issued if all of the registers associated with an input port do not use the same clock.

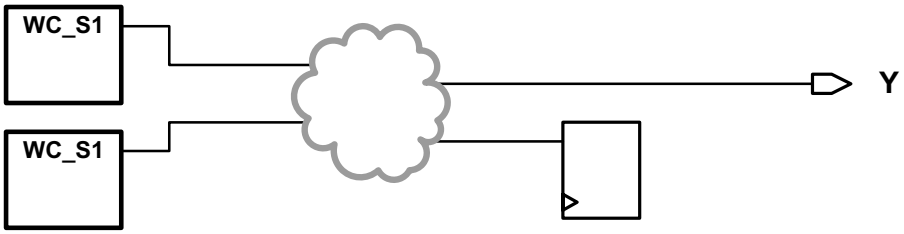
For an output port, the tool determines the number of registers in the fanin of the port. It does not include any additional surrounding registers. Figure 8-23 shows an output port with an associated register count of two.

Figure 8-23 Output Port Register Count Computation Example



In this example, if the reuse threshold is set to a value of two or higher, the tool replaces the registers with shared wrapper cells, as shown in Figure 8-24.

Figure 8-24 Output Port Registers After Wrapper Cell Replacement



The tool places the shared wrapper cells for the fanin registers in the output wrapper chain.

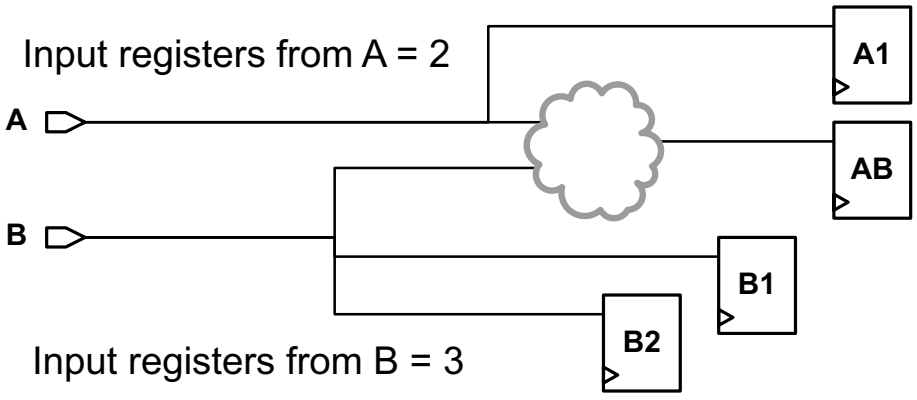
In the maximized reuse flow, dedicated wrapper cells are added to output ports according to the following rules:

- If the sum of the output fanin registers exceeds the reuse threshold, then a dedicated wrapper cell is added to the output port.
- When the number of the output register cells exceeds the reuse threshold value for a three-state output port, dedicated wrapper cells are added to both data-out and enable paths.
- If a port is driven by a CTL register and no netlist exists for the CTL model, a dedicated wrapper cell is added to the port.
- If a port is driven by a CTL register when a netlist exists for the CTL model and if the pin of the CTL model connects to a register inside the CTL model, then a dedicated wrapper cell is added to the port.
- For three-state and bidirectional ports, only a functional register that directly controls the pad is considered as a shared wrapper cell for the enable path of the pad. There can be inverting or noninverting buffers between the register and the pad enable pin. If no such register is found, a dedicated wrapper cell is added at the driver cell enable pin to control the port.

A warning is issued if all of the registers associated with an output port do not use the same clock.

If the same register is in the fanin or fanout of multiple ports, the register is replaced with a shared wrapper cell if any port meets the threshold. In Figure 8-25, input port A has two associated registers, and input port B has three associated registers. Register AB is in the fanout of both ports.

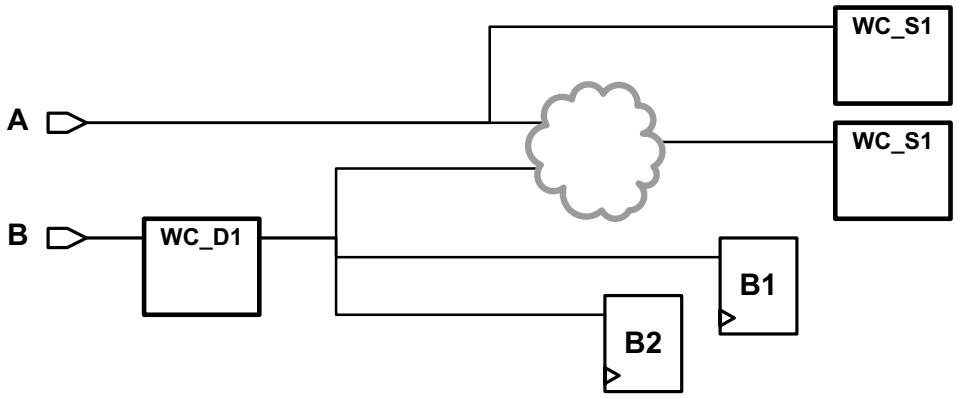
Figure 8-25 Register in Fanout of Multiple Ports



In this example, if the reuse threshold is set to a value of two, the tool replaces the registers for input port A with shared wrapper cells, and inserts a dedicated wrapper cell at input port

B, as shown in Figure 8-26. Register AB is replaced with a shared wrapper cell even though it is also in the fanout of the dedicated wrapper cell for input port B.

Figure 8-26 Register in Fanout of Multiple Ports After Wrapper Cell Replacement

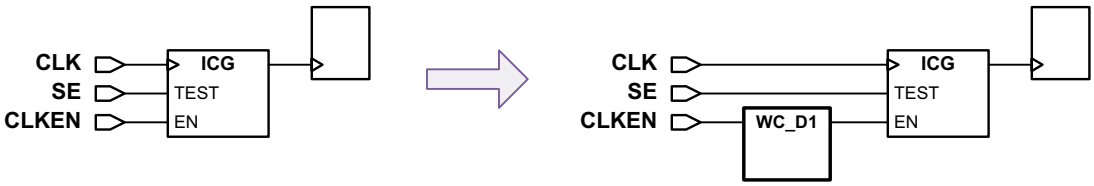


Registers that become shared wrapper cells for one port do not affect the associated register count for other ports. In Figure 8-26, the associated register count for input port B is three even though register AB is wrapped for input port A.

If a register is classified as both an input shared register and an output shared register, the register becomes an input shared register and is removed from the output shared register list. Unlike the simple core-wrapping flow, no dedicated wrapper cell is placed at the output port.

A dedicated wrapper cell is added to an input port that drives the enable signal of a clock-gating cell. In Figure 8-27, an integrated clock-gating cell has both a functional enable pin and a test-mode pin. A dedicated wrapper cell is added for the functional enable signal driven by input port CLKEN. No wrapper cells are inserted for the clock signal or the global test-mode signal.

Figure 8-27 Integrated Clock Gating Cell Enable Signals

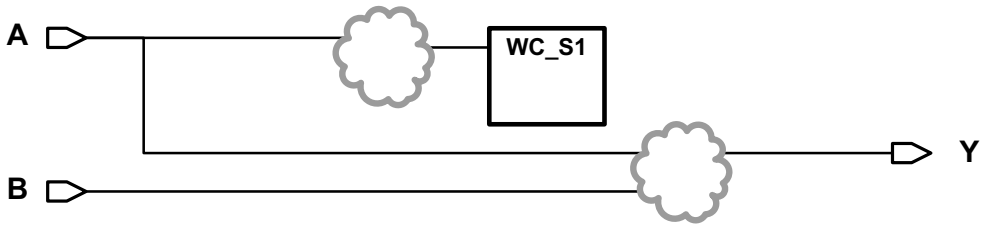


A warning message is issued if such a clock-gating enable signal is detected:

```
Warning: Port 'CLKEN' is connected to a clock gating cell 'UICG';
  a dedicated wrapper cell is added to the port. (TEST-1183)
```

No wrapper cell is added to a port that is bounded only by combinational logic unless you explicitly specify a dedicated wrapper cell for that port. See Figure 8-28.

Figure 8-28 Combinational Feedthrough Paths



An information message is issued if a feedthrough port is detected and no dedicated wrapper cell is manually specified for the port:

```
Information: No I/O registers are found for port 'B';
  not adding any dedicated wrapper cells to the port. (TEST-1180)
Information: No I/O registers are found for port 'Y';
  not adding any dedicated wrapper cells to the port. (TEST-1180)
```

Overriding the Reuse Threshold

The reuse threshold can be set on a port-by-port basis with the `set_boundary_cell` command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -reuse_threshold N -ports {port_list}
```

This command sets the reuse threshold value to *N* (where *N* >= 0) for all ports in the *port_list* specification.

To ignore the reuse threshold and force the use of a dedicated wrapper cell for one or more ports, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_D1
```

If a port exceeds the reuse threshold and you want to manually specify some port-associated registers as shared wrapper cells, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -include {IO_register_cell_list} -ports {port_name}
```

This command causes the cells listed in *IO_register_cell_list* associated with the port *port_name* to be used as shared wrapper cells.

To manually specify some registers as shared wrapper cells for all associated ports that exceed the reuse threshold, omit the `-ports` option:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -include {IO_register_cell_list}
```

When using the `-include` option with the `-ports` option, any specified ports exceeding the reuse threshold do not get a dedicated wrapper cell. When using the `-include` option

without the -ports option, all ports associated with the specified registers exceeding the reuse threshold do not get a dedicated wrapper cell. You can exceed the reuse threshold when manually specifying shared wrapper cell registers with this option.

If a port meets the reuse threshold and you want to manually exclude some port-associated registers as shared wrapper cells, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper
            -exclude {IO_register_cell_names} -ports {port_name}
```

This command causes the registers listed in *IO_register_cell_names* to be excluded for the port *port_name*. If a register is excluded for the specified port but qualifies to be wrapped for another port, the register is still wrapped.

To manually exclude some registers as shared wrapper cells for all associated ports that meet the reuse threshold, omit the -ports option:

```
dc_shell> set_boundary_cell -class core_wrapper
            -exclude {threshold_value}
```

Note:

The -include and -exclude options of the set_boundary_cell command can cause a port to become partially wrapped.

Applying a Combinational Depth Threshold

For a core-wrapped design, any logic that exists between the wrapper chain and the I/O ports becomes logic that is external to the block for testing purposes. The -reuse_threshold option of the set_wrapper_configuration command applies a maximum fanout-breadth or fanin-breadth threshold limit for input and output shared wrapper cells, respectively. However, this reuse threshold only indirectly limits the logic depth that can exist outside the wrapper chain.

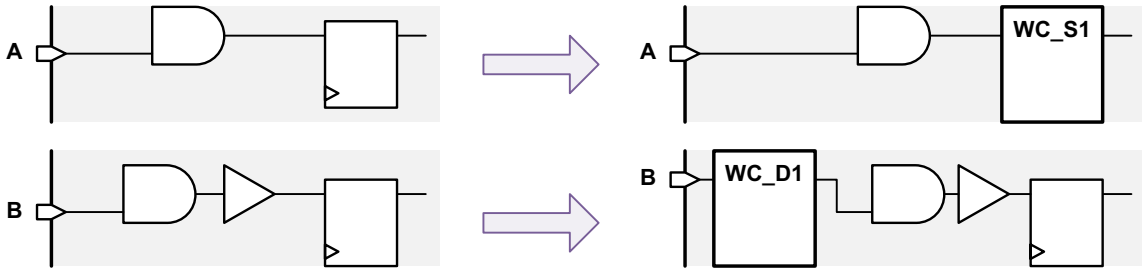
You can use the -depth_threshold option of the set_wrapper_configuration command to directly specify the maximum number of combinational cells, including buffers and inverters, that can exist between a port and its associated registers. For example,

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -depth_threshold 2
```

If this depth is exceeded, a dedicated wrapper cell is used for that port. This can be used to prevent too much logic from being placed on the external side of the wrapper chain.

Figure 8-29 shows the wrapper cell insertion behavior when the -depth_threshold value is set to 1.

Figure 8-29 Core Wrapping With a Combinational Depth Threshold Value of 1



Using Dedicated Wrapper Cells

Dedicated wrapper cells are used for any ports that exceed the reuse or combinational depth thresholds. You can also manually force the use of a dedicated wrapper cell with the following command:

```
dc_shell> set_boundary_cell -class core_wrapper \
            -ports port_list -type WC_D1
```

In the simple core-wrapping flow, dedicated wrapper cells use the dedicated wrapper clock. In the maximized reuse flow, the tool attempts to identify the dominant clock domain associated with the port and uses that clock for the dedicated wrapper cell.

The following rules guide the automatic selection of a clock for dedicated wrapper cells in the maximized reuse flow:

- A port's dedicated wrapper cell uses the same clock as the flip-flops associated with that port.
- If the port is associated with flip-flops of multiple clock domains, the dominant clock is used.
- If a dominant clock is not found, any user-specified wrapper clock, defined using the `set_dft_signal -type wrp_clock` command, is used.
- If no user-specified wrapper clock has been defined, a dedicated wrapper clock is created and used.

To specify that a particular clock, *`clock_name`*, is to be used for a specific dedicated wrapper cell on a port, *`port_name`*, use the following command:

```
dc_shell> set_boundary_cell -class core_wrapper -type WC_D1 \
            -shift_clk clock_name -ports port_name
```

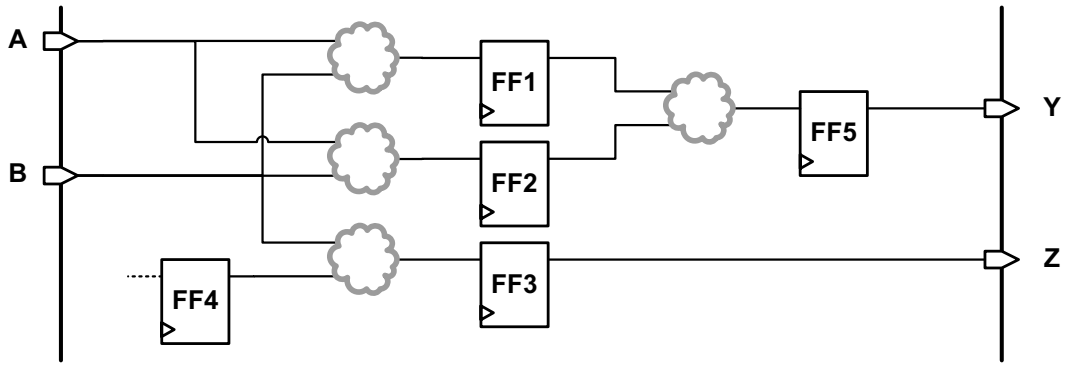
This specification overrides the automatic clock selection process. The specified clock can belong to any part of the design hierarchy.

Previewing Shared Wrapper Cells

In the maximized reuse flow, the `preview_dft` command shows additional information about how register reuse is applied.

The preview report uses annotations to indicate when a shared wrapper cell is associated with multiple ports. Figure 8-30 shows a design to be core-wrapped in the maximized reuse flow, and Example 8-2 shows the corresponding wrapper chain report from the `preview_dft` command.

Figure 8-30 Design Example With Common Shared Wrapper Cells



Example 8-2 Wrapper Chain Preview Report with Common Shared Wrapper Cells

```
dc_shell> preview_dft -test_wrappers all
...
                Wrapper    Wrapper     Control     Safe    Wrapper
Index   Port    Function   Cell Type   Cell Impl   Value   Clock     Cell Name
-----   ----    --------   ---------   ---- ----   -----   -------   ---------
 4      A       input      WC_S1        -   INP     -      CLK       FF2_reg
 4      B       input      WC_S1            INP     -      CLK       FF2_reg(d)
 3      A       input      WC_S1        -   INP     -      CLK       FF1_reg
 3      B       input      WC_S1            INP     -      CLK       FF1_reg(d)
 2      B       input      WC_S1        -   INP     -      CLK       FF3_reg
 2      Z       output     WC_S1            INP     -      CLK       FF3_reg(*)
 1      B       output     WC_S1        -   INP     -      CLK       FF4_reg
 0      Y       output     WC_S1        -   INP     -      CLK       FF5_reg
```

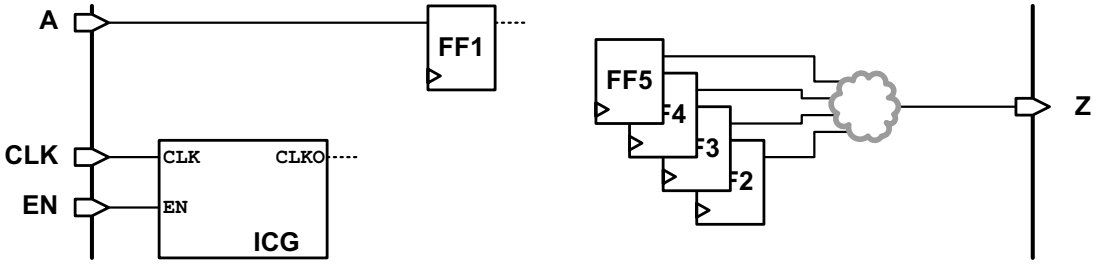
Shared registers that are common to multiple input ports and multiple output ports are shown with the letter d (d) for all subsequent entries after the first register entry; these marked entries do not count against the total wrapped register count. Shared registers that are common to both an input port and an output port but are used only as input wrapper cells are shown with an asterisk (*) for the output port.

Note the following:

- FF1 and FF2 have duplicate entries because they are listed twice, one time for input port A and one time for input port B. The duplicate entries are marked with the letter d (d).
- FF3 acts as both an input wrapper cell and as an output wrapper cell. After wrapping, it is placed in the input wrapper chain; the output entry is marked as a duplicate entry with an asterisk (*).

The preview report also indicates the reason why a dedicated wrapper cell is used at a port instead of a shared wrapper cell. Figure 8-31 shows a design to be core-wrapped in the maximized reuse flow, and Example 8-3 shows the corresponding wrapper chain report from the `preview_dft` command.

Figure 8-31 Design Example With Dedicated Wrapper Cells



Example 8-3 Wrapper Chain Preview Report with Dedicated Wrapper Cells

```
dc_shell> preview_dft -test_wrappers all
...
Note: Dedicated wrapper cells are grouped into a hierarchical instance named:
"top_Wrapper_inst" (Module name: "top_Wrapper_inst_design")

Dedicated Wrapper
Cell Reason          Description
-----------------    -----------
TEST-1183            Port drives enable pin of clock-gating cell
TEST-1185            Number of I/O registers exceeds reuse threshold

Wrapper Length:       3

               Wrapper   Wrapper
Index   Port   Function  Cell Type  ...  Cell Name
-----   ----   --------  ---------       ---------
 2      A      input     WC_S1           FF1_reg
 1      EN     input     WC_D1           top_Wrapper_inst/top_EN_wrp0_1 (TEST-1183)
 0      Z      output    WC_D1           top_Wrapper_inst/top_Z_wrp0_0 (TEST-1185)
```

In the wrapper chain report, each port with a dedicated wrapper cell has a reason message code indicating why the dedicated wrapper cell is used. A short description for each reason is shown in a legend table before the report. For more information on a dedicated wrapper cell reason, see the man page. For a complete list of dedicated wrapper cell reasons, see SolvNet article 038531, “What Are the Reasons for Dedicated Wrapper Cells to Be Inserted?”

Wrapper Signal Behavior

Table 8-2 describes the shift and capture behavior of the wrapper cells for the various test modes. In the maximized reuse flow, dedicated wrapper cells use capture signals, but shared wrapper cells have no capture signal.

Table 8-2 Test-Mode Behavior of Wrapper Cells in the Maximized Reuse Flow

	Shift (inputs)	Shift (outputs)	Capture (inputs)	Capture (outputs)	Safe control (inputs)	Safe control (outputs)
Mission mode	S[a]	S	0	0	0	0
Standard scan	S	S	0	0	0	0
Compressed scan	1	S	1	0	0	1
wrp_if	1	S	1	0	0	1
wrp_of	S	1	0	1	1	0
wrp_safe	S	S	0	0	1	1

a. The default name for this wrapper signal is wrp_shift.

The test modes operate as follows:

- Standard scan modes – Only core internal chains are active; the wrapper chain is transparent.
- Compressed scan modes – Both wrapper chains and core internal chains are active; they are all compressed by the scan compression codec.
- wrp_if – Both wrapper chains and core internal chains are active.
- wrp_of – Only wrapper chains are active; core internal chains are loaded with 0.
- wrp_safe – The safe values are driven from the core inputs into the core, and from the core outputs into the surrounding logic.

Compressed scan modes, defined with the `-usage scan_compression` option, configure the wrapper chain cells to operate in an inward-facing mode, just as the wrp_if mode does.

Standard scan modes, defined with the `-usage scan` option, configure dedicated wrapper cells to be transparent and shared wrapper cells to be in functional mode. These modes are not used for core-level testing; define and configure them only if core-level scan segments are needed by a top-level test mode.

Using the Pipelined Scan-Enable Feature

The pipelined scan-enable feature can be used in the maximized reuse flow. By default, two signals are used to enable shifts for designs using wrapper cells:

- test_se – Pipelined, used for internal flip-flops
- wrp_shift – Pipelined, used for wrapper flip-flops (shared or dedicated)

When the pipelined scan-enable feature is enabled, pipelined scan-enable structures are added to both of these signals. A pipelined scan-enable cell is used for each clock domain used by the input and output wrapper cells, with pipeline registers clocked by that clock. Pipelined scan-enable cells are not shared between input and output wrapper cells.

For example, if the design has two clock domains driving input wrapper cells and three clock domains driving output wrapper cells and if the number of wrapper cells within each of these domains meets the required pipeline fanout limit, a total of five pipelined scan-enable cells are created. Two of the pipelined scan-enable cells drive input wrapper cells; the remaining three pipelined scan-enable cells drive output wrapper cells. To create this pipelined structure, use the following commands:

```
dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable

dc_shell> set_scan_configuration \
            -pipeline_scan_enable true -pipeline_fanout_limit P
```

To use the same scan-enable for all registers, define the signal as both a scan-enable signal and a wrapper shift signal using the `set_dft_signal` command:

```
dc_shell> set_dft_signal -view spec -type ScanEnable -port SE

dc_shell> set_dft_signal -view spec -type wrp_shift -port SE

dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable

dc_shell> set_scan_configuration \
            -pipeline_scan_enable true -pipeline_fanout_limit P
```

In this case, pipelined scan-enable cells are not shared between internal cells, input wrapper cells, or output wrapper cells.

To use separate scan-enables for input and output wrapper chains, use the following commands:

```
dc_shell> set_dft_signal -view spec -type ScanEnable -port SE
```

```
dc_shell> set_dft_signal -view spec -type wrp_shift -port WSE_I

dc_shell> set_dft_signal -view spec -type wrp_shift -port WSE_O

dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable \
            -input_shift_enable WSE_I -output_shift_enable WSE_O

dc_shell> set_scan_configuration \
            -pipeline_scan_enable true -pipeline_fanout_limit P
```

Note:

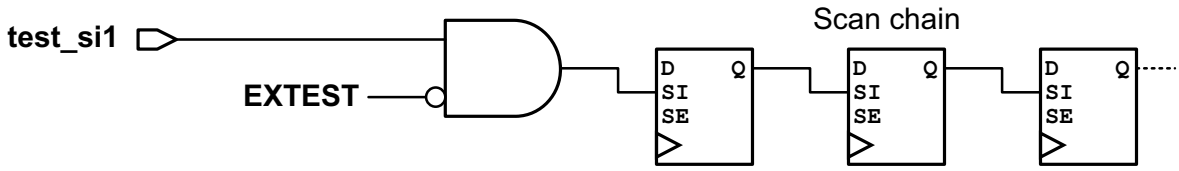
A single global_pipe_se signal is used for both internal and wrapper chains for all of these scenarios.

For more information on the pipelined scan-enable feature, see “Pipelined Scan-Enable Architecture” on page 7-34.

Low-Power Features

To minimize power consumption in wrp_of (EXTEST) mode, all core scan chain cells are loaded with logic 0. This reduces toggle activity inside the block as clock pulses are issued to test external logic. To accomplish this, the logic shown in Figure 8-32 drives the first scan cell inputs of the internal scan chains.

Figure 8-32 Scan Data Gating Logic For Low-Power EXTEST

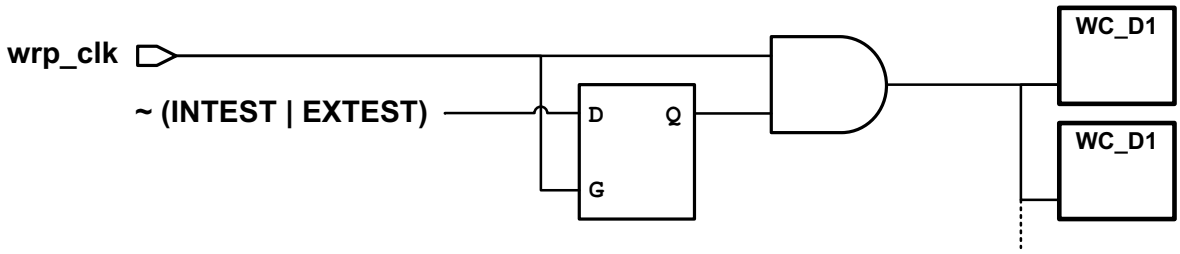


To add low-power clock gating to wrapper cells clocked by the dedicated wrapper clock, use the following command:

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -gate_dedicated_wrapper_cell_clk enable
```

The tool adds a clock-gating cell to the dedicated wrapper cell clock source so that the clock is off when the wrapper cell is not in use, as shown in Figure 8-33. Shared wrapper cells clocked by their preexisting functional clock signals are not affected.

Figure 8-33 Clock Gating Logic For Low-Power Dedicated Clock Operation



Low-power clock-gating logic for dedicated wrapper clocks uses discrete gating logic.

Hierarchical Core Wrapping

The hierarchical core wrapping feature can be used to build a core that contains only wrapper and core chains, but does not contain any test control module (TCM) or wrapper mode logic. This core can then be used at a higher level of integration with other wrapper chains or compression architectures. This hierarchical wrapping feature is only available in the maximized reuse flow.

To enable hierarchical wrapping, use the following commands:

```
dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable -hier_wrapping enable
```

By default, hierarchical wrapping is disabled.

When hierarchical wrapping is enabled, a single mode, Internal_scan, is created. This mode creates both wrapper chains and core internal chains. In addition, the following core-level input ports and signals are created:

- Three separate shift signals are created to control input wrapper chains, output wrapper chains, and core internal chains.
- If safe states are specified, separate input and output safe control signals are created for input and output wrapper cells.
- In the maximized reuse flow, shared wrapper cells do not use a capture signal. However, if there are dedicated wrapper cells in the input and output wrapper chains, separate input and output capture-enable signals are created for input and output wrapper cells.

By default, the tool creates these signals using default port and signal names. To use existing placeholder ports for these signals instead, define them with the `set_dft_signal` command by specifying the following signal types with the `-type` option:

- `wrp_shift`
- `wrp_ded_capture_in`

- wrp_ded_capture_out
- wrp_safe_in
- wrp_safe_out

The test model generated after hierarchical wrapping includes attributes that identify input and output wrapper chains, input and output wrapper scan-enable signals, input- and output-dedicated safe control signals, and input- and output-dedicated wrapper capture-enable ports.

Note:

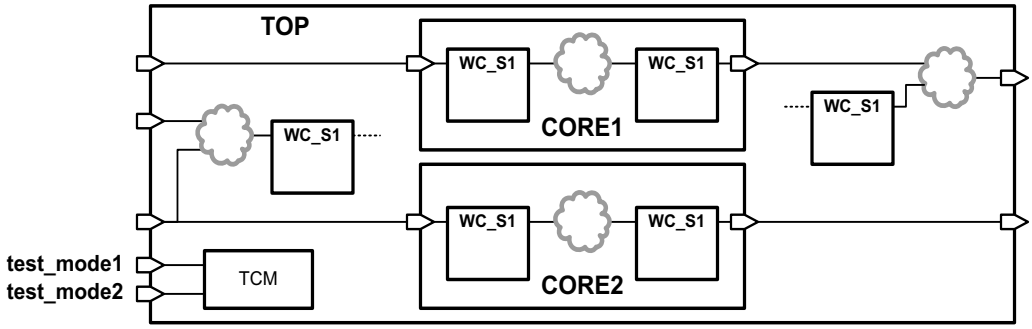
Multiple test modes are not allowed at the core level when using hierarchical wrapping to create the core.

After the core is created, to enable integration of hierarchically wrapped cores at a higher level of the hierarchy, use the following commands:

```
dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -maximize_reuse enable
```

During integration, the tool detects any hierarchical wrapped cores using the previously stored test attributes. Wrapper cells are incrementally added as needed to augment core-level wrapper capabilities, as shown in Figure 8-34. The tool creates the normal set of wrapper test modes, selected by a test control module, according to user specifications. As with typical wrapping flows, multiple test modes, including scan compression, are allowed at this level.

Figure 8-34 Integrating Wrapper-Only Cores at a Higher Level



Core-level input wrapper chains are mixed with integration-level input wrapper cells, and core-level output wrapper chains are mixed with integration-level output wrapper cells. Core-level wrapper shift, capture, and safe control signals are connected to appropriate integration-level wrapper logic.

Core-level wrapper chains can be integrated only one time. There is no support for recursive wrapping flows that propagate wrapper capabilities up through multiple hierarchy levels.

Limitations of the Maximized Reuse Flow

Note the following limitations related to the behavior of the scan-enable signal in capture mode:

- Preexisting scan-enable control of clock-gating cells

  If the design has I/O registers controlled by clock-gating cells that are controlled by the scan-enable port, these I/O registers might not shift during the capture mode operation of the INTEST and EXTEST modes because scan-enable might not be active in capture mode.

- Preexisting scan-enable control of set and reset

  If the set or reset pins of the I/O registers are controlled by the scan-enable port, the I/O registers might not shift during the capture mode of operation of the INTEST and EXTEST modes because scan-enable might not be active in capture mode.

Delay Test Core Wrapping Flow

You can use the delay test core wrapping flow to insert a wrapper chain capable of applying at-speed ATPG vectors to the wrapped core. To enable this flow, use the `-delay_test` option of the `set_wrapper_configuration` command:

```
dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration -class core_wrapper \
            -delay_test true
```

The delay test core wrapping flow is a variant of the simple core wrapping flow that uses the same dedicated and shared wrapper cells, but it builds a different test controller. This test controller configures the wrapper chains into the various modes required to perform at-speed testing of the wrapped core and top-level logic surrounding the core using transition patterns. Input and output wrapper cells are in the same chain, but they have separate input and output shift and capture controls.

Note:

The delay test core wrapping is a legacy flow that is documented for completeness. For best results, use the maximized reuse flow instead.

Wrapper Signal Behavior

In a delay-test wrapper insertion flow, two top-level control signals are created. These signals are used as the shift and capture signals for the input and output wrapper cells in the various test modes that are created. See Figure 8-35.

Figure 8-35 Decoding Logic for Delay Test Shift and Capture Signals

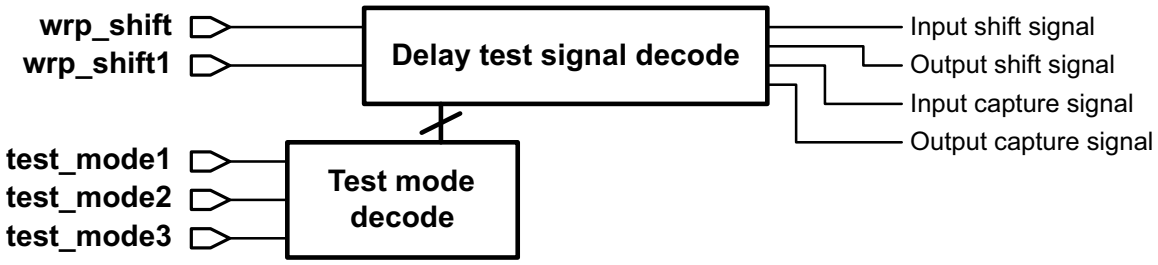


Table 8-3 describes the shift and capture behavior of the wrapper cells for the various test modes.

Table 8-3 Test-Mode Behavior of Wrapper Cells in the Delay Test Core Wrapping Flow

	Shift (inputs)	Shift (outputs)	Capture (inputs)	Capture (outputs)	Safe control (inputs)	Safe control (outputs)
Mission mode	0	0	0	0	0	0
Standard scan	0	0	0	0	0	0
Compressed scan	S1[a]	S1	~S1	0	0	1
wrp_if	S1	S1	~S1	0	0	1
wrp_if_delay	S1	S2[b]	~S1	0	0	1
wrp_if_scl_delay	S1	S2	~S1	0	0	1
wrp_of	S1	S1	0	~S1	1	0
wrp_of_delay	S1	S2	0	~S2	1	0
wrp_of_scl_delay	S1	S2	0	~S2	1	0
wrp_safe	0	0	0	0	1	1

a. The default name for this wrapper signal is wrp_shift.

b. The default name for this wrapper signal is wrp_shift1.

The behaviors used in the INTEST modes (wrp_if*) are:

- Input chains shift in shift cycles and shift in capture cycles.
- Output chains shift in shift cycles and capture in capture cycles.
- Separate shift enable signals are used for input, output, and core chains.
- Modes are used for launch-on-last-shift transition ATPG for the wrapped core.
- Any specified safe values are driven at the core outputs.

The behaviors used in the EXTEST modes (wrp_of*) are:

- Wrapper chains are configured in EXTEST.
- Input chains shift in shift cycles and capture in capture cycles.
- Output chains shift in shift cycles and shift in capture cycles.
- Separate shift enable signals are used for input, output, and core chains.
- Modes are used for launch-on-last-shift transition ATPG of the wrapped core.
- Any specified safe values are driven at core inputs.

Creating User-Defined Core Wrapping Test Modes

Normally, when core wrapping is enabled, DFT Compiler creates a default set of core wrapping test modes as described in "Core Wrapping Test Modes" on page 8-10. You can also create user-defined test modes for wrapped cores using the `define_test_mode` command.

Table 8-4 shows the test mode usage types you can specify with the `-usage` option of the `define_test_mode` command. You can define one or more test modes for each usage. When you define a test mode for a given usage, the default test mode is not created for that usage.

Table 8-4 Test Mode Usage Types for Wrapped Cores

Test mode usage	Test mode description	Default test mode name
`wrp_if`	Inward-facing uncompressed scan mode	wrp_if
`scan_compression`	Inward-facing compressed scan mode	ScanCompression_mode
`wrp_of`	Outward-facing uncompressed scan mode	wrp_of
`wrp_safe`	Safe mode	wrp_safe

Table 8-4 Test Mode Usage Types for Wrapped Cores (Continued)

Test mode usage	Test mode description	Default test mode name
`scan`	Unwrapped standard scan mode	Internal_scan[a]

a. This default unwrapped test mode is not created when core wrapping is enabled; it can only be created for a wrapped core by defining a user-defined test mode.

You can configure individual user-defined test modes using the `-test_mode` option of DFT configuration commands. The `-test_mode` option cannot reference a default test mode unless that mode name was explicitly defined with the `define_test_mode` command.

Example 8-4 defines a set of user-defined core wrapping test modes. The inward-facing compressed scan mode, defined with `-usage scan_compression`, specifies the base mode as the inward-facing uncompressed scan mode, defined with `-usage wrp_if`.

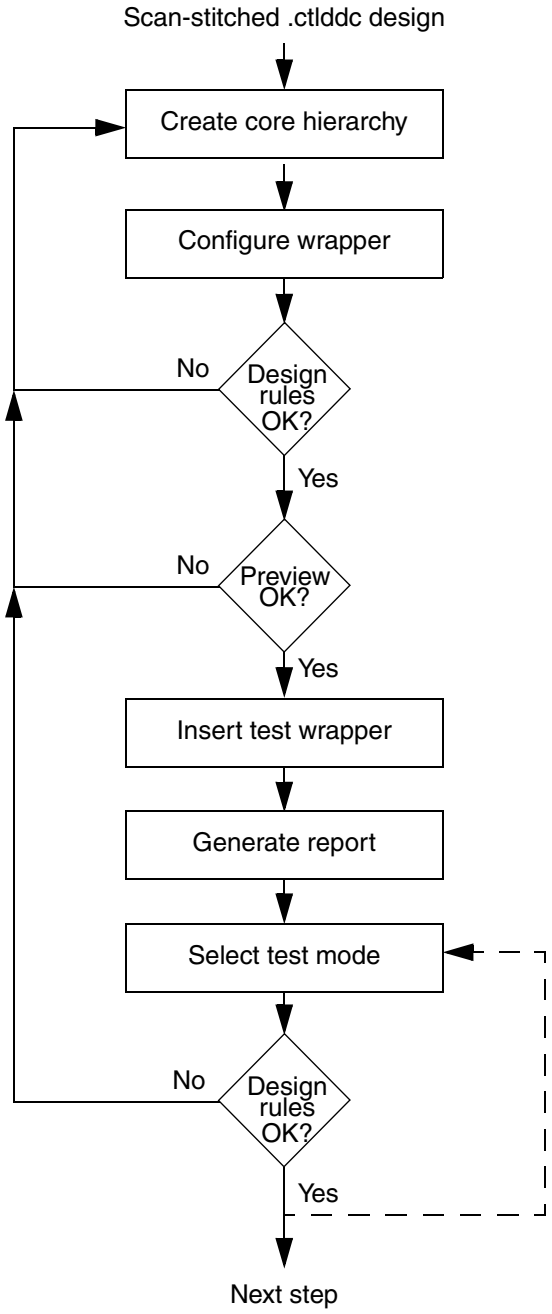
Example 8-4 User-Defined Core Wrapping Test Modes

```
# define test modes
define_test_mode MY_INTEST      -usage wrp_if
define_test_mode MY_INTEST_COMP -usage scan_compression
define_test_mode MY_EXTEST      -usage wrp_of

# configure scan modes
set_wrapper_configuration  -test_mode MY_INTEST \
  -class core_wrapper -chain_count 1 -mix_cells true
set_scan_configuration     -test_mode MY_INTEST \
  -chain_count 2 -clock_mixing mix_clocks \

set_wrapper_configuration  -test_mode MY_EXTEST \
  -class core_wrapper -chain_count 2 -mix_cells false
set_scan_configuration     -test_mode MY_EXTEST \
  -chain_count 2 -clock_mixing mix_clocks

set_scan_compression_configuration \
  -test_mode MY_INTEST_COMP -base_mode MY_INTEST \
  -chain_count 12
```

Wrapping Cores With Existing Scan Chains

To wrap a core that is already scan-stitched, you should use the scan-stitched core flow. Figure 8-36 illustrates this flow.

Figure 8-36 Scan-Stitched Core Flow Diagram



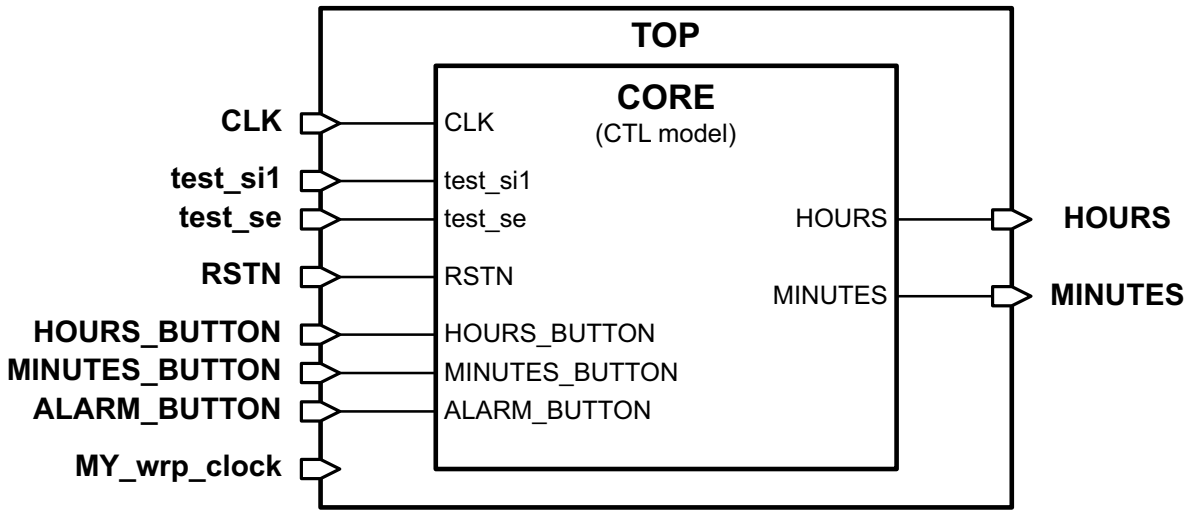
Start the flow with a CTL test model of the core. If you use a test model, DFT Compiler cannot touch the scan logic in the model during wrapping.

1. Create an enclosing top-level hierarchy that instantiates the core.

Figure 8-37 shows the hierarchy you should create. You must create an empty top-level design with the same input and output pins as the core design, instantiate the test model within the top-level design, and connect all of the input and output pins. You can also include unconnected placeholder ports for wrapper signals in the top-level design.

You can use a text editor or an automated script to create a netlist file that accomplishes this step.

Figure 8-37 Creating Core Hierarchy



2. Configure the core wrapper.

   Use the `set_dft_configuration -wrapper enable` command to enable core wrapping. If you do not issue this command, the other core wrapping commands will not have any effect. To set the configuration for wrapping, use the `set_wrapper_configuration` command. You can override the wrapper configuration on any ports by using the `set_boundary_cell` command.

```
dc_shell> set_dft_configuration -wrapper enable

dc_shell> set_wrapper_configuration \
            -class core_wrapper -maximize_reuse true
```

3. Define the wrapper signals and configure the wrapper chains.

   By default, control signals are added to the design to control the wrapper configuration. To specify existing placeholder ports for these signals, use the `set_dft_signal` command to specify the names and connections of these signals. These wrapper signals must exist in the top-level design you previously created.

```
dc_shell> set_dft_signal -view spec \
            -type wrp_clock -port MY_wrp_clock
```

4. Configure the wrapper chain characteristics.

You can use the set_wrapper_configuration and set_boundary_cell commands to specify any wrapper chain characteristics. You can use the set_scan_path command to specify the order of the wrapper cells.

```
dc_shell> set_wrapper_configuration -class core_wrapper \
            -chain_count 2

dc_shell> set_boundary_cell -class core_wrapper \
            -ports {CLKOUT} -type none

dc_shell> set_scan_path \
            -class wrapper WC0 \
            -ordered_elements [list ordered_port_list]
```

5. Check test design rules.

   Use the dft_drc command to check test design rules.

   ```
   dc_shell> create_test_protocol
   dc_shell> dft_drc
   ```

6. Preview the wrapper and scan cells before inserting them.

   Use the preview_dft command to report on the wrapper and scan cells before you actually insert them.

   ```
   dc_shell> preview_dft -test_wrappers all
   ```

7. Insert the wrapper cells.

   Use the insert_dft command to insert the wrapper cells into the design and stitch the wrapper chain.

   ```
   dc_shell> insert_dft
   ```

8. (Optional) Select the test mode.

   You should check the design rules for each test mode created. Use the current_test_mode command to set the test mode to each of the modes of operation:

   ```
   dc_shell> current_test_mode wrp_if
   ```

9. Check that the design is ready for ATPG by using the dft_drc command.

   ```
   dc_shell> dft_drc
   ```

   This verifies that the scan chains and wrapper cells operate correctly for the current test mode. You can repeat this step for additional test modes.

Core Wrapping Scripts

The following script examples illustrate core wrapping.

Core Wrapping With A Dedicated Wrapper

Example 8-5 wraps a core with a dedicated wrapper.

Example 8-5 Script Example for Dedicated Wrapper

```
read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port clk_st

# Enable and configure wrapper client
set_dft_configuration -wrapper enable
set_wrapper_configuration -class core_wrapper \
    -style dedicated \
    -use_dedicated_wrapper_clock true  \
    -safe_state 1

# Set scan chain count as desired
set_scan_configuration -chain_count 10

# Create the test protocol and run pre-drc
create_test_protocol
dft_drc -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration

# Preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_wrp_if.rpt
```

```
report_dft_signal -view existing_dft -port *

report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_model -output ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan \
  -output stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if \
  -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_of \
  -output stil/wrp_of.spf
```

Core Wrapping With A Shared Wrapper

Example 8-6 wraps a core with a shared wrapper.

Example 8-6 Script Example for Shared Wrapping

```
read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port clk_st

# Enable and configure wrapper client
set_dft_configuration -wrapper enable

# Configure for shared wrappers, using existing cells and create glue
# logic around existing cells
set_wrapper_configuration -class core_wrapper \
    -style shared \
    -shared_cell_type WC_S1 \
    -use_dedicated_wrapper_clock true  \
    -safe_state 1 \
    -register_io_implementation in_place

# Set scan chain count as desired
set_scan_configuration -chain_count 10

# Create the test protocol and run pre-drc
create_test_protocol
dft_drc -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration
```

```
# Preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_wrp_of.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all \
    > reports/xg_wrap_dedicated_wrp_if.rpt

report_dft_signal -view existing_dft -port *

report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_model -output ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan \
  -output stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if \
  -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_of \
  -output stil/wrp_of.spf
```

Core Wrapping With A Delay Test Wrapper

Example 8-7 wraps a core with a dedicated delay wrapper.

Example 8-7 Script Example for Dedicated Delay Wrapping

```
read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port clk_st
```

```
# Enable and configure wrapper client
set_dft_configuration -wrapper enable
set_wrapper_configuration -class core_wrapper \
    -style dedicated \
    -use_dedicated_wrapper_clock true  \
    -safe_state 1 \
    -delay_test true
# Set scan chain count as desired
set_scan_configuration -chain_count 10

# Create the test protocol and run pre-drc
create_test_protocol
dft_drc -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration

# Preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none

insert_dft

current_test_mode Internal_scan

report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of.rpt

current_test_mode wrp_of_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_delay.rpt

current_test_mode wrp_of_scl_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_scl_delay.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if.rpt

current_test_mode wrp_if_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if_delay.rpt
```

```
current_test_mode wrp_if_scl_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_wrp_if_scl_delay.rpt

report_dft_signal -view existing_dft -port *
report_area

change_names -rules verilog -hierarchy

write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_model -output ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan \
  -output stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if \
  -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_if_delay \
  -output stil/wrp_if_delay.spf
write_test_protocol -test_mode wrp_if_scl_delay \
  -output stil/wrp_if_scl_delay.spf
write_test_protocol -test_mode wrp_of \
  -output stil/wrp_of.spf
write_test_protocol -test_mode wrp_of_delay \
  -output stil/wrp_of_delay.spf
write_test_protocol -test_mode wrp_of_scl_delay \
  -output stil/wrp_of_scl_delay.spf
```

Core Wrapping With Shared Delay Wrapper

Example 8-8 wraps a core with a shared delay wrapper.

Example 8-8 Script Example for Shared Delay Wrapping

```
read_ddc ddc/des_unit.ddc
current_design des_unit
uniquify
link

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} \
    -port clk_st

# enable and configure wrapper client
set_dft_configuration -wrapper enable

# Configure for shared wrappers, using existing cells and create glue
# logic around existing cells
set_wrapper_configuration -class core_wrapper \
    -style shared \
    -shared_cell_type WC_S1 \
    -use_dedicated_wrapper_clock true  \
    -safe_state 1 \
    -register_io_implementation in_place \
```

```
    -delay_test true

# Set scan chain count as desired
set_scan_configuration -chain_count 10

# Create the test protocol and run pre-drc
create_test_protocol
dft_drc -verbose

# Report the configuration of the wrapper utility, optional
report_wrapper_configuration

# Preview all test structures to be inserted
preview_dft -test_wrappers all
preview_dft -show all
report_dft_configuration

# Run scan insertion and wrap the design
set_dft_insertion_configuration -synthesis_optimization none
insert_dft

current_test_mode Internal_scan

report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_Internal_scan.rpt

current_test_mode wrp_of
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of.rpt

current_test_mode wrp_of_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_delay.rpt

current_test_mode wrp_of_scl_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_of_scl_delay.rpt

current_test_mode wrp_if
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_path_wrp_if.rpt

current_test_mode wrp_if_delay
report_scan_path -view existing_dft -cell all > \
    reports/
    xg_wrap_dedicated_delay_path_wrp_if_delay.rpt

current_test_mode wrp_if_scl_delay
report_scan_path -view existing_dft -cell all > \
    reports/xg_wrap_dedicated_delay_wrp_if_scl_delay.rpt
report_dft_signal -view existing_dft -port *
report_area
change_names -rules verilog -hierarchy
```

```
write -format ddc -hierarchy -output ddc/scan.ddc
write -format verilog -hierarchy -output vg/scan_wrap.vg
write_test_model -output ddc/des_unit.scan.ctlddc
write_test_protocol -test_mode Internal_scan \
  -output stil/Internal_scan.spf
write_test_protocol -test_mode wrp_if_delay \
  -output stil/wrp_if_delay.spf
write_test_protocol -test_mode wrp_if_scl_delay \
  -output stil/wrp_if_scl_delay.spf
write_test_protocol -test_mode wrp_if \
  -output stil/wrp_if.spf
write_test_protocol -test_mode wrp_of \
  -output stil/wrp_of.spf
write_test_protocol -test_mode wrp_of_delay \
  -output stil/wrp_of_delay.spf
write_test_protocol -test_mode wrp_of_scl_delay \
  -output stil/wrp_of_scl_delay.spf
```

9

On-Chip Clocking Support

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

OCC support includes phase-locked loops, clock shapers, clock dividers and multipliers and so on. In the scan-ATPG environment, scan chain `load_unload` is controlled through an automatic test equipment (ATE) clock. However, internal clock signals that reach state elements during capture are PLL-related.

OCC flows can use either the user-defined clock controller and clock chains or the DFT-inserted OCC clock controller. If you use an existing user-defined clock controller, you would need a set of user-defined commands to identify the existing clock controller outputs with their corresponding clock chain control bits.

This chapter includes the following sections:

- OCC Flows
- Reporting Clock Controller Information
- DRC Support
- DFT-Inserted OCC Controller Configurations
- Waveform and Capture Cycle Example
- Limitations

Note:

This chapter covers flows that are intended for a DFT-to-TetraMAX implementation. For information on using OCC controllers in a non-DFT-to-TetraMAX implementation, see the *TetraMAX Online Help*.

Background

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

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

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

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

- The test-mode control logic for the OCC controller operates correctly and has been connected properly
- Test-mode clocks from the OCC circuitry can be efficiently used by TetraMAX ATPG for at-speed test generation
- OCC circuitry can operate asynchronously to other shift clocks from the tester
- TetraMAX patterns do not require additional modifications to use the OCC and to run properly on the tester

Supported DFT Flows

On-chip clocking (OCC) is supported in the following DFT flows:

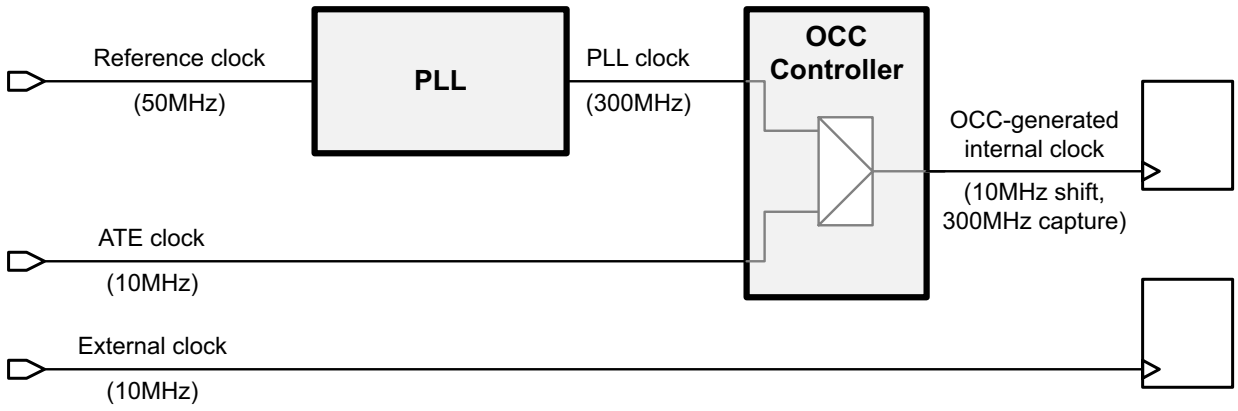
- Basic scan flow
- Top-down, nonhierarchical adaptive scan insertion flow
- Bottom-up basic scan flow with OCC controller stitching at the top level
- Bottom-up hierarchical adaptive scan synthesis flow, which represents subblocks with test models that are OCC controller stitched at the top level
- Hierarchical adaptive scan synthesis (HASS) and Hybrid flows, which stitch the cores at the top level with integration at the top level

Clock Type Definitions

Note the following clock definitions as they apply to OCC controller clocks in this chapter. Figure 9-1 shows an example of each clock type.

- *Reference clock* – The frequency reference to the phase-locked loop (PLL). It must be maintained as a constantly pulsing and free-running oscillator, or the circuitry will lose synchronization.
- *PLL clock* – The output of the PLL. It is also a free-running source that runs at a constant frequency that might or might not be the same as the reference clock.
- *ATE clock* – Shifts the scan chain, typically more slowly than a reference clock. This signal must preexist, or you must manually add this signal (that is, port) when inserting the OCC. The period for this clock is determined by the `test_default_period` variable. Usually the ATE clock is not used as a reference clock, but it must be treated as a free-running oscillator so that it does not capture predictable data while the OCC controller generates at-speed clock pulses. The ATE clock is called a dual clock signal when the same port drives both the ATE clock and the reference clock.
- *Internal clock* – The OCC controller is responsible for gating and selecting between the PLL and ATE clocks, thus creating the internal clock signal to satisfy ATPG requirements.
- *External clock* – A primary clock input of a design that directly clocks flip-flops through the combinational logic, without the use of a PLL clock. The period for this clock is determined by the `test_default_period` variable.

Figure 9-1 Clock Types Used in the OCC Controller Flow



Capabilities

The following OCC features are available:

- Synthesis of individual or multiple clock controllers and clock chains, using the DFT-inserted OCC controller
- Support of pre-DFT DRC, scan chain stitching, and post-DFT DRC in documented OCC support flows
- Support of a PLL-bypass configuration when an external (ATE) clock is used for capture, thus bypassing the PLL clock(s)
- Generation of STIL procedure files with internal clock control details for use with the TetraMAX tool
- Support of post-DFT DRC, scan chain shifting, and adaptive scan
- Support of user-defined clock controller logic and clock chains that are already instantiated in the design

OCC Controller Structure and Operation

OCC controller types and operation is covered in the following sections:

- OCC Controller Types
- OCC Controller Signal Operation
- Logic Representation of an OCC Controller
- Scan-Enable Signal Requirements for OCC Controller Operation

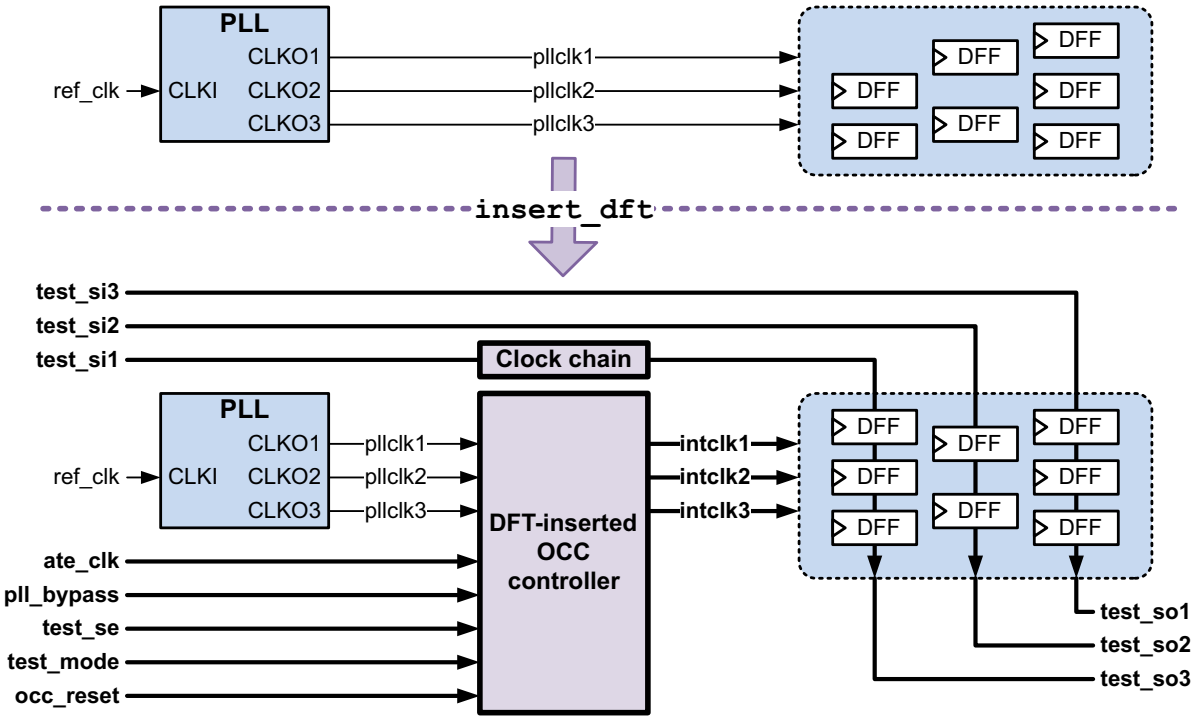
OCC Controller Types

You can use DFT-inserted or user-defined OCC controllers, as described in the following flows:

- DFT-Inserted OCC Controller Flow

  The `insert_dft` command performs insertion and synthesis of a DFT-inserted OCC controller and clock chain. The OCC controller design is validated and incorporated into the resulting test protocol. This flow is shown in Figure 9-2.

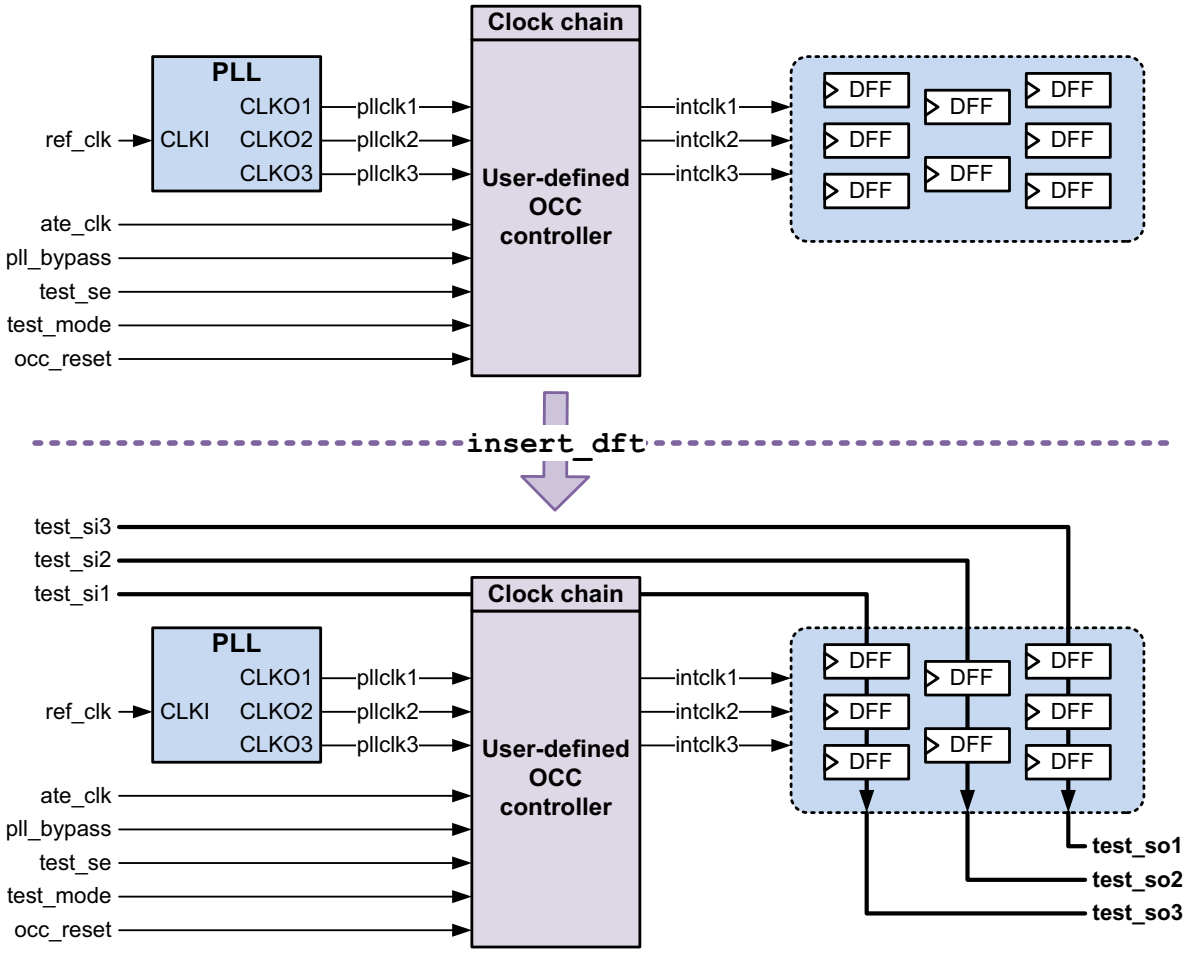
Figure 9-2 OCC and Clock Chain Synthesis Insertion Flow



- Existing User-Defined OCC Controller Flow

  An existing user-defined OCC controller and clock chain is described to DFT Compiler using the `set_dft_signal` command. The OCC controller design is validated and incorporated into the resulting DFT logic and test protocol. This flow is shown in Figure 9-3.

Figure 9-3 User-Defined Clock Controller and Clock Chain Flow



OCC Controller Signal Operation

For Figure 9-2 on page 9-6 and Figure 9-3 on page 9-7, note the following:

- The reference clock (`refclk1`) is always free-running. It is used as a test default frequency input to the PLL.

- The PLL clocks (`pllclk1`, `pllclk2`, and `pllclk3`) are free-running clock outputs from the on-chip clock generator; they can be divided, shaped, or multiplied. They are used for the launch and capture of internal scanable elements that become internal clocks.
- The ATE clock (ATEclk) shifts the scan chain per tester specifications. Each PLL might have its own ATE clock.

  See "Waveform and Capture Cycle Example" on page 9-36 for a waveform diagram that demonstrates the relationship between the various clocks.
- The OCC controller (occ_int) serves as an interface between the on-chip clock generator and internal scan chains. This logic typically contains clock multiplexing logic that allows internal clocks to switch from a slow ATE clock during shift to a fast PLL clock during capture.
- Internal clocks (`intclk1`, `intclk2`, and `intclk3`) are outputs of the PLL control logic driving the scan cells. Each internal clock is controlled or enabled by the clock chain and is connected to the sequential elements within the design.
- The OCC Bypass signal (piX) allows connection of the ATE clock signal directly to the internal clock signals, thus bypassing the PLL clocks.
- The ScanEnable signal (piE) enables switching between the ATE shift clock and output PLL clock signals. ScanEnable must be inactive during every capture procedure, as described in "Scan-Enable Signal Requirements for OCC Controller Operation" on page 9-10. You can use individual ScanEnable signals for each PLL clock signal.
- The TestMode signal (pi1) must be active in order for the circuit to work.
- The OCC Reset (pi2) is used during test setup and initialization.
- The clock chain is a scan chain segment of one or more scan cells. This chain allows for a per-pattern clock selection mechanism by ATPG. Clock selection values are loaded into the clock chain as part of the regular scan load process.

  Note the following:

  - The clock chain is clocked by the falling edge of the internal clock.
  - For standard scan designs, the clock chain can be a dedicated scan chain or a segment within a scan chain.
  - For compressed scan designs, the clock chain can be a dedicated uncompressed scan chain or a segment within a compressed scan chain. For more information, see the "Scan Compression and OCC Controllers" section in Chapter 2, "Using Adaptive Scan Technology," in the *DFTMAX Compression User Guide*.

DFT Compiler inserts the OCC controller and clock chain and then gates each clock controller output to a corresponding internal clock.

In Figure 9-3 on page 9-7, you first need to instantiate any user-defined blocks used as a clock controller or clock chain in the design. When you do this,

- Ensure that the global and clock signals of the user-defined block are connected to the rest of the design.
- Confirm that connections exist for the internal and external clocks, the reset and mode signals of a clock controller, and the clock signals to a clock chain. This enables the block to be controlled from the top level in the pre-DFT netlist.

Note:

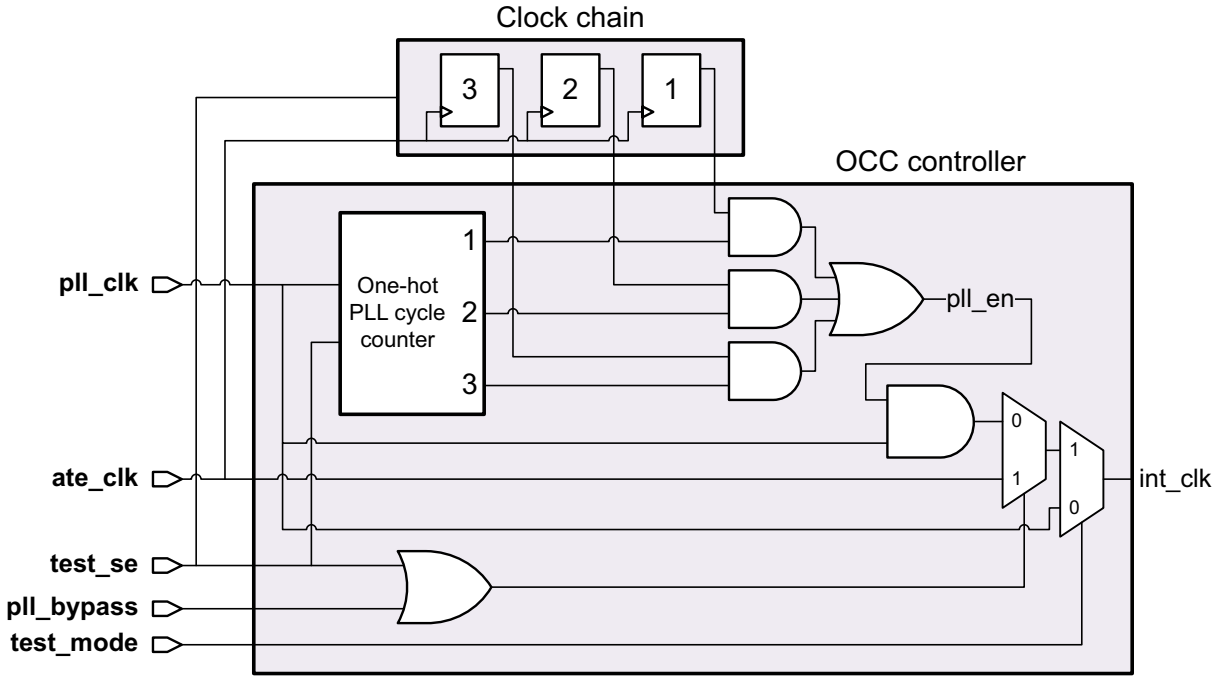
In this design, the clock chains that provide access to clock control bits are embedded within the clock controller, but they could exist as a separate block also.

You can insert and validate the clock controller cells in the RTL, choose a preferred synthesis flow, and then run DFT Compiler with a gate-level netlist.

Logic Representation of an OCC Controller

Figure 9-4 shows the logic structure of an OCC controller and clock chain.

Figure 9-4 Logic Representation of an OCC Controller



Note: This is a logic view, not a functional implementation.

An OCC controller is designed to deliver up to a user-specified number *N* of at-speed clock pulse cycles during capture. A PLL cycle counter generates *N* successive one-hot enable

signals, each of which is gated by the output of a clock chain scan cell. This logic structure provides ATPG with the flexibility to control which cycles deliver an at-speed clock pulse. For an OCC controller that handles multiple OCC generators, the clock chain contains a set of *N* scan cells for each clock.

Note that the figure shows the conceptual operation of an OCC controller. Implementation details, such as cleanly switching between the PLL clock and ATE clock, are not shown.

For more information about the logic structure and operation of the DFT-inserted OCC controller, see SolvNet article 034274, "DFT-inserted OCC Controller Data Sheet".

Scan-Enable Signal Requirements for OCC Controller Operation

The scan-enable signal switches the OCC controller between the ATE shift clock and output PLL clock signals. Therefore, for proper operation, *the scan-enable signal must be held in the inactive state in all capture procedures*.

If you use the STIL procedure file created by the tool, the protocol already meets this requirement. The tool constrains all scan-enable signals to the inactive state in the capture procedures, excluding any scan-enable signals defined with the `-usage clock_gating` option of the `set_dft_signal` command.

If you use a custom STIL procedure file, make sure that all scan-enable signals used by OCC controllers are constrained to the inactive state in all capture procedures.

Enabling On-Chip Clocking Support

To enable OCC support, use the `-clock_controller` option of the `set_dft_configuration` command, as shown in the following example:

```
dc_shell> set_dft_configuration \
                -clock_controller enable
```

OCC Flows

This section covers the following OCC flows in DFT Compiler:

- DFT-Inserted OCC Controller Flow
- Existing User-Defined OCC Controller Flow
- Hierarchical On-Chip Clocking Flow

DFT-Inserted OCC Controller Flow

If you have a design that contains an OCC generator, such as a PLL, but not an OCC controller and clock chain, DFT Compiler can insert both the OCC controller and clock chain. Note that this clock controller design supports only one ATE clock per OCC controller. This section describes the flow associated with this type of implementation.

The PLL clock is expected to already be connected in the design being run through this flow. DFT Compiler will disconnect this PLL clock at the hookup location and insert the newly synthesized clock controller at this location.

This section has the following subsections:

- Defining Clocks
- Defining Global Signals
- Configuring the OCC Controller
- Specifying Scan Configuration
- Performing Timing Analysis
- Script Example

Defining Clocks

You need to define the reference, PLL, and ATE clocks by using the `set_dft_signal` command. Note that this command does not require you to specify the primary inputs.

This section has the following subsections:

- Reference Clocks
- PLL-Generated Clocks
- ATE Clocks

Reference Clocks

The following example shows how to define a PLL reference clock that has the same period as the `test_default_period` variable (assumed to be 100 ns).

```
dc_shell> set_dft_signal -view existing_dft \
               -type MasterClock -port my_clock \
               -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
               -type refclock -port my_clock \
               -period 100 -timing [list 45 55]
```

Note that you only need to define the reference clock with signal type `MasterClock` when the reference clock has the same period as the `test_default_period` variable. Otherwise, this signal definition is not needed and not accepted.

To define a reference clock that has a period other than the `test_default_period`, use the following command:

```
dc_shell> set_dft_signal -view existing_dft \
               -type refclock -port my_clock \
               -period 10 -timing [list 3 8]
```

Note:

When the reference clock period differs from the `test_default_period`, do not define the signal as any signal type other than `refclock`.

Also note the following caveats associated with the `test_default_period` when defining a reference clock:

- If the reference clock period is an integer divisor of the `test_default_period`, then patterns can be written in a variety of formats, including STIL, STIL99, and WGL.
- If the reference clock is not an integer divisor to the `test_default_period`, the only format that can be written in a completely correct way is STIL. Other formats, including STIL99, cannot include the reference clock pulses, and a warning is printed, indicating that these pulses must be added back to the patterns manually.
- Do not define a reference clock period or timings with resolution finer than 1 picosecond. The TetraMAX tool cannot work with finer timing resolutions.

PLL-Generated Clocks

For DFT Compiler to correctly insert the OCC, you must define the PLL-generated clocks as well as the point at which they are generated. The following examples show how to define a set of launch and capture clocks for internal scannable elements controlled by the OCC controller:

```
dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk1 \
            -view existing_dft

dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk2 \
            -view existing_dft

dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk3 \
            -view existing_dft
```

ATE Clocks

The following examples show how to define the signal behavior of the ATE-provided clock required for shifting scan elements:

```
dc_shell> set_dft_signal \
            -type ScanClock \
            -port ATEclk \
            -timing [list 45 55] \
            -view existing_dft

dc_shell> set_dft_signal \
            -type Oscillator \
            -port ATEclk \
            -view existing_dft
```

If you want to identify a precise hookup pin where the ATE clock signal should be connected, use the `-hookup_pin` option of the `set_dft_signal` command and specify the hierarchical name of the pin. In this case, the `set_dft_signal -type ScanClock` command should be issued separately with the `-view spec` option, in addition to the two commands with the `-view existing_dft` option, because this defines how the signal is to be connected to the hookup pin.

Note:

You can use the same clock port as both the ATE clock and PLL reference clock. However, caveats apply. For more information, see SolvNet article 037838, "How Can I Use the Same Clock Port for the ATE and PLL Reference Clocks?".

The following example shows how to specify the hookup pin.

```
dc_shell> set_dft_signal -view spec -type ScanClock \
            -port ATEclk -hookup_pin U2005_ate_clk_2/Z
```

Note:

ATEclk needs to be defined as `-type ScanClock` and `-type Oscillator`.

Defining Global Signals

You must identify the top-level interface signals that control the OCC controller. This includes the OCC bypass, OCC reset, and ScanEnable signals. You must also define a dedicated TestMode signal that activates the OCC controller logic. In the OCC controller insertion flow, these signals are defined with the `-view spec` option because they will be implemented and connected by the `insert_dft` command.

The following examples show how to define a set of OCC controller interface signals for the design example:

```
dc_shell> set_dft_signal \
            -type pll_reset \
            -port pi2 \
            -view spec
```

```
dc_shell> set_dft_signal \
            -type pll_bypass \
            -port piX \
            -view spec

dc_shell> set_dft_signal \
            -type ScanEnable \
            -port piE \
            -view spec

dc_shell> set_dft_signal \
            -type TestMode \
            -port pi1 \
            -view spec
```

The TestMode signal must be a dedicated signal for the OCC controller. It must be active in all test modes and inactive in mission mode. It cannot be shared with TestMode signals used for other purposes, such as AutoFix or multiple test-mode selection.

Note:

You can specify an internal hookup pin for any of these OCC controller interface signals by using the `-hookup_pin` option of the `set_dft_signal` command. You cannot specify internal hookup pins for ATE clocks or reference clocks.

Configuring the OCC Controller

To configure the OCC controller, use the `set_dft_clock_controller` command. Note the following syntax and descriptions:

```
int set_dft_clock_controller
  [-cell_name cell_name]
  [-design_name design_name]
  [-pllclocks ordered_list]
  [-ateclocks clock_name]
  [-chain_count integer]
  [-cycles_per_clock integer]
```

Table 9-1 set_dft_clock_controller Command Syntax

Option	Description
`-cell_name` *cell_name*	Specifies the hierarchical name of the clock controller cell.
`-design_name` *design_name*	Specifies the OCC controller design name. You must specify `snps_clk_mux`.
`-pllclocks` *ordered_list*	Specifies the ordered list of hierarchical PLL clocks you want to control.

Table 9-1 set_dft_clock_controller Command Syntax (Continued)

Option	Description
`-ateclocks` *clock_name*	Specifies the ATE clock (port) you want to connect to the OCC controller. Note: You cannot specify multiple clocks per controller.
`-chain_count` *integer*	Specifies the number of clock chains. The default number of clock chains is one.
`-cycles_per_clock` *integer*	Specifies the maximum number of capture cycles per clock. You should specify a value of two or greater. Capture cycles are cycles during capture when capture clocks are pulsed. Typically, for at-speed transition testing, there are two capture cycles: one is used for launching a transition and the other for capturing the effect of that transition.
`-test_mode_port` *port_name*	Specifies the test-mode port used to enable the clock controller. Use this option if you have multiple test-mode ports and you want to use a specific port to enable the clock controller. The specified port must be defined as a TestMode signal using the `set_dft_signal`command.

The following example shows a typical usage of the `set_dft_clock_controller` command:

```
dc_shell> set_dft_clock_controller \
                -cell_name occ_int \
                -design_name snps_clk_mux \
                -pllclocks { pll/pllclk1  pll/pllclk2 \
                 pll/pllclk3 } \
                -ateclocks  { ATEclk }  \
                -cycles_per_clock 2 -chain_count 1
```

By default, the OCC controller uses an AND/OR-based clock selection structure to glitchlessly select the clocks. The AND cells gates a clock signal, and the OR cells combine the gated signals together. To use discrete latch-based clock gating for the gating portion, set the following variable to `true`:

```
set_app_var test_occ_insert_clock_gating_cells true
```

When using DFTMAX serializers with this variable setting, the `test_elpc_unique_fsm` variable must be set to its default of `true`. For details on this variable, see the section on using serializers with OCC controllers in Chapter 6, “Adaptive Scan With Serializer,” of the *DFTMAX Compression User Guide*.

To use an integrated clock-gating cell, enable latch-based clock gating using the `test_occ_insert_clock_gating_cells` variable. Then, set the desired cell type using the `test_icg_p_ref_for_dft` variable. For information about this variable, see the *DFTMAX Compression User Guide*. For information on clock gating styles, see the *Power Compiler User Guide*.

Specifying Scan Configuration

Specify the `set_scan_configuration` command to define scan ports, scan chains, and the global scan configuration.

To specify scan constraints in your design, use the following command:

```
set_scan_configuration -chain_count <#chains>...
```

If the current design is to be used as a test model later in a hierarchical flow, it is important to avoid clock mixing. Such mixing can cause the clock chain of the OCC controller to mix with flip-flops of opposite polarity on a single scan chain. As a result, this scan chain cannot be combined with scan chains of other test models and the minimum scan chain count at the top level is increased. This problem is worsened when multiple OCC controllers are added to the design or when multiple subdesigns of the top-level design will have OCC controllers.

Performing Timing Analysis

After DFT insertion completes, you must ensure that the OCC controller logic is properly constrained for timing analysis.

If you are using a MUX-based clock-gating method to select the clocks, you must use the `set_clock_gating_check` command to manually specify a clock-gating check at the MUX gate. This check is needed to check the timing between the fast-clock-enable registers and the “FastClock” gates (multiplexers between the fast clocks and the slow clocks). MUX-based clock gating is used when the `test_occ_insert_clock_gating_cells` variable is set to its default of `false`.

For more information on performing timing analysis in a DFT-inserted OCC controller flow, see SolvNet article 022490, “Static Timing Analysis Constraints for On-Chip Clocking Support.” Also see the two sections titled “Special Considerations” in SolvNet article 034274, “DFT-Inserted OCC Controller Data Sheet.”

Script Example

Example 9-1 shows DFT Compiler performing prescan DRC, scan chain stitching, and post-scan DRC. The STIL procedure file generated at the end of the DFT insertion process contains PLL clock details suitable for the TetraMAX tool.

Example 9-1 Flow Example for DFT-Inserted OCC Controller and Clock Chain

```
read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# top level free running clock
set_dft_signal -view existing_dft -type refclock \
   -port refclk1  -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
   -port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
   -port  ATEclk -timing [list 45 55]

set_dft_signal -view existing_dft -type Oscillator \
   -port  ATEclk

# Define the PLL generated clocks --
# these are the launch/capture clocks for internal scannable
# elements and are controlled by occ controller
set_dft_signal -view existing_dft -type Oscillator \
  -hookup_pin pll/pllclk1

set_dft_signal -view existing_dft -type Oscillator \
  -hookup_pin pll/pllclk2

set_dft_signal -view existing_dft -type Oscillator \
  -hookup_pin pll/pllclk3

# Enable PLL capability
set_dft_configuration -clock_controller enable

# The following command specifies the OCC controller
# design to be instantiated. The DFT Compiler synthesized
# clock controller is named snps_clk_mux
set_dft_clock_controller -cell_name snps_pll_controller \
   -design_name snps_clk_mux -pllclocks { pll/pllclk1 \
   pll/pllclk2 pll/pllclk3 } -ateclocks  { ATEclk }  \
   -cycles_per_clock 2 -chain_count 1

set_scan_configuration -chain_count 30 -clock_mixing no_mix
create_test_protocol
dft_drc
report_dft_clock_controller -view spec
preview_dft -show all
insert_dft
dft_drc
```

```
# Run DRC with external clocks enabled during capture
# (PLL bypassed)
set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output scan_pll.stil
```

Existing User-Defined OCC Controller Flow

If you have a design that contains an OCC generator and it already instantiates an existing user-defined OCC controller and clock chain, DFT Compiler can analyze the OCC controller, validate the functionality, and incorporate it into the test protocol.

This section has the following subsections:

- Defining Clocks
- Defining Global Signals
- Specifying Clock Chains
- Specifying Scan Configuration
- Script Example

Defining Clocks

Use the `set_dft_signal` command to define the following clock signals for Figure 9-3:

- Reference Clocks
- PLL-Generated Clocks
- ATE Clocks
- Clock Chain Configuration and Control-Per-Pattern Information

Reference Clocks

A reference clock definition is used primarily as an informational device. It provides correct data for the protocol file to pass TetraMAX DRC. For some special cases, a reference clock signal might not be needed.

The following example shows how to define a PLL reference clock that has the same period as the test_default_period variable (assumed to be 100 ns).

```
dc_shell> set_dft_signal -view existing_dft \
            -type MasterClock -port my_clock \
            -timing [list 45 55]

dc_shell> set_dft_signal -view existing_dft \
            -type refclock -port my_clock \
            -period 100 -timing [list 45 55]
```

Note that you only need to define the reference clock with signal type MasterClock when the reference clock has the same period as the test_default_period variable. Otherwise, this signal definition is not needed and not accepted.

To define a reference clock that has a period other than the test_default_period, use the following command:

```
dc_shell> set_dft_signal -view existing_dft \
               -type refclock -port my_clock \
               -period 10 -timing [list 3 8]
```

Note:

When the reference clock period differs from the test_default_period, do not define the signal as any signal type other than refclock.

Also note the following caveats associated with the test_default_period when defining a reference clock:

- If the reference clock period is an integer divisor of the test_default_period, then patterns can be written in a variety of formats, including STIL, STIL99, and WGL.
- If the reference clock is not an integer divisor of the test_default_period, the only format that can be written in a completely correct way is STIL. Other formats (including STIL99) cannot include the reference clock pulses. A warning is printed, indicating that these pulses must be added back to the patterns manually.
- Do not define a reference clock period or timings with resolution finer than 1 picosecond. The TetraMAX tool cannot work with finer timing resolutions.

PLL-Generated Clocks

PLL clocks are the output of the PLL. This output is a free-running source that also runs at a constant frequency, which might not be the same as the reference clock's. This information is forwarded to the TetraMAX tool through the protocol file to allow the verification of the clock controller logic.

The following commands show how to define a PLL clock for the design example:

```
dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk1 \
            -view existing_dft

dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk2 \
            -view existing_dft

dc_shell> set_dft_signal
            -type Oscillator \
            -hookup_pin PLL/pllclk3 \
            -view existing_dft
```

ATE Clocks

An ATE clock signal can be pulsed several times before and after scan shift (scan-enable signal inactive) to synchronize the clock controller logic in the capture phase and back into the shift phase.

The following commands show how to define ATE clocks for the design example:

```
dc_shell> set_dft_signal \
            -type ScanClock \
            -port ATEclk \
            -timing [list 45 55] \
            -view existing_dft

dc_shell> set_dft_signal \
            -type Oscillator \
            -port ATEclk \
            -view existing_dft
```

Note:

You can use the same clock port as both the ATE clock and PLL reference clock. However, caveats apply. For more information, see SolvNet article 037838, “How Can I Use the Same Clock Port for the ATE and PLL Reference Clocks?”.

A user-defined OCC controller can use ATE clock synchronization logic which differs from the DFT-inserted OCC controller. In these cases, you might need to set the `test_ate_sync_cycles` variable to a nondefault value. For more information, see SolvNet article 035708, “What Does the test_ate_sync_cycles Variable Do?”.

Clock Chain Configuration and Control-Per-Pattern Information

You must specify the correlation between the internal clock signals driven from the clock controller outputs, the signals driven from the clock generator (PLL) outputs, and the signals provided by the user-defined clock chain. This information indicates how the clock signal is

enabled by the clock chain control bits in each clock cycle. The correspondence between the controlled internal clock signals and clock chain control bits is identified in the protocol file for TetraMAX ATPG to generate patterns.

You must specify

- All user-defined clock controller outputs referencing the internal clocks
- A corresponding set of clock chain control bits, ATE clock, and clock generator output (PLL) for each clock controller output

See the following example:

```
dc_shell> set_dft_signal -type Oscillator \
            -hookup_pin occ_int/intclk1 \
            -ate_clock ATEclk \
            -pll_clock PLL/pllclk1  \
            -ctrl_bits [list 0 occ_int/FF_cyc1/Q 1 \
                             1 occ_int/FF_cyc2/Q 1 \
                             2 occ_int/FF_cyc3/Q 1] \
            -view existing_dft
```

Note:

The `-ctrl_bits` option is used to provide a list of triplets that specify the sequence of bits needed to enable the propagation of the clock generator outputs. The first element of each triplet is the cycle number (integer) indicating the cycle where the clock signal will be propagated. The second element is the pin name (a valid design hierarchical pin name) of the clock chain control bit. The third element is the active state (0 or 1) of the control bit signal. For more information about this option, see the `set_dft_signal` man page.

Note:

The `-view existing_dft` option is used because connections already exist between the referenced port and the clock controller.

Defining Global Signals

You must identify the top-level interface signals that control the OCC controller. This includes the OCC bypass, OCC reset, and ScanEnable signals. You must also define a dedicated TestMode signal that activates the OCC controller logic. In the user-defined OCC controller flow, these signals are defined with the `-view existing_dft` option because they already exist and must be described to DFT Compiler.

The following examples show how to define a set of OCC controller interface signals for the design example:

```
dc_shell> set_dft_signal \
            -type pll_reset \
            -port pi2 \
            -view existing_dft
```

```
dc_shell> set_dft_signal \
            -type pll_bypass \
            -port piX \
            -view existing_dft

dc_shell> set_dft_signal \
            -type ScanEnable \
            -port piE \
            -view existing_dft

dc_shell> set_dft_signal \
            -type TestMode \
            -port pi1 \
            -view existing_dft
```

The TestMode signal must be a dedicated signal for the OCC controller. It must be active in all test modes and inactive in mission mode. It cannot be shared with TestMode signals used for other purposes, such as AutoFix or multiple test-mode selection.

Note:

You can specify an internal hookup pin for any of these OCC controller interface signals by using the `-hookup_pin` option of the `set_dft_signal` command. You cannot specify internal hookup pins for ATE clocks or reference clocks.

Specifying Clock Chains

Specify clock chains by using the `set_scan_group` command. This ensures that the sequential cells are treated as a group and are logically ordered.

```
dc_shell> set_scan_group clk_chain \
            -class OCC \
            -include_elements \
            [list occ_int/FF_1 occ_int/FF_2] \
            -access [list ScanDataIn occ_int/si \
                          ScanDataOut occ_int/so \
                          ScanEnable occ_int/se] \
            -serial_routed true
```

In this flow, you should insert the clock chain to be clocked by the falling edge of the internal clock in such a way that it can be placed at the head of a scan chain. When defining the clock chain, use the `set_scan_group -class OCC` command to specify the special treatment of the clock chain, and avoid using the `set_scan_path` command so that the scan architect has maximum flexibility in putting the clock chain in the best location on the scan chains. The `-class OCC` option allows the clock chain to be recognized if the module is incorporated as a test model in the integration flow.

In case a separate scan path is required for the clock chain, the `set_scan_path -class OCC` command is also available. If you are using adaptive scan, the clock chain does not need to be a separate scan path, but if you want to define it as such, two `set_scan_path`

commands are required, one with the `-test_mode Internal_scan` option and one with the `-test_mode ScanCompression_mode` option. The process of combining adaptive scan with clock controllers results in a multiple test-mode architecture; therefore, both modes must be specified.

If the current design is to be used later as a test model in a hierarchical flow and your scan configuration allows clock mixing, you should make sure that the clock chains are kept separate from other scan chains. Otherwise, the top-level scan architecture might require too many scan chains because the submodule scan chains are incompatible with each other. To force the clock chains to be separate, use the `set_scan_path` command with the `-class occ` and `-complete true` options. For more information, see SolvNet article 018046, "How Can I Control Scan Stitching of OCC Controller Clock Chains?"

Specifying Scan Configuration

Specify the `set_scan_configuration` command to define scan ports, scan chains, and the global scan configuration by using the following command:

```
set_scan_configuration -chain_count num_chains ...
```

If the current design is to be used later as a test model in a hierarchical flow, it is important to avoid clock mixing. If you must mix clocks, use the method described in "Specifying Clock Chains" on page 9-22 to avoid problems integrating the clock chains at the next higher level of hierarchy.

Script Example

When you run a design that contains a user-defined OCC controller and clock chains, a STIL procedure file is generated, as shown in Example 9-2.

Example 9-2 Flow Example for Existing OCC Controller and Clock Chain

```
read_verilog mydesign.v
current_design mydesign
link

# Define the PLL reference clock
# top level free running clock
set_dft_signal -view existing_dft -type refclock \
   -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
   -port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
   -port  ATEclk -timing [list 45 55]
```

```
set_dft_signal -view existing_dft -type Oscillator \
   -port  ATEclk

# Define the PLL generated clocks
set_dft_signal -view existing_dft -type Oscillator \
    -hookup_pin pll/pllclk1

set_dft_signal -view existing_dft -type Oscillator \
    -hookup_pin pll/pllclk2

set_dft_signal -view existing_dft -type Oscillator \
    -hookup_pin pll/pllclk3

# Enable PLL capability
set_dft_configuration -clock_controller enable

#  Specify clock controller output and control-per-pattern information
set_dft_signal -type Oscillator -hookup_pin occ_int/intclk1 \
  -ate_clock ATEclk -pll_clock PLL/pllclk1 \
  -ctrl_bits [list 0 occ_int/FF_1/Q 1 \
                   1 occ_int/FF_2/Q 1] \
  -view existing_dft

#  Define the existing clock chain segments
set_scan_group clk_chain -class occ\
    -include_elements [list occ_int/FF_1  \
                            occ_int/FF_2]  \
    -access [ list ScanDataIn occ_int/si  \
                   ScanDataOut occ_int/so
                   ScanEnable occ_int/se] \
    -serial_routed true

#  Specify global controller signals
set_dft_signal -type pll_reset -port pi2  -view existing_dft
set_dft_signal -type pll_bypass -port piX -view existing_dft
set_dft_signal -type ScanEnable -port piE -view existing_dft

# Define the TestMode signals
set_dft_signal -type TestMode -port pi1 -view existing_dft

# Registers inside the OCC controller must be nonscan or else the
# internal clock will not be controlled correctly. However,the
# clock chain must be scanned. Use set_scan_element false on the
# former but not on the latter.
set_scan_element false occ_int/clock_controller

set_scan_configuration -chain_count 30 -clock_mixing no_mix
create_test_protocol
dft_drc
report_dft_clock_controller -view existing_dft
preview_dft -show all
insert_dft
dft_drc
```

```
# Run DRC with external clocks enabled during capture
# (PLL bypassed)

set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output top.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output scan_pll.stil
```

Hierarchical On-Chip Clocking Flow

The integration flow supports test models that include OCC controllers. Either the DFT-inserted OCC controller flow or the existing user-defined OCC controller flow can be used for the block containing the OCC controller. The hierarchical integration flow works the same for either case for the levels above the block with the controller.

Use the `set_dft_configuration -clock_controller enable` command at all hierarchical levels above those that contain the OCC controllers. Top-level connections for the test model's `pll_reset`, `pll_bypass`, ATE clock, and TestMode (for the OCC controller) signals should be either all preconnected or all left unconnected so that the `insert_dft` command can make the connections. The DFT architect cannot recognize partially connected conditions reliably and might make mistakes if some, but not all, of these signals are already connected.

When OCC controller insertion is performed on the current design, and the current design contains test models which have their own OCC controllers, the `insert_dft` command must make the top-level connections to the test model OCC control signals. OCC controller insertion is not supported when test model OCC control signals are already connected.

Top-level integration uses signal types to determine the correct connections to make. Most DFT signals used by OCC controllers, including the `pll_reset` and `pll_bypass` types, can be preconnected at the top level by the user and specified as `-view exist`, or they can be connected by the `insert_dft` command if preferred.

However, the ATE clock signal has no special signal type in the test model, so user guidance is required. To give this guidance, use the `-connect_to` option of the `set_dft_signal` command.

The following commands define an ATE clock and two submodule connections in an integration flow where the `insert_dft` command must make the OCC connections to the cores containing the OCC controllers:

```
set_dft_signal -view existing_dft -type ScanClock \
   -port ATEclk -timing [list 45 55] \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view spec -type ScanClock \
   -port ATEclk \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
   -port ATEclk
```

This example uses both the `-view existing_dft` and `-view spec` options for the `ScanClock` signal type because timing can only be attached to an existing clock. The `-view spec` option specifies that a change to the design is needed during DFT insertion.

The following commands define an ATE clock and two submodule connections in an integration flow where the OCC connections to the cores already exist:

```
set_dft_signal -view existing_dft -type ScanClock \
   -port ATEclk -timing [list 45 55] \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
   -port ATEclk
```

When the ATE clock signal is already connected to a core containing an OCC controller, only the `-view existing_dft` option should be used.

Note that reference clocks, other than ATE clocks, are not connected by the `insert_dft` command because they are considered to be functional signals rather than test signals. The only effect of specifying them is that they are defined in the test protocol.

Script Example for Cores Without Preconnected OCC Signals

Example 9-3 shows a script example that performs hierarchical integration of a DFT-inserted OCC controller where the `insert_dft` command must make the OCC connections to the cores containing the OCC controllers.

Example 9-3 Script Example for Integration Without Preconnected OCC Signals

```
# Read the test models for the OCC-inserted submodules
read_ddc subdesign1.ddc
read_ddc subdesign2.ddc

# Read the top level of the design
read_verilog topdesign.v
```

```
current_design topdesign
link

# Define the PLL reference clock
# top level free running clock
# The PLL reference clock must always be preconnected
set_dft_signal -view existing_dft -type refclock \
   -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
   -port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
   -port ATEclk -timing [list 45 55] \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view spec -type ScanClock \
   -port ATEclk \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
   -port ATEclk

# Enable PLL capability
set_dft_configuration -clock_controller enable

# Specify global controller signals, all in spec view
set_dft_signal -type pll_reset -port pi2 -view spec
set_dft_signal -type pll_bypass -port piX -view spec
set_dft_signal -type ScanEnable -port piE -view spec

# Define the PLL TestMode signals
set_dft_signal -type TestMode -port pi1 -view spec

create_test_protocol
dft_drc
preview_dft -show all
insert_dft
dft_drc

# Run DRC with external clocks enabled during capture
# (PLL bypassed)
set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output topmodule.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output
 topmodule.scan.stil
```

Script Example for Cores With Preconnected OCC Signals

Example 9-4 shows a script example that performs hierarchical integration of an existing OCC controller where the OCC connections to the cores already exist.

Example 9-4 Script Example for Integration With Preconnected OCC Signals

```
# Read the test models for the OCC-inserted submodules
read_ddc subdesign1.ddc
read_ddc subdesign2.ddc

# Read the top level of the design
read_verilog topdesign.v
current_design topdesign
link

# Define the PLL reference clock
# top level free running clock
# The PLL reference clock must always be preconnected
set_dft_signal -view existing_dft -type refclock \
   -port refclk1 -period 100 -timing [list 45 55]

set_dft_signal -view existing_dft -type MasterClock \
   -port refclk1 -timing [list 45 55]

# Define the ATE clock
# the ATE-provided clock for shift of scan elements
set_dft_signal -view existing_dft -type ScanClock \
   -port ATEclk -timing [list 45 55] \
   -connect_to [list u1/ATEclk u2/ATEclk]

set_dft_signal -view existing_dft -type Oscillator \
   -port ATEclk

# Enable PLL capability
set_dft_configuration -clock_controller enable

# Specify global controller signals, all in existing DFT view
set_dft_signal -type pll_reset -port pi2 -view existing_dft
set_dft_signal -type pll_bypass -port piX -view existing_dft
set_dft_signal -type ScanEnable -port piE -view existing_dft

# Also define ScanEnable in spec view so it is used for top-level logic
set_dft_signal -type ScanEnable -port piE -view spec

# Define the PLL TestMode signals
set_dft_signal -type TestMode -port pi1 -view existing_dft

create_test_protocol
dft_drc
preview_dft -show all
insert_dft
```

```
dft_drc

# Run DRC with external clocks enabled during capture
# (PLL bypassed)
set_dft_drc_configuration -pll_bypass enable
dft_drc

change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output topmodule.scan.v

# Write out combined PLL enabled/bypassed test protocol:
write_test_protocol -output topmodule.scan.stil
```

Reporting Clock Controller Information

Use the `report_dft_clock_controller` command to generate reports.

DFT-Inserted OCC Controller Flow

For DFT-inserted OCC controllers and clock chains, use the `report_dft_clock_controller -view spec` command to output a report. This report displays the options that you set for the `set_dft_clock_controller` command.

Example 9-5 shows a clock controller report for the DFT-inserted OCC controller flow.

Example 9-5 Report Example from the report_dft_clock_controller -view spec Command

```
****************************************
Report : Clock controller
Design : des_chip
Version: G-2012.06
Date   : Fri Sep  7 05:42:06 2012
****************************************

========================================
TEST MODE: all_dft
VIEW     : Specification
========================================
Cell name:            pll_controller_0
Design:               snps_clk_mux
Chain count:          1
Cycle count:          2
PLL clock:            u_pll/clkgenx2 u_pll/clkgenx3
ATE clock:            ateclk
```

Existing User-Defined OCC Controller Flow

For existing user-defined OCC controllers and clock chains, after you define the internal clock signals and the corresponding control-per-pattern information, use the `report_dft_clock_controller -view existing_dft` command to report what you have specified. In Example 9-6, the report shows information about the clock generator signal, the ATE clock, the OCC controller output, and the clock chain control bits.

Example 9-6 Report Example from the report_dft_clock_controller Command

```
**************************************
Report : Clock controller
Design : des_chip
Version: G-2012.06
Date   : Fri Sep  7 05:18:55 2012
**************************************

Clock controller: ctrl_0
============================================================
  Number of bits per clock: 4
  Controlled clock output pin: duto/clk
  ------------------------------------------------------------
  Clock generator signal: dutp/PLLCLK
  ATE clock signal: i_ateclk
  Control pins:
              cycle 0  duto/snps_clk_chain_0/FF_0/Q  1
              cycle 1  duto/snps_clk_chain_0/FF_1/Q  1
              cycle 2  duto/snps_clk_chain_0/FF_2/Q  1
              cycle 3  duto/snps_clk_chain_0/FF_3/Q  1
  ============================================================
============================================================
```

DRC Support

D-rules (Category D – DRC Rules) support PLL-related design rule checks. The checked rule and message text correspond by number to TetraMAX PLL-related C-rules (Category C – Clock Rules). The rules are as follows:

- D28 – Invalid PLL source for internal clock
- D29 – Undefined PLL source for internal clock
- D30 – Scan PLL conditioning affected by nonscan cells
- D31 – Scan PLL conditioning not stable during capture
- D34 – Unsensitized path between PLL source and internal clock
- D35 – Multiple sensitizations between PLL source and internal clock

- D36 – Mistimed sensitizations between PLL source and internal clock
- D37 – Cannot satisfy all internal clocks off for all cycles
- D38 – Bad off conditioning between PLL source and internal clock

Enabling the OCC Controller Bypass Configuration

Use the `set_dft_drc_configuration` and `write_test_protocol` commands to enable the OCC controller bypass configuration for design rule checking. The `-pll_bypass` option of the `set_dft_drc_configuration` command enables post-scan insertion DRC with constraints that put the OCC clock controller in bypass configuration.

The syntax is as follows:

```
set_dft_drc_configuration -pll_bypass enable | disable
```

The default setting is `disable`.

To perform DRC of both bypass configurations, PLL active and PLL bypassed, use the following commands:

```
insert_dft

set_dft_drc_configuration -pll_bypass disable ;# already the default
dft_drc

set_dft_drc_configuration -pll_bypass enable
dft_drc

write_test_protocol my_design.spf
```

The test protocol written by the `write_test_protocol` command contains information for PLL bypass as well as for PLL enabled. In the TetraMAX tool, use the `run_drc -patternexec` command to select the operating mode to use.

DFT-Inserted OCC Controller Configurations

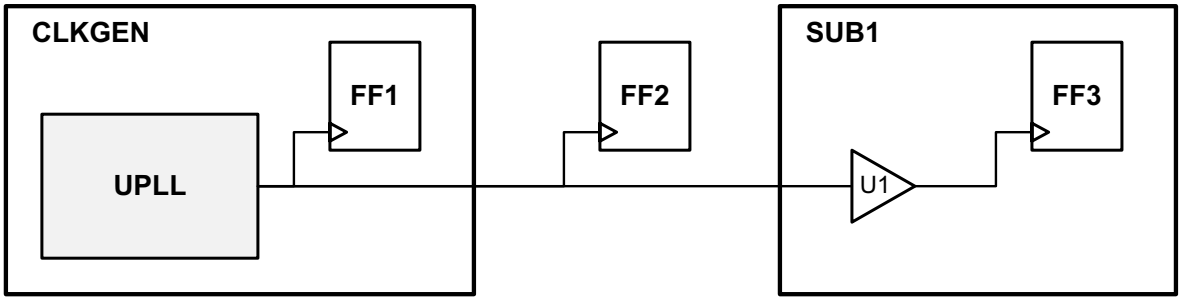
This section shows DFT-inserted OCC controller configurations and the associated configuration commands, as described in the following subsections:

- Single OCC Controller Configurations
- Multiple DFT-Inserted OCC Controller Configurations

Single OCC Controller Configurations

This section shows the results of using various configurations of the `set_dft_clock_controller` command on the design example shown in Figure 9-5.

Figure 9-5 Design Example for Single OCC Controller Insertion



The following configuration examples are applied to this design:

- Example 1 – Controller inserted at the output of UPLL, within the CLKGEN block.
- Example 2 – Controller inserted at the output of the CLKGEN block.
- Example 3 – Controller inserted at the output of the buffer.

Example 1

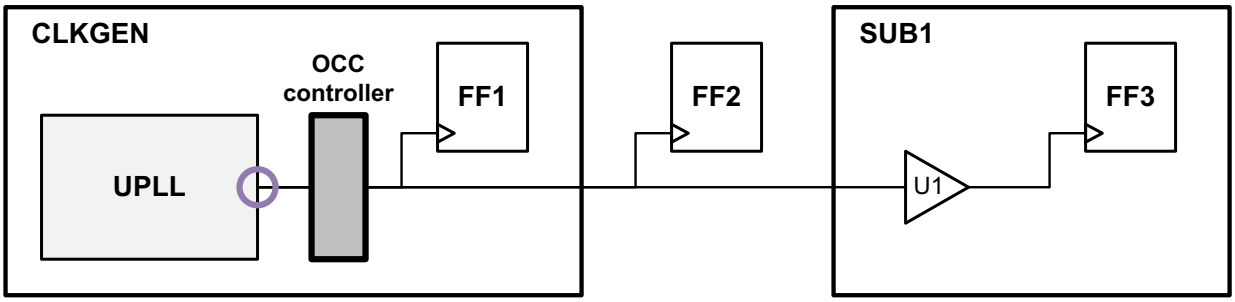
The first example, shown in Figure 9-6, uses the following configuration:

```
dc_shell> set_dft_clock_controller \
                -pllclocks {CLKGEN/UPLL/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of PLL, within the clkgen1 block.
- The clocks of all flip-flops are controllable.

Figure 9-6 Controller Inserted at Output of PLL Within CLKGEN Block



Example 2

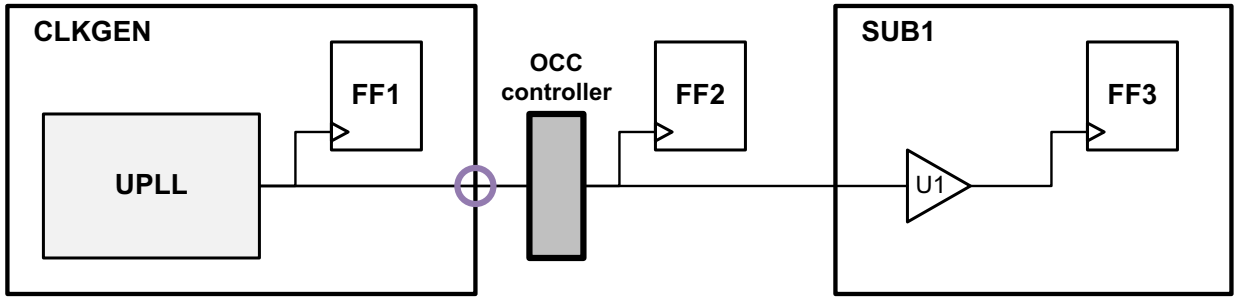
The second example, shown in Figure 9-7, uses the following configuration:

```
dc_shell> set_dft_clock_controller \
                -pllclocks {CLKGEN/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of the CLKGEN block.
- The FF1 clock remains uncontrollable.

Figure 9-7 Controller Inserted at Output of CLKGEN Block



Example 3

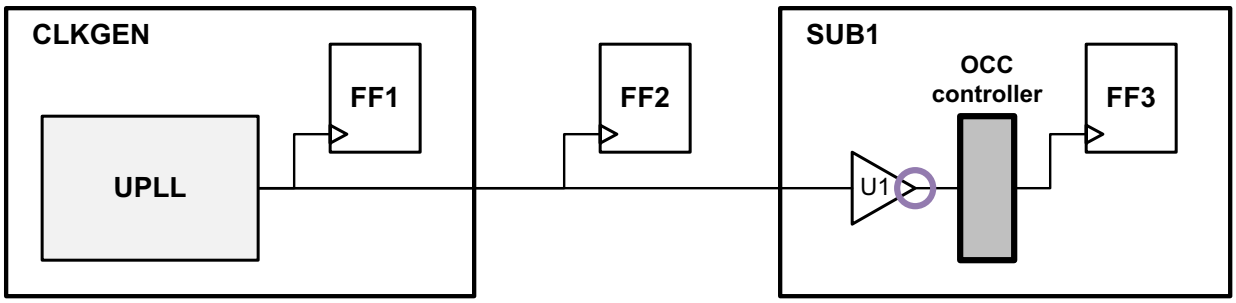
The third example, shown in Figure 9-8, uses the following configuration:

```
dc_shell> set_dft_clock_controller \
                -pllclocks {SUB1/U1/Z}
```

In this case, the following occurs:

- The controller is inserted at the output of buffer U1.
- The FF1 and FF2 clocks remain uncontrollable.

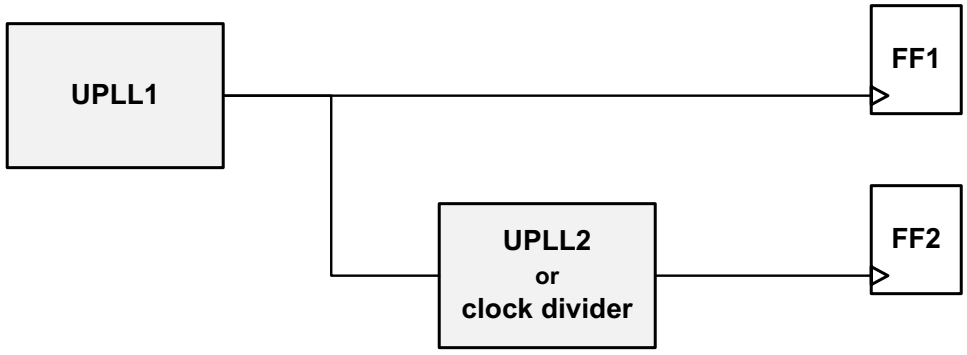
Figure 9-8 Controller Inserted at Output of the Buffer



Multiple DFT-Inserted OCC Controller Configurations

This section shows the results of configuring multiple DFT-inserted OCC controllers for the design example shown in Figure 9-9.

Figure 9-9 Design Example for Multiple OCC Controller Insertion



When multiple PLLs exist in a design, the reference clock input to each PLL cell must be a free-running clock in test mode. Care must be taken to insure that an OCC controller is not inserted at a location that would block a free-running clock to a downstream PLL cell.

In this design example, the primary PLL named UPLL1 receives the incoming reference clock and generates a PLL output clock. This PLL output clock then feeds either a second PLL or clock divider cell, creating a second cascaded PLL output clock.

The following configuration examples are applied to this design:

- Example 1 – Controller incorrectly inserted, at the output of UPLL1.
- Example 2 – Controller correctly inserted, at the output of a buffer driven by UPLL1.

Example 1

The first example, shown in Figure 9-10, uses the following configuration:

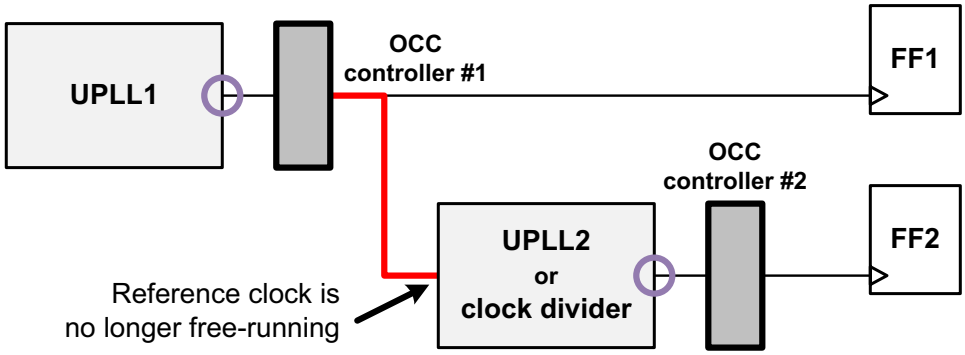
```
dc_shell> set_dft_clock_controller \
                -pllclocks {UPLL1/clkout}
dc_shell> set_dft_clock_controller \
                -pllclocks {UPLL2/clkout}
```

In this case, the following occurs:

- The controller is inserted at the output of UPLL1.
- As a result, the free-running clock to UPLL2 is blocked by the first OCC controller, causing incorrect operation of UPLL2.

The incorrect operation of UPLL2 might only be detectable during Verilog simulation of the resulting netlist.

Figure 9-10 Free-Running Clock Blocked to UPLL2



Example 2

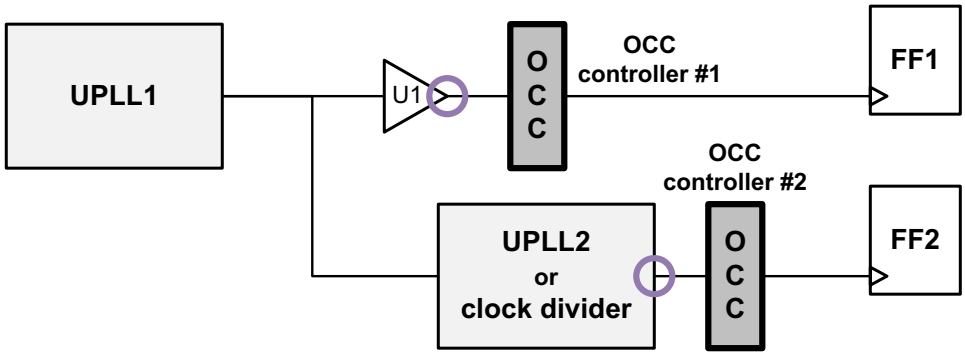
The second example, shown in Figure 9-11, uses the following configuration:

```
dc_shell> set_dft_clock_controller \
                -pllclocks {U1/Z}
dc_shell> set_dft_clock_controller \
                -pllclocks {UPLL2/clkout}
```

This example uses a buffer to isolate the downstream fanout that the first OCC controller should drive. In this case, the following occurs:

- The controller is inserted at the output of buffer U1 driven by UPLL1.
- As a result, the free-running clock from UPLL1 propagates to UPLL2, allowing correct operation of UPLL2.

Figure 9-11 Free-Running Clock Propagates to UPLL2

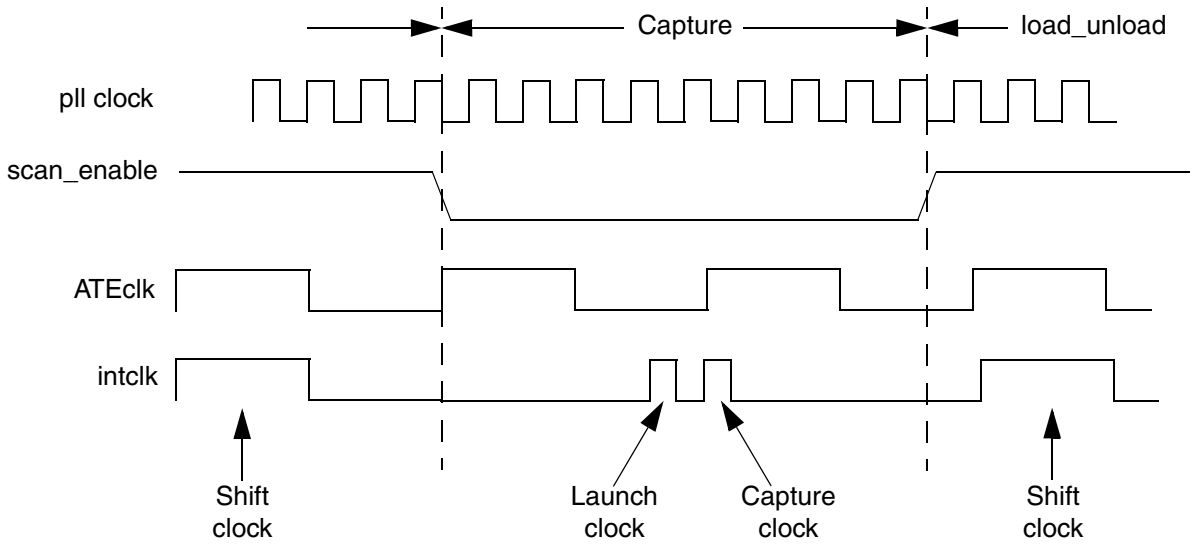


You must ensure that the isolation buffer is not optimized away by applying a `set_dont_touch` command, applying a `set_size_only` command, or using hierarchy. After the clock controller is inserted, the resulting test protocol references the specified pin as a PLL pin.

Waveform and Capture Cycle Example

Figure 9-12 shows an example of the relationship between various clocks when the design contains an OCC generator and an OCC controller.

Figure 9-12 Desired Clock Launch Waveform Example



For information about `pll clock`, `ATEclk`, and `intclk`, see “Clock Type Definitions” on page 9-4.

Limitations

Note the following limitations:

- Inferencing internal PLL or any reference clocks is not supported. For prescan DRC, you must explicitly define your reference clock, ATE clock, and PLL clocks.
- When multiple OCC controllers are used, all controllers must use the same `-cycles_per_clock` option value.

- You cannot mix the DFT-inserted and user-defined OCC controller types in the same DFT insertion run. However, this restriction does not apply to cores that already contain OCC controllers. In integration flows, you can mix OCC controller types across cores and between cores and the top level.
- The OCC controller generated by DFT Compiler does not test faults across internal clock domains.
- When inserting new OCC controllers into your design, you must synthesize and insert multiple clock OCC controllers with a single `insert_dft` command.
- The only supported scan style is multiplexed flip-flop.
- Post-DFT DRC is not supported in Hierarchical Adaptive Scan Synthesis (HASS) flows.
- Fast-sequential patterns with OCC support cannot measure the primary outputs between system pulses. The measure primary output is placed before the first system pulse and measures only Xs. You have to use pre-clock-measure, with the strobe being placed before the clock.
- External clocks, which have a direct connection to scan flip-flops, cannot serve as ATE clocks for the OCC controller.
- The `set_dft_clock_controller -ateclocks` command accepts only one port. The user can have multiple OCC controllers, but only one port can be specified at the `-ateclocks` option per controller.
- Postclock strobe measure (that is, end-of-cycle measure) is not compatible with PLL reference clocks.
- If you are using a MUX-based clock-gating method to select the clocks, you must use the `set_clock_gating_check` command to manually specify a clock-gating check at the MUX gate. The Design Compiler, IC Compiler, and PrimeTime tools do not infer clock-gating checks for MUX gates. This check is needed to check the timing between the fast-clock-enable registers and the “FastClock” gates (multiplexers between the fast clocks and the slow clocks).

  In the DFT-inserted OCC controller flow, MUX-based clock gating is used when the `test_occ_insert_clock_gating_cells` variable is set to its default of `false`. For more information, see SolvNet article 022490, “Static Timing Analysis Constraints for On-Chip Clocking Support.”
- When a pipelined scan-enable signal is used with OCC flows, the `insert_dft` command fails to make some connections properly. To use these features together, you must check and correct the connections so that the following requirements are met:
  - The scan-enable connections to the OCC controller and clock chain must use the unpipelined scan-enable signal. That is, use the input to the scan-enable pipeline register instead of its output.

- The clock connection to the scan-enable pipeline register in OCC controller clock domains must be connected to the internal clock output of the OCC controller block.
- In hierarchical OCC controller flows, the OCC controller can be inserted or defined at the core level, then scan-enable pipeline registers can be inserted during the top-level integration phase. In this case, the OCC controller is correctly connected at the core level, but the connections are incorrectly made during top-level integration.

  To ensure correct operation, you must design the core to provide multiple scan-enable signals. Connect the OCC controller to the unpipelined scan-enable signal, and use the pipelined scan-enable signal for the remaining connections.

  To satisfy the requirement that the scan-enable pipeline register must be clocked by the OCC controller's clock output, you must also pass the OCC internal clock to an output of the core, then use it for the clock connection of the top-level scan-enable pipeline register.

- In hierarchical OCC controller flows, if you insert a DFT-inserted OCC controller during integration, the OCC signals of cores containing OCC controllers must be left dangling to be completed by DFT insertion; they cannot be preconnected.

10

Exporting Data to Other Tools

This chapter describes how to export output data from DFT Compiler to other tools, such as TetraMAX ATPG.

This chapter includes the following sections:

Verifying DFT Inserted Designs for Functionality

After DFT insertion, the resulting scan-inserted design is verified for functional equivalence with respect to the nonscan design. This is done to ensure that DFT insertion did not introduce any logic errors. Verification is accomplished by using the Synopsys Formality tool.

The following subsections describe the verification process:

- Verification Setup File Generation
- Test Information Passed to the Verification Setup File
- Script Example
- Formality Tool Limitations

Verification Setup File Generation

By default, Design Compiler synthesis automatically creates a verification setup file in your working directory. The automated setup file has the extension .svf and is named default.svf. This file tracks any design changes that are required for the verification process and assists the Formality tool in compare-point matching and verification.

The automated setup file is stored in binary format.

Use the `set_svf` command to generate a Formality setup information file for efficient compare-point matching in the Formality tool.

The syntax is as follows:

```
set_svf
     file_name
     [-append]
     [-off]
```

Argument Definitions

`file_name`

Specifies the file into which Formality setup information is recorded. You must specify a file name unless the `-off` option is specified.

`-append`

Appends to the specified file. If another Formality setup verification file is already open, then it will be closed before opening the specified file. If `-append` is not used, then `set_svf` overwrites the named file, if it exists.

`-off`

Stops recording Formality setup information to the currently open file. To resume recording into the same file, you must reissue the `set_svf` command with the `-append` option.

Test Information Passed to the Verification Setup File

When you run the `insert_dft` command, the following DFT specific information is recorded in the verification setup file:

- Scan-enable signals are disabled.
- Test modes are disabled wherever they are used (for example, AutoFix/ adaptive scan).
- Constants are passed to the file.
- Core wrapper shift `(wrp_shift)` is disabled.
- The TCK, TMS, and TRST ports of BSD Compiler are held at 0 and the TDO port is not verified.

The setup information is reported in the assumptions summary report.

For more information, see the *Formality User Guide*.

Script Example

Example 10-1 shows you how to use a verification setup file for functionality checking in the Formality tool.

Example 10-1 Formality Script Example For Equivalence Checking

```
#Enable Automatic Setup to Disable Scan/Test
set synopsys_auto_setup true

#Set your verification setup file
set_svf ./my_svf_file

# READ LIBRARIES
foreach file $link_library {read_db $lib}

# Read Reference Design
create_container pre_dft
read_ddc ./outputs/des_unit.pre_dft.ddc
set_top des_unitset_reference_design pre_dft:/WORK/des_unit

# Read Implementation Design
create_container post_dft
```

```
read_ddc ./outputs/des_unit.post_dft.ddc
set_top des_unit
set_implementation_design post_dft:/WORK/des_unit
# Match compare points and Verify
match
verify
```

Formality Tool Limitations

The following features are not supported.

- Internal pins support
- DBIST

Exporting a Design to TetraMAX ATPG

This section describes the steps that take a design from DFT Compiler to TetraMAX ATPG, in which you use the automatic test pattern generation (ATPG) capability to generate test vectors. It has the following subsections:

- Before Exporting Your Design
- Exporting Your Design to TetraMAX ATPG

Before Exporting Your Design

Before exporting your design to TetraMAX ATPG, be aware of the following:

- You need to make sure all design rule violations have been corrected. Use the `dft_drc` command to detect and correct scan design rule violations.
- Designs must have valid scan chains to be recognized by TetraMAX ATPG. Any nonscan sequential cell or capture violation has the potential to lower fault coverage. Use the `report_scan_path -chain all` command to verify that all scan chains are intact before exporting a design.
- All cells on the scan chain must be controllable and observable. This is because controllable-only or observable-only chains identified by DFT Compiler are not written to the STIL procedure file, and TetraMAX ATPG does not recognize them. If you have problems with the scan chain, you must fix them by using DFT Compiler.

- TetraMAX ATPG does not accept designs in which the original source was VHDL and arrays of arrays are used in top-level buses.
- Be aware of dependent slave operation. TetraMAX ATPG reports an S29 warning for circuits and protocols generated by DFT Compiler if the circuit has scan-out lock-up latches inserted by DFT Compiler, and these latches are closed at the end of a cycle. The `dft_drc` command, which runs DFT Compiler design rule checking, similarly reports diverging scan chains for such lock-up latches

Before exporting your design to TetraMAX ATPG, you should also be aware of the differences in which DFT Compiler and TetraMAX ATPG handle your design. These differences are explained in the following sections:

- Support for DFT Compiler Commands in TetraMAX ATPG
- Creating Generic Capture Procedures

Support for DFT Compiler Commands in TetraMAX ATPG

To export a DFT Compiler flow to the TetraMAX ATPG flow, TetraMAX ATPG must translate some DFT Compiler commands into TetraMAX ATPG commands. The TetraMAX ATPG flow supports the following DFT Compiler commands:

- `set_dft_signal`
- `read_test_protocol`

The following DFT Compiler command has no impact on the STIL protocol and is therefore ignored in the DFT Compiler to TetraMAX flow:

- `set_test_assume`

Creating Generic Capture Procedures

DFT Compiler allows you to write out a protocol file with generic capture procedures to be used in TetraMAX ATPG.

The generic capture procedures consist of the following procedures:

- multiclock_capture()
- allclock_capture()
- allclock_launch()
- allclock_launch_capture()

Advantages of Generic Capture Procedures

This is the preferred format for the protocol file to be given to TetraMAX ATPG due to the following advantages:

- The single cycle capture procedure is efficient.
- It matches the event ordering (force PI, measure PO, pulse clock) in the TetraMAX tool without any manual modifications.
- Stuck-at and at-speed ATPG can use a single common protocol file.
- The stuck-at _default_WFT_ WaveformTable is used as a template for modifying the timing of the at-speed WaveformTables.

Writing a Protocol File With Generic Capture Procedures

The criteria for writing out a protocol file with generic capture procedures are as follows:

- To write out protocol files, use the `create_test_protocol` command.

  ```
  create_test_protocol -capture_procedure [single_clock | multi_clock]
  ```

  Choose `multi_clock` to create a protocol file that uses one generic capture procedure for all capture clocks. The default is `multi_clock`.

- You must use preclock strobe timing to write out a single vector generic capture procedure. Otherwise, the capture procedures will have three-vectors.

  The following variables are set to their defaults to define the timing in the protocol file.

  - `$test_default_delay 0`
  - `$test_default_bidir_delay 0`
  - `$test_default_strobe 40`
  - `$test_default_period 100`

  To use preclock strobe timing, the value of $test_default_strobe must be set to be before the time of the leading edge of any of the test clocks that you have defined.

WaveformTables

The STIL procedure file, written out, using the generic capture procedures, has several different WaveformTables:

- _default_WFT_
- _multiclock_capture_WFT_
- _allclock_capture_WFT_

- _allclock_launch_WFT_
- _allclock_launch_capture_WFT_

The timings of these different WaveformTables are identical when they are written out by DFT Compiler.

The WaveformTables are suitable for the following procedures:

- The test_setup macro and load_unload procedures, which use _default_WFT_
- The capture procedures that neither launch nor capture transition fault effects and use _multiclock_capture_WFT_. Generic capture procedures, using the internal clocks also use _multiclock_capture_WFT_ because the PLL pulse trains are generated internally and independently of the external timing.

Observe the following criteria when using the protocol file for at-speed testing:

- The timings in the _allclock_ WaveformTables should be changed to get at-speed transition fault testing on the external clocks. Ensure that you do not change the period or the timings of the reference clocks, or else the PLLs might lose lock. You should change only the rise and fall times of the external clocks.
- Each two-clock transition fault test consists of a launch cycle using _allclock_launch_WFT_ followed by a capture cycle using _allclock_capture_WFT_. The active clock edges of these two cycles should be close to each other. Make sure that the clock leading-edge comes after the all_outputs strobe time, and adjust the time for all values (L, H, T and X) in the _allclock_capture_WFT_ if necessary.
- The _allclock_launch_capture_WFT is used only when launch and capture are caused by opposite edges of the same clock. Note the timing is from the leading edge of the clock to the same clock's trailing edge. However, this timing only occurs in full sequential ATPG and can be ignored in most cases.

Writing a Protocol File With a Multiclock Capture Procedure

You can control multiple clock capture by specifying a single generic capture procedure, called `multiclock_capture`, in an SPF file. This procedure, which enables you to map all capture behaviors irrespective of the number of clocks present, is the default procedure.

In addition to supporting capture operations that contain multiple clocks, this procedure also eliminates the need to manually define a full set of clock-specific capture procedures. The `multiclock_capture` is used for stuck-at ATPG but can be used for transition delay and path delay ATPG if the allclock procedures are not present in the protocol file.

Example 10-2 shows a `multiclock_capture` procedure.

Example 10-2 Example of a multiclock Capture Procedure

```
Procedures {
    "multiclock_capture" {
        W "_multiclock_capture_WFT_";
        C {
            "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
            "all_outputs" = \r165 X;
            "all_bidirectionals" = ZZZ;
        }
        F {
            "i_scan_block_sel[0]" = 1;
            "i_scan_block_sel[1]" = 1;
            "i_scan_compress_mode" = 0;
            "i_scan_testmode" = 1;
        }
        V {
            "_po" = \r168 #;
            "_pi" = \r179 #;
        }
    }
}
```

As is the case with all capture procedures, the single-vector form of `multiclock_capture` requires the timing in the WaveformTable to follow the TetraMAX event order for captures (preclock strobe timing). This means that all input transitions must occur first, all output measures must occur next, and all clock pulses must be defined as the last event.

Example 10-3 shows a `multiclock_capture` WaveformTable.

Example 10-3 Example of a multiclock_capture WaveformTable

```
WaveformTable "_multiclock_capture_WFT_" {
        Period '100ns';
        Waveforms {
            "all_inputs" {
                0 {
                    '0ns' D;
                }
            }
            "all_inputs" {
                1 {
                    '0ns' U;
                }
            }
            "all_inputs" {
                Z {
                    '0ns' Z;
                }
            }
            "all_bidirectionals"{
                T {
```

```
        '0ns' Z;
        '40ns' T;
    }
  }
"all_bidirectionals" {
    L {
        '0ns' Z;
        '40ns' L;
    }
}
"all_outputs" {
    X {
        '0ns' X;
        '40ns' X;
    }
}
"all_outputs" {
    H {
        '0ns' X;
        '40ns' H;
    }
}
"all_outputs" {
    T {
        '0ns' X;
        '40ns' T;
    }
}
"all_outputs" {
    L {
        '0ns' X;
        '40ns' L;
    }
}
"i_scan_clk_sclk2_in" {
    P {
        '0ns' D;
        '45ns' U;
        '55ns' D;
    }
}
"i_scan_clk_sclkd" {
    P {
        '0ns' D;
        '45ns' U;
        '55ns' D;
    }
}
"i_resetin" {
    P {
        '0ns' U;
        '45ns' D;
        '55ns' U;
```

```
                    }
                }
            }
}
```

Writing a Protocol File With a Single Clock Capture Procedure

You can write out a protocol file that uses a single clock capture procedure for all clocks (also known as the legacy three-vector capture procedure) by specifying the `-capture_procedure single_clock` option-argument of the `create_test_protocol` command. Example 10-4 shows a single_clock procedure.

Example 10-4 Example of a single_clock Capture Procedure

```
Procedures {
    "capture_clk_st"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
           "all_outputs" = \r46 X;}
        F {"test_mode" = 0;}
        "forcePI":   V {"_pi" = \r61 #;}
        "measurePO": V {"_po" = \r46 #;}
        "pulse":     V {"clk_st" = P;}
       }
    "capture_pclk"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
           "all_outputs" = \r46 X;}
        F {"test_mode" = 0;}
        "forcePI":   V {"_pi" = \r61 #;}
        "measurePO": V {"_po" = \r46 #;}
        "pulse":     V { "pclk" = P;}
       }
    "capture_rstn"{
        W "_default_WFT_";
        C {"all_inputs" = 0\r59 N 0;
           "all_outputs" = \r46 X; }
        F {"test_mode" = 0;}
        "forcePI":   V {"_pi" = \r61 #;}
        "measurePO": V {"_po" = \r46 #;}
        "pulse":     V {"rstn" = P;}
       }
      }
```

Using Allclock Capture Procedures

The allclock procedures are used for transition delay and path delay ATPG. One set of generic allclock procedures can replace clock specific at-speed capture-launch procedures. DFT Compiler copies the _default_WFT_ WaveformTable to each of the WaveformTables of

the allclock procedure. You need to modify the timing information in the allclock WaveformTables to synchronize with the timing that will be used for the at-speed testing in TetraMAX ATPG.

By default, an allclock procedure applies to a single vector, although it doesn't have to carry the redundant clock parameter. An allclock procedure may reference any WaveformTable for each operation.

For examples of allclock procedures, see Example 10-5, Example 10-6, and Example 10-7.

Example 10-5 Example of Allclock Procedures

```
    "allclock_capture" {
         W "_allclock_capture_WFT_";
         C {
              "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
              "all_outputs" = \r165 X;
              "all_bidirectionals" = ZZZ;
         }
         F {
              "i_scan_block_sel[0]" = 1;
              "i_scan_block_sel[1]" = 1;
              "i_scan_compress_mode" = 0;
              "i_scan_testmode" = 1;
         }
         V {
              "_po" = \r168 #;
              "_pi" = \r179 #;
         }
     }
     "allclock_launch" {
         W "_allclock_launch_WFT_";
         C {
              "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
              "all_outputs" = \r165 X;
              "all_bidirectionals" = ZZZ;
         }
...

     F {
...
         }
         V {
              "_po" = \r168 #;
              "_pi" = \r179 #;
         }
     }
     "allclock_launch_capture" {
         W "_allclock_launch_capture_WFT_";
         C {
              "all_inputs" = 00 \r91 N 111 \r14 0 \r33 N 1 \r32 N;
              "all_outputs" = \r165 X;
              "all_bidirectionals" = ZZZ;
```

```
        }
        F {
            "i_scan_block_sel[0]" = 1;
            "i_scan_block_sel[1]" = 1;
            "i_scan_compress_mode" = 0;
            "i_scan_testmode" = 1;
        }
        V {
            "_po" = \r168 #;
            "_pi" = \r179 #;
        }
    }
```

In Example 10-6, a _default _WFT timing is copied to an `allclock _capture` procedure.

Example 10-6 allclock_capture WaveformTable With Default Timing

```
WaveformTable "_allclock_capture_WFT_" {
        Period '100ns';
        Waveforms {
 "all_inputs" {
                0 {
                    '0ns' D;
                }
            }
...
"all_bidirectionals" {
                L {
                    '0ns' Z;
                    '40ns' L;
                }
            }
...
"all_outputs" {
                L {
                    '0ns' X;
                    '40ns' L;
                }
            }
            "i_HEAD_PIPE_REG_CLK" {
                P {
                    '0ns' D;
                    '45ns' U;
                    '55ns' D;
                }
            }
...
```

Example 10-7 shows an updated WaveformTable timing copied to an `allclock_capture` procedure.

Example 10-7 allclock_capture Procedure WaveformTable With Updated Timing

```
WaveformTable "_allclock_capture_WFT_" {
```

```
        Period '10ns';
        Waveforms {
 "all_inputs" {
                0 {
                    '0ns' D;
                }
            }
...
"all_bidirectionals" {
                L {
                    '0ns' Z;
                    '4ns' L;
                }
            }
...
"all_outputs" {
                L {
                    '0ns' X;
                    '4ns' L;
                }
            }
            "i_HEAD_PIPE_REG_CLK" {
                P {
                    '0ns' D;
                    '5ns' U;
                    `6ns' D;
                }
            }
...
```

Limitation in the Generic Capture Procedures

A limitation associated with the capture procedures is the following:

- The _default_WFT timing is copied to the allclock WaveformTables (launch_WFT, capture_WFT, launch_capture_WFT). You need to modify these WaveformTables with the correct timing before running at-speed ATPG (transition delay, path delay).

For more information on generic capture procedures, see the “STIL Procedure Files” topic in the *TetraMAX Online Help*.

Exporting Your Design to TetraMAX ATPG

The following steps work only if your design is completely chain-routed and you have successfully validated the scan chains by inspecting the outputs generated by the `dft_drc` and `report_scan_path -chain all` commands.

To export your design to TetraMAX ATPG, do the following:

1. Before starting any work with DFT Compiler, including scan insertion, set the test timing variables to the values specified by your ASIC vendor. If your ASIC vendor does not have specific requirements, the following defaults achieve the best results from TetraMAX ATPG:

   ```
   dc_shell> set_app_var test_default_delay 0
   dc_shell> set_app_var test_default_bidir_delay 0
   dc_shell> set_app_var test_default_strobe 40
   dc_shell> set_app_var test_default_period 100
   ```

   These are the default settings; you do not need to add them to your script.

2. Identify the netlist format that you are exporting to TetraMAX ATPG, using the `test_stil_netlist_format` environment variable. The syntax is

   ```
   set_app_var test_stil_netlist_format db |verilog | vhdl
   ```

   It is important that you identify the netlist format because TetraMAX ATPG handles difficult names, such as those for buses and escaped characters in Verilog, according to the language used in the netlist.

3. Guide netlist formatting by setting the environment variables that affect how designs are written out.

   Note:
   Set the environment variables before you write out the netlist or STIL procedure file.

   For example, if you want vectored ports in your Verilog design to be bit-blasted, set the `verilogout_single_bit` variable to true. For more information about environment variables that affect how designs are written out, see the *HDL Compiler for Verilog User Guide* or the *HDL Compiler for VHDL User Guide*.

4. Check for design rule violations by entering the `dft_drc` command:

   ```
   dc_shell> dft_drc
   ```

   For information on these commands, see Chapter 5, “Pre-Scan Test Design Rule Checking.”

   Fix any design rule violations. Repeat the `dft_drc` command until no design rule violations are found.

5. Write out the netlist. For example, to write out a Verilog netlist, use the following command:

   ```
   dc_shell> write -format verilog -hierarchy \
                 -output filename.v
   ```

6. Write out the test protocol file, using the STIL protocol. Enter the following command:

   ```
   dc_shell> write_test_protocol -output protocol.spf
   ```

All of the information that TetraMAX ATPG requires to create ATPG vectors, such as scan pins and constrained signals, is found in the STIL procedure file.

SCANDEF-Based Reordering Flow

DFT Compiler can generate SCANDEF information that describes scan chain ordering requirements. You can use this SCANDEF information in the IC Compiler tool to perform scan chain reordering and to fix timing violations based on physical information, or you can perform scan chain reordering using other place-and-route tools.

This section has the following subsections:

Overview

Additional routing is required when you add scan chains to designs. To meet die size and timing requirements, you need to reduce the amount of routing as much as possible. One way to do this is to reorder scan chains.

The physical scan chain implementation in the IC Compiler tool consists of the ability, first, to read and check the integrity of the SCANDEF information and, second, to perform physical scan chain optimization.

Reordering of scan chains in place-and-route tools requires information about the scan chains that exist in the design. This is because there can be aspects of the scan chains, such as lock-up latches, clock-mixing, or user-defined scan paths or scan segments, that must be considered during reordering. This information can be generated from DFT Compiler in SCANDEF form. The SCANDEF information specifies a list of “stub” chains that can be reordered by another tool. The boundaries of these stub chains can be I/O ports,

lock-up latches, or multiplexers. Therefore, the stub chains are not usually identical to scan chains, so the number of scan chains (stubs) defined in the SCANDEF information does not necessarily match the chain count specified during scan chain architecting.

Generation of SCANDEF Information

The tasks you must perform to generate SCANDEF information are described in the following sections:

- Reading and Compiling the Design
- Specifying the Scan Configuration
- Writing Out the SCANDEF Information

Reading and Compiling the Design

You must first read your design into dc_shell. If your design is an HDL source file, you must compile it, using either the `compile` or the `compile -scan` command. For more information, see the Design Compiler documentation and Chapter 5, “Pre-Scan Test Design Rule Checking.”

Specifying the Scan Configuration

Make sure you properly set up the intended scan design by using the `set_scan_configuration` command. With this command, you describe such items as clock mixing, scan style, and the number of scan chains.

For more information on the `set_scan_configuration` command, see Chapter 6, “Architecting Your Test Design.”

Writing Out the SCANDEF Information

In the step before place and route, the `insert_dft` command generates SCANDEF information for the place-and-route tools. To generate the SCANDEF information,

1. Execute the `insert_dft` command.
2. If you are using Design Compiler topographical mode, perform an incremental compile with the `compile_ultra -incremental -scan` command.
3. Execute the `change_names` command with the necessary name rules.
4. Generate the SCANDEF information with the `write_scan_def` command:

```
dc_shell> write_scan_def -output filename.scandef
```

This command writes out the SCANDEF information to the specified file name. It also annotates the current design in memory with the SCANDEF information.

5. Write out the required output files, depending on your layout flow:

   - If you are using the IC Compiler tool, write out the design with the `write -format ddc` command. The resulting .ddc file contains the SCANDEF information. the IC Compiler tool does not need the SCANDEF file from the previous step, but you can use the file for reference.
   - If you are using a layout tool other than the IC Compiler tool, write out the design with the `write -format verilog` command. The layout tool also needs the SCANDEF file from the previous step.

Note:

You must execute the `write_scan_def` command to annotate the scan ordering information onto the current design, even when using the .ddc flow.

Next, run your place-and-route tools to generate a new scan chain routing order.

Generating SCANDEF Information for Typical Flows

The complete flow capturing the generation of SCANDEF information is as follows:

```
read_file -format ddc top.ddc
current_design top
set_scan_configuration -style multiplexed_flip_flop
set_dft_signal -view existing_dft -type ScanClock \
  -port clock -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
change_names ...
write_scan_def -output my_def.scandef
write_test_protocol -output test_mode.spf
write -format verilog -hierarchy -output top.v
write -format ddc -hierarchy -output top.ddc
```

Generating SCANDEF Information for Hierarchical Flows

The following flow is used at the top level to generate SCANDEF information in hierarchical flows:

```
## Reading top-level design
read_verilog top.v
## Reading test-models
read_test_model -format ddcblock.ctlddc
```

```
current_design top
link
set_scan_configuration -style multiplexed_flip_flop
set_dft_signal -view existing_dft -type ScanClock \
  -port clock -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
change_names ...
write_scan_def -output my_def.scandef
write_test_protocol -output test_mode.spf
## Removing the test-model blocks so that Design Compiler
## does not write empty modules for the blocks
remove_design block
write -format verilog -hierarchy -output top.v
write -format ddc -hierarchy -output top.ddc
```

Hierarchical SCANDEF Flow Support

Hierarchical Scan Synthesis flows represent a DFT-inserted core by a core test language model (CTL model). You can use CTL models instead of the netlist representation of the subblocks during chip-level scan integration. In such cases, the SCANDEF information represents the scan segments within these subblocks as black-box segments. This representation permits repartitioning of the segment as a whole but does not permit the reordering of cells within the segments. This feature allows DFT Compiler to write the scan cells of the subblocks, so the tool can reorder the subblock scan cells along with the chip-level scan cells.

DFT Commands

Use the `write_scan_def` command with the `-expand_elements` option at the chip level to write SCANDEF information that includes the scan cells inside cores.

```
dc_shell> write_scan_def -expand_elements list_of_instances
```

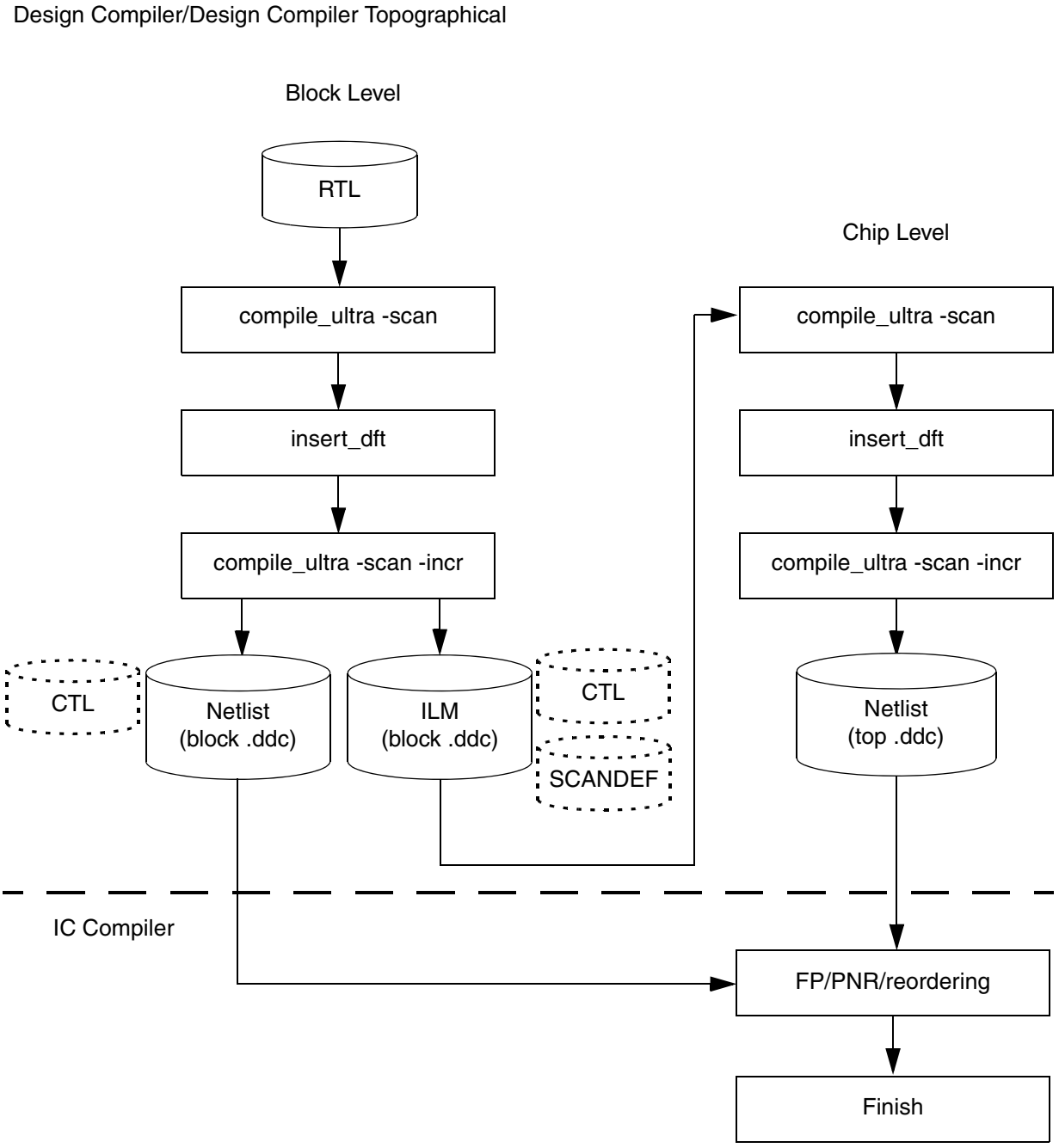
The `-expand_elements` option is used to specify a list of instances abstracted into CTL models. The scan segments of these models must be treated as “flat,” fully visible segments when generating the chip-level SCANDEF information.

The default behavior is to treat all instances abstracted into CTL models as black boxes. This means that they are represented in the SCANDEF information by the `BITS` construct.

Hierarchical SCANDEF Flows

Figure 10-1 shows the hierarchical SCANDEF flow.

Figure 10-1 Hierarchical SCANDEF Flow



In this flow, the block is logically optimized and DFT-complete before chip-level integration. Test information of the block is abstracted in CTL models. (You can also choose to abstract

the block into an ILM representation.) During chip integration, you can choose to use the full netlist or the ILM representation of the block to perform optimizations.

The block CTL model can be used to insert chip-level DFT without disturbing the block-level DFT structures in a hierarchical scan synthesis flow. Using the block-level CTL model or the complete netlist, the chip-level SCANDEF information can be generated by treating the block-level scan segments either as black-box segments or as flat, fully visible segments. The IC Compiler tool uses the full netlist representation of the block to perform global physical optimizations. You can choose which blocks are flattened at the chip-level SCANDEF generation by using the `-expand_elements` option of the `write_scan_def` command.

The following hierarchical SCANDEF flows are supported:

- Flow 1 – The block-level CTL models are used for DFT insertion at the chip level.
- Flow 2 – The complete block-level netlists (.ddc files) along with the block-level SCANDEF information are used for DFT insertion at the chip level.
- Flow 3 – The complete block-level netlists (.ddc files) are used for DFT insertion at the chip level.

Flow 1

Block level

```
read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hierarchy
write_scan_def -output sub1.scandef
write_test_model -output sub1.ctlddc -format ddc
write -f verilog -hierarchy -output sub1.v
write_test_protocol -output sub1_scan.spf
```

Chip level

```
read_verilog ./top.v
read_ddc sub1.ctlddc
read_ddc sub2.ctlddc
current_design TOP
link
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
```

```
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
write_scan_def -output top.scandef \
    -expand_elements [list inst_sub1 inst_sub2]
```

Note that `inst_sub1` and `inst_sub2` are the instance names of modules sub1 and sub2, respectively.

Flow 2

Block level

```
read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
     -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hierarchy
write_scan_def -output ./sub1.scandef
write -format ddc -output ./sub1.ddc -hierarchy
```

Chip level

```
read_verilog ./top.v
read_ddc sub1.ddc
read_ddc sub2.ddc
current_design TOP
link
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
write_scan_def -output top.scandef \
    -expand_elements [list inst_sub1 inst_sub2]
```

Note that `inst_sub1` and `inst_sub2` are the instance names of modules sub1 and sub2, respectively.

Flow 3

Block level

```
read_ddc sub1_test_ready.ddc
current_design sub1
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
compile_ultra -incremental -scan
dft_drc
change_names -rules verilog -hierarchy
write -format ddc -output ./sub1.ddc -hierarchy
```

Chip level

```
read_verilog ./top.v
read_ddc sub1.ddc
read_ddc sub2.ddc
current_design TOP
link
set_scan_configuration -chain_count 10
set_dft_signal -port clock -type Scanclock -view existing_dft \
    -timing [list 45 55]
create_test_protocol
dft_drc
preview_dft
insert_dft
write_scan_def -output top.scandef\
    -expand_elements [list inst_sub1 inst_sub2]
```

Note that `inst_sub1` and `inst_sub2` are the instance names of modules sub1 and sub2, respectively.

Limitations

Note the following limitations of hierarchical SCANDEF flows:

- The expanded hierarchical SCANDEF is based solely on the unexpanded hierarchical SCANDEF; that is, the expanded SCANDEF can only include cells that are represented by the `BITS` construct in the unexpanded version.
- In Flow 1, where you use block-level test models for chip-level DFT insertion, you need to write SCANDEF information at the block level before writing out a CTL model.
- In Flow 3, where you use the block-level netlists for chip-level DFT insertion and do not write SCANDEF information before writing the netlist, the block-level scan constraints, such as `scan_path` or `scan_group`, are not considered when expanding.

Impact of DFT Configuration Specification on SCANDEF Generation

The impact of the specified DFT configuration on the final SCANDEF information is shown in the following test cases.

Consider a design with a set of scan flip-flops, f1, f2, f3, f4.

Case 1

```
set_scan_path chain1 -exact_length 2
```

The scan elements in this scan chain can be swapped with those of other scan chains. The SCANDEF information is as follows:

```
- CHAIN1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
           f2 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so ;
+ PARTITION CLOCK1_45_55
```

Case 2

```
set_scan_path chain1 \
  -head_elements {f1} -tail_elements {f3} -include_elements {f4}
```

The SCANDEF information is as follows:

```
- CHAIN1
+ START f1 Q
+ FLOATING f4 ( IN SDI ) ( OUT Q )
+ STOP f3 SD1 ;
```

In this case, because we have some include elements, the “def” scan chain does not have the `PARTITION` attribute.

Case 3

```
set_scan_path chain1 -include_elements {f4} -exact_length 2
```

This case is a combination of case 1 and case 2. The element f4 must belong to the scan chain and cannot be used for partitioning. Elements that are added are swapped with other scan chains. However, due to the DEF format limitation, consider all the elements as `INCLUDE` constructs. Therefore, if f2 is the added element, the SCANDEF is as follows:

```
- CHAIN1
+ START PIN test_si
+ FLOATING f4 ( IN SDI ) ( OUT Q )
           f2 (IN SDI ) ( OUT Q )
+ STOP PIN test_so ;
```

Case 4

```
set_scan_path chain1 -ordered_elements {f2 f3}
```

The SCANDEF information is as follows:

```
- CHAIN1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
           f4 ( IN SDI ) ( OUT Q )
+ ORDERED f2 ( IN SDI ) ( OUT Q )
           f3 ( IN SDI ) ( OUT Q )
+ PARTITION CLOCK1_45_55
+ STOP PIN test_so ;
```

Within a stub chain, all FLOATING elements are listed first, followed by all ORDERED elements.

Case 5

```
set_scan_path chain1 -include_elements {f1 f2} -complete
```

In this case, the scan chain is not modified. The SCANDEF information is as follows:

```
- CHAIN1
+ START PIN test_si
+ ORDERED f1 ( IN SDI ) ( OUT Q )
           f2 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so ;
```

Case 6

```
set_scan_group group1 -include_elements {f1 f2} -serial_routed true
set_scan_path chain1 -include_elements {group1 f3 f4}
```

The SCANDEF is as follows:

```
- CHAIN1
+ START PIN test_si
+ ORDERED f1 ( IN SDI ) ( OUT Q )
           f2 ( IN SDI ) ( OUT Q )
+ FLOATING f3 ( IN SDI ) ( OUT Q )
            f4 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so
```

Case 7

```
set_scan_group group1 -include_elements {f1 f2} -serial_routed false
set_scan_path chain1 -include_elements {group1 f3 f4 f5 f6}
```

Consider that `test_si` and `test_so` signify the scan-in and the scan-out ports of the scan chain, chain1. Then, two subchains are considered and the SCANDEF information is as follows:

```
- SUB_GP1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
            f2 ( IN SDI ) ( OUT Q )
+ STOP     f3 SDI ;

-SUB_GP2
+ START f3 q ;
+ FLOATING f4 ( IN SDI ) ( OUT Q )
            f5 ( IN SDI ) ( OUT Q )
            f6 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so;
```

Case 8

Consider a top-level design with some flip-flops and a test model for a block—block1 with scan ports `test_si1` and `test_so1` and a scan chain of length 20. The SCANDEF is as follows:

```
- SUB_GP1
+ START PIN test_si
+ FLOATING f1 ( IN SDI ) ( OUT Q )
            f2 ( IN SDI ) ( OUT Q )
            block1 (IN test_si1) (OUT test_so1 ) ( BITS 20 )
+ PARTITION pll_clk3_clk_45_45
+ STOP f3 SDI ;
```

The value of the `BITS` parameter indicates the length of the scan chain in the CTL model. This length is maintained during partitioning.

Case 9

Consider a design in which the scan chain begins with a scan port `test_si` and ends at scan port `test_so`. When you run `insert_level_shifters`, consider the level shifters to be inserted at the test ports. The SCANDEF information is as follows:

```
- SUB_GP1
+ START LS_test_si Y
+ FLOATING f1 ( IN SDI ) ( OUT Q )
            f2 ( IN SDI ) ( OUT Q )
+ PARTITION pll_clk3_clk_45_45
+ STOP LS_test_so A;
```

Here, `LS_test_si` and `LS_test_so` are the inserted level shifters. So, the chain begins at the output of the `LS_test_si level` shifter and ends at the input of the `LS_test_so level` shifter.

Case 10

Consider a design in which the registers f1, f2, and f3 have been inferred as a shift register. The SCANDEF information is as follows:

```
- CHAIN1
+ START PIN test_si
+ ORDERED f1 ( IN SDI ) ( OUT Q )
          f2 ( IN D ) ( OUT Q )
          f3 ( IN D ) ( OUT Q )
+ FLOATING f4 ( IN SDI ) ( OUT Q )
+ STOP PIN test_so
```

The shift register is written using an `ORDERED` construct.

PARTITION Label Naming Convention

In a SCANDEF file, the `PARTITION` keyword specifies the partition name for a stub chain. Partition names describe how scan cells can be moved between different stub chains. Scan cells can be moved between partitions that share the same name. If a stub chain does not have a `PARTITION` keyword, its scan cells cannot be moved into other chains.

The partition naming convention for the different scenarios is as follows:

- MUX-D style without multivoltage

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_ <*launch_time_of_last_state_of_last_segment_of_chain*>

- MUX-D style with multivoltage

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_ <*launch_time_of_last_state_of_last_segment_of_chain*>_ <*voltage_domain*>_<*power_domain*>

- LSSD style when `test_lssd_no_mix` is FALSE

  SNPS_LSSD_<*clock_name*>_<*master_clock_name*>_<*slave_clock_name*>_ <*voltage_domain*>_<*power_domain*>

- LSSD style when `test_lssd_no_mix` is TRUE

  SNPS_LSSD_<*clock_name*>_<*chain_system_clock_name*>_<*master_clock_name*>_ <*slave_clock_name*>_<*voltage_domain*>_<*power_domain*>

- LSSD style with X-chains

  LSSD_X_<*clock_name*>_<*master_clock_name*>_<*slave_clock_name*>_ <*voltage_domain*>_<*power_domain*>

- Scan-enabled LSSD style without multivoltage

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_
  <*launch_time_of_last_state_of_last_segment_of_chain*>

- Scan-enabled LSSD style with multivoltage

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_
  <*launch_time_of_last_state_of_last_segment_of_chain*>_
  <*voltage_domain*>_<*power_domain*>

- Multiple test-mode SCANDEF generation

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_
  <*launch_time_of_last_state_of_last_segment_of_chain*>_
  M1[*_M2_...additional_modes*]

- Multiple test-mode SCANDEF generation when `test_enable_multi_mode_scan_def` is `true`

  <*clock_name*>_<*capture_time_of_first_state_of_first_segment_of_chain*>_
  <*launch_time_of_last_state_of_last_segment_of_chain*>_<*voltage_domain*>_
  <*power_domain*>_*M1*[*_M2_...additional_modes*]

For wrapper chains, WRPSI_, WRPSO_ and WRPS_ are the corresponding keywords used to represent the different wrapper chains.

If you specify the `-include_elements` option in a `set_scan_path` command for a scan chain, that chain in the SCANDEF information does not have the `PARTITION` label because scan chain membership has higher precedence than repartitioning.

Support for Other DFT Features

The following DFT features are supported:

- Standard scan and compressed scan
- Multiple test-mode scan
- Internal pins
- Memories with test models
- Multivoltage support
- Hierarchical flows (with test models)
- PLL flows

- Core wrapping
- Shift registers

Limitations With SCANDEF Generation

SCANDEF is not supported for the following:

- BSD Compiler
- Scan extraction flows that have combinational logic between two adjacent scan flip-flops