cadence®

Encounter® Test: Guide 5: ATPG

Product Version 12.1.101
February 2013

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Product Encounter® Test and Diagnostics contains technology licensed from, and copyrighted by:

1. IBM Corporation, and is © 1994-2002, IBM Corporation. All rights reserved. IBM is a Trademark of International Business Machine Corporation;.
2. The Trustees of Indiana University and is © 2001-2002, the Trustees of Indiana University. All rights reserved.
3. The University of Notre Dame and is © 1998-2001, the University of Notre Dame. All rights reserved.
4. The Ohio State University and is © 1994-1998, the Ohio State University. All rights reserved.
5. Perl Copyright © 1987-2002, Larry Wall

Associated third party license terms for this product version may be found in the `README.txt` file at downloads.cadence.com.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Contents

List of Tables

List of Figures

Preface

About Encounter Test and Diagnostics

Encounter® Test uses breakthrough timing-aware and power -aware technologies to enable customers to manufacture higher-quality power-efficient silicon, faster and at lower cost. Encounter Diagnostics identifies critical yield-limiting issues and locates their root causes to speed yield ramp.

Encounter Test is integrated with Encounter RTL Compiler global synthesis and inserts a complete test infrastructure to assure high testability while reducing the cost-of-test with on-chip test data compression.

Encounter Test also supports manufacturing test of low-power devices by using power intent information to automatically create distinct test modes for power domains and shut-off requirements. It also inserts design-for-test (DFT) structures to enable control of power shut-off during test. The power-aware ATPG engine targets low-power structures, such as level shifters and isolation cells, and generates low-power scan vectors that significantly reduce power consumption during test. Cumulatively, these capabilities minimize power consumption during test while still delivering the high quality of test for low-power devices.

Encounter Test uses XOR-based compression architecture to allow a mixed-vendor flow, giving flexibility and options to control test costs. It works with all popular design libraries and automatic test equipment (ATE).

Typographic and Syntax Conventions

The Encounter Test library set uses the following typographic and syntax conventions.

- Text that you type, such as commands, filenames, and dialog values, appears in Courier type.

  Example: Type `build_model -h` to display help for the command.

- Variables appear in Courier italic type.

  Example: Use `TB_SPACE_SCRIPT=`*`input_filename`* to specify the name of the script that determines where Encounter Test binary files are stored.

- Optional arguments are enclosed in brackets.

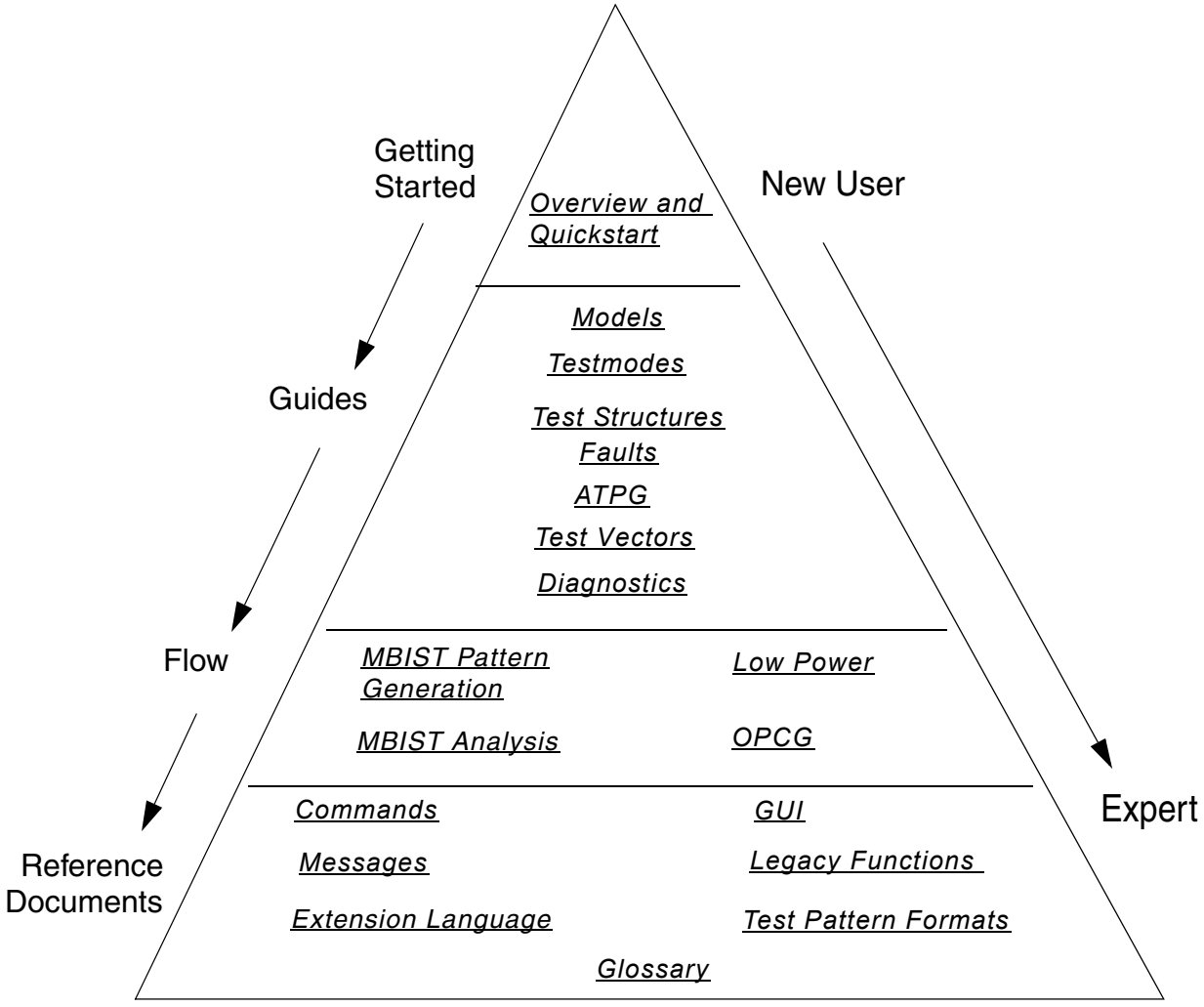
Example: `[simulation=gp|hsscan]`

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable list boxes, appear in Helvetica italic type.

  Example: Select *File - Delete - Model* and fill in the information about the model.

Encounter Test Documentation Roadmap

The following figure depicts a recommended flow for traversing the documentation structure.



Getting Help for Encounter Test and Diagnostics

Use the following methods to obtain help information:

1. From the *<installation_dir>*`/tools/bin` directory, type `cdnshelp` at the command prompt.
2. To view a book, double-click the desired product book collection and double-click the desired book title in the lower pane to open the book.

Click the *Help* or *?* buttons on Encounter Test forms to navigate to help for the form and its related topics.

Refer to the following in the *Encounter Test: Reference: GUI* for additional details:

- "Help Pull-down" describes the *Help* selections for the Encounter Test main window.
- "View Schematic Help Pull-down" describes the Help selections for the Encounter Test View Schematic window.

Contacting Customer Service

Use the following methods to get help for your Cadence product.

- Cadence Online Customer Support

  Cadence online customer support offers answers to your most common technical questions. It lets you search more than 40,000 FAQs, notifications, software updates, and technical solutions documents that give step-by-step instructions on how to solve known problems. It also gives you product-specific e-mail notifications, software updates, service request tracking, up-to-date release information, full site search capabilities, software update ordering, and much more.

  Go to http://www.cadence.com/support/pages/default.aspx for more information on Cadence Online Customer Support.

- Cadence Customer Response Center (CRC)

  A qualified Applications Engineer is ready to answer all of your technical questions on the use of this product through the Cadence Customer Response Center (CRC). Contact the CRC through Cadence Online Support. Go to http://support.cadence.com and click the *Contact Customer Support* link to view contact information for your region.

- IBM Field Design Center Customers

Contact IBM EDA Customer Services at 1-802-769-6753, FAX 1-802-769-7226. From outside the United States call 001-1-802-769-6753, FAX 001-1-802-769-7226. The e-mail address is edahelp@us.ibm.com.

Encounter Test And Diagnostics Licenses

Refer to "Encounter Test and Diagnostics Product License Configuration" in *Encounter Test: Release: What's New* for details on product license structure and requirements.

Using Encounter Test Contrib Scripts

The files and Perl scripts shipped in the `<ET installation path>/ etc/tb/ contrib` directory of the Encounter Test product installation are not considered as "licensed materials". These files are provided AS IS and there is no express, implied, or statutory obligation of support or maintenance of such files by Cadence. These scripts should be considered as samples that you can customize to create functions to meet your specific requirements.

What We Changed for this Edition

The section *Producing True-Time Vectors from OPCG Logic* has been moved to the new flow document *Encounter Test: Flow: OPCG.*

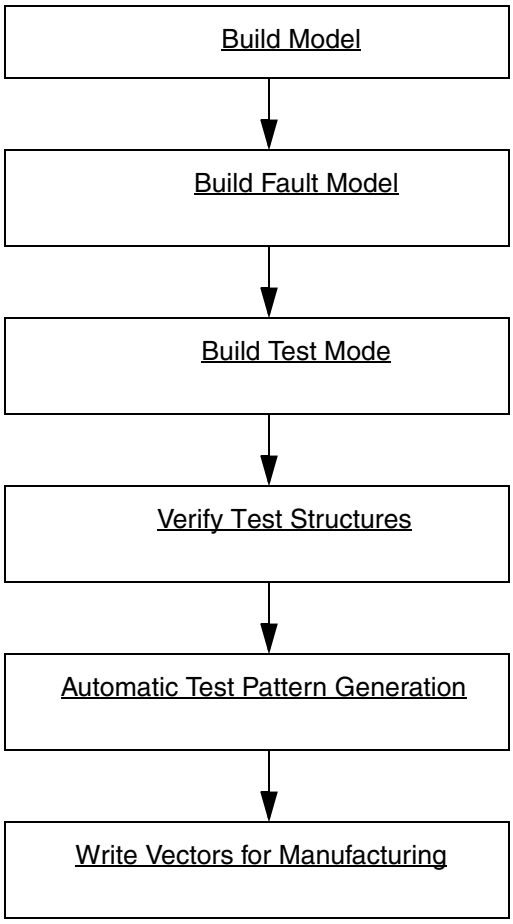
1

Introduction to Automatic Test Pattern Generation

Test pattern generation is the process of determining what stimuli to apply to a design to demonstrate the design's correct construction. Application of these test vectors is used to prove the design contains no manufacturing induced defects. The test vectors may be either automatically generated (by Encounter Test), or they can be manually generated.

The following figure shows where the test pattern generation fits in a typical Encounter Test flow.

Figure 1-1 Encounter Test Process Flow



Encounter Test automatic test pattern generation (ATPG) supports designs with compression and low power logic in a static, untimed, or timed environment. The following types of tests are supported:

- Scan Chain - refer to "Scan Chain Tests" on page 28
- Logic - refer to "Logic Tests" on page 29
  - Static
  - Dynamic (with or without SDF timings)
- Path - refer to "Path Tests" on page 30
- IDDq - refer to "IDDq Tests" on page 30

- Driver and Receiver - refer to Parametric Test in the *Encounter Test: Reference: Legacy Functions*
- IO Wrap tests
- IEEE 1149.1 JTAG, Boundary Scan Verification patterns
- Core tests - refer to "Core Tests" on page 31
- Low power tests - refer to Creating Low Power Tests in the *Encounter Test: RAK: Low Power* for more information.
- Commit test patterns - refer to "Committing Tests" on page 245

In addition to the above-mentioned types of tests, Encounter Test provides the following features:

- Compacting and manipulating test patterns
- Compressing test patterns
- Simulating and fault grading of test patterns
- Fault sub-setting
- Generating Low Power Tests

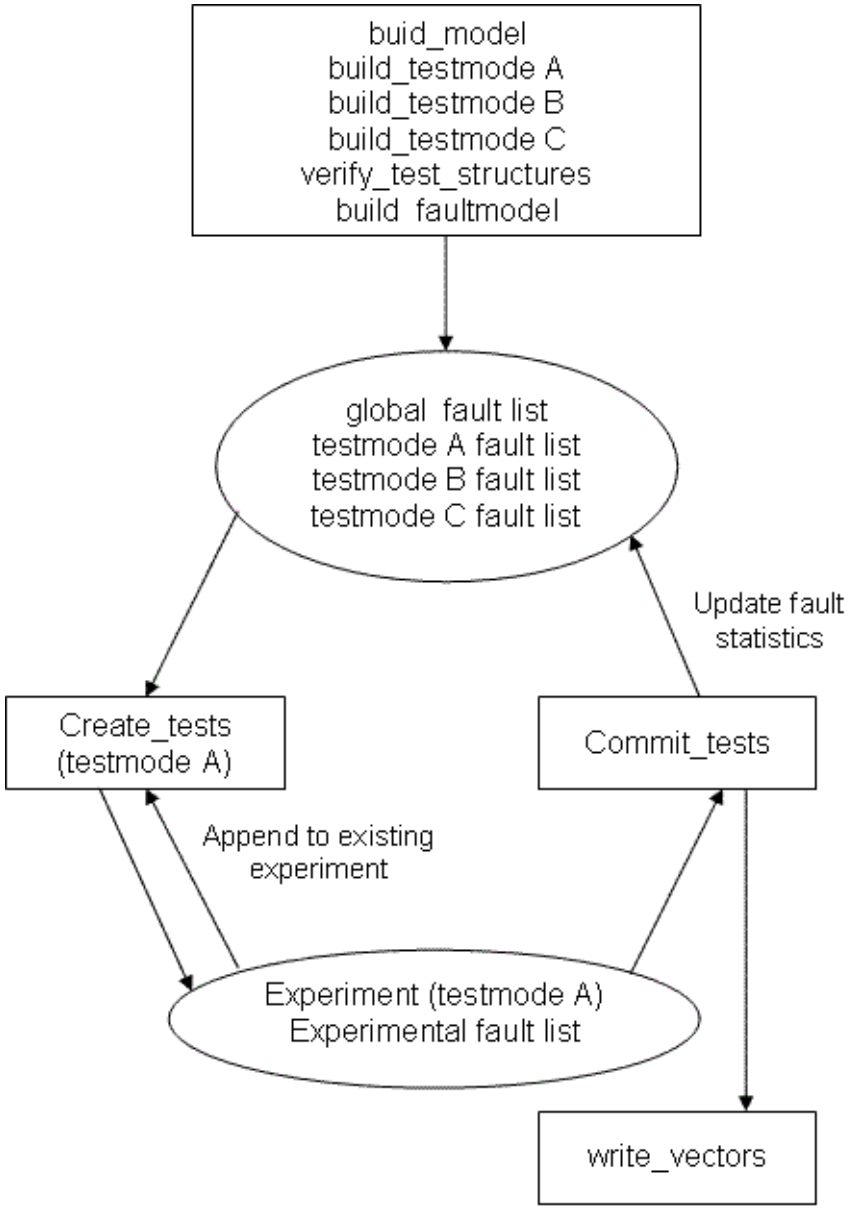
It is recommended that you perform Test Structure Verification (TSV) to verify the conformance of a design to general design guidelines. Non conformance to these guidelines result in poor test coverage or invalid test data.

ATPG Process Overview

The prerequisites to automatic test pattern generation (ATPG) in Encounter Test are building the logic model, a test mode, and a fault model. A fault is selected from the Fault List and sent to the test generation engine. Based on the input constraints of the test mode and optional input parameters, the engine either generates a test for the fault, classifies it as untestable, or exits on reaching process limits.

The following figure shows this iterative process.

Figure 1-2 ATPG flow



Generated tests are passed to the active compaction step and accumulated. Test compaction merges multiple test patterns into one if they do not conflict with a user selectable effort.

Note: If the generated tests are applied at different oscillator frequencies, the test compaction function does not produce any warning message but applies all the tests at the same oscillator frequency.

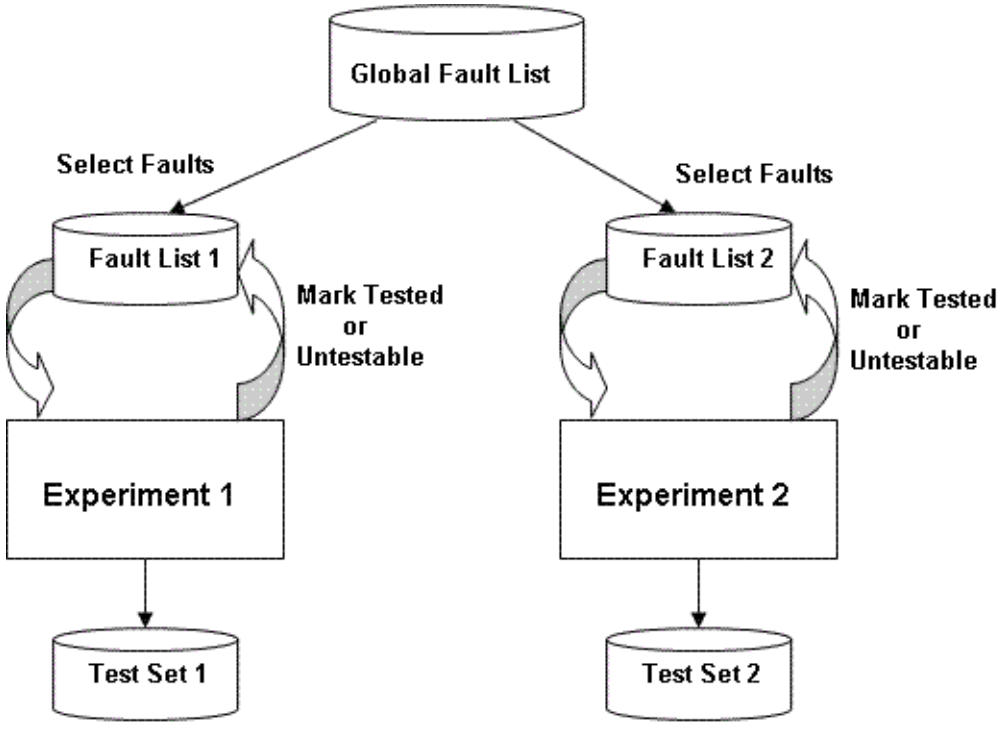
When the test group reaches certain limit, such as 32 or 64 vectors, the tests are passed to the fault simulator that runs a good fault-free and a faulty machine simulation to predict the output states and identify tested faults. The tested faults are marked off in the fault list. A fault is considered to be tested when the good machine value differs from the fault machine value at a valid measure point.

The results of test pattern generation, active compaction, and fault simulation are collectively called an experiment. An experiment can be saved or committed if the user is satisfied with the pattern and test coverage results (refer to "Committing Tests" on page 245 for more information). It can be appended to, as in top-off or add-on patterns, or it can be discarded and started from scratch. More complex procedures involving multiple fault lists, test modes, experiments, and cross mark-offs of faults can be done with this type of process flow.

This process repeats till the unprocessed fault list is empty or user-defined process limits are reached. Encounter Test then reports a summary of the test generation process with the number of test sequences and the test coverages achieved.

The following figure shows the fault list processing:

Figure 1-3 Fault List Processing



Notes:

1. To perform Test Generation using the graphical interface, refer to "ATPG Pull-down" in the *Encounter Test: Reference: GUI*.
2. Also refer to "Static Test Generation" on page 57.

Additional Pattern Compaction

Pattern compaction is an optional step that is useful for reducing test pattern count even further than test compaction. It repeats the fault simulation step by processing the test patterns in a sorted or reverse order from how they were first simulated. The faults tested by the experiment are first reset, and then as faults are tested, subsequent patterns may fail to test new faults and are then discarded.

General Types of Tests

Encounter Test supports the following various types of tests:

Scan Chain Tests

These tests are used to verify the correct operation of scan chains. Since most scan based test vectors presume the correct operation of the scan chains, it is important (for diagnostic purposes) to first ensure that scan chains are working before the application of any test vectors that presume they are working. The Encounter Test stored pattern test generation application can automatically generate scan chain tests to check for the correct operation of scan chains. If the design is operating in a level sensitive scan design (LSSD) mode, a scan chain LSSD flush test may also be generated automatically.

These scan chain tests may be static, dynamic, or dynamic timed. See "Test Vector Forms" on page 254 for more information on these test formats.

You can generate scan chain tests for designs using test compression, which will verify the correct operation of the spreader, compactor, channels, and masking logic.

The scan test walks each scannable flop through all possible transitions by loading an alternating sequence (00110011?.), and simulating several pulses of the shift clock(s).

- LSSD Flush Test

  For Level Sensitive Scan Design (LSSD) designs, it is possible to turn on all the shift A clocks and all of the shift B clocks simultaneously. In this state of the design, any logic

values placed on the scan inputs will “flush” through to the scan output pins of the respective scan chains. The LSSD flush test generated by Encounter Test sets all scan input pins to zero and lets that value flush through to the scan outputs. Then all scan input pins are set to one and that value is let flush through to the scan outputs. Finally, all the scan input pins are set back to zero.

LSSD flush tests are sometimes used to screen for different speed chip samples. Since the scan chain usually traverses every portion of the chip, the LSSD flush test may be a reasonable gauge of the overall chip speed.

A LSSD flush test is not generated if Logic Test Structure Verification (TSV) has not been run. TSV tests determine whether a LSSD flush test can be applied. A LSSD flush test cannot be applied if:

- Either the shift A or shift B clocks are chopped
- The scan chain contains one or more edge-sensitive storage elements (flip-flops).
- The shift clocks are ANDed with other shift clocks which could result in unpredictable behavior when they are all ON.

Logic Tests

These tests are used to verify the correct operation of the logic of the chip. This is not to be confused with functional testing. The objective of the logic tests is to detect as many manufacturing defects as possible.

Logic tests can be static or dynamic (with or without timings). You can specify the timings through a clock constraint file (refer to “Clock Constraints File” on page 129 for more information) or an SDF file.

See “Test Vector Forms” on page 254 for additional detail on these test formats.

Static logic tests are the conventional mainstream logic tests. They detect common defects such as stuck-at and shorted net defects. They may also detect open defects. Encounter Test does not target the CMOS open defects, but stuck-at fault tests detect most of the CMOS open defects.

Dynamic tests are used to detect dynamic, or delay types of defects. Dynamic tests specify certain timed events, and this format can be applied to test patterns targeted for static faults. The converse is not supported; Encounter Test does not allow static-formatted tests when dynamic faults are targeted.

Static tests for scan designs have the following general format:

```
Scan_Load
Stim_PI
Pulse one or more clocks
Measure_PO (optional)
Scan_Unload
```

In general, dynamic tests will contain the following events:

```
Scan_Load
Stim_PI
```

Dynamic events:

```
Pulse launch clock
Pulse capture clock  ( this the at speed, timed, quickly)
Measure_PO (optional)
Scan_Unload
```

The dynamic events section is the only part of the test that can be applied quickly (at speed). The other events in the test are applied slowly.

The example pattern format mentioned above represents a typical delay test. Other delay test formats can also be produced.

Path Tests

Path tests produce a transition along an arbitrary path of combinational logic. The paths to be tested may be specified using the `pathfile` keyword or selected by Encounter test by specifying the `maxpathlength` keyword.

Path delay tests may or may not be timed. Paths may be tested in various degrees of strictness ranging from Hazard-Free Robust(most strict), Robust, Nearly-Robust, and Non-Robust(least strict). Refer to "Path Tests" on page 142.

Refer to the following for additional information:

- "Delay and Timed Test" on page 83.
- "create_path_delay_tests" in the *Encounter Test: Reference: Commands*

IDDq Tests

For static CMOS designs, it is possible to detect certain kinds of defects by applying a test pattern and checking for excessive current drain after the switching activity quietens down. This is called IDDq testing (IDD stands for the current and q stands for quiescent). Refer to "Create IDDq Tests" on page 207 for more information.

Parametric(Driver/Receiver) Tests

Parametric tests exercise the off-chip drivers and on-chip receivers. For each off-chip driver, objectives are added to the fault model for DRV1, DRV0, and if applicable, DRVZ.

For each on-chip receiver, objectives are added to the fault model for RCV1 and RCV0 at each latch fed by the receiver. These tests are typically used to validate that the driver produces the expected voltages and that the receiver responds at the expected thresholds.

IO Wrap Tests

These tests are produced to exercise the driver and receiver logic. The tests use the chip's internal logic to drive known values onto the pads and to observe these values through the pad's receivers. IO wrap tests may be static or dynamic. Static IO wrap tests produce a single steady value on the pad. Dynamic IO wrap test produce a transition on the pad.

IEEE 1149.1 Boundary Scan Verification Tests

Verification patterns are produced to validate the IEEE 1149.1 standard functions. No faults are processed and no specific defects are targeted.

IEEE 1149.1 boundary scan verification ensures that:

- The design is fully compliant with the IEEE 1149.1 standard.
- The Boundary Scan Design Language (BSDL) matches the design and visa versa, and that both are compliant with the IEEE 1149.1 Standard.

In regard to IEEE 1149.6 standard, Encounter Test also supports these boundary scan structures ensuring compliance to the IEEE 1149.6 standard. Verification is limited to the digital IEEE 1149.6 constructs.

Encounter Test ATPG can be run on testmodes containing 1149.x constructs. Refer to "1149.1 Boundary Chain Test Generation" on page 228 for more information.

Core Tests

Encounter Test supports core testing as per IEEE 1500 standard.

Low Power Tests

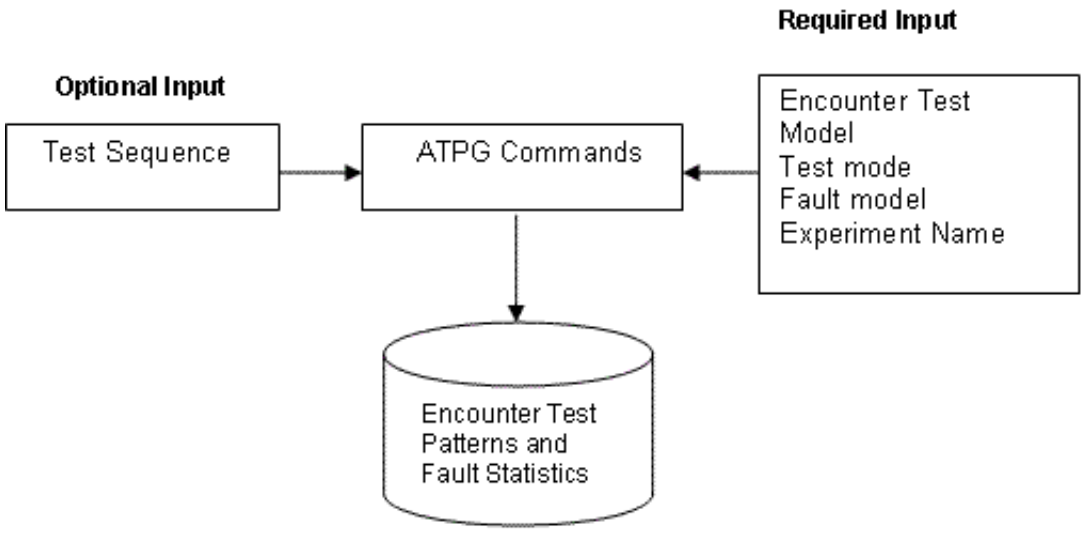
Refer to *Encounter Test: RAK: Low Power* for details on configuring a low power design and producing low power test patterns.

Committing Tests

This is an Encounter Test concept where a set of test patterns and their corresponding tested faults are saved and stored. All subsequent runs start at the test coverage achieved by the saved patterns. Refer to "Committing Tests" on page 245 for more information

Inputs for ATPG

The following figure represents the required and optional input for ATPG processing.



Test Generation Restrictions

The restrictions of test generation are as follows:

- Non conformance of designs to Encounter Test LSSD guidelines and GSD (General Scan Design) guidelines can result in invalid test data and reduced test coverage. The Test Structure Verification process verifies the conformance to these guidelines.
- There is no test generation override option to process the dynamic faults that were excluded from the fault model or from the test mode.

Invoking ATPG

Use the following syntax to invoke ATPG through command line:

```
create_<test type>_tests EXPERIMENT=<experiment name> TESTMODE=<testmode name>
WORKDIR=<directory>
```

For example, to create logic test, use the following command:

```
create_logic_tests EXPERIMENT=name TESTMODE=name  WORKDIR=<directory>
```

Refer to create_logic_tests in the *Encounter Test: Reference: Commands* for more information.

Use the following command to commit the test results:

```
commit_tests INEXPERIMENT=<name> TESTMODE=<testmode name> WORKDIR=<directory>
```

Refer to commit_tests in the *Encounter Test: Reference: Commands* for more information.

To invoke ATPG using Graphical User interface, select *ATPG - Create Tests - <Test type>*

For example, to generate scan chain tests using GUI, select:

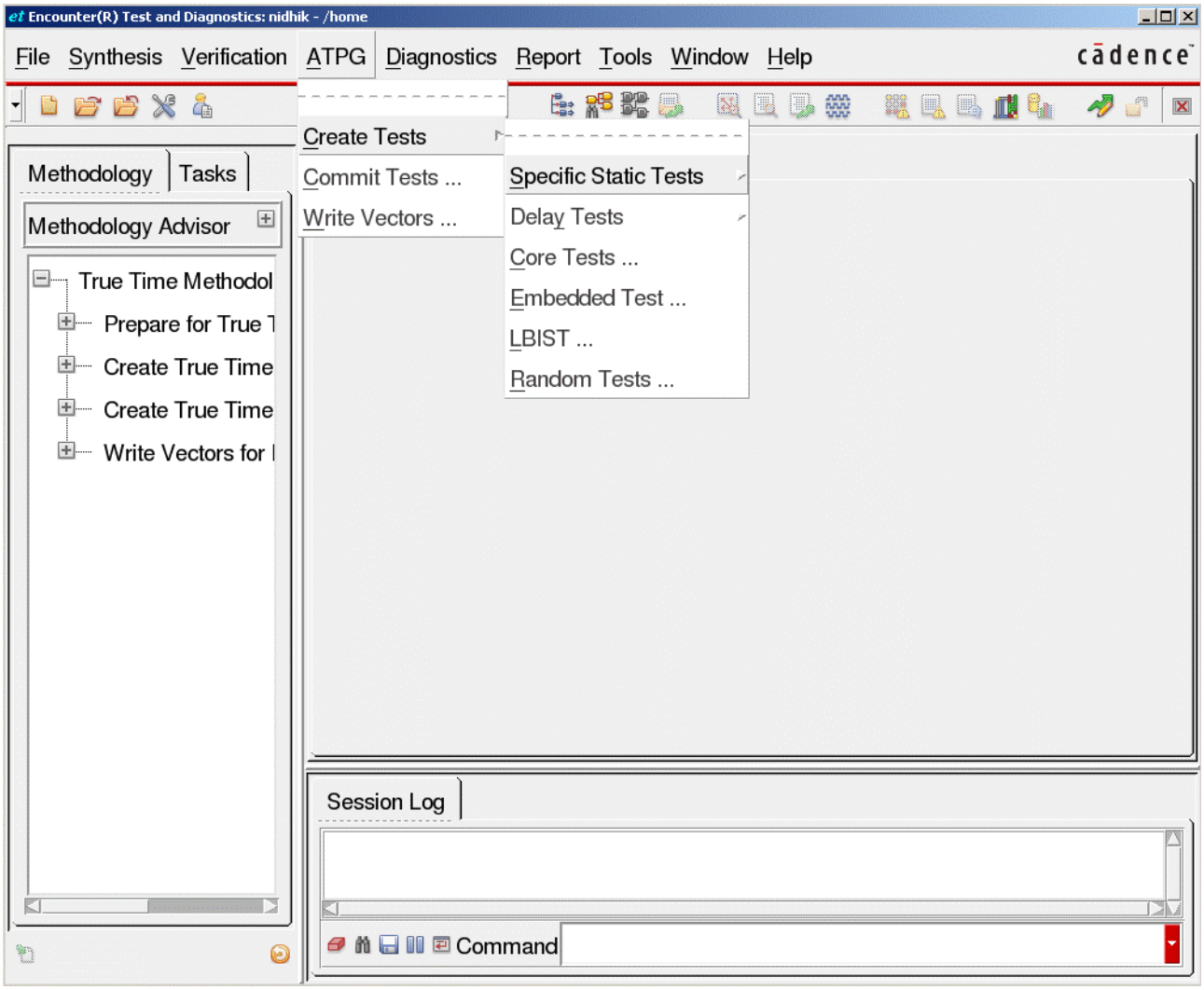
ATPG - Create Tests - Specific Static Tests - Scan Chain

To commit tests, select:

ATPG - Commit Tests

The following figure shows the ATPG menu in GUI:

Figure 1-4 ATPG Menu in GUI



Stored Pattern Test Generation (SPTG)

Stored Pattern Test Generation (SPTG) is an approach for component manufacturing test. It performs test pattern generation, test compaction, and fault simulation to create test patterns that can be applied to a design (using a stored pattern tester) to test for defects. You can experiment and save the results of experiments.

The following concepts are important to this discussion.

- Test pattern generation is the creation of a set of test patterns, sometimes referred to as test vectors.

- Static compaction is the merging of several test patterns into a single test pattern to target multiple defects. This reduces the number of test patterns required to test a design.
- Fault simulation accomplishes the following:
  - Computes the expected response values for a defect free design. The test patterns, including the expected response values, are written to an output file. This file contains information to be applied on a stored pattern tester.
  - Determines which faults are detected by the test patterns.

    Fault simulation acts as a filter. For example, if a test pattern is simulated and it does not detect any additional remaining faults (in other words, is ineffective), you can choose to not write the test pattern to the output file. Also, if any test pattern could potentially damage the product, you can choose to not write the pattern to the output file.

    The results of fault simulation are used to compute test coverage (the percentage of the faults that are detected by the tests). Refer to "Fault Statistics" in the *Encounter Test: Guide 4: Faults* for details on the calculation of test coverage.

Stored Pattern Test Generation applies the concepts of test pattern generation, compaction, and fault simulation in the process shown in Figure 1-2.

True-Time Test: An Overview

Encounter Test supports both static and delay testing. For delay testing, Encounter test has flows for both timed and untimed pattern generation. The timed and untimed test flows are very similar with only slight differences in the commands and options. For timed ATPG, True-Time Test provides an alternative methodology to the Encounter Test standard delay test methodology by using a streamlined and simplified two-phase process for generating timed patterns. This methodology filters out a large number of specialized and potentially problematic sequences that detect few faults and promote numerous iterations at the tester.

The following are the three phases of True-Time Test:

1. Logic model preparation
2. Pre-analysis of the chip, refer to "True-Time Test Pre-Analysis" on page 36.
3. Test pattern generation, refer to "True-Time Test Pattern Generation" on page 37.

Designing a Logic Model for True-Time Test

When building a logic model for True-Time Test, use the `build_model` command keyword `truetime=yes` to enable additional checks. These checks help validate that the technology cell levels of hierarchy are correctly identified to increase the likelihood of matching with the Standard Delay File (SDF).

True-Time Test Pre-Analysis

Prepare Timed Sequences determines a set of optimal parameters to run test generation and fault simulation for timed tests. These parameters include the following:

- The best clock sequences
- The frequency per sequence
- Maximum path length for each selected sequence

You do not need to perform this step for static ATPG.

The clock sequences are combined into a multi-domains sequence that simultaneously tests the combined sequences. Refer to preceding section in this chapter for related information.

The determination of best sequences is performed by generating test patterns for a statistical sample of randomly selected dynamic faults. Each unique test pattern is evaluated to ascertain how many times the test generator used it to test a fault. The set of patterns used most often are considered the best sequences. Normally, the top four or five will test 80 percent of the chip. An additional option, `maxsequences`, is available to use more sequences, if desired.

The presence of prepared sequences is denoted by sequences named `DelayTestClock`*n* in the sequence definition.

The maximum path length is determined by generating a distribution curve of the longest data path delays between a random set of release and capture scan chain latch pairs. The area under the curve is accumulated from shortest to longest and the maximum path length is the first valley past the cutoff percentage. This method constrains the timings to ignore any outlying paths that over inflate the test timings. Additional options are available to control this step by providing the cutoff percentage for the curve and a maximum path length to override the calculation.

The best sequences, and their timings and constraints are stored in the `TBDseq` file for the test mode. Refer to "TBDpatt and TBDseqPatt Format" in the *Encounter Test: Reference: Test Pattern Formats* for related information.

To perform pre-analysis using commands, refer to "prepare_timed_sequences" in the *Encounter Test: Reference: Commands*.

To perform pre-analysis using the graphical user interface, refer to "Prepare Timed Sequences" in the *Encounter Test: Reference: GUI*.

True-Time Test Pattern Generation

Encounter Test supports both static and delay ATPG.

To generate patterns using static ATPG, refer to "Static Test Generation" on page 57.

The recommended flow for delay ATPG for timed tests is to have test generation run off the results from pre-analysis done using Prepare Timed Sequences. This is not a required step, but is highly recommended. If the pre-analysis phase is performed using Prepare Timed Sequences, the applications automatically detect and process prepared sequence definitions unless otherwise specified (through the `useprep` keyword). Create Logic Delay Tests provides an option to exclude the sequence definitions produced by Prepare Timed Sequences.

Usually, all of the best sequences are processed in the order of most to least productive. However, the option is there to process an individual sequence or an ordered list of sequences. The number of faults processed for each job can be controlled by specifying `maxfaults=` on the command line.

To generate patterns, refer to the following:

- "Create Logic Delay Tests" in the *Encounter Test: Reference: GUI*
- ""create_logic_delay_tests" in the *Encounter Test: Reference: Commands*.

Types of True-Time ATPG

Automatic True-Time Testing

The Automatic True-Time flow automatically controls most delay test parameters. If SDF information is used for this flow, clocking sequences, clock timings, and transition constraints are automatically determined. To run this flow, perform the normal preparation required for running timed delay tests and then use the `prepare_timed_sequences` command to start the process. The minimum required inputs to `prepare_timed_sequences` are a test mode and delay model. To achieve higher control of the process, specify any of the available

test generation and timing options. Refer to "prepare_timed_sequences" in the *Encounter Test: Reference: Commands* for details on all associated keywords.

After the `prepare_timed_sequences` step is competed, run `create_logic_delay_tests`. The information from `prepare_timed_sequences` is automatically used.

At-Speed and Faster Than At-Speed True-Time Testing

The At-Speed and Faster Than At-Speed flows are run the same way as the Automatic True-Time Test flow with the addition of an input clock constraints file which contains user-specified timings for each clock domain to be used. For the At-Speed flow, specify the functional speeds of each domain within the clock constraints file. For the Faster Than At-Speed flow, it is recommended to start at the fastest speed, then perform subsequent runs with incrementally less aggressive timings until arriving back at a functional system speed. This ensures that defects are marked off only down paths with a minimum of slack. Longer paths which are not measurable at the aggressive times are not used to mark off faults. Therefore, only faults which only feed long paths are run at slower timings.

Static ATPG

Encounter Test has full support for a static ATPG flow. The processing is very simple and straight forward and has been used on 1000s of chips. Refer to "Static Test Generation" on page 57 for more information.

2

Using the True-Time Use Model Script

Encounter Test provides a script named `true_time` to perform Encounter Test true time on a design. The script takes a design through build model, test pattern generation using ATPG, and writes out patterns in desired language. The script also gives you an option of using a portion or the entire Encounter Test flow. For example, you can use the script to just perform ATPG and not build model. This script is designed and maintained to allow the best flow through Encounter Test for the general market place.

Note: You might get more optimized results using direct system calls, but the script is designed in a way to allow good results without having the expertise in all the Encounter Test domains.

The script provides the following:

- Support for the four basic use models (static, fixed time, at-speed, and faster than at-speed)
- Ability to perform static top off after delay ATPG
- Optional pattern sorting
- Perform optional SDC
- Perform up to three faster than at-speed runs
- Support full scan and compression modes (OPMISR+, XOR)
- Support for user-modified TDRs, modedefs, user sequences, and lineholds
- Support for industry compatible fault modeling:
  - Full fault
  - Cell boundary
- Support for optional cell delay template (to fix SDF related problems)
- Perform testing with memories defined as blackboxes

Some other features of the script are:

- Consistent with each release:
  - Updated with new features for each release
  - Removes obsolete commands and keywords
  - Contains recommended keyword settings
- Helps you set up Encounter Test True Time
  - The setup file (`tt_setup`) contains a limited set of user inputs
  - The script executes the required commands
  - A single setup keyword may relate to several Encounter Test keywords or commands (`test_through_memories` -> `allowjustifytimeframes=`*`n`*, `dynonly2clocks=no,...`)
- Helps determine scope of evaluation
- The setup file serves as a common checklist for all evaluations
- Prints the following output to xterm:
  - Message summary only if warnings or errors
  - Total fault counts
  - Scan chain summary messages
  - Test coverage and pattern count totals
  - Complete logs also available
- Stops on severe warnings
  - Makes them more visible and encourages fixing them immediately
  - Script allows override if necessary
- Allows you to stop the execution after performing some steps
- Allows restarting the execution after performing some steps
- Saves commands in a file for additional editing, if required

Executing the Encounter Test True Time Script

The `true_time` command used to run the true-time script differs slightly from other Encounter Test commands. The command takes a setup file as input and reads the rest of the data from this file.

The syntax for the `true_time` command is given below.

```
true_time <setup_file>
```

where *`setup_file`* is the file with options and parameters to run through the Encounter True Time flow.

Prerequisite Tasks

Complete the following tasks before executing the `true_time` script:

- Create a working directory by using the `mkdir <directory name>` command.
- Fill in the setup file (*`<setup_file>`*) with required steps.
- Set up Encounter Test into your environment (using `et -c` or `et`)

Setup File Input

Encounter Test is shipped with the following two templates:

- *$Install_Dir*`/tb/etc/tb/contrib/tt_setup` - Used for delay and then static ATPG
- *$Install_Dir*`/tb/etc/tb/contrib/tt_setup_static` - Used only for static ATPG

Create a copy of either of these files in your working directory. The `tt_setup_static` template is a subset of `tt_setup`. The following topics discuss the various sections and inputs of the `tt_setup` file.

Script Control Information

This information controls the execution of the script and allows you to start or stop after certain steps and control log files.

The following table lists the parameters and the corresponding values for this section of the setup file:

Parameter	Description	Value
`RESTART=<value>`	Restart processing from a previous step. This is optional. If not specified, the script will start from `build_model`.	■ `build_model` (the beginning) ■ `build_testmode` ■ `read_sdc` (also does `read_sdf`) ■ `atpg` ■ `write_vectors` ■ `run_ncverilog`
`EXITBEFORE=<value>`	Exit script before the specified step. This is optional. If not specified, the script will stop at the end.	■ `build_model` (the beginning) ■ `build_testmode` ■ `read_sdc` (also does `read_sdf`) ■ `atpg` ■ `write_vectors` ■ `run_ncverilog` ■ `end` (the end)
`REMOVE_OLD_LOGFILES=<value>`	Clear the log file directory. This is optional. If not specified, the script will clear log files.	■ `yes` - clear the logs (default) ■ `no` - keep the logs
`SCRIPTFILE=<filename>`	Save the executed commands in a file. If not specified, the script will save commands in *`$WORKDIR`*`/run_et`	

Parameter	Description	Value
CONTINUE_WITH_SEVERE=<value>	Continue execution even if severe errors are found. This is optional.	■ yes - Continue even if severe messages are encountered (default) ■ no - Stop processing if severe messages are encountered
EXECUTE=<value>	Execute all the commands as the script is created. This is optional. If not specified, all commands are executed by default.	■ yes - Run all commands (default) ■ no - Create script, no execution.
LOGFILE=<filename>	Save the log of the script in the specified file. This is optional. If not specified, log from the script is not saved.	
METHNAME=<methodology name>	Create a methodology file using the information in the tt_setup file. The methodology file is stored in WORKDIR.	

Model Information

This section contains information used to create the Encounter Test model and fault model. The following information is required only for build_model and/or build_faultmodel steps.

Refer to build_model and build_faultmodel in the *Encounter Test: Reference: Commands* for more information on the respective parameters.

Parameter	Description	Value
`WORKDIR=<directory>`	Specify a working directory. Required for all steps.	
`DESIGNSOURCE=<file.v>`	Specify a fully qualified name of a netlist. Required only for the `build_model` step.	
`TECHLIB=<file1.v,file2.v>`	Specify a fully qualified name of libraries (these can be `,` or `.`). Required only for the `build_model` step.	
`TOPCELL=<name>`	Specify a top-level design cell. This is optional and if not specified, Encounter Test selects the last cell in the design source.	
`CREATEBLACKBOXES=<value>`	Allow the tool to automatically create blackbox for missing cells. This is optional, and the default value is `no`.	■ `no` - Do not create black boxes (default) ■ `yes` - Create black boxes
`BLACKBOXOUTPUTS=<value>`	Tie output of black boxes to a value. This is optional, and the default value is `x`.	`x, z, X , Z, 0, 1`
`INDUSTRYCOMPATIBLE=<value>`	Align fault models with other tools. This is optional, and the default value is `no`.	■ `no` - Do not create industry compatible fault model (default) ■ `yes` - Create special fault model

Parameter	Description	Value
`CELLFAULTS=<value>`	Build an entire fault model or cell fault model. This is optional, and the default value is `no`.	■ `no` - Create a full fault model (default) ■ `yes` - Create only cell faults
`OPTIMIZE=<value>`	Optimize the logic model for improved performance. This is optional and the default value is `dangling`.	■ `dangling` - Remove dangling logic (default) ■ `none` - Do not remove any logic
`VLOGPARSER=<value>`	Specify the verilog parser to use. This is optional. The default value is `IEEEstandard`, but Encounter Test also determines the best parser based on the comments in files.	■ `IEEEstandard` - verilog 2001 ■ `et` - verilog with old `/ /!` style attributes

Test Mode Information

The information in this section is used to create the Encounter Test test modes. The information in the following table is required only if you are using the `true_time` script to build a testmode(s).

Refer to build_testmode in the *Encounter Test: Reference: Commands* for more information.

Parameter	Description	Value
`ASSIGNFILE=<file>`	Specify the file containing test function pins (such as clocks and scan enables). When using compression, specify the fullscan test mode assign file. Required for the `build_testmode` step.	
`COMPRESSION=<value>`	If compression is present in design, specify the type of compression being used. Required for the `build_testmode` step.	■ `opmisrplus` - use OPMISRPLUS compression ■ `opmisrplus_topoff` - use OPMISRPLUS
`OPMISRPLUS=<value>`	Specify the compression with ATPG topoff using the fullscan mode	■ `xor` - use XOR compression ■ `xor_topoff` - use XOR compression with ATPG topoff using the fullscan mode
`COMPRESSIONASSIGNFILE=<file>`	Specify the file containing test function pins (such as clocks, scan enables) for compression testmode. Required for the `build_testmode` step to build compression testmode.	
`TESTMODE=<file name>`	Specify the custom mode definition file name. This is optional.	

Parameter	Description	Value
`TDRPATH=<directory>`	Specify the TDR directory to override the default TDR. This is optional.	
`MODEDEFPATH=<directory>`	Specify the directory containing custom mode definition file defined `TESTMODE` above. This is optional.	
`STILFILE=<file>`	This is an alternative method to specify test function pins and mode definition files (STIL SPF file). This is optional.	
`SEQDEF=<file>`	The file where TBDseqPatt mode initialization and customer scan protocols are found. This is optional.	
`COMPRESSIONTESTMODE= <file name>`	Specify the custom mode definition file for compression test mode. This is optional.	
`COMPRESSIONSEQDEF=<file>`	Specify the sequence definition file name or names that contain pre-defined input pattern sequences.	
`COMPRESSIONSTILFILE=<file>`	This is an alternative method to specify test function pins and mode definition files (STIL SPF file). This is optional	

ATPG Controls and Static ATPG Information

This information is used to control ATPG settings. The parameters listed in the following table are required only if you use the `true_time` script to run ATPG.

Refer to create_logic_tests and create_scanchain_tests in the *Encounter Test: Reference: Commands* and "Static Test Generation" on page 57 for more information.

Parameter	Description	Value
`ATPGTYPE=<value>`	Specify the type of ATPG to perform. This is a required parameter.	■ `static` - Static ATPG ■ `dynamic` - Delay ATPG ■ `dynamic_topoff` - Delay ATPG with static topoff ■ `dynamic_only` - Only simulate on dynamic faults
`EFFORT=<value>`	Specify the ATPG effort. This is optional, and the default value is `low`.	■ `low` - Low ATPG effort ■ `medium` - More effort ■ `high` - Run higher effort levels.
`COMPACTION=<value>`	Specify ATPG compaction effort. This is optional, and the default value is `medium`.	■ `medium` - Good amount of compaction ■ `low` - Less compaction ■ `high` - Higher level of compaction

Parameter	Description	Value
SORTPATTERNS=<value>	Perform sorting of resultant ATPG vectors. This reduces coverage slightly, but can drastically reduce pattern counts. The default value is no.	■ no - Do not run extra sorting ■ Yes - Do additional sorting
STATICSEQUENCES=<name>	Specify the name of test sequence for ATPG. This is imported by SEQUENCEFILE and is optional.	
LINEHOLD=<file>	Specify the user-specified line hold file. This is optional.	
MAXCPUTIME=2880	Specify the maximum time for ATPG steps (2 days). This is optional.	
SEQUENCEFILE=<file>	Specify the user-specified test sequence for ATPG. This is optional.	

Delay ATPG Information

This input is used to control delay ATPG settings. The parameters listed in the following table are required only if you use the true_time script to run delay ATPG.

Note: This section does not exist in the tt_setup_static file.

For more information on delay ATPG, refer to "Delay and Timed Test" on page 83.

Parameter	Description	Value
SDCPATH=<directory>	Specify the directory containing SDC file. This is optional and is used for fixed time, at-speed, and faster-than at-speed tests.	
SDCNAME=<name>	Specify the name of SDC file. This is optional and is used for fixed time, at-speed, and faster than at-speed tests.	
SDFPATH=<directory>	Specify the directory containing the SDF file. This is optional and is used for at-speed and faster than at-speed tests.	
SDFNAME=<name>	Specify the name of SDF file. This is optional and is used for at-speed, and faster than at-speed tests.	

Parameter	Description	Value
`CLOCKCONSTRAINTS=<file>`	Specify the clock constraint file with required clocks and frequencies. You need to specify only this parameter if using for at-speed test. For faster-than-at-speed, the values in the file should represent the faster-than-at-speed times. This is optional and is used for fixed time, at-speed and faster-than at-speed tests.	
`CLOCKCONSTRAINTS2=<file>`	When using faster-than-at-speed methodology, this would be the next fastest frequencies. This is optional and is used for faster than at-speed tests.	
`CLOCKCONSTRAINTS3`	When using faster-than-at-speed methodology, this would be the slowest frequencies. This is optional and is used for faster-than-at-speed tests.	
`ANALYZE=<value>`	Analyze defect sizes. This is optional and the default value is `no`.	■ `no` - Do not perform default size analysis ■ `yes` - Perform default size analysis
`DYNAMICSEQUENCES=<name>`	Specify the name of sequences to use for ATPG. This is optional.	

Parameter	Description	Value
`DYNAMICCOVERAGE=<value>`	Specify the type of dynamic faults to process, that is, just within a domain or within a domain and cross domains. The default value is `intradomain`.	■ `intradomain` - Perform ATPG on faults within domains ■ `alldomains` - Perform atpg on fault within domains and cross domains
`LASTSHIFTLAUNCH=<value>`	Specify whether to use launch of last shift. The default value is `no`.	■ `no` - Use launch off capture ■ `yes` - Use launch off shift
`DELAYTESTTHRUMEMORIES=<value>`	Specify whether to generate tests through memories. The default value is `no`.	■ `no` - Do not generate tests through memories ■ `yes` - Generate tests through memories
`EARLYMODE=<value>`	Use linear combination of delays for timing checks that use early mode timing. This is optional.	Format: *x,y,z* where *x,y,* and *z* add up to `1`. For example, `0,1,0` or `0,0.5,0.5`
`LATEMODE=<value>`	Use linear combination of delays for timing checks that use late mode timing. This is optional.	Format: *`x,y,z`* where *`x,y`* and *`z`* add up to `1`. For example, `0,1,0` or `0,0.5,0.5`
`TEMPLATESOURCE=<file>`	Used by `read_celldelay_template` to identify a list of required delays for a cell. This is optional.	

Parameter	Description	Value
`MAXSEQ=99`	Specify the maximum number of sequences for ATPG.	
`USEPREP=<value>`	Specify whether to run `prepare_timed_sequences` before delay ATPG. The default value is `yes`.	■ `yes` - Execute `prepare_timed_sequences` (default) ■ `no` - Do not execute `prepare_timed_sequences`

Path Delay ATPG Information

This information is used to control the path delay ATPG settings. The parameters listed in the following table are required only if you use the `true_time` script to run path delay ATPG. This section does not exist in the `tt_setup_static` file.

Parameter	Description	Value
`PATHTEST=<values>`	Generate a path test. This is optional and the default value is `no`.	■ `no` - Do not generate any path tests ■ `yes` - Generate path test in addition to transition tests ■ `only` - Generate path test without transition tests
`PATHFILE=<file>`	Specify the file containing list of paths. Required for path tests.	
`PATHNAME=<name>`	Specify an alternate fault model name. This is the last qualifier and is optional.	

Parameter	Description	Value
PATHTYPE=<value>	Specify the type of path tests to generate. This is optional and the default is nearlyrobust.	nearlyrobust, robust, nonrobust, hazardfree

Write Patterns Information

This information is used to control the writing of the test vectors into different formats.

For more information on Test Vector Formats, refer to write_vectors in the *Encounter Test: Reference: Commands* or "Test Vector Forms" on page 254.

Parameter	Description	Value
WRITEVERILOG=<value>	Specify whether to write out verilog patterns. This is optional, and the default option is no.	■ no - Do not write out patterns in Verilog (default) ■ yes - Write out the patterns in Verilog
VERILOG_SCAN=<value>	Specify the kind of patterns to write out. This is required if writing out Verilog. The default value is default.	■ serial - The expanded scan format ■ parallel - Where scan values are applied directly to and measured directly at the scan registers

Parameter	Description	Value
WRITESTIL=<value>	Specify whether to write out STIL patterns. This is optional, and the default value is no.	■ no - Do not write out patterns in STIL (default) ■ yes - Write out the patterns in STIL
WRITEWGL=<value>	Specify whether to write out WGL patterns. This is optional, and the default value is no.	■ no - Do not write out patterns in WGL (default) ■ yes - Write out the patterns in WGL

NC-Sim Information

This information is used to control the execution of NC-sim. The parameters listed in the following table require the writing out of test patterns in the Verilog format.

For more information on NC-sim, refer to "NC-Sim Considerations" on page 196.

Parameter	Description
NCVERILOG_DESIGN=<files>	Specify the design and techlib files separated by " , " to use for NCSIM separated,
NCVERILOG_OPTIONS=<optons>	Specify the ncverilog options to use in ncverilog separated by "y" NCVERILOG_DEFINE=<options>
NCVERILOG_DEFINE=<options>	Specify the +define options to use in ncverilog separated by " , "

Parameter	Description
NCVERILOG_DIRECTORY=<directory>	Specify the directory to find ncverilog or ncsim. This avoids a potential conflict between the ncverilog used by Encounter Test and the ncverilog used for simulation

Output

Before processing, the script parses the setup file to analyze data and checks the incorrect values:

```
INFO - COMPRESSION keyword not set to  recommended values: opmisrplus,
opmisrplus_topoff, xor, or xor_topoff. COMPRESSIONTESTMODE keyword not set.
No compression mode in the design.
Only TESTMODE=FULLSCAN_TIMED testmode is active.
```

Also, messages are reported to highlight the executing events:

```
INFO - Saving Encounter Test commands to script file: ./run_et
INFO - Using WORKDIR=.
....
Running commit_tests logic - to save logic patterns
********************************************************************************
Total Static Coverage: 74.01%; Total Patterns 36;
********************************************************************************
Completed Successfully. Continuing.
********************************************************************************
Running write_vectors Verilog

INFO (TVE-003): Verilog write vectors output file will be: ./testresults/verilog/
VER.FULLSCAN_TIMED.data.scan.ex1.ts1.   [end TVE_003]
INFO (TVE-003): Verilog write vectors output file will be: ./testresults/verilog/
VER.FULLSCAN_TIMED.mainsim.v.   [end TVE_003]
Completed Successfully. Continuing.
```

If any information is missing, for example, no NC-sim information supplied, the script exits before starting the step.

```
********************************************************************************
Setup file indicates exit before run_ncverilog. Exiting.
```

The output summary highlights the coverage and pattern count achieved during the run.

3

Static Test Generation

This chapter explains the concepts and commands to perform static ATPG with Encounter Test.

Several types of tests are available for static pattern test generation. Refer to "General Types of Tests" on page 28 for more information.

- To perform *Test Generation* using the graphical interface, refer to "ATPG Pull-down" in the *Encounter Test: Reference: GUI*.
- To perform *Test Generation* using command lines, refer to descriptions of commands for creating and preparing for tests in the *Encounter Test: Reference: Commands*.

The availability of test generation functions is dependent on licensing. Refer to "Encounter Test and Diagnostics Product Configuration" in *What's New for Encounter® Test* for details on the licensing server.

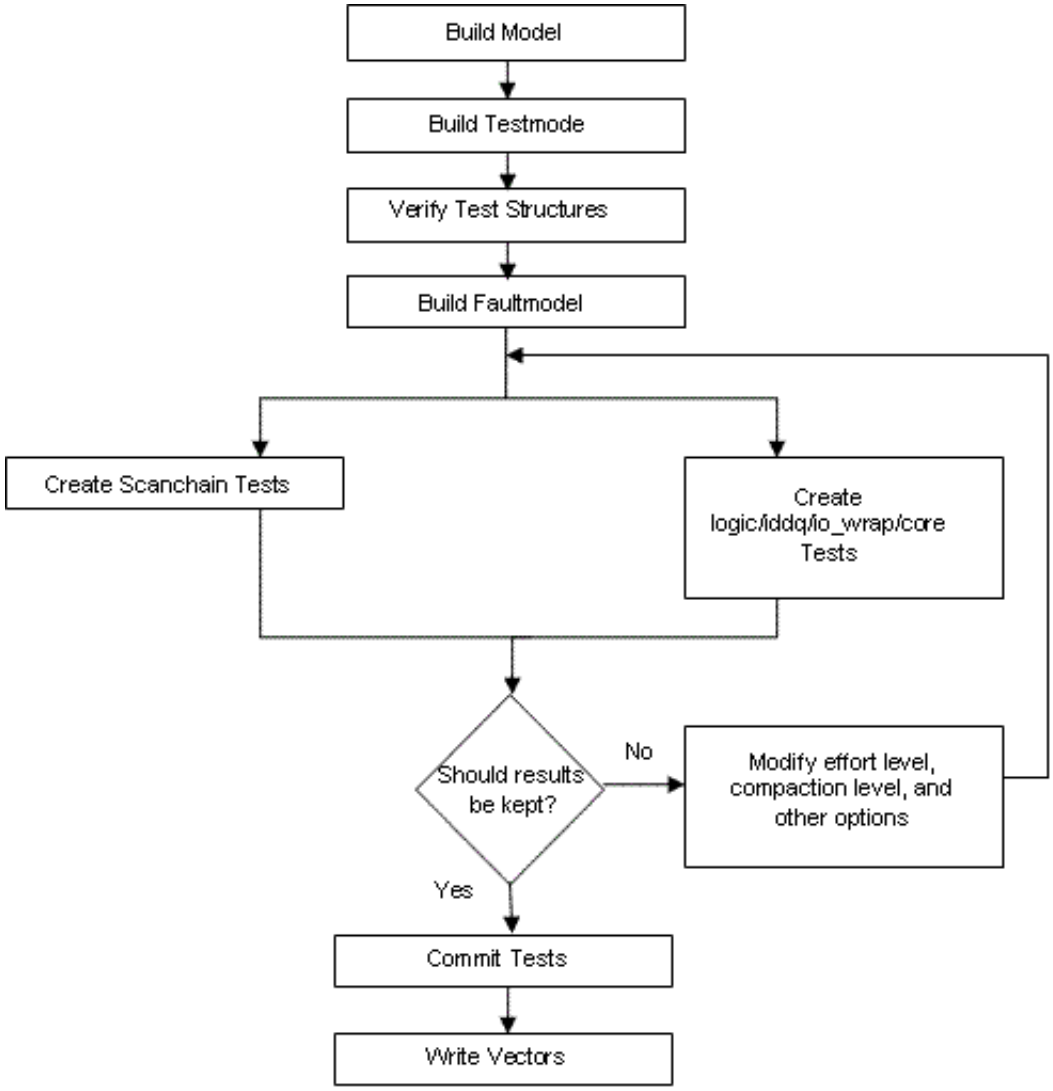
The chapter discusses the following ATPG tasks:

- Scan Chain and Reset Fault Tests
  - "Performing Scan Chain Tests" on page 60
    - "Performing Flush Tests" on page 63
- Logic Tests
  - "Performing Create Logic Tests" on page 64
- Domain Aware Test Generation
  - "Performing Domain Aware Test Generation" on page 66
- Exporting and Saving Patterns
  - "Committing Tests" on page 81
  - "Writing Test Vectors" on page 82

Static Test Pattern Generation Flow

The following figure shows a typical processing flow for running static test pattern generation.

Figure 3-1 Encounter Test Static Pattern Test Processing Flow



1. build_model - Reads in verilog netlists and builds the Encounter Test model

   For complete information on Build Model, refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.

2. build_testmode - Creates scan chain configurations of the design

For complete information, refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes*.

3. verify_test_structures - Verifies scan chains and checks for design violations

   Resolve any TSV violation before running ATPG.

   For complete information, refer to "Logic Test Structure Verification (TSV)" in the *Encounter Test: Guide 3: Test Structures*.

4. build_faultmodel - Creates the fault model for ATPG

   For complete information, refer to "Building a Fault Model" in the *Encounter Test: Guide 4: Faults*.

5. create_scanchain_tests - Creates test patterns to validate and test faults on the scan chain path

   For complete information, refer to "Scan Chain Tests" on page 28.

6. commit_tests - Saves the patterns and marks off faults from the scan chain test

   The scan chain test is written added to the master pattern set for this testmode so that subsequent ATPG runs do not need to generate tests for these faults.

   For complete information, refer to "Utilities and Test Vector Data" on page 245.

7. create_logic_tests - Creates patterns to test the remaining static faults

   An experiment is created that contains the required type of tests. Most often these will be logic tests.

   For more complete information of the various test types, refer to "General Types of Tests" on page 28.

   Refer to "Advanced ATPG Tests" on page 207 for information on other types of ATPG capabilities such as Iddq, core test, and parametric testing.

8. Should the results be kept?

   If the results of the experiment are satisfactory, the experiment can be committed, appending it's test patterns and fault detection to the master pattern set for the testmode.

   - ❑ Yes - commit tests
   - ❑ No - If the results are not satisfactory, another experiment can be run with different command line options.

   You can also analyze untested faults. For complete information, refer to "Deterministic Fault Analysis", in the *Encounter Test: Guide 4: Faults.*

9. write_vectors - Writes out the patterns in WGL, Verilog, or STIL format

   For complete information, refer to "Writing and Reporting Test Data" on page 181.

Performing Scan Chain Tests

This command generates a scan chain test targeting static faults along the scan paths. After creating the scan patterns, you can commit them to save the patterns and fault status for future ATPG runs. For more information, refer to "Scan Chain Tests" on page 28.

For compression modes, the scan chain tests test many of the faults in the compression networks. This also applies to testing faults in any existing masking logic.

To create scan chain tests using the graphical interface, refer to "Create Scan Chain Tests" in the *Encounter Test: Reference: GUI*.

To perform create scan chain tests using command lines, refer to "create_scanchain_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `create_scanchain_tests` command is given below:

```
create_scanchain_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode
- `experiment`= name of the test patterns

Prerequisite Tasks

Before executing `create_scanchain_tests`, build a design, testmode, and fault model. Refer to the *Encounter Test: Guide 1: Models* for more information.

Important Information from Log

The output log contains a summary of the number of patterns generated with their representative coverage. Static faults should be tested.

```
*************************************************************************
     ----Stored Pattern Test Generation Final Statistics----
                 Testmode Statistics: FULLSCAN
```

```
                        #Faults   #Tested   #Redund   #Untested   %TCov   %ATCov
Total Static             908       425       0         437         46.81   46.81

                     Global Statistics

                        #Faults   #Tested   #Redund   #Untested   %TCov   %ATCov
Total Static             1022       425       0         551        41.59    41.59
*************************************************************************

                    ----Final Pattern Statistics----

Test Section Type                          # Test Sequences
-----------------------------------------------------------
Scan                                        1
-----------------------------------------------------------
Total                                       1
```

Debugging No Coverage

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

Additional Tests Available

There are additional scripts to help test the set and reset faults on scan flops. These ATPG commands are not supported officially but are available to achieve higher test coverages.

An Overview to Scan Chain Patterns

The following is the structure of a static scan chain test mode:

- `Scan_Load` - Load the scan chain bits with repeating 0011 pattern
- `Stim_PI` - Stay in scan state
- Static scan chain shift #1
  - `Stim_PI` - Load the next value on scan inputs to continue 0011 pattern from scan input
  - `Pulse` - Pulse the scan clocks

- `Measure_PO` - Measure values on scan outputs

...

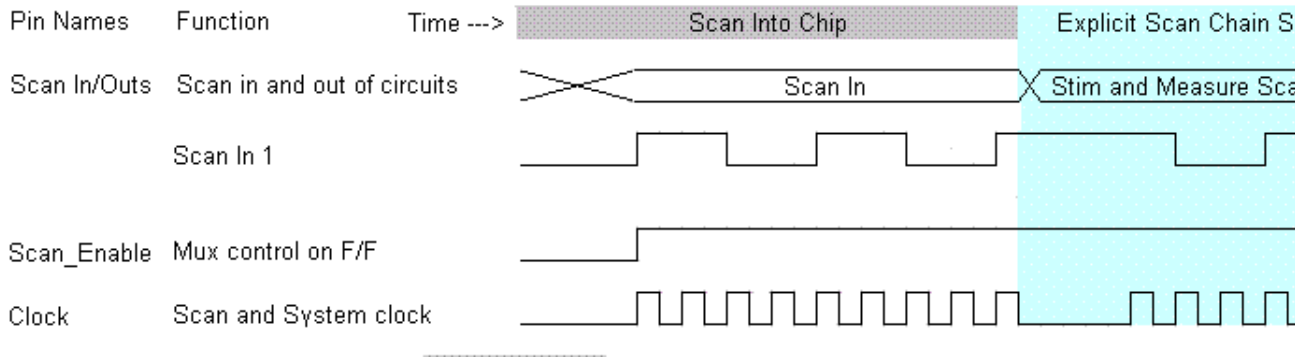
- Static scan chain shift #5
  - `Stim_PI` - Load the next value on scan inputs to continue 0011 pattern from scan input.
  - `Pulse` - Pulse the scan clocks
  - `Measure_PO` - Measure values on scan outputs
  - `Scan_Unload` - Observe all scan bits

In summary:

- The static scan chain test is always in the scan state.
- All clocks are configured at scan speeds.
- Additional clocking and sequences are added while testing for compression, MISRs, or masking.
- The order of sequence can change slightly based on custom-scan protocols and custom-scan preconditioning.

The following figure depicts a static scan chain wave form:

Figure 3-2 Static Scan Chain Wave Form



Performing Flush Tests

This command generates a flush test and test static faults along the scan paths. This test is applicable only to LSSD clock designs. After creating the flush patterns, you can commit them to save the patterns and fault status for future ATPG runs. Refer to "LSSD Flush Test" on page 28 for more information.

To create flush tests using the graphical interface, refer to "Create LSSD Flush Tests" in the *Encounter Test: Reference: GUI*.

To create flush tests using the command line, refer to "create_lssd_flush_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `create_lssd_flush_tests` command is given below:

```
create_lssd_flush_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode
- `experiment`= name of the test patterns

Prerequisite Tasks

Before executing `create_lssd_flush_tests`, build a design, testmode, and fault model. Refer to the *Encounter Test: Guide 1: Models* for more information.

Debugging No Coverage

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

Performing Create Logic Tests

Create logic tests generates static ATPG patterns. For more information, refer to “Logic Tests” on page 29.

To perform Create Logic Tests using the graphical interface, refer to "Create Logic Tests" in the *Encounter Test: Reference: GUI*.

To perform Create Logic Tests using the command line, refer to “create_logic_tests” in the *Encounter Test: Reference: Commands*.

The syntax for the `create_logic_tests` command is given below:

```
create_logic_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated

Prerequisite Tasks

Complete the following tasks before executing Create Logic Tests:

1. Import a design into the Encounter Test model format. Refer to “Performing Build Model” in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See “Performing Build Test Mode” in the *Encounter Test: Guide 2: Testmodes*.
3. Build a fault model including static faults. See “Building a Fault Model” in *Encounter Test: Guide 4: Faults* for more information.

Output

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

```
*************************************************************************
        ----Stored Pattern Test Generation Final Statistics----

                  Testmode Statistics: COMPRESSION_ILL

              #Faults     #Tested     #Redund     #Untested     %TCov     %ATCov

Total Static   1857         1489        149         171          80.18     87.18

                             Global Statistics

              #Faults     #Tested     #Redund     #Untested     %TCov     %ATCov

Total Static  1950          1489        149         264          76.36     82.68
*************************************************************************

                    ----Final Pattern Statistics----

Test Section Type                          # Test Sequences
-------------------------------------------------------
Logic                                        43
-------------------------------------------------------
Total                                        43
```

Specify `reportoutput=industry` to generate the report in the industry compatible coverage format. A sample report for static (stuck at 0/1) faults generated using `reportoutput=industry` is given below:

```
*************************************************************************
                               #Faults  #Tested  #Possibly  #Redund  #Untested
Static Testmode                  68       32        16         0        20
Static Global                    68       32        16         0        20
Static Test Coverage            47.06%
Static Fault Coverage           47.06%
Static ATPG Effectiveness       76.47%
*************************************************************************
```

- Static Test Coverage is the percentage of detected faults out of detectable faults. It is calculated as `#tested in mode/(#test mode faults - #redundant - #globally ignored)`
- Static Fault Coverage is the percentage of detected faults out of all faults. It is calculated as `#tested globally/(#global faults + #globally ignored)`
- Static ATPG Effectiveness is the percentage of ATPG-resolvable faults out of all faults. It is calculated as `(#Tested + #Redundant + #globally ignored + #ATPG Untestable + #Possibly Tested)/(#global faults + #globally ignored)`

Debugging Low Coverage

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

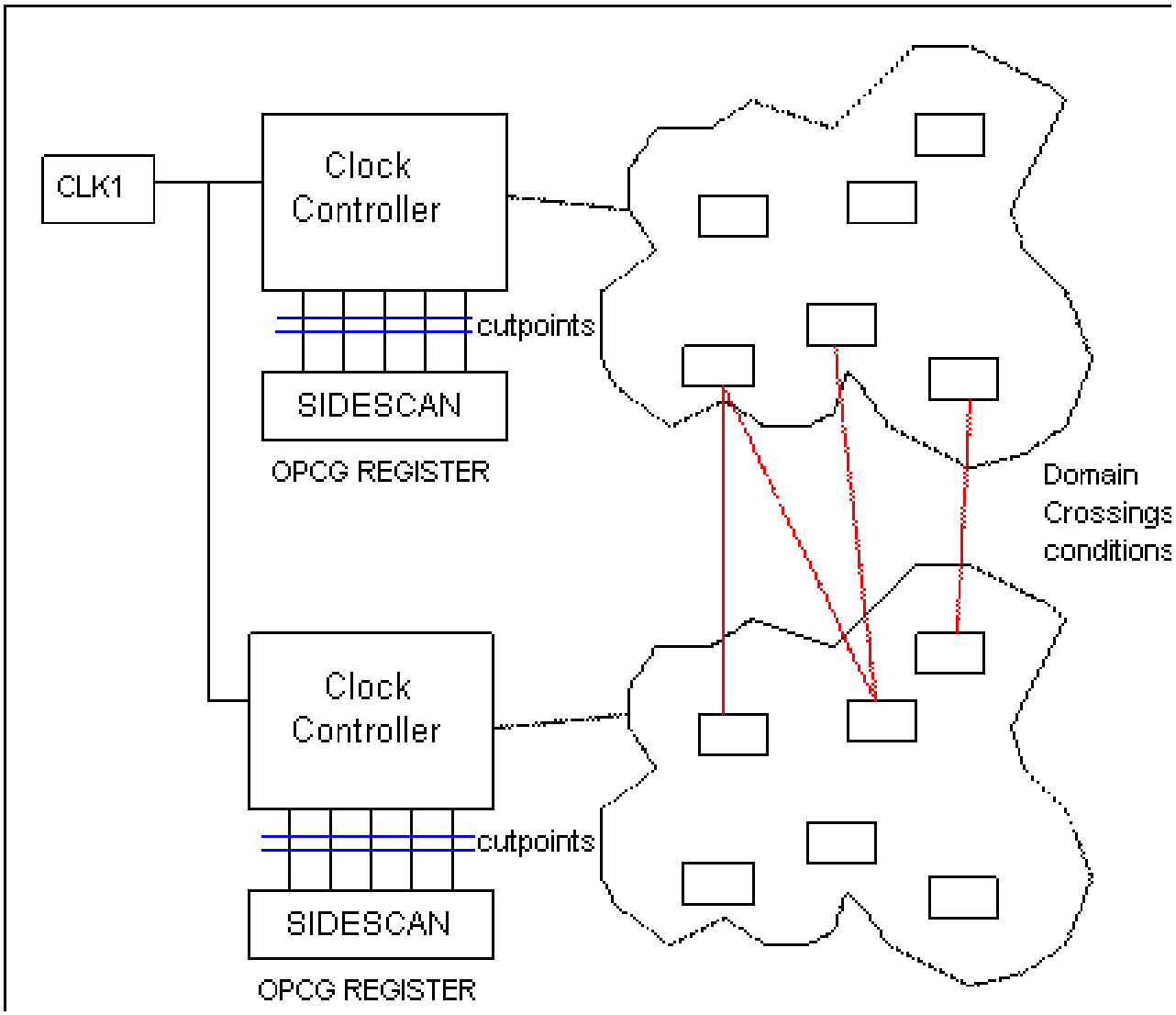
Note: Another tool to analyze low ATPG coverage is deterministic fault analysis. Refer to Analyze Faults in the *Encounter Test: Guide 4: Faults* for more information.

Performing Domain Aware Test Generation

Encounter Test supports for testing designs that contain multiple asynchronous clock domains fed from a single clock source.

Figure 3-3 on page 67 depicts a scenario where a tester-driven clock primary input (CLK1) feeds into clock controller circuitry that controls the clocks for two separate internal domains. The clocks to the domains are controlled by registers that are part of the clock controller circuitry. The registers in this example are shown as sidescan registers that are loaded every time the clocking sequence needs to change their values, although it is also possible to control the clocks through registers loaded during scan.

Figure 3-3 Controlling Domains through Sidescan Registers



As shown in Figure 3-3 on page 67, if both the internal domains are pulsed together during test generation, it might result in an edge race, thus causing a transition from one domain to be captured by the other during a single clock pulse.

Encounter Test allows you to automatically find domain crossings and to protect against edge races during test generation, whether domains are driven from separate PIs, PPIs, or are derived internally from one or more common PIs or PPIs. Additional capabilities include:

- Control (enable/disable) domains through user-specified programming information
- Automatically identify domain crossings

- Check programming registers to determine whether they prevent domain crossings
- Use an SDC to protect domain crossings
- Automatically generate constraints to protect domain crossings (if an SDC is not available)
- Identify groups of domains that may be pulsed together
- Define arbitrary clock domains within a design

Controlling Clock Domains During Test Generation

If a design contains registers to enable or disable the clocking of various domains, they need to be set for test generation. For the following, Encounter Test automatically determines how to program the registers:

- Internal Domain Registers
- OPCG Domains created by Cadence RTL Compiler DFT

For the following, you need to provide the programming for the registers:

- Scannable Registers not defined as Internal Domain Registers
- OPCG Domains inserted by the User

Scannable Registers

If the clock control registers are part of a scan chain, they should be set as LINEHOLD for a given ATPG experiment, or should be included in a Scan_Load event as part of a user-specified test sequence.

```
create_logic_tests LINEHOLD=<mylinehold.file>
```

where `<mylinehold.file>` includes the following:

```
HOLD latch.pin1 = 1 ;
HOLD latch.pin2 = 1 ;
HOLD latch.pin3 = 0 ;
…
```

Internal Domains

Internal domain registers are defined during build_testmode and are loaded via a user-specified setup sequence using a Load_OPCG_Controls event (to load the register values) and a Linehold object. As the internal domain registers are hidden from ATPG via cutpoints,

the Load_OPCG_Controls event provides the means to do the register programming from the ATE, while the Linehold object is used by ATPG to identify the values that should be placed on the cutpoints. This process of loading internal domain programming registers via the setup sequence is known as sidescan.

```
read_sequence_definition importfile=<mysetup.file> testmode=…
create_logic_tests setupsequence=<mysetup>
<mysetup.file> contents:
TBDpatt_Format (mode=node, model_entity_form=name);
[ Define_Sequence Setup_domain1 (setup);
[ Lineholds () :
"ir3.pf.ppi1"=0
"ir3.pf.ppi2"=0
"ir1.pf.ppi1"=1
"ir1.pf.ppi2"=0
"ir2.pf.ppi1"=0
"ir2.pf.ppi2"=0
;
] Lineholds;
[ Pattern (pattern_type = static);
Event Load_OPCG_Controls:
"ir3.pf.CG"=0
"ir1.pf.CG"=1
"ir2.pf.BDI"=0
"ir3.pf.BDI"=0
"ir2.pf.CG"=0
"ir1.pf.BDI"=0
;
] Pattern;
] Define_Sequence Setup_domain1;
```

OPCG Domains

OPCG clock domain registers may be either included in the regular scan chains or loaded separately through sidescan. Loading registers that are included in the scan chains is described in section "Scannable Registers" on page 68. Registers loaded through sidescan are similar to Internal Domain Registers and must be loaded through a setup sequence (which is referenced from a user test sequence). However these sidescan registers are hidden behind a clock PPI (and perhaps some control signal PPIs). The clock PPI is pulsed explicitly in a user test sequence and should match what the programming dictates. As a result, ATPG does not need to know how to set the registers that control the clock PPI pulses to generate patterns; the Linehold object is not required in the setup sequence unless there are additional control signal PPIs whose values are derived from other OPCG programming registers not controlling the clock PPI. The Load_OPCG_Controls event loads the registers from the ATE.

Protecting Domain Crossings with SDC

Read the SDC into Encounter Test using the read_sdc command. SDC set_false_path/ set_multicycle_path statements are used to protect paths that have timing issues. The -hold constraints apply during both static and dynamic ATPG, while -setup constraints apply only to dynamic ATPG.

```
read_sdc testmode=<tm> sdc=<sdcfile>
```

To remove the SDC constraints for subsequent ATPG runs, use the following command:

```
remove_sdc testmode=<testmode>
```

Creating Domain Crossing Constraints Automatically

If an SDC is not used, `prepare_domain_constraints` identifies potential domain crossings and creates constraints that prevent capturing data transitions across domains. These constraints are similar to SDC `set_falsepath -hold` constraints. If cross-domain transitions occur during a given clock pulse, the capturing flops are X'd out. Domain crossing constraints apply during both static and dynamic ATPG.

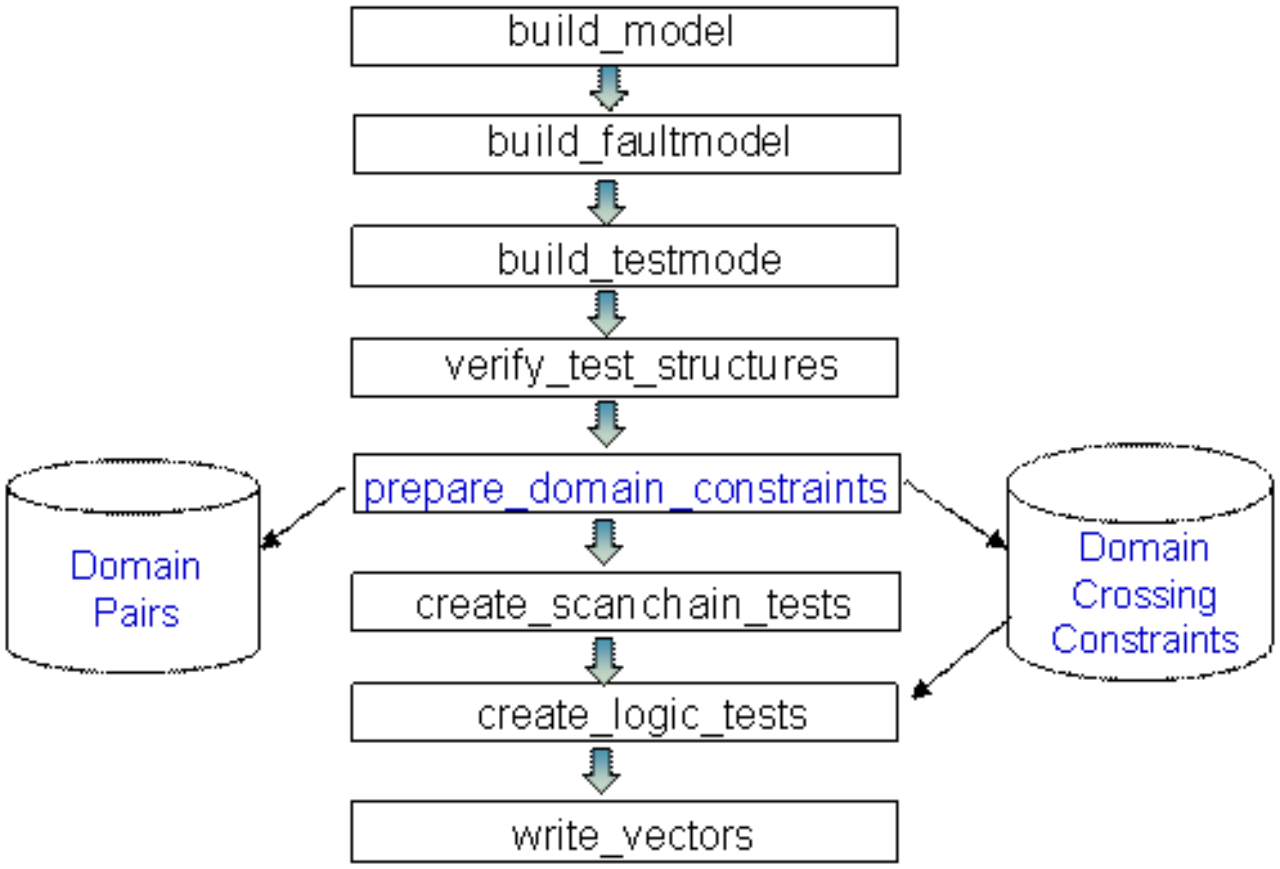
The basic command syntax is as follows:

```
prepare_domain_constraints testmode=<tm>
```

Refer to prepare_domain_constraints in the *Encounter Test: Reference: Commands* for more information on the command syntax.

The following figure shows how `prepare_domain_constraints` is integrated in the ATPG flow. You can run `prepare_domain_constraints` any time after `build_testmode` and before ATPG.

Figure 3-4 prepare_domain_constraints in ATPG Flow



The following are treated as potential clock sources when identifying domains:

- Clock Primary Inputs (PIs)
- Data Primary Inputs (PIs) whose transitions are launched through ATE internal clocks
- Clock Pseudo-Primary Inputs (PPIs)
- Internal Domain Roots
- User specified internal domain points (pin names)

Each clock source is traced forward to identify all latches, flops, RAMs or ROMs in the domain.

Note: Paths that include lockup latches or are rising edge feeding falling edge paths are typically considered safe and are not protected by constraints. To create constraints for these paths, specify `safelockup=no`.

To remove the domain crossing constraints so that they are not used during ATPG, run the following command:

```
remove_domain_constraints testmode=<tm>
```

User-specified Clock Domains

It is possible to identify a arbitrary point in the clock path as a clock domain root, even if it has not been defined as a PI, PPI, or an Internal Domain or OPCG Clock Domain Register. All the latches, flops, and memories that are fed from that point in the clock path are treated as a separate domain. Constraints are generated to protect this domain from all other clock domains (including the original clock source). To identify an arbitrary point as a clock domain root, create a file (for example, `myclocks`) with one statement for each arbitrary clock domain root, with the following format:

```
include pin <instance_pin_name> ;
```

To exclude a PI, PPI, Internal Domain, or OPCG Clock Domain pin from consideration as a clock domain, specify:

```
exclude pin <instance_pin_name> ;
```

Then run `prepare_domain_constraints` providing this file as input:

```
prepare_domain_constraints inclockfile=<myclocks>
```

Checking Clock Domain Control Registers

The `prepare_domain_constraints` command automatically checks some internal domain and OPCG clock domain register types to check if they adequately prevent transitions from crossing clock domains, when they are programmed. Registers defined in the testmode as block_input are checked to verify that data from another domain is not captured along the data path to the domain being checked. A register value name containing the string "block" is assumed to be the blocking state of the register.

Checking Domain Blocking Registers

Internal Domain Registers defined as clock_gate that have a register value name of functional_clock_gate are checked to see if data from another domain affects the clock path of the domain being checked.

If a domain crossing exists that cannot be blocked by either a "block_input" or "functional_clock_gate" register, TCE-401 warning message is generated.

Although the probability exists for invalid data to be captured by the receiving state element, `prepare_domain_constraints` automatically generates constraints to prevent invalid data from being captured.

Note: Paths that include lockup latches or are rising edge feeding falling edge paths are generally considered safe and fencing is not checked. To check these paths, specify `safelockup=no`.

To disable the checking, specify:

```
prepare_domain_constraints checkfencing=no …
```

Checking for Scan Path Transitions in Delay Test

`prepare_domain_constraints` provides a separate check for at-speed delay testing to ensure that transitions are not captured along the scan path. Typically, delay test is run with the Scan Enable signals held opposite to their Scan State value. However, if, for instance, the domain blocking logic was implemented incorrectly, it may be possible to capture data along the scan path by mistake. To check for scan path transitions, specify:

```
prepare_domain_constraints checkscancapture=yes
```

The warning message TCE-416 is printed for each flop that can capture data along the scan path.

A falsepath setup constraint is also created for each of these paths to protect it during delay test.

Reporting Domain Pairs

`prepare_domain_constraints` lists all the interacting domain pairs in the file, *<workdir>*`/testresults/domainPairs.`*<testmode>*. It may be used to identify problems with fencing that need to be fixed in the design, or to determine the domains that can be clocked together. The first entry in the domainPairs file is the header with the following format:

```
/ Domain Pair Crossing File - Created By Encounter Test
// ------------------------------------------------------------------------
-------------------------------------
// Version       : <Encounter(R) Test and Diagnostics version being used>
// Creation Date : <date>
// Created By    : <user name>
// Top Level Cell: <cell name>
// indomainfile  : none
```

The next entry in the file is the following data:

```
//------------------------------------------------------------------------
//List of Domains (sorted by decreasing size)
//------------------------------------------------------------------------
//   Type     # Nodes    Pin Name    (Cell Name)
//  -------   --------  ----------   ------------
     PPI       12305     <pin1>       OPCG_CELL
     PI        11658     <pin2>
     PPI       3460      <pin3>       OPCG_CELL
     PI        200       <pin4>
```

The list of clock domains gives an indication of the size of each domain. The number of nodes does not include inactive logic or nodes that are set to a constant value (such as a TI or TC). Following the list of clock domains are the domain pairs and their fencing status.

```
//------------------------------------------------------------------------
//Domain Pair Fencing Status
//format: <fence_type> domain_root <r1> [domain_pll <p1>]
//domain_root <r2> [domain_pll <p2>] ;
//<fence_type> = SAFE        - domain crossing is marked safe by user.
//NOT_SAFE    - domain crossing is marked not safe by user
//                              (domains will not be grouped).
//FENCED      - domain crossing is fenced by block_inputs/clock_gating
//                              registers.
//NOT_FENCED  - domain crossing is not fenced by a clock_gate register
//                              or by a block_inputs register.
//NOT_VERIFIED- domain crossing not checked for fencing
//                              (checkfencing=no specified).
//------------------------------------------------------------------------
FENCED domain_root <domain1> domain_root <domain2> ;
NOT_FENCED domain_root <domain1> domain_root <domain3> ;
…
```

In addition to the above data, the domainPairs file also includes `STIMGO` statements that list the pins that are defined as `GO` signals in the testmode (when a design has OPCG logic, `GO` signals are used to intiate the clock pulses).

If domains are driven from a PLL, the PLL output is also listed. The domainPairs file may be given a different name by specifying the keyword `domainfile=`*`<filename>`*.

Note: Names in the `domainPairs.`*`<testmode>`* file may be enclosed in quotation marks.

Reporting Flop to Flop Domain Crossings

Specify the following to display the summary matrix of the number of crossings between domains:

```
prepare_domain_constraints...reportdomainmatrix=yes
```

The following information is printed in the output log:

```
INFO (TCE-402): Clock Domain Identification Started. [end TCE_402]
Domain#     Type       Pin Name
--------   --------   ----------
1          INTERNAL   ir1.clkout
2          INTERNAL   ir2.clkout
3          INTERNAL   ir3.clkout
4          PI         clk
5          PI         olc
INFO (TCE-403): Clock Domain Identification Completed. 5 Domains Identified. [end
TCE_403]
INFO (TCE-404): Domain Crossing Identification Started. [end TCE_404]
Domain Crossing Matrix:
                 1         2         3         4         5
      ________  ________  ________  ________  ________
    1 |        .         .         1         .         .
    2 |        .         .         1         .         .
    3 |        .         .         .         .         .
    4 |        .         .         .         .         .
    5 |        .         .         .         .         .
      ________  ________  ________  ________  ________
                1         2         3         4         5
INFO (TCE-405): Domain Crossing Identification Completed. [end TCE_405]
```

The report indicates that there is one domain crossing between domains 1 (`ir1.clkout`) and 3 (`ir3.clkout`), and one domain crossing between domains 2 (`ir2.clkout`) and 3 (`ir3.clkout`).

To see the individual flop to flop domain crossings, specify:

```
prepare_domain_constraints …. reportdomaincrossings=yes
```

The domain crossings are written to the log file in the following format:

```
INFO (TCE-419): Domain ir1.__i2.DOUT, State element id1.if4.__i0, feeds domain
                ir3.__i2.DOUT, state element: id3.if1.__i0.  [end TCE_419]
INFO (TCE-419): Domain ir2.__i2.DOUT, State element id2.if4.__i0, feeds domain
                ir3.__i2.DOUT, state element: id3.if1.__i0.  [end TCE_419]
```

Reporting Clock Domain Groupings

`prepare_domain_constraints` collects non-crossing and safe domains together into groups of domains that can be pulsed together. These groups are reported in the `domainPairs.<testmode>` file in the testresults directory.

```
// ------------------------------------------------------------------------
// Recommended Domain Groupings
//
//  format: Group n (yy%): r1 r2 ... rn ;
//    n   - group number
//    yy  - percentage of nodes clocked by this group (vs all nodes clocked by all
//          groups)
//    ri  - A domain root in the group
// ------------------------------------------------------------------------

Group 1 (23.17%):
    pin1
    pin2
;

Group 2 (18.5%):
    pin3
    pin4
;
…
```

To disable the grouping report, specify `writegroups=no`.

To control the amount of switching caused by the domain groupings (for low power ATPG), specify `maxgrouppercent=`*`nn`* where *`nn`* is an integer percentage between `0` and `100`. `prepare_domain_constraints` determines the number of nodes clocked by each domain and avoids grouping domains that would cause the percentage of nodes to exceed the value specified for `maxgrouppercent`.

To prevent two non-overlapping domains from being grouped together, specify the domain pair as NOT_SAFE (refer to "Declaring Unsafe Domain Pairs" on page 78 for more information).

Declaring Safe Domain Crossings

There may be domains in the design that can be pulsed together even though they appear to interact. If you decide that such domains are safe to be pulsed together, provide the information to `prepare_domain_constraints` so that the domain registers are not checked and constraints are not created to protect the crossings.

Identifying Safe Domain Pairs

To identify safe domain pairs, run `prepare_domain_constraints`. Copy the `domainPairs.<testmode>` file and update it - replacing the existing `fence_type` with the value `SAFE`.

Original domainPairs file (`domainPairs.<testmode>`):

```
FENCED       domain_root <domain1> domain_root <domain2> ;
NOT_FENCED   domain_root <domain1> domain_root <domain3> ;
Modified domainPairs file (mysafe.domains):
FENCED       domain_root <domain1> domain_root <domain2> ;
SAFE      domain_root <domain1> domain_root <domain3> ;
```

Then rerun `prepare_domain_constraints` using this file as input:

```
prepare_domain_constraints indomainfile=mysafe.domains
```

You can either leave other domain pairs (those not marked SAFE or NOT_SAFE) in the input file or remove them. They are ignored when read in as input.

Note: Ensure to specify the `inclockfile` if specified in the original run so that any user-defined clock domains is recognized.

Safe Capture Flops

You can declare domain crossings as safe at a more granular level by identifying individual capture flops that are exempt from checking. Adding an Encounter Test `SUPPRESS_MSG` attribute to a specific technology cell or instance prevents any potential TCE-401 or TCE-416 messages and also prevents the creation of constraints required to protect the flop. A sample usage is shown below:

```
(* TYPE="CELL" *)
module domain(out, bi, so, in1, in2, in3, clk, se, si);
output out, so;
input in1, in2, in3, clk, se, si, bi;
wire q1, q2, q3, q4, a1, o1;

// Suppress message on if1
(* SUPPRESS_MSG="CCE401" *)
scanflop if1 (q1, bi? ~q1 : in1, se, si, clk);

// Suppress message on if2
(* SUPPRESS_MSG="CCE416" *)
scanflop if2 (q2, bi? ~q2 : in2, se, q1, clk);

scanflop if3 (q3, in3, se, q2, clk);
and ia1 (a1, q1, q2);
or  io1 (o1, a1, q3);
scanflop if4 (out, o1, se, q3, clk);
assign so = out;
endmodule
```

Refer to the *Encounter Test: Guide 3: Test Structures* for more information on the `SUPPRESS_MSG` attribute.

Declaring Unsafe Domain Pairs

There may be domains in the design that do not cross but for some reason you do not want them to be grouped together. Similar to declaring safe domain crossings, specify the value `NOT_SAFE` for `indomainfile`. If the domains do not cross, there will not be an entry for the pair in the original domainPairs file - so it must be added before marking it NOT_SAFE. Now when you rerun `prepare_domain_constraints`, the command puts the domains in separate groups.

Checking Constraints For Register Programming

You can provide a specific set of register programming information when running `prepare_domain_constraints`. This programming information is used when the domain crossings are identified. This can be useful to get an early indication of the constraints that are going to be used during ATPG given the specific set of programming values. If the programming successfully prevents all domain crossings, no constraints should be identified. If constraints are identified, the programming does not prevent all possible domain crossings.

Below are some sample specifications of programming information:

```
prepare_domain_constraints setupsequence=<mysetup>// internal domains
prepare_domain_constraints linehold=<mylineholds>// scanchain registers
prepare_domain_constraints testsequence=<ts>// OPCG domains
```

Note: If you run `prepare_domain_constraints` with specific programming data, you must still provide this programming data during ATPG if `writesetup=no` is specified. If `writesetup=yes` is specified, the programming data will be included in the setupsequences created by `prepare_domain_constraints`.

Checking Domain Crossings for Macro

`prepare_domain_constraints` checks for domain crossings if run on a macro. The clock sources may be any of the sources supported for a complete design (such as PIs, PPIs, and internal_domains.)

Chip-level PI clock domains that cross are normally not pulsed together in the same test, so they are not checked or protected by default. Macro PI clocks may come from the same chip-level clock source and should be checked. To check macro PI clocks, specify the following:

```
prepare_domain_constraints macro=yes
```

Asynchronous Set/Reset Clocks

To exclude set/reset or other asynchronous clocks from checking when `macro=yes` is specified, add the `TB_DOMAIN_CROSSING="NOT_SAFE"` attribute to the PI or specify the domain as `NOT_SAFE` in the `indomainfile`.

For example, to specify that the identified reset clock should not be pulsed with other clocks (and does not need to be protected), specify:

```
module myrlm (out, in1, in2, reset_clock, clock1, … );
       output out;
…
(* TB_DOMAIN_CROSSING = "NOT_SAFE" *)
       input reset_clock;  //  reset_clock will not be pulsed with other clocks,
                           //  so constraints are not necessary
input clock1;
…
endmodule
```

Fencing Signals on Macro

If the clock source comes into the macro as a primary input, there is a probability that any fencing signals may also be coming into the macro as primary inputs. You can correlate PIs carrying block_inputs and functional_clock_gates type signals with their corresponding PI clock by adding attributes to the fencing logic pins in the Verilog design source:

```
module myrlm (out, in1, in2, block_inputs1, clock1, functional_clock_gate2, clock2…
);
output out;
…
(* TB_SE_INPUT = "clock1" *)
input block_inputs1;  //  block_input signal for clock1
input clock1;

(* TB_SEFCG = "clock2" *)
(* TB_POLARITY = "0" *)
input functional_clock_gate2; // func. clock gate for clock2
// 0 value enables fencing (blocks the gating)
input clock2;
…
endmodule
```

`prepare_domain_constraints` sets the block_inputs and functional_clock_gates PI signals when checking the fencing for the domains associated with them. The default value placed on these PIs when performing the fencing checks is logic 1, but that may be overridden by placing a `TB_POLARITY` attribute on the pin, as in the example given above.

Checking PI to Flip-Flop Paths

`prepare_domain_constraints` includes macro data PIs when checking for domain crossings. That is, paths between PIs and capturing flops are considered as domain crossings. Data PIs are typically assumed to belong to some unknown clock domain called "The PI Domain".

Associating Data PIs with PI Clock Domains

You can associate a data PI with a specific PI clock, if you know that the chip-level source of the PI will be clocked by the same clock as the clock PI. To associate a data PI with a PI clock, add the `TB_DOMAIN_CLOCK` attribute to the data PI:

```
module myrlm (out, in1, in2, block_inputs1, clock1, functional_clock_gate2, clock2…
);
output out;
…
input clock1;
(* TB_DOMAIN_CLOCK = "clock1" *)
input in1; // in1 is fed by a flop clocked by the clock1 domain
…
endmodule
```

Exempting data PIs from Checking

If you do not want to include a specific data PI in the domain crossing checking, make it exempt by adding the `TB_DOMAIN_EXEMPT` attribute on the data PI:

```
module myrlm (out, in1, in2, block_inputs1, clock1, functional_clock_gate2, clock2…
);
output out;
…
input clock1;
(* TB_DOMAIN_EXEMPT = "yes" *)
input in1; // domain crossings from in1 to anywhere are ignored.
…
endmodule
```

Using Checkpoint and Restart Capabilities

At times, the ATPG job can be terminate abnormally before completing. This section discusses the techniques provided by Encounter Test to recover the application data in such cases.

A checkpoint allows the application to save its current results. You can set the checkpoints for the application by using the `checkpoint=<value>` keyword, where value is the number of

minutes between each checkpoint. The default is 60 minutes. For example, if you specify `checkpoint=120`, Encounter Test sets up checkpoints and saves application data after every two minutes.

You can use the checkpoint data to restart in case of network or machine failure during the execution of the application.

Note: Some processes cannot be interrupted to take a checkpoint. If one of these processes takes longer than the specified number of minutes, the next checkpoint will be taken as soon as the process ends.

Using the Andrew File System (AFS) might occasionally result in the checkpoint files getting out of sync, which cannot be restored for restart. However, this problem has never occurred when using local disk or NFS.

To restart the application using checkpoint data, re-run the same ATPG command after adding the following:

`restart=no|yes|end`

Specifying `restart` allows you to restart an application that produced a checkpoint file before it ended abnormally. If a checkpoint file exists, the default is to restart the application from the checkpoint and continue processing (`restart=yes`).

Use `restart=end` to restart the application from the existing checkpoint but immediately cleanup, write out results files, and then end.

Use `restart=no` to start the application from the beginning instead of using an existing checkpoint file.

If you use the `restart` keyword, Encounter Test ignores any other specified keywords, that is, only the keyword values from the original command line are used.

Committing Tests

By default, Encounter Test makes all test generation runs as uncommitted tests in a test mode. Commit Tests moves the uncommitted test results into the committed vectors test data for a test mode. Refer to "Performing on Uncommitted Tests and Committing Test Data" on page 246 for more information on the test generation processing methodology.

Writing Test Vectors

Encounter Test writes the following vector formats to meet the manufacturing interface requirements of IC manufacturers:

- Standard Test Interface Language (STIL) - an ASCII format standardized by the IEEE.
- Waveform Generation Language (WGL) - an ASCII format from Fluence Technology, Inc.
- Verilog - an ASCII format from Cadence Design Systems., Inc.

Refer to "Writing and Reporting Test Data" on page 181 for more information.

4

Delay and Timed Test

This chapter provides information on the concepts and commands to perform delay and timed ATPG with Encounter Test.

With current technologies, more and more defects are being identified that cause the product to run slower than its designed speed. As stuck-at testing is a slow test that is only designed to check the final result, paths with above average resistance or capacitance due to impurities or a bad etch can go undetected. These defects can change the speed of a path to the point where it is outside of the typically thin tolerance designed in to the functional circuitry. This can result in a defective product.

Delay defects take the form of spot and parametric defects. An example of the spot defect is a partially open wire that adds resistance to the wire thereby slowing the signal passing through the wire. A parametric defect is a change in the process that causes a common slight variation across multiple gates, which in turn causes the arrival of a signal or transition to take longer than expected. An example of a problem causing a parametric failure would be an increase in the transistor channel length.

Encounter Test provides Manufacturing (slow to rise and fall faults) and Characterization (path delay) Tests for identifying spot and parametric defects. Encounter Tests provides a use model script that allows you to create the model to dynamic ATPG test generation. Refer to "Using the True-Time Use Model Script" on page 39 for more information on the script

The following sections discuss the details about the commands that constitute the true-time use model.

The following steps represent a typical delay ATPG use model flow for manufacturing defects:

- Scan chain and reset fault testing
  - create_scanchain_delay_test - refer to "Creating Scan Chain Delay Tests" on page 88
- Logic Testing
  - read_sdf - refer to "Performing Build Delay Model (Read SDF)" on page 92

- prepare_timed_sequences - refer to "Performing Prepare Timed Sequences" on page 123
  - AT Speed and Faster than AT Speed Tests can be achieved by specifying the appropriate clock frequency in a clock constraint file.
- create_logic_delay_test - refer to "Create Logic Delay Tests" on page 126

- Exporting and saving patterns
  - commit_tests - refer to "Committing Tests" on page 245
  - write_vectors - refer to "Writing Test Vectors" on page 181

The following steps represent a typical delay ATPG use model flow for Characterization defects (path test)

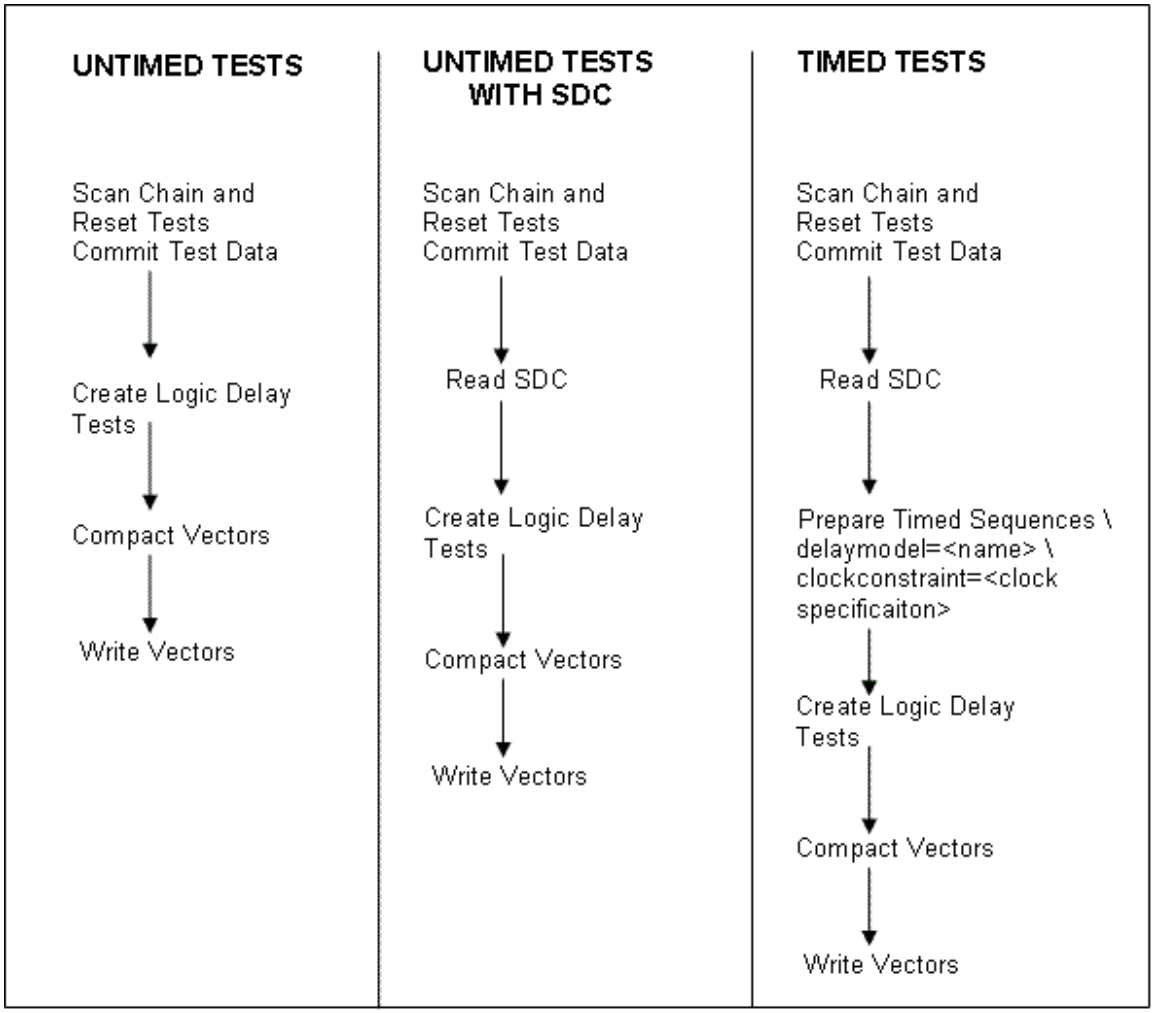
- create_path_delay_tests - refer to "Performing Create Path Tests" on page 138
- Exporting and saving patterns
  - commit_tests - refer to "Committing Tests" on page 245
  - write_vectors - refer to "Writing Test Vectors" on page 181

Refer to the following sections for more information on delay faults:

- "Delay Defects" on page 144
- "Delay ATPG Patterns" on page 145
- "Testing Manufacturing Defects" on page 85
- "Timing Concepts" on page 96
- "Delay Path Calculation" on page 105
- "Characterization Test" on page 137
- "Dynamic Constraints" on page 153

Refer to the following figure for a high level flow of the various delay test methodologies:

Figure 4-1 Flow for Delay Test Methodologies



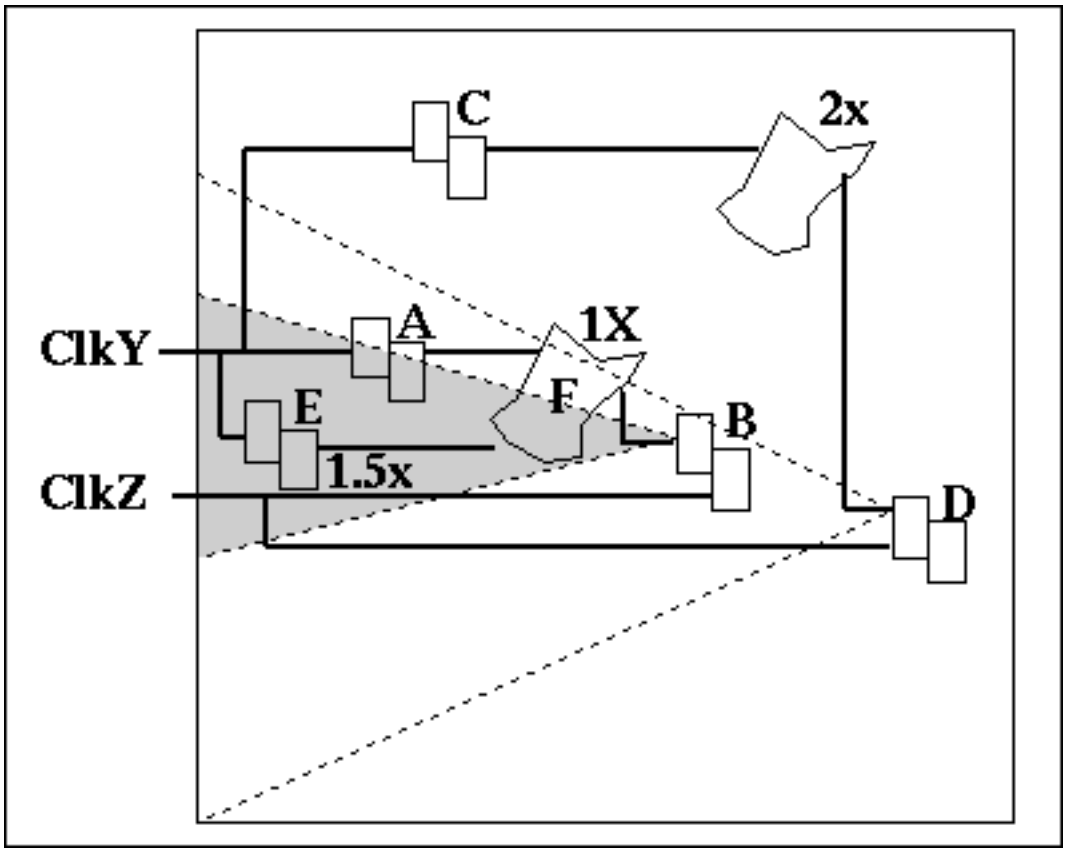
Note: It is recommended to use Timed Tests whenever possible.

Testing Manufacturing Defects

Manufacturing delay test is a full-chip transition fault test useful for finding random defects across the entire chip. As designs get faster, technology smaller, and more paths closer to the critical arrival time, manufacturing delay test assists in assuring quality levels. True-Time Test is a streamlined method for running manufacturing test.

In Figure 4-2, a transition fault F exists in the logic between registers A and B. See Figure 4-19 for an example of a transition fault.

Figure 4-2 Delay Testing and Effects of Clocking and Logic in Backcones



- The logic between A and B operates at a speed of 1x.
- The transition to test the fault is released (launched) by the C; lY from register A, and captured by the ClkZ in register B. However, this ClkY then ClkZ sequence may initiate other transitions in the design. For example, a transition in logic from register C to register D. The logic in this part of the design may operate at a speed of 2x (that is, twice as slow).
- By default, Encounter Test will time all ClkY-ClkZ sequences to the longest (slowest) path in the system. This is to ensure the tests will work in manufacturing (the responses will be accurate at the tester).

The following methods impact the tests and their timings:

- Lineholds, see “Linehold File” on page 171

- Test function input pin switching (1->0, 0 ->1). See "Identifying Test Functions Pins in the *Encounter Test: Guide 2: Testmodes* for related information.
- Clock domains (controlled by lineholds or constrained timings). See CLOCKDOMAIN in the *Encounter Test: Guide 2: Testmodes* for more information.
- Early or late modes in the standard delay file, refer to "Timing Concepts" on page 96
- Constrained timings, as described below.

It is realistic to assume that the tests produced to accommodate all timings will not effectively test the fastest (1x) transitions. Those transitions must be operated twice as fast as the 2x transitions if they are to operate at system speed in the design. Encounter Test can do this through a technique known as Constrained Timings. This technique allows you to specify options to exclude consideration of any paths greater than a certain cycle time (length). If a register is fed by any path longer than the specified cycle time, the register is marked as not observable causing no further detections to be possible at this register for this given timing sequence. If a transition can occur in the specified cycle time or less, the register will be measured and fault effects can be detected.

The timings specified with constrained timings can be *tight* timings for a given clock sequence that allows the shorter paths to complete while the longer paths are ignored. When consecutively run, tests for all timings (for all timing domains in the functional design) can be generated for manufacturing by appending runs with shorter-to-longer timings after each other. When appending runs, the fault lists (detected and undetected) are transferred between the runs to accelerate follow-on processes. This approach detects faults at the fastest speeds that they can be measured. The result is an increased pattern count but with higher quality test patterns in terms of ability to detect real defects.

Lineholds and/or test inhibits can also be used to *shut down* paths in the design. Once a path is no longer active, it is no longer considered by Encounter Test. Only the longest active path will be used for any given clock sequence. Automatic, At-Speed, and Faster Than At-Speed flows can automatically generate constraints for each clocking sequences which can also shut down long paths with respect to timing. While the test generator will target the longer paths, the simulator can still detect faults on shorter paths.

Since the SDF data can contain best, normal and worst case timings, these values can be used as is or can be linearly scaled to test for particular process points (like Vdd and/or temperature). When this is done, Early Mode and Late Mode linear combination settings can be used during test generation to affect the distribution of delays (variance of normal distribution).

The methodology most used to produce a range of tests for given cycle times on the design is to make runs with different constrained timings. That is, run with the fastest (shortest) cycle times first. Subsequent runs are then used to increase the constrained times until the slowest

(longest) cycle times are reached. A common approach with this technique is when multiple clock domains exist on the design. The cycle times for each clock domain are used as the specification of the constrained timings.

Manufacturing Delay Test Commands

This section discusses the various commands used to test manufacturing delays in a design.

Creating Scan Chain Delay Tests

This command generates a scan chain test targeting dynamic faults along the scan paths. After the scan patterns are created, they can be committed to save the patterns and fault status for future ATPG runs.

To create dynamic scan chain tests using the graphical interface, refer to Create Dynamic Scan Chain Tests in the *Encounter Test: Reference: GUI*.

To perform Build Delay Model using command line, refer to "create_scanchain_delay_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `create_scanchain_delay_tests` command is given below:

```
create_scanchain_delay_tests workdir=<directory> testmode=<modename>
experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode
- `experiment`= name of the test patterns

Prerequisite Tasks

Before executing `create_scanchain_delay_tests`, build a design, testmode, and fault model. Refer to the *Encounter Test: Guide 1: Models* for more information.

Output

The output log, as shown below, will contain a summary of the number of patterns generated and their representative coverage. Both static and dynamic faults should be tested.

```
*************************************************************************
           ----Stored Pattern Test Generation Final Statistics----
               Testmode Statistics: FULLSCAN
                       #Faults  #Tested  #Redund  #Untested  %TCov   %ATCov
Total Static             908      425       0        437       46.81   46.81
Total Dynamic            814      316       0        498       38.82   38.82
                     Global Statistics
                       #Faults  #Tested  #Redund  #Untested  %TCov   %ATCov
Total Static             1022     425       0        551       41.59   41.59
Total Dynamic            1014     316       0        698       31.16   31.16
*************************************************************************
                  ----Final Pattern Statistics----
Test Section Type                          # Test Sequences
-------------------------------------------------------
Scan                                         2
-------------------------------------------------------
Total                                        2
```

Debugging No Coverage

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

Additional tests

There are additional scripts to help test the set and reset faults on scan flops. These ATPG commands are not supported officially but are available to achieve higher test coverages.

Delay Scan Chain Overview

When generating a delay scan chain test (`create_scanchain_delay_tests`), Encounter Test generates a total of two scan patterns, which consist of the following structure:

- `Scan_Load` - Load the scan chain bits with repeating 1000111 patterns. This tests the slow to rise and slow to fall transition faults in the scan chain.
- `Stim_PI` - Stay in scan state
- Delay shift #1
  - `Stim_PI` - Load the next value on scan inputs to continue 1000111 pattern from scan input

- `Pulse` - Pulse the scan clocks
- `Measure_PO` - Measure values on scan outputs
- `Stim_PI` - Load the next value on scan inputs to continue 1000111 pattern from scan input
- `Pulse` - Pulse the scan clocks
- `Measure_PO` - Measure values on scan outputs

`.....`

- Delay shift #8
  - `Stim_PI` - Load the next value on scan inputs to continue 1000111 pattern from scan input.
  - `Pulse` - Pulse the scan clocks
  - `Measure_PO` - Measure values on scan outputs
  - `Stim_PI` - Load the next value on scan inputs to continue 1000111 pattern from scan input.
  - `Pulse` - Pulse the scan clocks
  - `Measure_PO` - Measure values on scan outputs
- `Scan_Unload` - Observe all scan bits
- `Scan_Load` - Load the scan chain bits with repeating 110 patterns. This tests the clock slow-to-turn-off faults.
- `Stim_PI` - Stay in scan state
- Static Shift
  - `Stim_PI` - Load the next value on scan inputs to continue `110` pattern
  - Pulse - Pulse scan clocks
  - Stim_PI - Load the next value on scan inputs to continue `110` pattern.
- Delay Shift #1
  - `Pulse` - Pulse scan clocks
  - `Stim_PI` - Load the next value on scan inputs to continue 110 pattern from scan input
  - `Pulse` - Pulse scan clocks

- `Measure_PO` - Measure values on scan outputs
- `Stim_PI` - Load the next value on scan inputs to continue `110` pattern from scan input

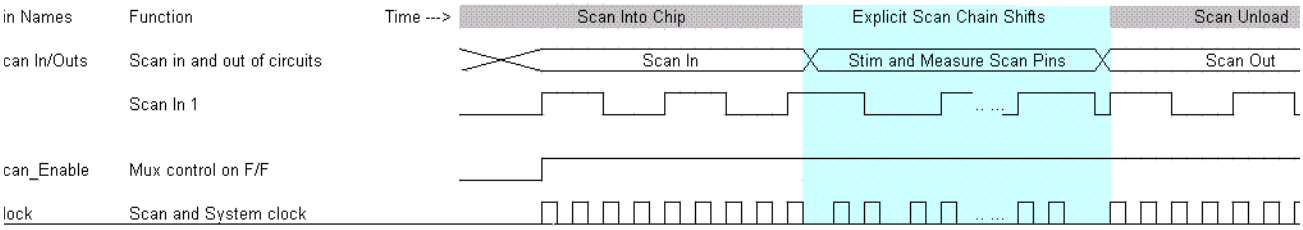
- `.....`
- Delay Shift #5
  - `Pulse` - Pulse scan clocks
  - `Stim_PI` - Load the next value on scan inputs to continue 110 pattern from scan input.
  - `Pulse` - Pulse scan clocks
  - `Measure_PO` - Measure values on scan outputs
  - `Stim_PI` - Load the next value on scan inputs to continue 110 pattern from scan input
- Static Pulse
  - `Pulse` - Pulse scan clocks
  - `Scan_Unload` - Observe all scan bits

In summary:

- The delay scan chain test is always in the scan state.
- All clocks are configured at scan speeds.
  - Clocks are not at system speed as scan paths are typically not timed to system speeds.
- Additional clocking and sequences are added while testing for compression, MISRs, or masking.
- The order of sequence can slightly change based on custom-scan protocols and custom-scan preconditioning.

The following figure shows a delay scan chain test diagram:

Figure 4-3 Dynamic Test Waveform



Performing Build Delay Model (Read SDF)

This command reads a Standard Delay File (SDF) and creates an Encounter Test delay model that represents the timings. For more information about the data retrieved and used from the SDF, refer to:

- "Timing Concepts" on page 96
- "Delay Path Calculation" on page 105
- "Specifying Wire Delays" on page 108

To perform Build Delay Model using the graphical interface, refer to "Build Delay Model" or "Read SDF" in the *Encounter Test: Reference: GUI*.

To perform Build Delay Model using command lines, refer to "build_delaymodel" or "read_sdf" in the *Encounter Test: Reference: Commands*.

The syntax for the `read_sdf` command is given below:

```
read_sdf workdir=<directory> testmode=<modename> delaymodel=<name>
sdfpath=<directory> sdfname=<sdf name> clockconstraintsfile= <file name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for test
- `delaymodel`= Internal Encounter Test name of the delay model. Multiple names can exist for a delay model at the same time.
- `sdfpath`= directory in which sdf file exists
- `sdfname`= name of sdf file

- `clockconstraints` = the file that includes clocking constraints. The clock domains that are outside the domain specified in the file are not used in the delay test generation. This ensures that the command only generates messages that are relavant to the clock domains being tested.

  Alternatively, you can specify either of the following keywords to limit the clock domains that will be checked while building delay model:

  - `testsequence` - The release/capture clocks are extracted from the specified sequences and used to determine the relevant clock domains
  - `dynseqfilter` - the type of sequences that the dynamic test generation is allowed to generate. The default value is any for LSSD testmode, which means that the command will only check for intra domains in such case.

Prerequisite Tasks

Complete the following tasks before executing Build Delay Model:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.

   When importing a design model with the intention of running timed delay tests, ensure that your levels of hierarchy are correct such that the highest level of hierarchy with the `CELL` or `BOOK` attribute is the layer at which you have delay information.

2. Create a Test Mode. Refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.

   The test mode is required to establish the locations of clock pins and can also be used to establish a design state through tied values and lineholds. The delay model data is not tied to a particular test mode and may be used with any test mode. Maintain awareness that any checks performed when building the delay model will all be interpreted through the selected test mode. Therefore, if a given test mode has certain paths disabled, the delay model build procedure will not check for delays along these paths, nor will it include any delay information along these paths in the delay model. This could result in missing delay information in another test mode where these paths are enabled.

   If using multiple test modes for a design, the recommended practice is to build a delay model for each test mode unless the test modes are very similar.

3. Create an SDF file.

   Use the tool of choice such as ETS (Encounter Timing System) to create an SDF file for the expected conditions (voltage levels, temperature) to run at the tester.

   Note: When reading an SDF, Encounter Test expects to map its delay information to the

cell or block level of hierarchy within the Encounter Test design model. Refer to "Delay Timing Concepts" on page 128 for more information.

Optional Input Files

- `TDMcellDelayTemplateDatabase` file

  This is not a required input and is generated as required. Refer to "Customized Delay Checking" on page 152 for details.

  This binary file contains a list of timing arcs that are expected to be present for various kinds of cells. The information is used for checking the completeness of the SDF. It can be customized or automatically generated. The recommended timing arcs are derived from the circuit model within Encounter Test. The application checks the topological paths that can exist through each cell (to determine IOPATHs and INTERCONNECTs required) and the pins that feed flops and latches (to determine where SETUP and HOLD checks are needed).

  If the delay information in this file does not match the information in the SDF, Encounter Test generates the warning message TDM-041.

  Note: Sometimes Encounter Test asks for over-specifications, such as setup checks on reset lines (where you will never want to create a delay test) and some functionally disabled paths (test-only paths). You can ignore the missing delay information warnings as long as you do not create delay tests in these areas.

  It is recommended that you provide the complete delay information because missing delays (which were recommended during read_sdf) will result in the timing engine not considering these paths during timings. Any missing piece of information tells the timing engine that anything downstream from the point of missing information will not affect the timing or has no timing requirements of its own. Therefore, for example, if delay information is missing for a real path, which is a multi-cycle path, the timing engine will be unable to realize that it is a multi-cycle path, and will expect to measure it successfully at speed (resulting in a miscompare).

Output

This task generates a list of cells with missing or unnecessary delays. The task also creates an Encounter Test delay model in the tbdata directory.

Debugging Common Messages

Messages with IDs TDM-013, TDM-014, TDM-041, TDM-051, TDM-055 represent delay model errors and need to be resolved, failing which may cause incomplete timings.

The following table lists some common delay model error messages and the corresponding debugging technique:

Table 4-1 Common Delay Model Error Messages

Error Message	Description
TDM-205	This message reports the number of timing templates created using the `read_sdf` command. The number reported by this message should match the number of different library cells used in the design. Refer to TDM-205 in the *Encounter Test: Reference: Messages* for more information.
TDM-041	A recommended delay was not specified. `0`ns delay paths may not be entered within an SDF. This message is generated if a delay between a driver and receiver is not in the SDF by default. The option `read_sdf interconnectdelay=0` allows Encounter Test to assume that an unspecified path has `0` time delay. The message explicitly states the driver and receiver instance names. You may also get this message if your clock path has reconvergent fanout. There are designs where multiple parallel clock drivers are wired together and then drive the FF's. Refer to TDM-041 in the *Encounter Test: Reference: Messages* for more information.

Error Message	Description
TDM-012	Unable to create a delay between the pins. This is between two ports on the same cell instance. Probable causes for this error can be that one port is tied to a logic value or the instance referenced in the SDF is not the highest level cell in the hierarchy. Refer to TDM-012 in the *Encounter Test: Reference: Messages* for more information.

TDM-055	The specified SDF delays to a layer that is not a technology cell.
	
	`Block cell name: <viewName>`
	
	`Instance name: <instName>`
	
	`SDF line number: <sdfLine>`
	
	This indicates missing cell definitions in your netlist. Ensure that all cells in your techlib are defined.
	
	Refer to TDM-055 in the *Encounter Test: Reference: Messages* for more information.

Encounter Test supports the ability to customize the SDF compatibility checks. Refer to "Customized Delay Checking" on page 152 for information on creating TDMcellDelayTemplateDatabase files.

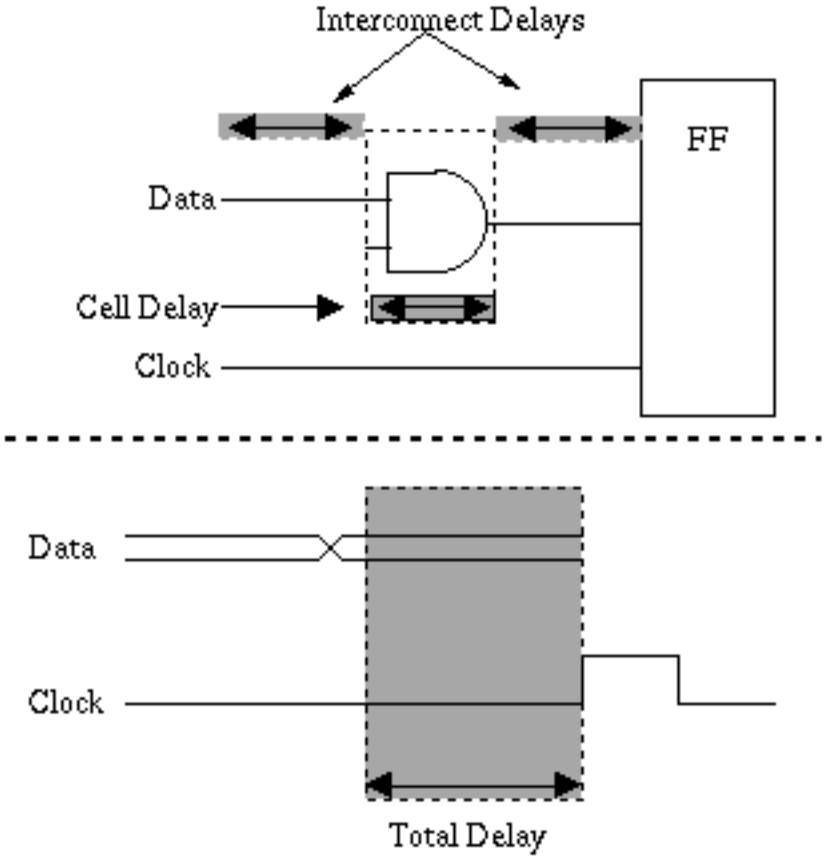
Timing Concepts

The positioning of the events within the dynamic pattern (timed section) ensures that the arrival times of each signal at a latch or PO occur at the proper time. This requires that the delays of the product's cells and interconnects be known accurate. The information is imported into Encounter Test using the Standard Delay File (SDF) and stored in Encounter Test's delay model. The delay information is generated and exported to the SDF by a timing tool, such as IBM's Einstimer or the Encounter Timing System (ETS). The method in which the timing tool is run (delay mode, voltage, and temperature parameters) has a great impact on the effectiveness of the timed tests at the tester. The range of parameters specified to the timing tool should accurately reflect the range of conditions to be applied at the tester. SDF information is required for the At-Speed and Faster Than At-speed True-Time methodologies. Automatic True-Time Test can be run without SDF timings, but will benefit from SDF timing information.

Build Delay Model creates a delay model for use in generating timing for transition tests, path delay tests, and for use in delay test simulation.

For delay testing, the SDF should contain information about all delays in the design, cell delays, and interconnect delays (see Figure 4-4).

Figure 4-4 SDF Delays that Impact Optimized Testing and the Effect in the Resulting Patterns



Note that Encounter Test reads an SDF, it expects to find delay information at the highest level of hierarchy that is a "cell" or "block" in the Encounter Test design model. This is the only valid layer of hierarchy for delay descriptions other than the very top level of the design (for delays to and from PIs and POs). Delays that are described at any other layer of hierarchy are not successfully imported into Encounter Test.

Use the Encounter Test GUI to check the hierarchical layer of a given block. Select Information Window option *Loc: Hierarchical Level* and mouse over a block to query it.

Refer to the "Information Window Display Options" in the *Encounter Test: Reference: GUI* for details.

If the levels of hierarchy in your Encounter Test design do not correlate to the SDF, force a block to be at a given level of hierarchy by adding the `TYPE` attribute to the block. For example, to force a given module to be a "CELL" in Verilog, code the following:

```
module myBlock ( in1, in2, in3, out1 ); //! TYPE="CELL"
```

Refer to the *Encounter Test: Guide 1: Models* for additional information on adding attributes to the model.

The SDF can also contain best, nominal, and worst case timings or can be produced for certain process conditions that will be used in manufacturing, such as temperature and voltage.

Build Delay Model reads the SDF and creates a delay model for Encounter Test.

Encounter Test supports SDF version 2.1, however the following 2.1 specification is not supported:

- `INSTANCE` statements with wildcards

The following features of SDF 2.1 are tolerated but not used in Encounter Test:

- `PATHPULSE` and `GLOBALPATHPULSE`
- `PATHCONSTRAINT`, `SUM`, `DIFF`, `COND`, `SKEWCONSTRAINT`
- `CORRELATION`
- `SKEW`

IEEE 1497 SDF Standard Support

Encounter Test Build Delay Model supports the following IEEE 1497 version 2.1 constructs:

COND	The COND construct allows any path delay to be made conditional, that is, the delay value applies only when the specified set of values is true. This allows for the state-dependency of path delays where the path appears more than once in the timing model with conditions to identify the applicable design state. Only the defined design values from the test mode or linehold files are honored during evaluation of conditional statements.
CONDELSE	The CONDELSE statement provides the default conditional delay to be used if none of the conditions evaluate to a logical True.
RECOVERY	The RECOVERY statement is not currently handled.
RETAIN	In an IOPATH, the RETAIN keyword specifies the time that an output port shall retain its previous logic value after a change at a related input port. The system will reads RETAIN delays and replaces the best case delay of the corresponding IOPATH delay will the best case numbers of the RETAIN triplet.
REMOVAL	The REMOVAL statement behavior is identical to the SDF 2.1 HOLD statement and is treated as an alias of the HOLD statement. Delays are merged if a HOLD and REMOVAL exists between the same two pins.
RECREM	The RECREM statement behavior is identical to the SDF 2.1 SETUPHOLD statement and is treated as an alias of the SETUPHOLD statement. Delays are merged if a SETUPHOLD and RECREM exists between the same two pins.
BIDIRECTSKEW	The BIDIRECTSKEW construct specifies limit values for bidirectional signal skew timing checks. A signal skew limit is the maximum allowable delay between two signals, which if exceeded causes devices to behave unreliably.

SCOND and CCOND	An alternative syntax can be used for SETUPHOLD and RECREM timing checks. This associates the conditions with the stamp and check events in the timing analysis tool. A stamp event defines the beginning of a measured interval, and a check defines the end of a measured interval. Separate conditions can be supplied for the stamp and check events using the SCOND and CCOND keywords. SCOND or CCOND or both SCOND and CCOND take precedence over COND. These keywords are read in for the SETUPHOLD and RECREM checks and the conditions will be evaluated in the same way as they are evaluated for the COND statement. Example: `(SETUPHOLD d clk (5) (7) (CCOND enb))` If enb is evaluated to a logical zero, then both the delays will be dropped. The construct is only honored when the specifying conditionals=yes for build_delaymodel.
PATHPULSE and PATHPULSEPERCENT	These keywords specify the pulse propagation limits (r-limit and the e-limit). Encounter Test does not use of these parameters; the SDF is parsed however the delays are ignored.

e-limit and r-limit within IOPATH statements	PATHPULSE numbers are sometimes specified with an IOPATH delay. In this case, Encounter Test parses the statement and ignores the e-limits and the r-limits.
	Example:
	`(IOPATH a y ((1:1:2) (2) (3)) ((2:3:4) (4) (5)) )`
	Both the r-limit and e-limit values are ignored in this case. The delay is read as:
	`(IOPATH a y (1:1:2) (2:3:4) )`

Note: Some of the preceding definitions are taken from the content of the document *P1497 DRAFT Standard for Standard Delay Format (SDF) for the Electronic Design Process*.

The following timing specifications are supported:

- DELAY
- TIMINGCHECK

Ignored SDF Statements and Timing Specifications

The following constructs are not tolerated and not used by Encounter Test:

- PERIODCONSTRAINT
- PATHCONSTRAINT
- DIFF
- SUM
- EXCEPTION
- NAME
- ARRIVAL
- DEPARTURE
- WAVEFORM
- SLACK
- SKEWCONSTRAINT

The following timing specifications are tolerated but not used by Encounter Test:

- TIMINGENV
- LABEL

Annotating SDF Information

SDF information is typically created using a static timing analysis tool. You can annotate this information to the Encounter Test database and automatically determine the locations of multi-cycle paths, setup violations, and hold violations.

Note: SDF annotation does not automatically determine the locations of false paths, therefore, it is recommended to use an SDC file with SDF information.

The annotated SDF information is useful for:

- Running at-speed patterns
- Running faster-than-at-speed

- Creating interdomain test patterns. SDF information ensures that paths between clock domains that cause setup or hold violations are correctly masked and will not cause miscompares.

The SDF information is mapped to the Encounter Test database by locating the correct cells in the design and attaching the information there. The hierarchy definitions in the database must match the hierarchy assumed in the SDF information for the annotation to be successful.

Tip

It is important to have complete and correctly mapped data, therefore, investigate all the messages reported during this process rigorously. Any incomplete or incorrect information can result in patterns that will fail good devices on the tester.

Use the following parameters to read and use SDF delay data. The following parameter points to the directory where the SDF file resides:

`SDFPATH=`*`sdf_directory_name`*

The following parameter points to the name of the file containing the SDF delay data, propagation delays, and timing check information for the design:

`SDFNAME=`*`sdf_file_name`*

Delay Model Timing Data

The following sections describe the types of timing data that are created in the delay model.

I/O Path and Device Delays

These delays model the connection from the input pins to output pins of a cell. The I/O path delays model a delay for a path or paths from a specific input pin to output pin. The I/O path delay can include the explicit transitions that occur at the beginning and end of the paths.

The device delays are more general than I/O paths. They state that any of the paths from any of the cell's inputs to any of the cell's outputs or a specific output pin are covered by this delay value. The device delays contain no notion of the phase of the transitions either at the input or the output.

Wire Delays

These delays describe the wire connections between different entities (`hierBlocks`) on a single product or a hierarchy of products (cell, chip, and card). There are three types of wire delays: interconnect, net, and port. Interconnect delay runs from any source pin on a net to

any sink pin on the same net, at the same or different levels of the package hierarchy. Net and port delays run only between pins in the same net at a single level of the package hierarchy. The exact meaning of a net delay depends upon how it is specified. If a source pin is specified, the delay is from this pin to any sink pin on the net (at this package level). If the net name is specified, the delay is for any source pin to any sink pin on the net (again, at the same package level). A port delay is used from any source to a single sink pin at the same package level. Refer to "Specifying Wire Delays" on page 108 for more information.

Timing Check Delays

These delays describe a required minimum time between two events on one or two input pins of a given cell (often relating to a memory element). Maintaining the minimum time between the two events ensures that the cell(s) will perform correctly. In the functional mode there may be timing check delays between input pins of non-memory elements but they are ignored by the test applications except when checking the Minimum Pulse Width or Skew Constraints.

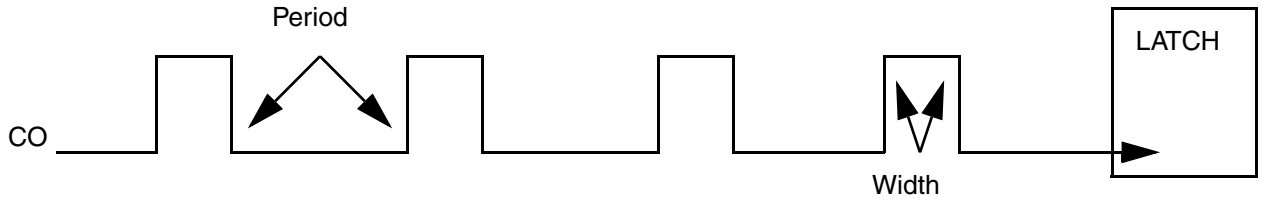
Minimum Path Pulse Width

The minimum size pulse (or glitch) that will allow the memory element inside a given cell to correctly operate.

Period and Width

These are features of a clock's edges when they arrive at a memory element. The width is the minimum size and polarity the pulse must be for the memory element to properly clock in data. The period is the minimum time required between a given edge of a clock pulse (leading to leading or trailing to trailing).

Figure 4-5 Period and Width for Clocking Data into a Memory Element



Setup, Hold, and Recovery Time

The setup and hold times describe the minimum duration of time that must be maintained between two signals that are arriving at different inputs of a cell. One pin must be a clock (C0), and the second is usually the data pin (D0), but may be a clock (C1). The setup time is the duration of time that the data (or C1 clock) must be stable before the specified clock edge arrives at the clock (C0) pin.

The hold time is the amount of time the data pin must be stable after the clock edge arrives at the clock (C0) pin.

Recovery time is treated the same as setup time by Encounter Test.

Figure 4-6 Setup Time

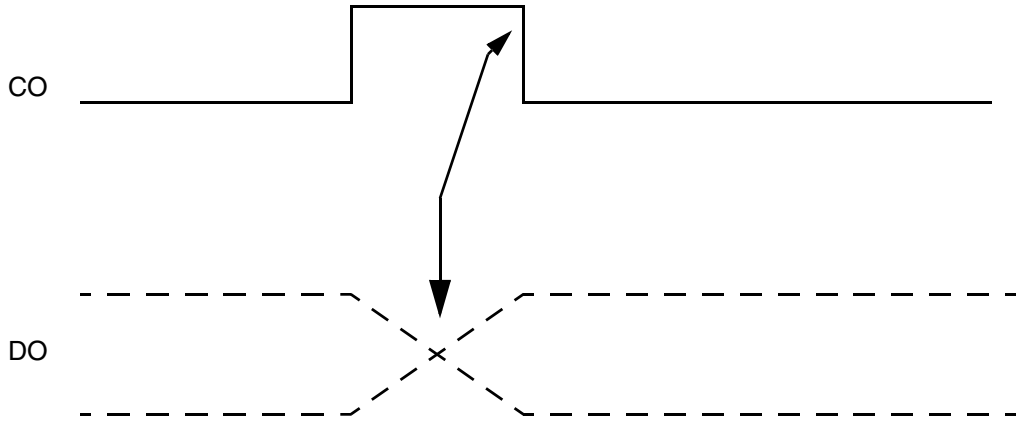
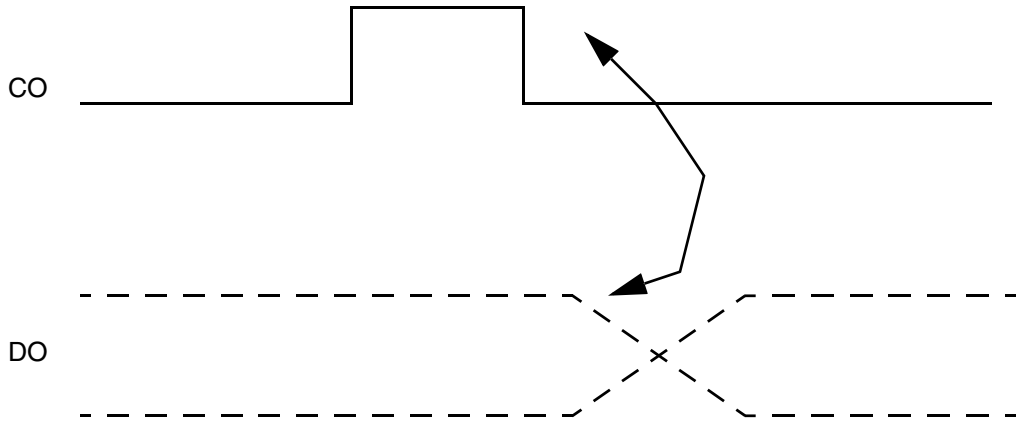


Figure 4-7 Hold Time



No Change

The SDF uses this to define a window surrounding a clock pulse during which some given pin must be stable.

Notes:

1. To perform Build Delay Model using the graphical interface, refer to "Build Delay Model" in the *Encounter Test: Reference: GUI*.
2. To perform Build Delay Model using command lines, refer to "build_delaymodel" in the *Encounter Test: Reference: Commands*.

Also refer to "Performing Build Delay Model (Read SDF)" on page 92.

Delay Path Calculation

The following terms and definitions will aid in the understanding of path delay calculation:

- early mode - The least amount of time required to evaluate logic. It is a user-specified combination chosen of the best, nominal and worst case delays found in the SDF.
- late mode - The most amount of time required to evaluate logic. It is a user-specified combination chosen of the best, nominal and worst case delays found in the SDF.
- combination - Also called a linear combination, three values or weights are used to skew the importance of the best, nominal and worst case delay numbers from the SDF. For example, `earlymode=(a,b,c)` is used as follows:

  ```
  earlymode delay = ((a x best case) + (b x nominal) + (c x worst))
  ```

  This allows targeting the early mode or late mode arrival time to a particular portion of the process curve. For example (0,1,0) uses the nominal values, while (0,.5,.5) will use the average of the nominal and worst delays.
- Arrival Time - The sum of the delays from a source to sink along the fastest or slowest path.

The following is brief description of the timing algorithm used in Encounter Test. It describes a clock to clock sequence but the same basic idea is used with pi to clock and clock to po sequences.

1. A test sequence consisting of launch and capture events is presented to the timer.
2. The pulse width requirements are calculated. Pulse timing checks are present at the cell level. These requirements are backed up to the primary inputs to account for any pulse width shrinkage which may have occurred.

3. The arrival times for all the latch data and clock inputs are calculated for the best, early, late and worst cases. The user has control of early and late. Best and worst are the very fastest and slowest (respectively) and are used only in special cases that are not discussed here.

   The arrival times take into account tie values, lineholds on primary inputs and in certain cases, lineholds on latches and the values on test function primary inputs. The values are used to disable paths that are not to be timed.

4. Linear dependencies (Setup, Minimum Pulse, and Hold time tests) are calculated. See Figure 4-8.

   The times are distance in time between transitions, (i.e. the leading edge of the release clock to the trailing edge of the capture clock). Also since the launch clock launches transitions that will be captured at many latches, there are many dependencies. Only the largest one that describes the relationships between edges is kept.

   When running with `maxpathlength` or a `clockconstraints file` (constrained timings), the dependencies that do not meet the timing constraints are dropped and the involved capture latches are ignored. When running the Automatic True-Time flow, the maximum path length is automatically determined for each clocking sequence. In the At-Speed and Faster Than At-Speed flows, a clock constraints file is user-specified.

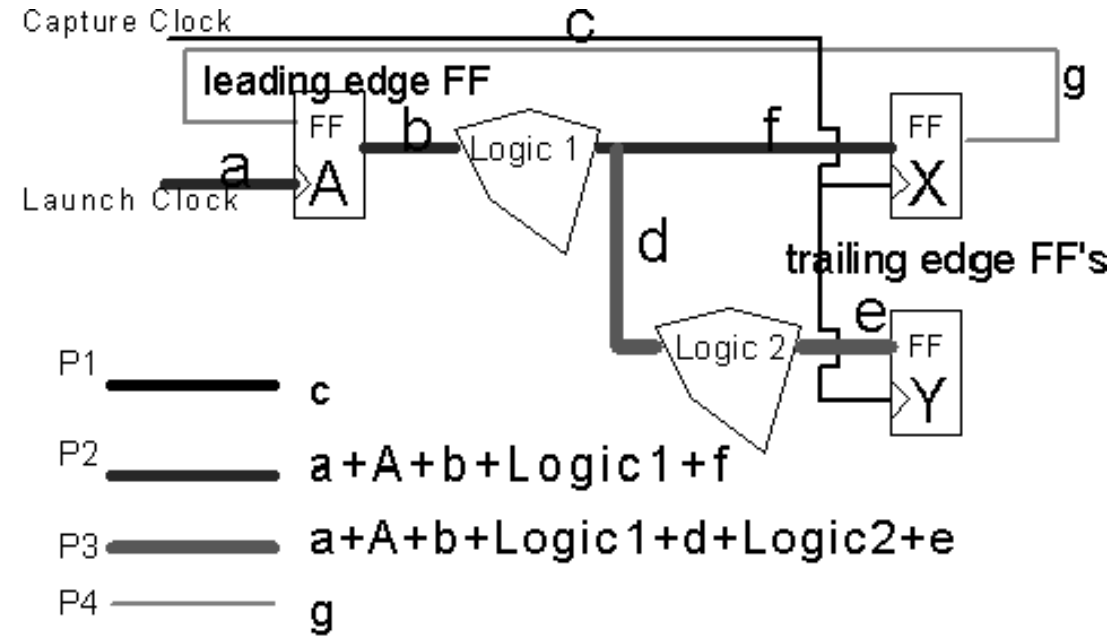
   In Figure 4-8, the sink “P3” would be a candidate to be ignored to test paths that are smaller that 2 Logics. The figure also shows setup and hold test that traverse logical paths between FFs, however these are not the only types of relationships; they also occur between clocks that feed the same cell or at clock gates. For example, a clock-to-clock relationship is used in a sequence with a scan clock release and a system capture clock capture. There is a relationship from the trailing edge of the release clock to the leading edge of the capture clock.

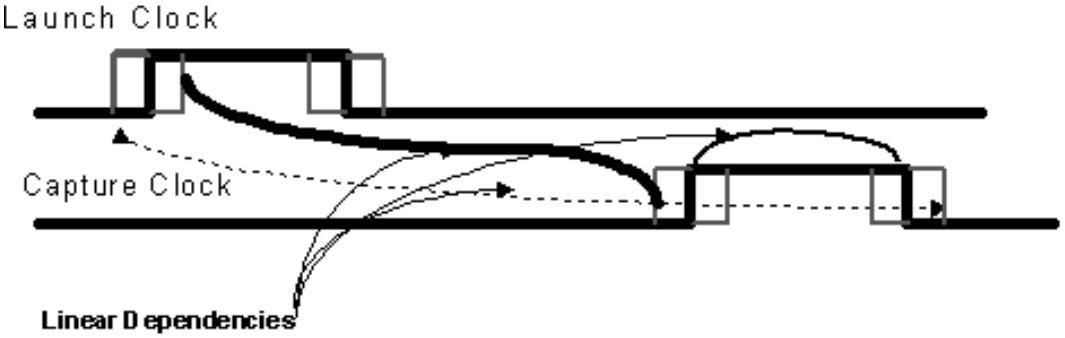
5. The set of linear dependencies are optimized into the timings (timeplates). In this step, the setup, hold and pulse dependencies are combined into one set of timings to be applied at the primary inputs. Though the maximum path lengths that are greater than the specified `maxpathlength` have been discarded, the combined set of edges still may be longer than the `maxpathlength`. For example, if the following dependencies are used in the figure, the total delay will be greater than the largest of the parts:

   Setup Time = 4 ns
   hold time = 2 ns
   pulse width = 2.5 ns.

   The total delay is 7 ns not 4 ns. The reason for this is (hold time + (2 x pulsewidth) = 7 ns.

Figure 4-8 Scenario for Calculating Path Delay





Setup Time Test

------ = maximum of (late mode P2 minus early mode P1) or
(late mode P3 minus early mode P1) +
2 times the tester accuracy.

Minimum Pulse Test

—— = maximum of TDR or pulse width calculated back from the FFs. So if TDR is 5, FFx is 4 and FFy=10 the pulse will be 10 + 2 times the tester accuracy. Calculations in early and late mode.)

Hold Time Test

▬▬ = (late mode P1 + late mode X + late mode P4 - early mode a) + 2 times the tester accuracy.

Note: in some cases the ▬▬ relationship may be setup or nochecks

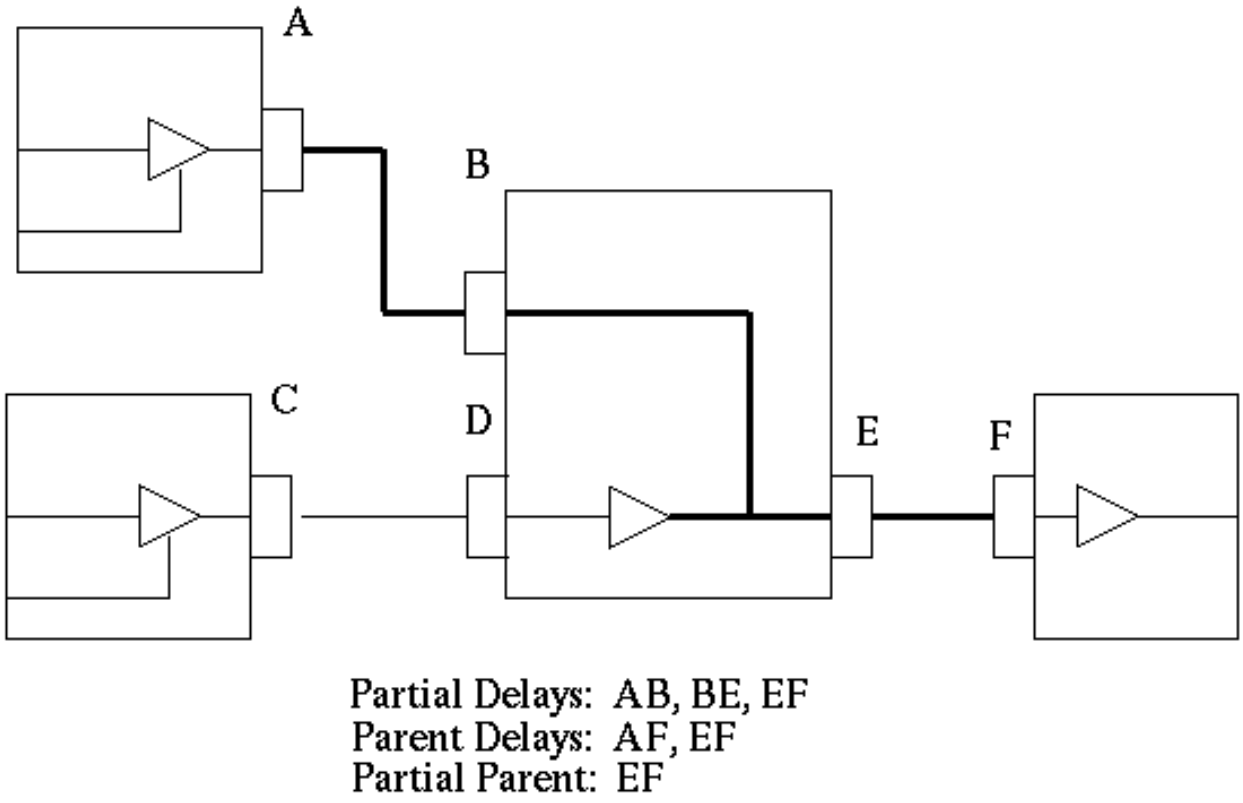
Specifying Wire Delays

Encounter Test allows you to specify delays across wires which pass directly through a cell to generate wire path delay data.

The following terms and corresponding definitions are associated with wire delays:

Wire Cell	A cell containing a wire that leads directly from an input to an output with no intervening primitives.
Partial Delay	A delay passing through the wire of a wire cell that could be incorporated into a larger delay passing through the same wire.
Parent Delay	Any interconnect delay leading from and going to a primitive block. Some parent delays can be broken down into two or more partial delays.
Partial Parent	A delay that exhibits properties of a partial delay and a parent delay. Figure 4-9 displays examples of the preceding terms.

Figure 4-9 Wire Delay Scenarios



Note that PORT and NET delays do not include the length of wire within the cell; they only include the interconnecting wires between the cells.

Delays through wire cells can either be specified as `INTERCONNECT` delays which span several cells, or each segment can be specified in `IOPATH` and `INTERCONNECT` delays. If delays are specified by parts, then all of the partial delays that comprise the parent delay must be specified or the delay information will be incomplete.

For example, in Figure 4-9, the delays can be specified as one `INTERCONNECT` from `A` to `F`, or they can be specified as `INTERCONNECT` delays from `A` to `B` and `E` to `F`, as well as an `IOPATH` delay from `B` to `E`.

Performing Read SDC

The `read_sdc` command reads and verifies design constraints from a Design Constraint File (SDC). For more information about the data retrieved and used from the SDF, refer to "Design Constraints File" on page 128.

Note: Any existing SDC in the current testmode from a previous run should be removed before reading the current SDC. Refer to "remove_sdc" in the *Encounter Test: Reference: Commands* for more information on this.

While reading and parsing an SDC file, Encounter Test creates design constraints. For more information on Encounter Test handling and processing of these constraints, refer to "Dynamic Constraints" on page 153.

To perform Read SDC using the graphical interface, refer to "Read SDC" in the *Encounter Test: Reference: GUI.*

To perform Read SDC using command line, refer to "read_sdc" in the *Encounter Test: Reference: Commands.*

The syntax for the `read_sdc` command is given below:

```
read_sdc workdir=<directory> testmode=<modename> sdcpath=<directory> sdc=<file name>
```

where

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `sdcpath`= directory where the sdc file exists
- `sdc`= name of the sdc file

Prerequisite Tasks

Complete the following tasks before executing Build Delay Model:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.

   Note: If SDC is required on multiple test modes for a design, you should read an SDC into each testmode.

2. Create an SDC file

   Note: Use any tool such as ETS (Encounter Timing System) or Einstimer to create an SDC file for the expected conditions (voltage levels, temperature) to run at the tester.

Output

Encounter Test stores the constraints internal to the testmode. When running ATPG, add `usesdc=yes` at the command line to use the SDC.

Common Debug Using SDC

Refer to "Constraint Checking" on page 154 for information on the test generator and simulators handle design constraints.

Design Constraints File

The design constraints file supplements or replaces the SDF or for delay tests that consider small delay defects, particularly for incorporating faster technologies with specialized timing algorithms such as on-product clocks for at-speed testing. The functional characteristics associated with these technologies, also known as *designer's intent*, are recorded in a design constraints file. True-Time Test accepts a Synopsys® Design Constraints (SDC) file that configures clock and delay constraint parameters.

Table 4-2 lists the SDC statements accepted by Encounter Test.

Table 4-2 SDC Statements and their Functions

SDC Statement	Function
`set_case_analysis`	Identifies points in the design to hold at a known state. The values may be specified for any hierarchical pin. Refer to "Set Case Analysis" on page 112.
`set_disable_timing`	Removes timing arcs and indicates the logic can be *don't cared*, that is, set to `X` for ATPG and simulation. Refer to "Set Disable" on page 113.

SDC Statement	Function
`set_false_path`	Identifies paths that are expected to be inactive in the functional mode; checking is not required. There are two types of false paths: ■ Paths through which transitions (including glitches) cannot propagate ■ Paths for which tests are not to be created. A common use is to remove inter-domain paths. Refer to "Set_False_Path" on page 113.
`set_multicyle_path`	Identifies logic that exceeds one cycle to complete. Refer to "Set_Multicycle_Path" on page 114.
No statement	Describes logical constraints that are expected in the functional process. Refer to "Boolean Constraints" on page 116. **Note:** Configured in the linehold file.

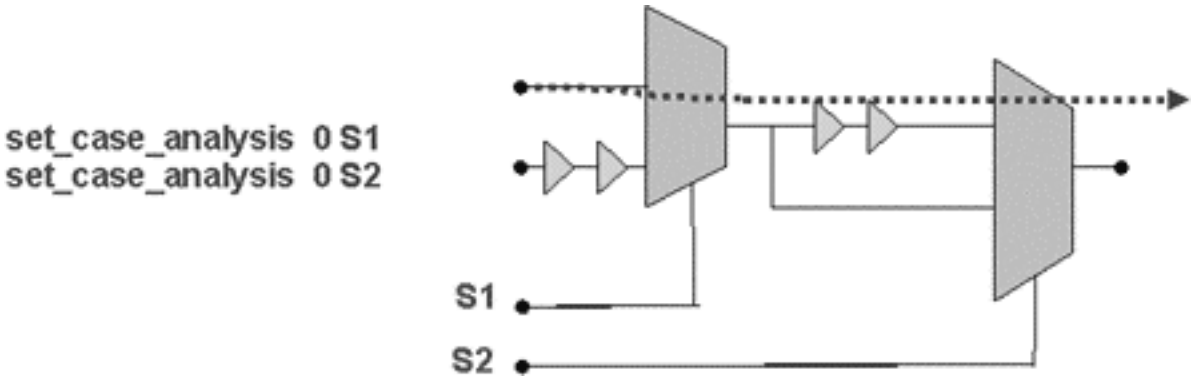
Design Constraint Syntax

The following syntax is accepted in the input design constraints file:

Set_Case_Analysis

The syntax is `set_case_analysis` *`value port_list`*. Figure 4-10 on page 113 depicts a configured example.

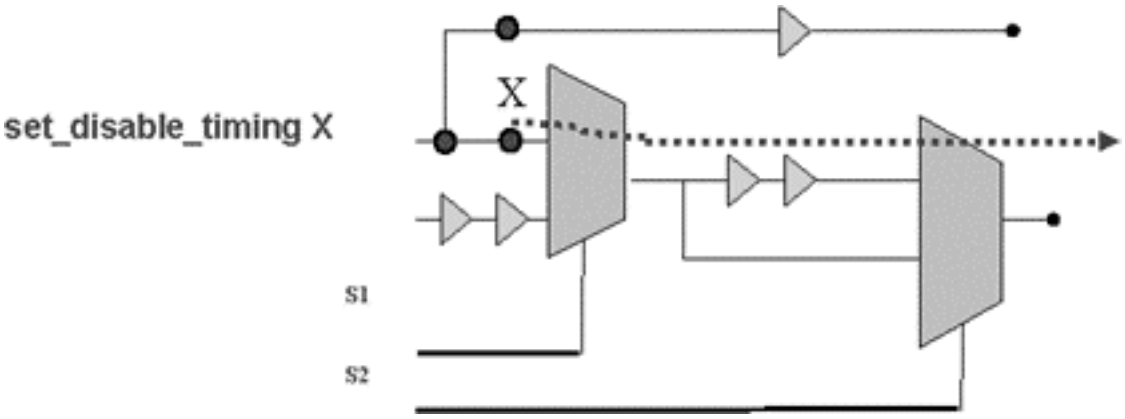
Figure 4-10 set_case_analysis Example



Set_Disable

The syntax is `set_disable_timing` *`port_list`*. Figure 4-11 depicts a configured example.

Figure 4-11 set_disable Example



Set_False_Path

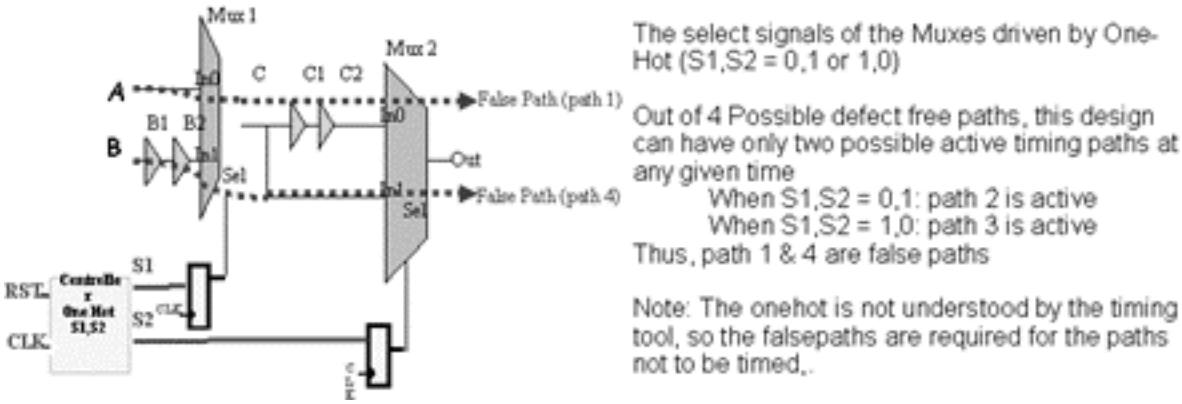
The syntax is `set_false_path -from` *`pin_A`* `-to` *`pin_B`*. The set_false_path statement has the following optional keywords:

- -setup : identifies the constraint as a long path constraint
- -hold : identifies the constraint as a short path constraint

If neither of these keywords is specified, the constraint is considered as both -setup and -hold constraint (as against considered as -setup constraint till previous release).

Figure 4-12 on page 114 depicts a configured example.

Figure 4-12 set_false_path Example

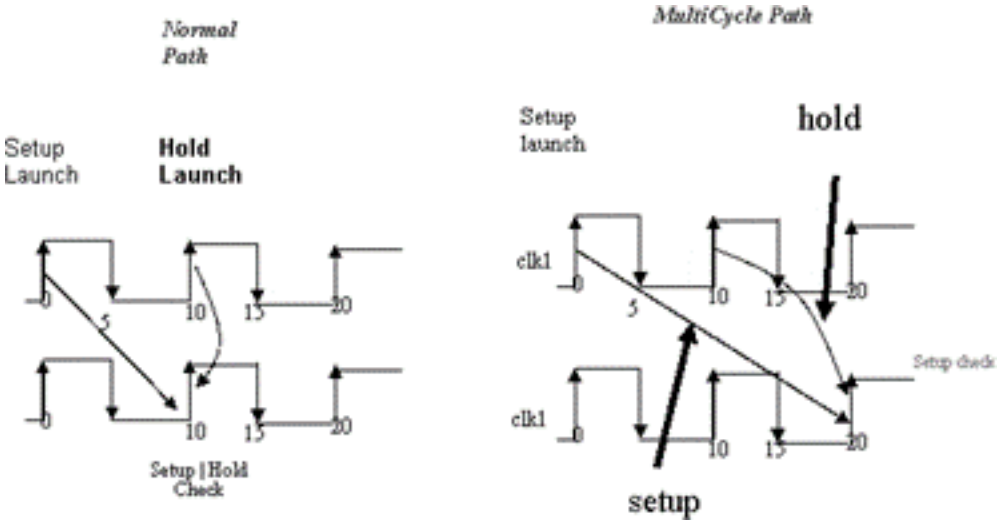


Refer to "False and Multicycle Path Grouping" on page 115 for additional information.

Set_Multicycle_Path

The syntax is `set_multicycle_path` -from *`path`*. Figure 4-13 depicts a configured example of a normal path and a multicycle path.

Figure 4-13 set_multicycle_path Example



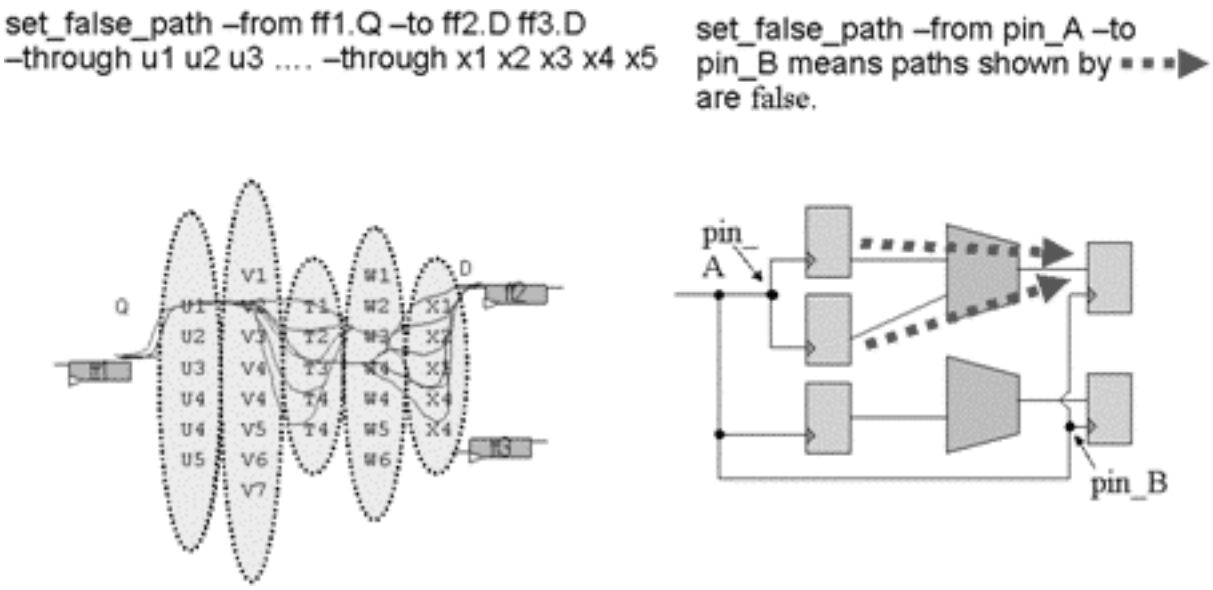
Refer to "False and Multicycle Path Grouping" on page 115 for additional information.

False and Multicycle Path Grouping

Falsepath and multicycle path statements are not required to correspond to individual combinational paths. An individual false or multicycle path statement may represent many paths using one `falsepath` or `multicycle` statement. Specify multiple locations at the `-from`, `-to`, and `-through` points. The paths can traverse as desired as long as the path goes through at least one point from the `-from`, `-to`, and `-through` point list.

Specify either the `-from` or `-to` point on a clock path. The paths start or end at the flip-flop/latches downstream of the clock pin. Figure 4-14 depicts a configured example where the `-from` or `-to` are clock pins. The dashes identify the falsepaths.

Figure 4-14 false and multicycle Path Grouping Example



Boolean Constraints

The Boolean constraints are `equiv`, `onehot`, `onecold`, `zeroonehot`, and `zeroonecold`. The following is the syntax to describe boolean constraints in either the design constraints file or linehold file(s):

`[Boolean]` *`constrainttype`* `[()` *`entitylist`* `())] [-`*`action`*`] [-`*`scope`*`];`

`constraintype` is one of the following:

`equiv` - instructs that all nodes must be the same value

`onehot` - instructs only this entity may be a logic 1; all others must be logic 0

`zeroonehot` - instructs that all entities are logic 0 or just one entity is a logic 1

`onecold` - instructs that only this entity may be a logic 0; all others must be logic 1

`zeroonecold` - instructs that all entities are logic 1 or only one entity is a logic 0

`entitylist` is the list of locations specified as follows:

A comma-delimited list of points in the netlist. The value affects the source of the net that feeds the pin or that is fed by the pin as seen in the following syntax:

[*`polarity] entity`*, *`[polarity] entity`*,...

Each *entity* is one of the following:

- PIN, the hierPin name
- NET, the hierNet_name
- BLOCK, the usage block name
- PPI, the pseudo-primary input name

The *polarity* is one of the following:

- *Not specified* instructs that it is positive logic
- ^ instructs to use the opposite of the expected negative value. For example, a,^b; enforces that a and b are always opposite each other.

-action is one of the following:

-ignore, the default, instructs to ignore all the scan bits fed by the last propagation entity in the propagation list.

-remove instructs to discard the sequence if a constraint is violated.

-scope is one of the following:

-dynamic, the default, instructs to only check or protect the dynamic pattern

-all instructs to check and protect across the entire test.

Design Constraints File Examples

The following is the sample content of an input SDC file:

```
create_clock -period 100 CLK
create_clock -period 100 -name "CLK2" CLK2MUX/Z
set_false_path -from  XPNDAO167P0/D1 -to XPNDAO138P0/Z
#  next is an example of a grouped false path
set_false_path -from  CLK -to CLK2
set_false_path ?fall -from  CLK2 -to CLK
set_false_path -from  CLK2 -through XPNDAO167P0/D1 -to XPNDAO138P0/Z
set_case_analysis 1 CLK2MUX/D1;
set_case_analysis 0 CLK2MUX/D0;
set_disable_timing  XPNDXORS660/SD
set_multicycle_path  1 -from XPNDAO194P1_C/A \
     -to XPNDAO119P0/D1 \
     -through  XPNDAO187P1/Z BOX841/Z
```

The following is the sample content of the output constraints file:

```
# ####################################################################
#  Created by RTL Compiler (RC) v05.10-p001_1 on Fri Jul 08 06:23:10 PM EDT 2005
```

```
# ######################################################################
# current_design PIXAVE4
case CLK2MUX.D0 = 0;
case CLK2MUX.D1 = 1;
# please include the create_clock statements a a comments.
#create_clock -name "CLK" -add -period 100.0 -waveform {0.0 50.0} [get_ports CLK]
#create_clock -name "CLK2" -add -period 100.0 -waveform {0.0 50.0} [get_pins
CLK2MUX/Z]false -from XPNDAO167P0.D1 -to  XPNDAO138P0.Z ;
false -from CLK -to CLK2MUX.Z ;
false ?from -from CLK2MUX.Z -to  CLK ;
false -from CLK2MUX.Z -through XPNDAO167P0.D1] -to XPNDAO138P0.Z ;
multicycle setup  -from XPNDAO194P1_C.A -through  XPNDAO187P1.Z XPNDAO119P0.D1
   BOX841.Z -to D1XPNDAO119P0.D1 ;
dontcare XPNDXORS660/SD;
```

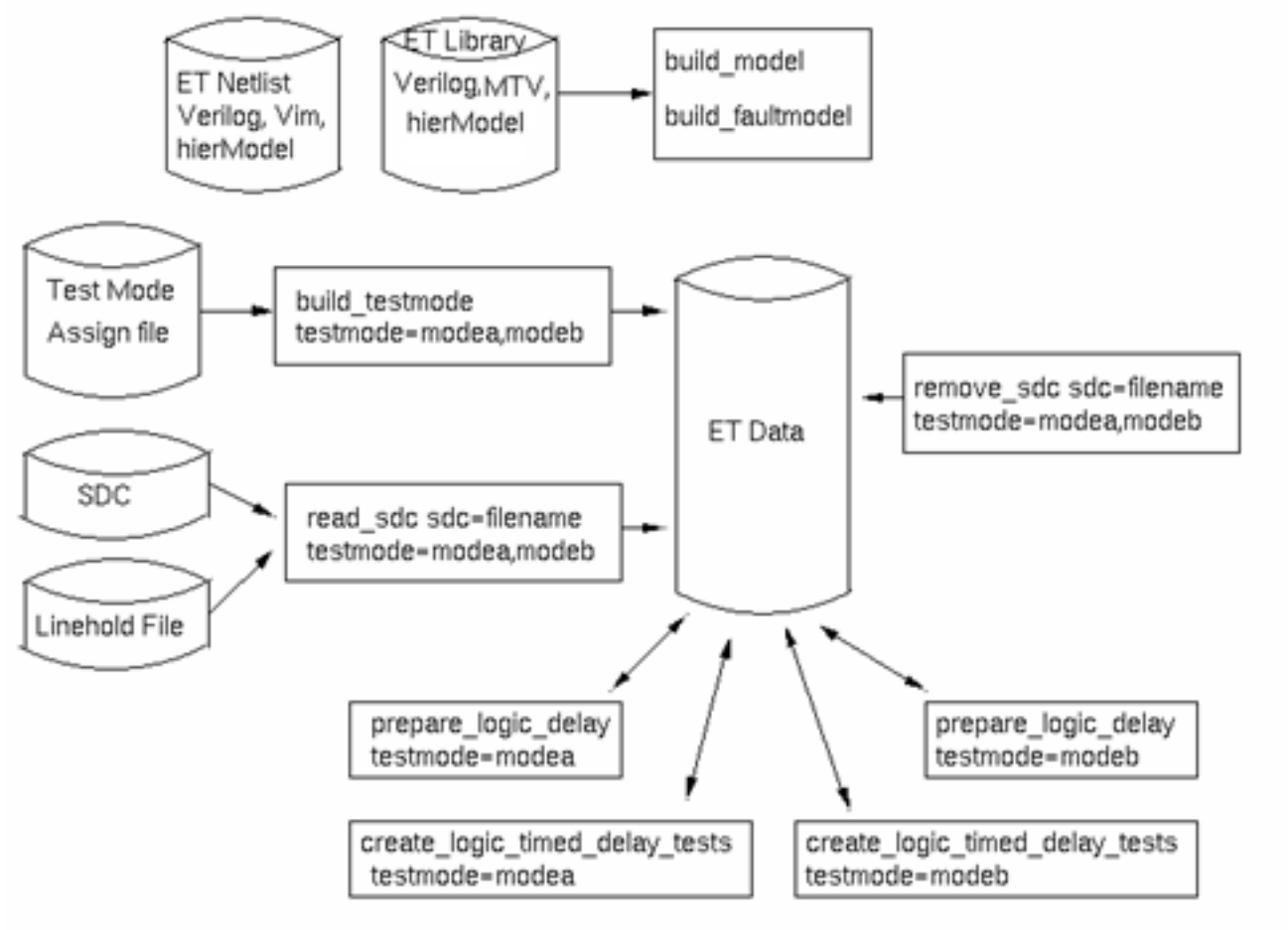
The following is a basic muticycle path example:

```
set_multicycle_path -to c1/b7/D 2;# Don't allow transitions to D pin of c1.b7
set_multicycle_path -hold 1 -to c1/b7/D ; # There is a hold violation on flop c1.b7
```

Using the SDC in True-Time Test

An overview of implementing an SDC constraints file in the True-time flow is depicted in Figure 4-15 on page 119.

Figure 4-15 Use Model with SDC Constraints File



The preceding flow occurs after a model and test mode are created and prior to test generation.

The following is a typical sequence of events for incorporating use of the SDC:

1. Develop/create an SDC that describes the design's intended function.
2. Use the SDC, lineholds, and test mode to customize True-Time Test generation to produce synthesis and timing analysis results.

Important

Ensure the `generate_clocks` statement occurs first in the SDC file or supporting TCL scripts.

A currently used constraints file may be changed using either of the following methods:

- Rebuilding a test mode
- Add or remove using Read SDC or Remove SDC

Note: The preceding techniques must be performed prior to test generation.

Use Read SDC to read and verify the constraints in a design constraints file and a linehold file. Read SDC incrementally updates existing constraints based on the content of the input constraints/linehold file(s).The verified constraints are stored in an output constraints file for subsequent use by test generation. The output files are in the following forms:

- `constraints.`*`testmode`*
- `constraints.`*`testmode.experiment`* to allow simultaneously running multiple experiments with different SDC files

If multiple test modes are specified, the SDC is verified using the first specified test mode. The results are applied to all specified test modes. Refer to the following to perform Read SDC and Remove SDC:

- "Read SDC" in the *Encounter Test: Reference: GUI*
- "read_sdc" in the *Encounter Test: Reference: Commands*
- "Remove SDC" in the *Encounter Test: Reference: GUI*
- "remove_sdc" in the *Encounter Test: Reference: Commands*

Note: An RTL Compiler license is required to run Read SDC.

Performing Remove SDC

The `remove_sdc` command removes the stored SDC data for a testmode or an experiment.

To perform Remove SDC using the graphical interface, refer to "Remove SDC" in the *Encounter Test: Reference: GUI*

To perform Remove SDC using command line, refer to "remove_sdc" in the *Encounter Test: Reference: Commands.*

The syntax for the `remove_sdc` command is given below:

```
Remove_sdc workdir=<directory> testmode=<modename>
```

where:

- `workdir` = name of the working directory

- `testmode`= name of the testmode from which to remove SDC

Note: The most commonly used keyword for the `remove_sdc` command is `Experiment`. Refer to "remove_sdc" in the *Encounter Test: Reference: Commands* for more information on this keyword.

An Overview to Prepare Timed Sequences

Prepare Timed Sequences determines a set of optimal parameters to run test generation and fault simulation for timed tests. These parameters include the following:

- The best clock sequences
- The frequency per sequence
- Maximum path length for each selected sequence

Note: You do not need to perform Prepare Timed Sequences for static ATPG.

The clock sequences are combined into a multi domain sequence that simultaneously tests the combined sequences. Refer to Table 4-3 on page 124 for more information.

The best sequences are determined by generating test patterns for a statistical sample of randomly selected dynamic faults. Each unique test pattern is evaluated to ascertain the number of times the test generator used it to test a fault. The set of patterns used most often are considered the best sequences. Typically, the top four or five patterns will test 80 percent of the chip. Specify the `maxsequences` option to use more sequences, if required.

The prepared sequences are represented by sequences named `DelayTestClockn` in the sequence definition. The following is an example of a sequence summary:

```
Best 5 Sequence Pattern Events Summary
DelayTestClockSeq1   292 out of 816 faults
    Static setup patterns:
      Stim_Latch
      Pulse_Clock -EC TheSysClock
    Dynamic pattern events:
      Pulse_Clock -ES TheReleaseClock
      Pulse_Clock -EC TheCaptureClock
    Static measure event:
Measure_Latch

DelayTestClockSeq2   58 out of 816 faults
    Static setup patterns:
      Stim_Latch_Plus_Random
      Stim_PI_Plus_Random
    Dynamic pattern events:
      Pulse_Clock -EC SystemClock
      Stim_PI
      Pulse_Clock -EC SystemClock
    Static measure event:
```

```
Measure_Latch
...

DelayTestClockSeq5   18 out of 816 faults
    Static setup patterns:
      Stim_Latch
      Stim_PI
    Dynamic pattern events:
      Pulse_Clock -ES TestClock1
      Pulse_Clock -ES TestClock2
    Static measure event:
      Measure_Latch
      Measure_PO
```

The maximum path length is determined by generating a distribution curve of the longest data path delays between a random set of release and capture scan chain latch pairs. The area under the curve is accumulated from the shortest to the longest and the maximum path length is the first valley past the cutoff percentage. This method limits the timings to ignore any outlying paths that over inflate the test timings. Additional options are available to control this by providing the cutoff percentage for the curve and a maximum path length to override the calculation.

The distribution summary is printed into the log, an example is given below:

```
                Random  Path   Length Distribution
Percentage      80      90     95     97     98     99     100
MaxPathLength   5300    6250   6500   7250   7500   7500   12350

Path length after area cutoff of  98 percent  = 7250 pico-seconds
plus two times accuracy of  250 ps            = 7750 pico-seconds
User provided maximum path length             = 7000 pico-seconds
Maximum Path Length to be used for timings    = 7750 pico-seconds
```

Dynamic constraints are automatically determined by generating timings for the set of best sequences. This occurs after the maximum path length has either been provided or calculated. The events for each sequence are applied and the calculated delays are evaluated for violations of the maximum allowed distance. For every detected violation, the sources are traced and a dynamic constraint is generated to specify to the test generator to disallow these sources to switch during the dynamic portion of the test. If no transitions occur in these paths, the cause of the timing violation is removed, thus resulting in faster timings. In addition to the violations, a set of ignored measure latches is generated for analysis. When the number of ignored measures exceeds 10 percent of all of the measures for the chip, the maximum path length for the sequence is increased to allow for better test coverage. A summary of this activity is printed in the output log, as shown in the following example.

```
                      Best 1 Sequence Timing Violations Summary

Test               #Ignore     Final Max  Final Max    Path Length
Sequence           #Violations Measures   Pulse Width
```

```
DelayTestClockSeq1 2197        5490       DEFAULT      (ppi_rts->ppi_rts 2000ps )
```

Performing Prepare Timed Sequences

This task determines a set of optimal parameters to run test generation and fault simulation for timed tests.

These parameters include the following:

- Best clock sequences
- frequency per sequence
- Maximum path length for each selected sequence

Encounter Test might create design constraints while running `prepare_timed_sequences` with a delay model. Refer to "Dynamic Constraints" on page 153 for more information on how Encounter Test handles and processes these constraints.

To perform pre-analysis using command line, refer to "prepare_timed_sequences" in the *Encounter Test: Reference: Commands*.

To perform pre-analysis using the graphical user interface, refer to "Prepare Timed Sequences" in the *Encounter Test: Reference: GUI*.

The syntax for the `prepare_timed_sequences` command is given below:

```
prepare_timed_sequences workdir=<directory> testmode=<modename>
delaymodel=<delaymodel name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `delaymodel=<name>` - Name of the delay model for the timed ATPG tests.

Note: The commonly used keywords for the `prepare_timed_sequences` command are:

- `clockconstraints =<file name>` - List of clocks and frequencies to perform ATPG. For more information, refer to "Clock Constraints File" on page 129.
- `dynseqfilter= <value>` - Type of clocking, for example launch of capture with same clocking or launch off shift. For more information on sequence types, refer to "Delay Test Clock Sequences" on page 149.

- `maxbestsequences=<integer 1 to 99>` - The default is to allow 99 different clocking sequences to be generated, if allowed in the design.
- `addseqs =syscapture|domain|domainorder` - Ensure that certain clocking sequences will be included in the generated sequences used for ATPG.
- `earlymode/latemode -` The ability to customize delay timings. The default is `0.0,1.0,0.0` for each option. For more information, refer to "Process Variation" on page 133.

Refer to "prepare_timed_sequences" in the *Encounter Test: Reference: Commands* for more information on these keywords.

The following table discusses the various methods of getting timings in the patterns:

Table 4-3 Methods to Generate Timed Patterns

Method	Encounter test will...
Specifying a clock constraint file with no delay model and no SDC	Use the clock timings found in the clock constraint file. All paths are treated as valid and can be achieved in the specified timings.
Specifying a clock constraint file with no delay model but with SDC	Use the clock timings found in the clock constraint file. All paths are treated as valid except for those found in the SDC.
Specifying a clock constraint file with delay model with SDC	Use the clock timings found in the clock constraint file. Encounter Test will also use the information in the delay model to identify paths that do not make this timing and then X them out. The SDC will be used to identify additional timing exemptions.
Specifying a delay model with no clock constraint file and SDC	Automatically identify the clocking frequency at which to time the design by finding the longest paths without any constraints. Encounter Test will create a path distribution curve data and clock timings and pick a clock timing of about 95% of the distribution. It will then time and ignore the longest paths based on the determined frequency.
Specifying neither a delay model, SDC, nor a clocking constraint file	Not generate timings in output patterns. Encounter Test will consider all the paths as valid and create only untimed transition tests.

Prerequisite Tasks

Complete the following tasks before executing Prepare Timed Sequences:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.
2. Build Encounter Test Testmode
3. Build Fault model with dynamic faults

Output

Encounter Test stores the test sequences internal to the testmode. The sequences can be examined by reporting the test sequence. Refer to report_sequences in the *Encounter Test: Reference: Commands* for more information.

Check the log file to see if a large number of constraints were added to the data base. A large number of constraints may prevent the test generator from ramping up the coverage, especially in signature-based testing.

Command Output

The following is an example of the three sequences identified by `prepare_timed_sequences` and used for ATPG:

```
Best 3 Sequence Pattern Events Summary
  DelayTestClockSeq1   21 out of 35 faults
    Static setup patterns:
      Stim_Latch
      Stim_PI_Plus_Random
    Dynamic pattern events:
      Pulse_Clock -ES CLK1
      Pulse_Clock -ES CLK1
    Static measure event:
      Measure_Latch
  DelayTestClockSeq2   11 out of 35 faults
    Static setup patterns:
      Stim_Latch
      Stim_PI_Plus_Random
    Dynamic pattern events:
      Pulse_Clock -ES CLK3
      Pulse_Clock -ES CLK3
    Static measure event:
      Measure_Latch
  DelayTestClockSeq3   3 out of 35 faults
    Static setup patterns:
```

```
  Stim_Latch
  Stim_PI_Plus_Random
Dynamic pattern events:
  Pulse_Clock -ES CLK2
  Pulse_Clock -ES CLK2
Static measure event:
  Measure_Latch
```

Create Logic Delay Tests

Refer to the following for more information on delay test concepts and faults:

- "Delay Timing Concepts" on page 128
- "Delay Defects" on page 144
- "True-Time Test: An Overview" on page 35

Encounter Test might create and use design constraints while running `create_logic_delay_tests` with a delay model or SDC. Refer "Dynamic Constraints" on page 153 to for more information.

To perform create logic delay tests using the graphical interface, refer to "Create Logic Delay Tests" in the *Encounter Test: Reference: GUI*

To perform create logic delay tests using command line, refer to "create_logic_delay_tests" in the *Encounter Test: Reference: Commands*.

The syntax for the `create_logic_delay_tests` command is given below:

```
creat_logic_delay_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that to be generated

The most commonly used keywords for the `create_logic_delay_tests` command are:

- `clockconstraints=<file name>` - List of clocks and frequencies to perform ATPG. For more information, refer to "Clock Constraints File" on page 129.
- `dynseqfilter=<value>` - Type of clocking, for example, launch of capture with same clocking or launch off shift. For more information on sequence types, refer to "Delay Test Clock Sequences" on page 149.

- `maxbestsequences=<integer 1 to 99>` - Default is to allow 99 different clocking sequences to be generated, if allowed in the design.

The following are the timed options:

- `delaymodel=<name>` - Name of the delay model for timed ATPG tests.
- `earlymode/latemode -` Ability to customize delay timings. Default is `0.0,1.0,0.0` for each option. For more information, refer to "Process Variation" on page 133.

Refer to "create_logic_delay_tests" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Note: To more information on generating test patterns with system timings, refer to "Performing Prepare Timed Sequences" on page 123.

Prerequisite Tasks

Complete the following tasks before executing Create Logic Delay Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.
2. Build Encounter Test Testmode
3. Build Fault model with dynamic faults

Tip

For timed tests, it is recommended to run `prepare_timed_sequences` to precondition the clocking sequences and timing constraints.

Output

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information on testmode, global coverage and the number of patterns used to generate the results.

```
                  Experiment Statistics: FULLSCAN.prep2
                        #Faults   #Tested   #Redund  #Untested  %TCov   %ATCov
Total Static             908       229       0        640       25.22    25.22
Total Dynamic            814       111       0        699       13.64    13.64
```

```
                  Experiment Global Statistics: FULLSCAN.prep2
                        #Faults   #Tested   #Redund  #Untested  %TCov   %ATCov
Total Static             1022      229       0        754       22.41    22.41
Total Dynamic            1014      111       0        899       10.95    10.95
          ------------------------------------------------------------

              Total # effective tests generated:             11

INFO (TDA-001): System Resource Statistics.  Maximum Storage used during the run
and Cumulative Time in hours:minutes:seconds:
```

Delay Timing Concepts

A dynamic test may be either timed or not timed. When it is timed, Encounter Test defines the timing in a sequence definition which is a sort of pattern template and accompanies the test data. When the dynamic tests are not timed, the tests are structured exactly the same as for timed tests, but there is no timing template. In this case, the timing, if used at all, is supplied by some other means, such as applying the clock pulses at some constant rate according to the product specification.

The following section identifies the settings to control the timings for a test generation run. One situation where these may be useful is in testing different clock domains.

Path Length

Maximum and minimum timings represent the time between the clock edge that triggers the release event and the clock edge that captures the result. With reference to Figure 4-21 on page 148, note that this is not exactly equal to the path length, because the paths from the Clock A input to LatchA and LatchB may not be equal. Regardless, it is often convenient to think in terms of the path lengths. Paths longer than the maximum path length (after adjusting for the clock skew as previously noted) cannot be observed by the test since they would be expected to fail. Any latch or primary output that would have been the capture point for these long paths is given a value of X (no measure).

Paths shorter than the minimum path length will be observed, but some small delay defects could go undetected in a short path. Note that any dynamic test that involves paths of unequal lengths must be timed to the longest path, so this *short path* concern exists regardless of whether a minimum path length is specified.

Design Constraints File

This file supplements or replaces the SDF or for delay tests that consider small delay defects, specifically for incorporating faster technologies with specialized timing algorithms such as

on-product clocks for at-speed testing. Refer to "Design Constraint Syntax" on page 112 for more information on syntax.

Clock Constraints File

The clock constraints file guides the creation of clock sequences used for fixed-time, at-speed or faster than at-speed test generation. The delay test generator builds tester-specific test sequences using the clock constraints file information and the tester description rule (TDR). Multiple domains and inter-domain sequences can be defined in a single clock constraints file. Certain timing-related TDR parameters are also overridden using a clock constraints file. Statements supported are:

- Clock domain statements
- Return to stability statements
- TDR overrides statements

Command Line Syntax

```
create_logic_delay_tests...clockconstraints=<filename>
```

Clock Constraints File Syntax

The domains to be tested are specified by the clocks that control them. Several syntax variations are supported. Each clock domain statement is used to generate a specific clock sequence during ATPG. Multiple clock domain statements are not combined into a single sequence.

```
clockname {edge, width timeUnit} {speed timeUnit};
```

or

```
clockname1 {edge, width timeUnit} clockname2{edge, width timeUnit} {speed
timeUnit};
```

or

```
clockname {speed timeUnit};
```

or

```
clockname1 clockname2 {speed timeUnit};
```

where:

- `clockname` - Name of a clock pin. This can be a primary input pin or a pseudo primary input (PPI).

Note: Different clock PIs can be used within the same statement in the clock constraints file.

- If there is only one pin, it is used as both the launch and capture clocks.
- If there are two pins, the first pin is the launch clock and the second pin is the capture clock

- `edge` - Either `posedge` or `negedge`, referring to which edge of the clock pin is the active edge (opposite of stability state) for the domain.
- `width timeUnit` - The pulse width. Specified as an integer. Specify `timeUnit` in `ns`, `ps`, or `MHz` (for frequency).
- `speed timeUnit` - The time between the leading edges of the launch and capture clocks. As with width, the speed can be defined in `ns`, `ps`, or `MHz`.

Note:

- The clockname must have a space after it.
- Speed `timeUnit` is optional but the surrounding braces {} are required.

Examples

Example 1:

Tests intra-clock domain CLKA to CLKA

```
CLKA {posedge , 5 ns} CLKA {posedge , 5 ns} {50 Mhz} ;
```

Example 2:

```
CLKA {posedge , 5 ns} { 10 ns } ;
CLKB {posedge , 10 ns} { 20 ns } ;
CLKA {posedge , 5 ns} CLKB {posedge , 10 ns} {10 ns} ;
```

Tests intra-clock domain CLKA to CLKA, intra-clock domain CLKB to CLKB, and inter-domain CLKA to CLKB.

Return to Stability Statement

The return to stability (RTS) statement specifies when the trailing edge of a clock is to occur relative to the END of the tester cycle. Specify the RTS time for clock domains using one of the following formats:

```
RTS clockname {edge , speed timeUnit} ;
```

or

```
RTS ALLCLKS {edge , speed timeUnit} ;
```

where:

- `clockname` - The name of a clock pin. This can be a primary input pin or a pseudo primary input (PPI).
- `edge` - Either posedge or negedge, referring to which edge of the clock pin is the active edge (opposite of stability state) for the domain.
- `speed timeUnit` - The pulse width. Specify speed as an integer. Specify `timeUnit` in `ns`, `ps`, or `MHz` (for frequency).

Note:

- The clockname must have a space after it.
- The Return to Stability statement applies to all Clock Domain Statements that reference the same clock PI name.

Example:

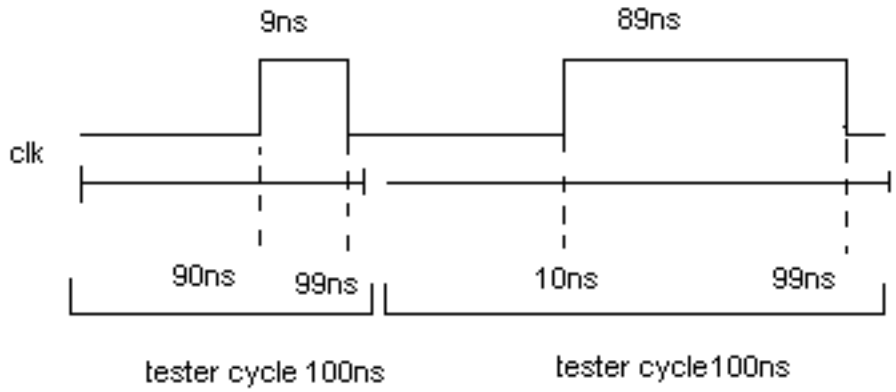
The following example shows how the clock domain statement is modified by a return to stability statement:

```
// CLKA clock domain is 50 MHz with 10 ns clock
CLKA   {posedge , 10 ns} CLKA  {posedge , 10 ns} {50 Mhz} ;

// Define the time consumed before the end of the tester cycle
// that CLKA returns to stability
rts CLKA {1 ns} ;
```

Resultant Clock Sequence

This example defines the timing for the CLKA launch and capture. If the TDR specifies that only one clock pulse per tester cycle is to be issued, the resulting sequence will place CLKA in two consecutive tester cycles:



The leading edge of the launch clock will be at time 90ns and the leading edge of the capture clock will be at time 10 in the next cycle. Both should have a pulse width of 10 ns.

However the RTS entry modifies this timing. It specifies that the trailing edge of both the launch and capture CLKA pulses should be at 99ns (1ns from the end of the cycle). To accommodate this, the launch pulse width is changed to 9 ns and the capture pulse width is changed to 89 ns.

Statements to Override TDR Parameters

In addition to clock domain specifications, certain TDR parameters can be overridden. The following statements may be used in the clockconstraints file:

- resolution {speed timeUnit} ;
- accuracy {speed timeUnit} ;
- period {speed timeUnit} ;

Resolution identifies the smallest increment of time on the tester. Accuracy is added to the time between release and capture timings. Tester period identifies the time for one tester cycle.

Example:

```
// tester resolution...smallest increment of time on the tester
```

```
resolution {10 ps} ;

// accuracy of the tester it will be added to the clk to clk time
accuracy {100 ps} ;

//  tester period also called the tester cycle
period {100 ns} ;

// CLKA clock domain is 50 MHz with 10 ns clock
CLKA   {posedge , 10 ns} CLKA  {posedge , 10 ns} {50 Mhz} ;
```

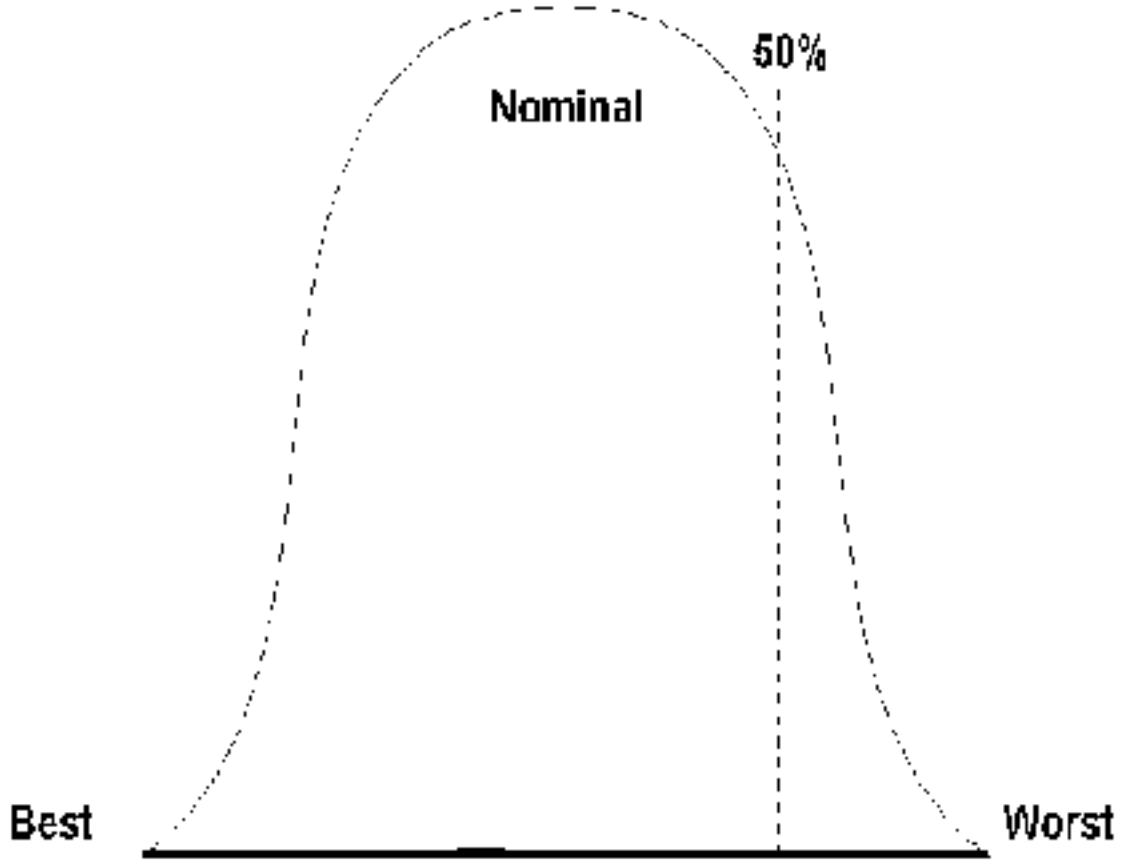
Process Variation

Control the timing calculation by selecting a point on the process curve. This *curve* is a mathematical representation of the delay value, which if measured on a sample of parts, would vary about some mean value; that is, the process curve is the statistical distribution of the average delay values of the chip. Figure 4-16 shows an example process curve for a given delay. Set the coefficients relating to best case, nominal case, and worst case delays. The delays are calculated as

```
delay=(best_case_coefficient * best_case_delay) +
    (nominal_case_coefficient * nominal_case_delay) +
    (worst_case_coefficient * worst_case_delay)
```

This formula allows the selection of either the best case, worst case, or nominal delays by setting one of the coefficients to 1 and the others to 0. However, Encounter Test accepts any decimal number for each of the coefficients to scale the delays (with a coefficient other than 1) or use averages (with two or more non-zero coefficients).

Figure 4-16 Process Variation



Create tests to sort the product based on its speed are created by making several different runs and selecting a different point on the process curve each time. To ensure that all received product is faster than the halfway point between nominal and worst case (see Figure 4-16), specify a process variation of (0,.5,.5). Not all manufacturers will do such sorting, therefore we recommend verifying whether an override is acceptable.

Pruning Paths from the Product

Use lineholds (`LH` attribute or linehold file) to hold a particular point to a static value. The effect of the linehold is to make a path unobservable which will cause it to be ignored for timing.

Verifying Clocking Constraints

The `verify_clock_constraints` command verifies the data in the clock constraint file against the Tester Descriptor Rule (TDR) syntax.

The syntax for the `verify_clock_constraints` command is given below:

```
verify_clock_constraints workdir=<directory> testmode=<modename>
clockconstraints=<filename>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `clockconstraints=<file name>` - List of clocks and frequencies to perform ATPG. Refer to "Clock Constraints File" on page 129 for more information.

Prerequisite Tasks

Complete the following tasks before executing Verify Clocking Constraints:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.
2. Build Encounter Test Testmode. Refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.

Output Files

None

Command Output

The output log contains information if the sequences match the data stored in the testmode.

A sample output is given below:

```
INFO (TTU-401): Verify Clock Constraints begins.   [end TTU_401]
INFO (TTU-130): Reading ClockConstraint file clock.domain.txt.   [end TTU_130]
INFO (TTU-402): Verify Clock Constraints is complete.   [end TTU_402]

Error in Syntax:

INFO (TTU-130): Reading ClockConstraint file clock.domain.txt.bad.   [end TTU_130]
ERROR  (TTU-418): For the statement specified on line 2 within the clock constraints
```

```
file:  clock.domain.txt.bad , the clock name CLKa2 was not found in the model.
Correct the clock constraint file value(s) and re-run. [end TTU_418]
ERROR (TTU-414): Clock_constraint file parse failed, due to syntax error at line
3. [end TTU_414]
INFO (TTU-402): Verify Clock Constraints is complete.   [end TTU_402]
```

Verifying Clock Constraints Information

Before running delay applications, it is highly recommended to check the clock constraint file against the TDR information to verify that the clock constraints are within the TDR guidelines. This reduces resource consumption and the subsequent effort to rectify the identified discrepancies at the end of long-running jobs.

The `create_logic_delay_tests`, `create_path_delay_tests`, `prepare_timed_sequences`, `create_wrp_tests`, and `time_vectors` commands automatically invoke the function to verify clock constraints whenever the application calls the clock constraint parser.

Another method to verify the clock constraints information with the TDR information is to use the `verify_clock_constraints` command. All the aforementioned commands and the `verify_clock_constraints` command check if the following criteria are met:

- The minimum pulse width value obtained from the clock constraint file should be equal to or greater than the specified value for the MIN_PULSE_WIDTH parameter in the TDR
- The maximum pulse width value obtained from the clock constraint file should be less than or equal to the specified value for the MAX_PULSE_WIDTH parameter in the TDR
- The minimum leading to leading edge width obtained from the clock constraint file should be equal to or greater than the specified value for the MIN_TIME_LEADING_TO_LEADING parameter in the TDR
- The minimum trailing to trailing edge width obtained from the clock constraint file should be equal to or greater than the specified value for the MIN_TIME_TRAILING_TO_TRAILING parameter in the TDR
- The specified pulse width and clock speed values for a clock in the clock constraints file should be unique (semantic check)

The above-mentioned clock constraint checks are invoked in the initial stages of the test generation and the applications quickly notify if any criterion is not met.

Note: For any of the above-mentioned checks, if the data to compare is not available from the TDR, the command does not perform the corresponding check.

Refer to verify_clock_constraints in the *Encounter Test: Reference: Commands* for more information.

Characterization Test

Characterization test is a type of path test used to primarily speed-sort a product. That is, using certain paths, the maximum speed at which the design can run is determined.

The goal of characterization test is to provide a starting point for schmoo-ing at the tester. Characterization test is based upon a path test, however many of the timing attributes of the Manufacturing Delay test apply. Many manufacturing sites have tools that facilitate changing the timings of the patterns therefore Encounter Test produces a basic *path test* that can be manipulated on the ATE. The default process for generating path tests when delays are available (a delay model has been built) is to generate tests for the specified longest (critical) paths and to individualize the patterns that test each path. Each test pattern can have its own unique timeplate that can be manipulated on the ATE independent of the timeplates used for other patterns. Encounter Test does not limit path tests (logic values and timings) to just testing and timings individual paths; multiple test paths can be simultaneously tested. When the paths are simultaneously tested, the speed of the tests is affected by the entire product. Constrained timings should be used so the timings do not take into account the paths longer than the specified paths.

An important consideration when performing path test is the pattern volume. In path test, the trade-off is between processing time, pattern count, test coverage and the number of paths that can be tested. A minimal pattern set is difficult with path tests. Compressing patterns is often not possible when focusing on a particular path and ignoring other capture registers in the design. This can cause data volume/tester time problems which must be balanced with the number of paths to test. As the number of paths increase, they should be grouped into paths with similar timings, then applied with the longest timing for that group. This means that some paths will have their timings relaxed but fewer timing changes will have to be made on the manufacturing test equipment.

Paths for characterization testing can be partially or fully specified. Encounter Test selects paths using delay information and or the gate level information. If delay information exists (a delay model was specified), the paths are determined by specifying the partial path and filling out the rest of the paths, enumerated from longest, shortest or random order. Also, if a path is given entirely at the cell level, it is considered only partially specified because the paths are stored at the cell level and there may be multiple paths within a cell. By default, Encounter Test finds a certain number of the longest paths upon which to attempt test generation based on a user-specified path. This is rarely an exhaustive set of paths due to the number of cells (and paths through those cells) in a specified path.

The specified path may not be the sole determining factor for the timing of the test. The logic in the back-cone of the capture register also affects the test's timing. In Figure 4-2, this is shown in the smaller dotted region. For a robust test, other logic which affects the path (feeds into the path) must be considered. If logic in the back-cone operates slower than the path, then that is the timing that must be used. This ensures that late values can only be due to the

cone of logic the target path exists within. To limit the effect of this, the timing is done using the actual pattern so only the logic experiencing transitions is timed.

The tests of paths can be specifically stored patterns generated for paths or previously generated patterns from another ATPG run such as dynamic stored pattern, WRPT or LBIST patterns that have been simulated against the defined set of faults. For example, LBIST patterns can be resimulated to investigate whether the longest paths are tested.

Performing Create Path Tests

This task generates dynamic ATPG patterns based on either a path list or starting/ending points.

Refer to "Path Tests" on page 142 for information on different types of path patterns.

To perform create path delay tests using the graphical interface, refer to Create Path Delay Tests in the *Encounter Test: Reference: GUI*.

To perform create path delay tests using the command line, refer to "create_path_delay_tests" in the *Encounter Test: Reference: Commands*

The syntax for the `create_path_delay_tests` command is given below:

```
create_path_delay_tests workdir=<directory> testmode=<modename> experiment=<name>
pathfile=<file>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated
- `pathfile`= name of paths to be generated. Refer to "Path File" on page 140 for more information.

The most commonly used general keywords for the `create_path_delay_tests` command are:

- `clockconstraints=<file name>` - List of clocks and frequencies to perform ATPG. For more information, refer to "Clock Constraints File" on page 129.
- `dynseqfilter=<value>` - Selects the type of clocking sequence and the path on which faults will be processed. For example, the value `repeat` selects a sequence that launches and captures with the same clock, and only paths in which all points are fed

and observed by the same clock will be processed. For more information on sequence types, refer to "Delay Test Clock Sequences" on page 149.

- `maxnumberpaths=<number>` - The maximum number of paths to generate for any one path group. There is a rising and falling group created for each specified path. The default is 20.
- `pathtype=nearlyrobust|robust|nonrobust|hazardfree` - Type of test to generate. Refer to "Path Tests" on page 142 for more information. The default is `nearlyrobust`.

The following are the timed options:

- `delaymodel=<name>` - Name of the delay model for timed ATPG tests.
- `earlymode/latemode` – Ability to customize delay timings. Default is `0.0,1.0,0.0` for each option. For more information, refer to "Process Variation" on page 133.

Options to simulate paths against existing patterns:

- `inexperiment=<name>` - input experiment name
- `tbcmd=sim` - whether to generate path tests (`tbcmd=tg`) or resimulate existing path tests (`tbcmd=sim`).

To simulate paths against existing patterns, use the `inexperiment` and `tbcmd` keywords.

Refer to "create_path_delay_tests" in the *Encounter Test: Reference: Commands* for more information on these keywords.

Prerequisite Tasks

Complete the following tasks before executing Create Path Delay Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in *Encounter Test: Guide 1: Models*.
2. Build Encounter Test testmode

Output Files

Encounter Test stores the test patterns in the experiment.

Command Output

The output log contains information about the number of paths tested and the type of generated test. A sample output log is given below:

```
Hzfree= Hazard Free Robust
Robust = Robust
NrRob = Nearly Robust
NoRob = Non Robust

*************************************************************************
----Dynamic Path Test Generation Final Statistics----
Final Experiment Statistics for Path Faults

#Faults  #Tested  #HzFree  #Robust   #NrRob   #NoRob  #NoTest  %TCov %HFree  %Rob
%NrRob %NoRob %NoTest
All Paths       4040      321        0        0        0      321        0   7.95
0.00   0.00   0.00   7.95   0.00
OR  Groups       920      138        0        0        0      138        0  15.00
0.00   0.00   0.00  15.00   0.00
AND Groups         0        0        0        0        0        0        0   0.00
0.00   0.00   0.00   0.00   0.00
*************************************************************************


----Final Pattern Statistics----

Test Section Type                        # Test Sequences
---------------------------------------------------------
AC Path                                  126
---------------------------------------------------------
Total                                    126
```

Path File

A path file is used as an input to the `prepare_path_delay` and `create_path_delay_tests` commands. It describes the paths for which the tests are to be generated. The path may consist of one entity or describe an entire path.

The syntax for the path file entry is:

```
pathname_identifier {type}  block/pin/net {edge} ......;
```

- `pathname_identifier` - The name that identifies the path.
- `type:`
  - `-setup` (default) path transition must stabilize before the capture clock.
  - `-hold` path transition arrives after the clock edge that launched the transition
- `edge:`
  - `r`, `+` Rising transition required at the previously specified block/pin/net

- `f, -` Falling transition required at the previously specified block/pin/net
- blank (default) Either of the transitions is allowed at the previously specified block/pin/net.

The guidelines for path file syntax are as follows:

- Names should not include spaces or semi-colons.
- A path specification should end with a semi-colon.
- Nodes in the path can be pins, blocks, or nets.
- Nodes should be separated by spaces.
- Names should be in Encounter Test format (that is, not Verilog or SDF formats).
- The range of nodes describing a path may include all to just one node. When a limited set of nodes describes a path, Encounter Test will develop one or more paths that represent the given path. The keywords `selectby` and `maxnumberofpaths` control the path selection.
- Paths through storage elements are not allowed.

The following example shows a path file consisting of four paths:

```
MyPath FF21.Q andGate1.outPin orGate1.outPin;
MyPath2 FF33.QN andGate1.outPin orGate2.outPin;
MyPath3 -setup FF33.QN r andGate1.outPin r orGate2.outPin r;
MyPath4 -hold FF33.datain r;
```

In the preceding example, `MyPath`, `MyPath2`, `MyPath3`, and `MyPath4` are the identifiers for each of the four paths. Following each name is the group of block/pin/nets that describe the points along the path.

`MyPath1`, `MyPath2`, and `MyPath3` are the paths tested using a setup test, which consists of a launch and capture clock. `MyPath3` includes the required transitions along the path. `MyPath1` and `MyPath2` will create two tests, one each for the rising and fall launch transitions, while `MyPath3` will only create paths that start with a rising transition, as `FF33.QN` is at the beginning of the path.

`MyPath4` represents a hold path. This is tested by a test sequence with a single dynamic clock pulse. The clock launches the transition at path source and the end of the path should not transition until after the same clock arrives at the latch that the path feeds.

Path Tests

These tests analyze whether a transition propagates through a specified user-defined path and a directed acyclic set of gates.

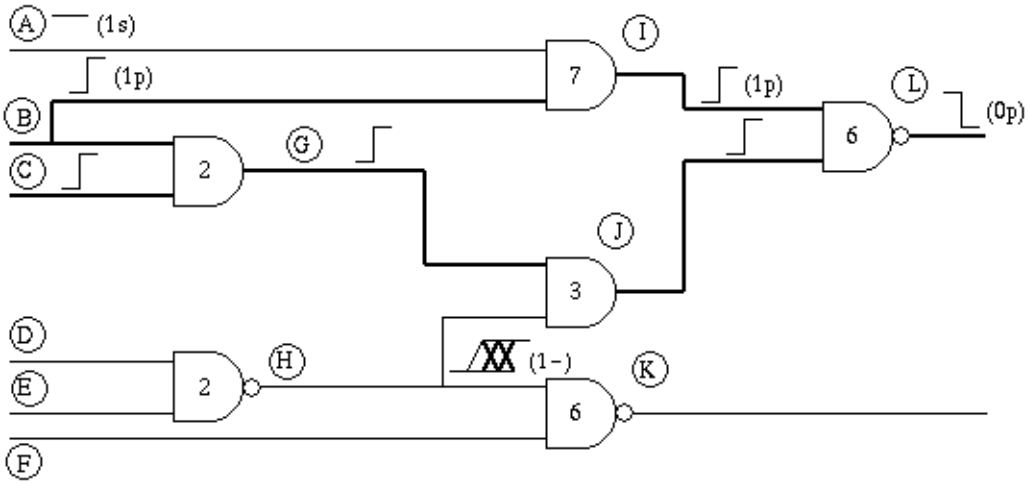
Hazard-Free Path Tests

Hazard-free tests require that the only possible way for the transition to arrive at the sink is without any interference from the gating signals that intersect the path. That is, the gating signals must be at a constant steady state; glitches are not allowed. To make a test hazard-free, all inputs into the path must be at a steady state (A=1s). Being at the steady state does not allow two paths to intersect. The design in Figure 4-17 can be made hazard-free by ensuring C and H are at a steady 1 (1s).

The benefits of these tests are:

- They allow the individual paths to be tested and diagnosed.
- They find very small defects.
- They can be used for characterization of paths to less than the system cycle. Refer to "Characterization Test" on page 137 for additional information.

Figure 4-17 Robust Path Test



Robust Path Tests

Robust path tests are “relaxed” hazard-free tests. The difference is that for robust tests the controlling to non-controlling transitions do not require a steady value from the gating signals. This means the off path pin may be unstable until the final state is achieved. Conversely, a non-controlling to controlling transition on the path requires the other pins to be at a steady value.

Robust path test tests the delay of a path independent of the delays of the other paths, including paths that intersect it. So if path B-I-L in Figure 4-17 is faulty, then regardless of the delay on the other paths, L will detect the 1 to 0 transition late. If a fault is present on B and C, then L will detect the transition late, however we are unable to diagnose from which path the fault has come.

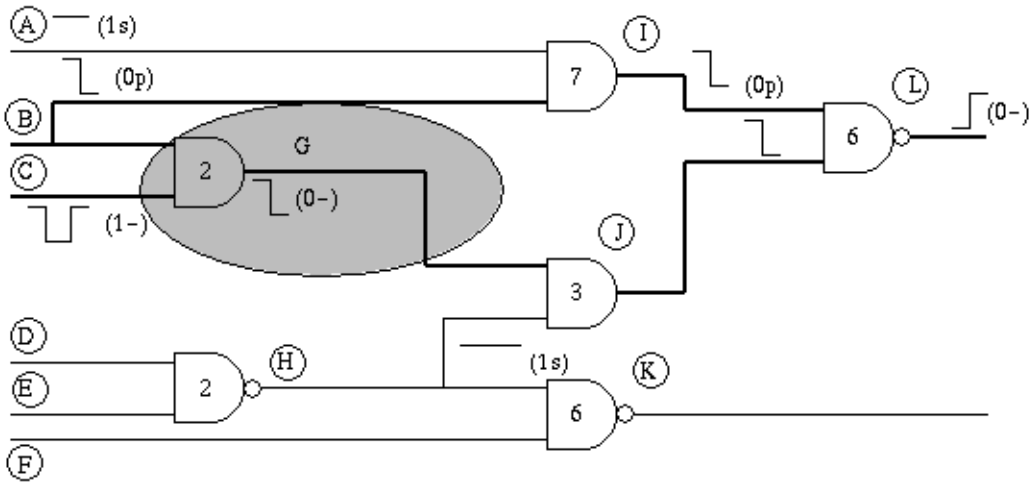
Non-Robust Path Tests

Non-robust path tests allow hazards on both controlling to non-controlling on the off path gating. The only requirement is that the initial and final state of the path's sink is a function of the transition of the transition of the source. For example, if the source does not change state, the sink will settle at the state that is a function of the source.

The non-robust tests create a path test, however a faulty value on another line can invalidate the test. An example is the hazard on C in Figure 4-18. If the glitch widens the value at L, the value still appears reasonable even though there is a fault on B. The glitch is typically caused by values changing from 0 to 1 and 1 to 0 on two inputs of AND or OR gates.

If the robust test is invalidated by a signal with a known definite glitch (a two input AND gate with both inputs transitioning in opposite directions would create a definite glitch or hazard), then the test is identified as a nearly-robust path test. A plain non-robust path test is one where off path gating requires a steady state value on an off path input, and it is impossible to get a steady state (for robustness) or definite hazard (for near robustness) on that pin.

Figure 4-18 Non-Robust Path Test



Delay Defects

Delay defects are modeled by dynamic faults. Catastrophic spot point defects are modeled by dynamic pin and pattern faults. The cell level defect and process variations use the dynamic path pattern fault. Both use a single transition to excite the fault effect. Dynamic pin and pattern faults require values at one gate. Path faults require a specific transition to be excited and the effect of that transition propagated along the chosen path. Path faults also have a variable detection criteria. They can be detected as hazard-free, robust, nearly-robust or non-robust.

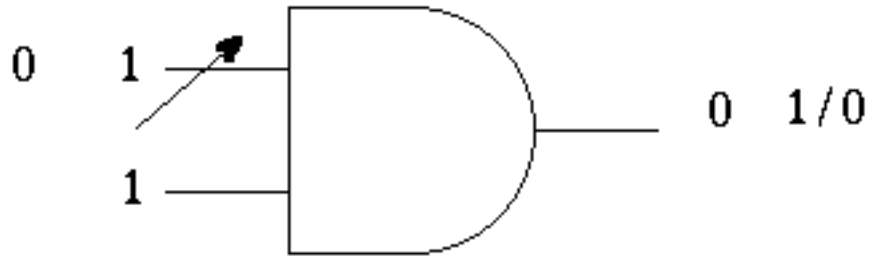
Refer to the following for related information:

- "Encounter Test Fault Model Overview" in *Encounter Test: Guide 4: Faults*.
- "Create Exhaustive Tests" on page 213

The following example shows an AND gate with a single (slow-to-rise) pin fault. The only pin required to change state is the faulty pin. The other input to the AND gate may or may not see a transition, but it must have a non-controlling value (1 for the AND gate) in the final state. The final output state is 1 in this example. If the defect exists, the output will be 0 for longer than expected time during which its state must be observed. The time the defect is present can be as small as a few hundred picoseconds or 10's of nanoseconds. Though there is a time component to determine whether the defect is active, the detection of the fault does not take

this into account. In our example, if the value (0) which is opposite the expected value (1) is captured in a latch (by a clock event) or PO (by a PO measure) immediately following the launch of the transition, the fault is detected.

Figure 4-19 Example of a Transition Fault



The path pattern faults model a defect or process variation that accumulate across multiple gates causing a signal to arrive late. Refer to "Path Pattern Faults" in the *Encounter Test: Guide 4: Faults.*

Delay ATPG Patterns

Manufacturing tests generate tests to detect random catastrophic spot delay defects. To ensure the random defects throughout the product are detected, dynamic faults are modeled at all gate or cell pins on the product. Dynamic faults are transition faults (slow-to-rise or slow-to-fall). See Figure 4-19 on page 145 for an example. These dynamic faults are targeted by True-Time Test flows as well as the fixed time delay test solution.

Parametric process problems cannot necessarily be observed at a single fault site. The delay effects of such a variation accumulate across the path. Characterization Test provides for testing process variations by testing specific paths. The paths may include be a complete set of logic between memory elements or smaller segments within technology cells. The paths are modeled by dynamic path pattern faults. The locations and paths of these faults can be user-determined or paths can be selected based on their path lengths.

An element that is essential to delay testing is that the test patterns must be applied with as little slack at the measure points (observable scan chains or POs) as possible. This can entail running at or near the system speed, or in some cases, faster than system speed for short paths. Encounter Test uses information contained within a Standard Delay File (SDF) as input to the True-Time Test methodologies. The SDF is used to automatically set the timings of PI switching and clock pulses at the tester.

In the True-Time Test methodology, the SDF is also used to automatically determine which memory elements have paths that are significantly longer than the rest of the paths within a

given clock domain (such as multi-cycle paths). These outliers are automatically made to not cause failures at the tester by either measuring X at these elements (ignoring the measure) or by constraining the logic that creates the long path during test pattern generation.

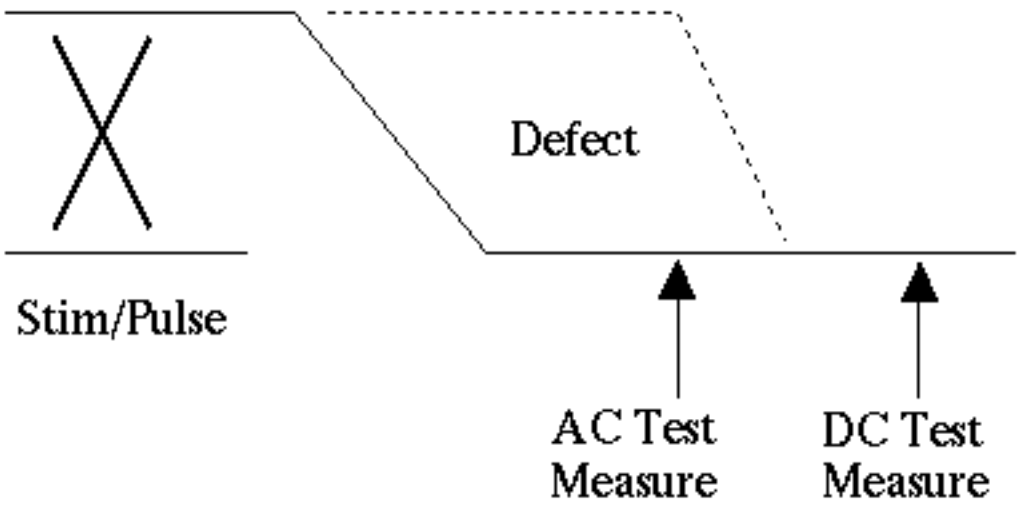
In the At-Speed and Faster Than At-Speed True-Time methodologies, the SDF is used in conjunction with a clock constraints file (described in "Clock Constraints File" on page 129). The clock constraints file specifies the desired operating speeds of each clock domain to be tested as well as several other user-defined timing relationships. By using these timing relationships, the outlier paths are determined the same way as in the automatic flow and the logic that contributes to paths that do not meet this user-defined timing are either ignored (measure X) or constrained during test pattern generation.

The At-Speed and Faster Than At-Speed flows require a method of achieving a high clock rate on the product during test. Sometimes this clocking can be provided by the tester, but it is more likely some hardware assistance will be required on the part itself. Additional On-Product Clock Generation support is available to achieve the system speeds. Refer to "On-Product Clock Generation (OPCG) Test Modes" in the *Encounter Test: Guide 2: Testmodes* for additional information.

Use of a design constraints file is another method to supplement At-Speed and On-Product test generation. Refer to "Design Constraints File" on page 111 for details.

The dotted line in Figure 4-20 shows the effect of a delay defect and the difference between a delay (AC) test and a static (DC) Test. A delay defect causes the expected switch time (represented by the solid line), to be delayed (represented by the dotted line). The difference between the delay and static tests is that the delay test generates tests to explicitly cause the designs to switch and when the result should be measured, while the static tests do not require a transition. In a static test, the measure can be done later since a static defect is considered to be tied high or low.

Figure 4-20 General Form of an AC Test

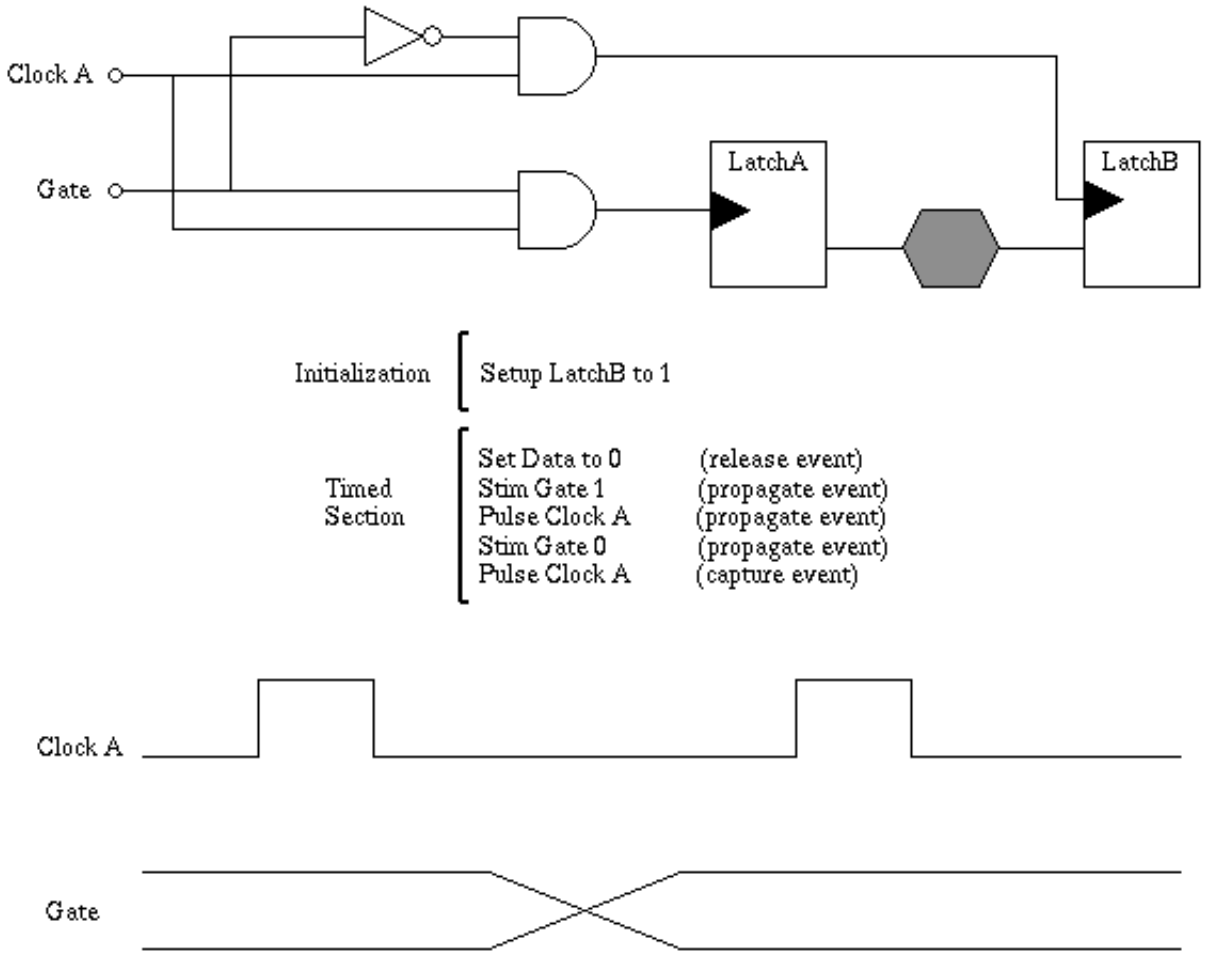


A basic delay test is a two pattern (or two clock, or two cycle) test. The first pattern is called the launch event because it launches the transition in the design from 0 to 1, or 1 to 0 (additional transitions of 0 to Z, 1 to Z, Z to 0, and Z to 1 are also possible in some designs). The second pattern is called the capture event because it captures, or measures, the transition at an observe point (register or primary input).

In its simplest form, a dynamic test consists of initiating a signal transition (the `release` event) and then some measured time later, observing the effect of this transition (the `capture` event) at some downstream observation point (typically a latch). For a transition to occur, the design must be in some known initial state from which the transition is made. Therefore, a dynamic test usually includes a setup or "`.initialization`" phase prior to the timed portion of the test. When the point of capture is an internal latch or flip-flop, observation of the test involves moving the latch state to an observable output.This is done with scan design by applying a scan operation to unload the scan chain.

The three phases of a dynamic test, initialization, timed, and observation are depicted in Figure 4-21 along with the three types of events in the timed portion. In general, a single dynamic pattern may have several events of each type.

Figure 4-21 Dynamic Test Sequence



A dynamic test may be either timed or not timed. When it is timed, Encounter Test defines the timing in a sequence definition which is a sort of pattern template and accompanies the test data. When the dynamic tests are not timed, the tests are structured exactly the same as for timed tests, but there is no timing template. In this case, the timing, if used at all, is supplied by some other means, such as applying the clock pulses at some constant rate according to the product specification. Refer to "Default Timings for Clocks" on page 184 for related information.

The optimized timed test calculated by using delays from the SDF and tester information from the Tester Description Rule is, depending on the tester's accuracy, a test that can run faster than the tester's cycle time. If the release and capture clocks are different, they can be placed in the same tester cycle and their pulses can overlap to obtain an optimized (at-speed) test. If the release and capture events both require the same clock to be pulsed, as is the case for most edge-triggered designs, then the two clock pulses must be placed into consecutive

tester cycles. By timing these two consecutive tester cycles differently, the pulses can be pushed out to near the end of the first cycle and pulled in near the beginning of the following cycle. This allows the pulses to occur with a cycle time much less than that of the tester and much closer to that of the product. This allows the test to run at the speed necessary to truly test the potential defect because it is ".at-[the]-speed". of the design, not at-the-speed (or at the mercy) of the test equipment.

Delay Test Clock Sequences

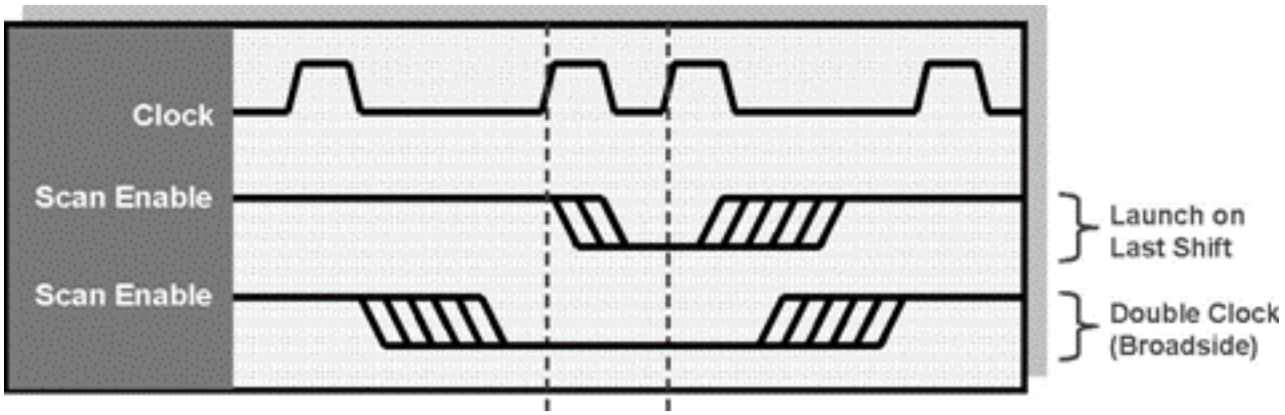
Transition fault testing is accomplished by passing a transition from a source flop, through the fault site, to a sink flop, and controlling the launch to capture timing. This requires design values to be set up in at least two time frames, the first to generate the transition and the second to capture the results.

There are two techniques for generating the two time frames:

- Launch on last shift requires that the final shift create a transition at the outputs of the required flops
- Double clocking (also known as functional release) uses two clocks in system operation (or capture) mode to create the transition and capture the results.

In Figure 4-22, the timings for the two transition generation techniques indicate that the clock is essentially the same. The scan can be run slow (to control power), but there must be two cycles of the clock at the test time and the time from active edge to active edge is controlled.

Figure 4-22 Delay Test Two-Frame Clocking



In launch-on-last-scan, the Mux-Scan "Scan Enable" signal must be toggled after the launch clock edge and before the capture clock edge. In double-clock, the scan is completed, and the clocks can even be held off for an indefinite period while the scan enable signal is toggled.

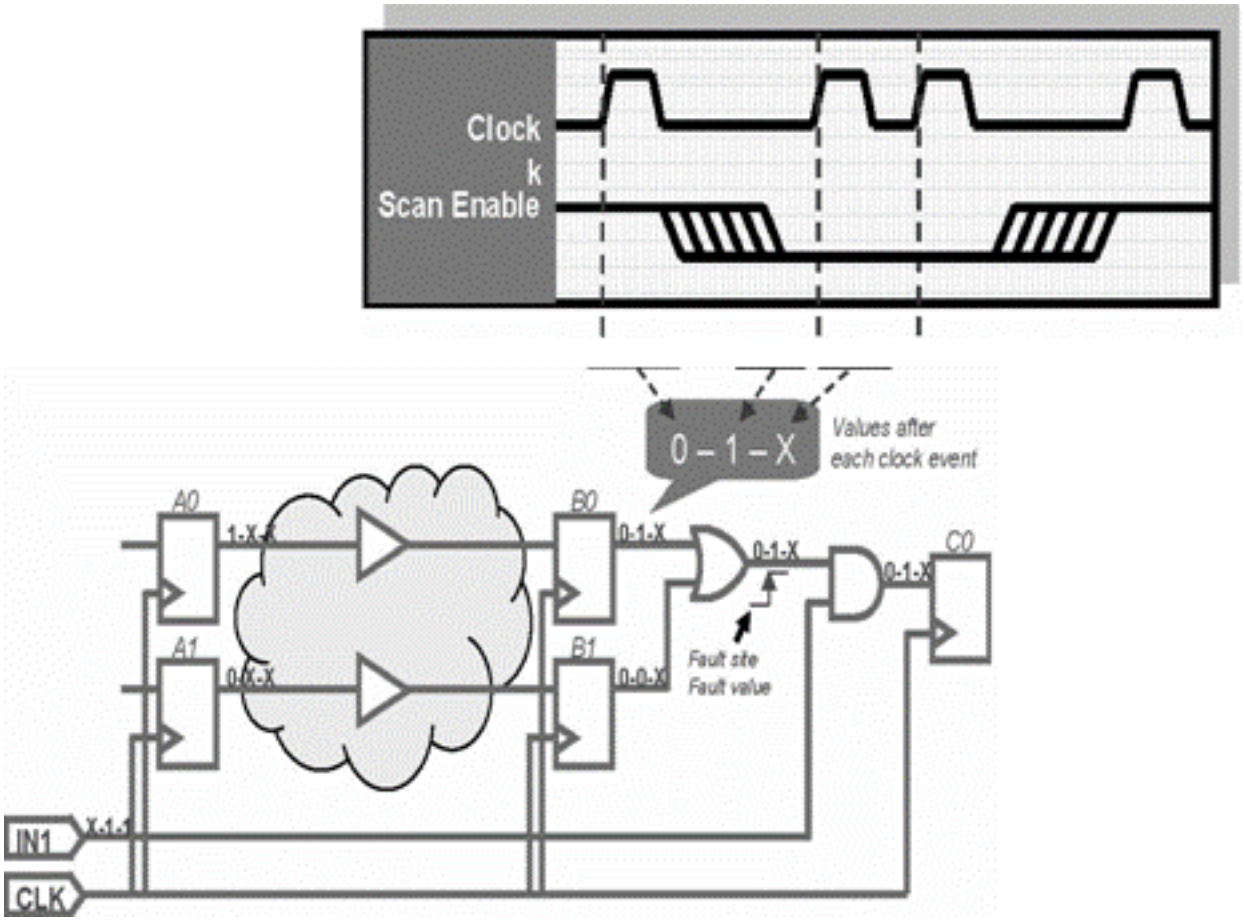
Then two timed clocks are issued, and there is again an indefinite amount of time to toggle the scan enable. This places no significant restrictions on the scan enable distribution and is easier timing in a Mux-Scan design but more difficult test generation.

The default for MuxScan designs is to use double-clock only. It is recommended to allow the test pattern generator to decide which method to use. If the SDF is provided, the timings of the `Scan_Enable` signal can be appropriately controlled. The keywords of the `create_logic_delay_tests` command for controlling clock sequence generation are:

- `dynfuncrelease=yes|no` where `yes` restricts the clock sequences to double-clock only, and `no` allows the test pattern generator to use either.
- `dynseqfilter=any|repeat|norepeat|onlyclks|clkpiclk|clockconstraint` where `repeat` will allow only patterns that repeat the same clock (within a domain), and `onlyclks` will also allow inter-domain sequences.

A method of calculating the values to scan into the flops in double-clock test generation is illustrated in Figure 4-23. Three levels of flops are depicted.

Figure 4-23 Delay Test Execution



To generate a test that detects a slow-to-rise fault on the output of the `OR` gate, a `0` or `1` transition must be created on the output of the `OR` gate using the following steps:

1. Place an initial value of `0` on the output of the OR gate by scanning a `0` into the `B0` flop and `B1` flops. The three values shown on each output pin are the values after the last shift clock, launch clock, and capture clock, respectively.
2. Place a value of `1` on the output of the `OR` gate. To accomplish this, the `B0` flop must capture a `1` at the launch event of the capture clock. Therefore, a value of `1` must be scanned in the `A0` flop in the previous scan shift. Similarly, for `B1` to capture a `0`, `A1` must have `0` scanned into it during the previous scan shift. Set input pin `IN1` to `1` to propagate the fault to `C0`.

The second pulse of the clock is the capture event. If the transition propagates with normal timing, a `1` is captured into the `C0` flop. If there is a delay defect, a `0` will be captured. During fault simulation, additional transition and static faults will also be detected by this pattern.

Customized Delay Checking

Customized delay checking is an advanced capability to review and remove log messages produced by Build Delay Model. It is highly recommended to first have a full understanding of the information conveyed by the messages in order to successfully use this capability.

Build Delay Model performs an integrity check is the data contained in the SDF file. The check verifies the created design that is patterned according to the SDF data is correctly modeled by Encounter Test. The secondary purpose of the check is to verify the completeness of the information within the SDF file.

The delay checking process is driven by information in a binary file named `TDMcellDelayTemplateDatabase`. This file can be created via any of the following:

- Automatic generation during the `build_delaymodel` process
- Automatic generation using `build celldelay template`
- Manual creation or update using `read_celldelay_template`

A `TDMcellDelayTemplateDatabase` file can be generated by either `build_delaymodel` or explicitly running `build_celldelay_template` with appropriate parameters to generate delays for the desired cells. A database file can be produced in readable form by specifying the `printtemplates=`*`filename`* keyword option for `build_celldelay_template`. The format of this file resembles an SDF file with all of the numerical delay information removed. The file describes which pins have timing relationships in each cell, but does not describe what those relationships actually are. The file contains an entry for top-level `CELLs` only. Each definition can be customized by the configuration of the pins on a given instance of the `CELL` as seen in the following example:

```
CELL AND
{
    IOPATH A RISING Z RISING
    IOPATH A FALLING Z FALLING
    IOPATH B RISING Z RISING
    IOPATH B FALLING Z FALLING
}

CELL AND
(
    TIE1 A
)
{
    IOPATH B RISING Z RISING
    IOPATH B FALLING Z FALLING
}
```

This shows two definitions for the same `AND` cell, but in the second case, the first input pin is tied to `1`. This latter definition will only apply instances of the `AND` cell in which the first input is tied to `1`. Otherwise, the first definition will apply, as it is more generic.

Once exported in this form, the definitions can be manually edited and read back in to the binary database used by `build_delaymodel` for delay integrity checking. To read a modified set of definitions back in, use `read_celldelay_template` and specify the your text file containing the definitions for `templatesource` keyword.

Run `build_delaymodel` to build the delay model with the new definitions for checking.

Dynamic Constraints

Dynamic constraints are intended to prevent pattern miscompares for transitions propagating through the following types of delays:

- Long paths (negative slacks or setup time violations)
- Short paths (hold time violations)
- Delaymux logic
- Unknown delays

The preceding delay types may miscompare due to:

- Long path is not measured within the expected time
- Short paths are evaluated earlier than expected
- The transition propagates through a delaymux

The inclusion of dynamic constraints is a method to constrain timings. Through dynamic constraints, transitions could be prevented down long paths, such that the clock-to-clock timing does not require expansion to accommodate the long path.

Constraints are included in the linehold and sequence definition files. The sequence definition file (`TBDseq`) includes constraints as keyed data. The keyed data is found on the define sequence and the key is CONSTRAINTS. The entire set of constraints for a sequence definition is found in the string that follows the key. See "Linehold Object - Defining a Test Sequence" on page 178 and "Keyed Data" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

Refer to "Linehold File Syntax" on page 173 for details on linehold syntax supporting the transition entity.

Constraint Checking

Timing constraints are enforced in the following two ways.

- The constraints are justified by the ATPG engine using the `constraintjustify` keyword.
- During simulation, the ATPG engine verifies the test patterns do not violate the constraints (using the keyword `constraintcheck`).

In some cases, the ATPG engine is unable to generate patterns in the presence of all the constraints, therefore, turning off `constraintjustify` allows the ATPG engine to generate patterns. In this case, the simulation engine can verify that the constraints were not violated, and if required, to take some action (either remove or keep them). The `constraintcheck` keyword controls this (defaults to `yes`).

If your delay test flow uses `prepare_timed_sequences`, then the ATPG engine will try to locate paths that will not be safe to measure within the frequency in design's clock constraints file. If the engine finds such paths, it creates "transition constraints" to prevent transitions from occurring along these long paths. If the test generator can successfully stop transitions from occurring along these paths, then the ATPG engine will not be required to create a timing X at the measure location of this path, which will improve coverage. If Encounter Test fails to prevent transitions down this path, it will have to measure an X on every flop that this long path feeds. Using `constraintcheck=yes` allows this.

If you do not specify `constraintcheck=yes`, but are using one or more transition constraints (as a result of `prepare_timed_sequences` or SDC), then it is highly recommended to use an SDF-annotated simulation, such as ncsim, to verify your patterns; otherwise the patterns might not work on the tester because of the violated constraints. The alternative is to use the default option and let Encounter Test perform the verification and correction (if required) in a single step.

In case of a performance issue, first examine the number of constraints that are being created in the `prepare_timed_sequences` phase. If this number is excessive, check the clock frequency(s), because it means that Encounter Test has identified too many paths that do not appear to meet your timings. Check specifically the number of violations reported by `prepare_timed_sequences` and compare this number to the number of flops in the given domain. If it is more than `1%` or `2%`, then your timings might be too aggressive.

Timed Pattern Failure Analysis for Tester Miscompares

A Timed Test Pattern is used to detect transition faults (or dynamic faults) at a speed that is as close as possible to the system clock speed. Encounter Test uses a Standard Delay File

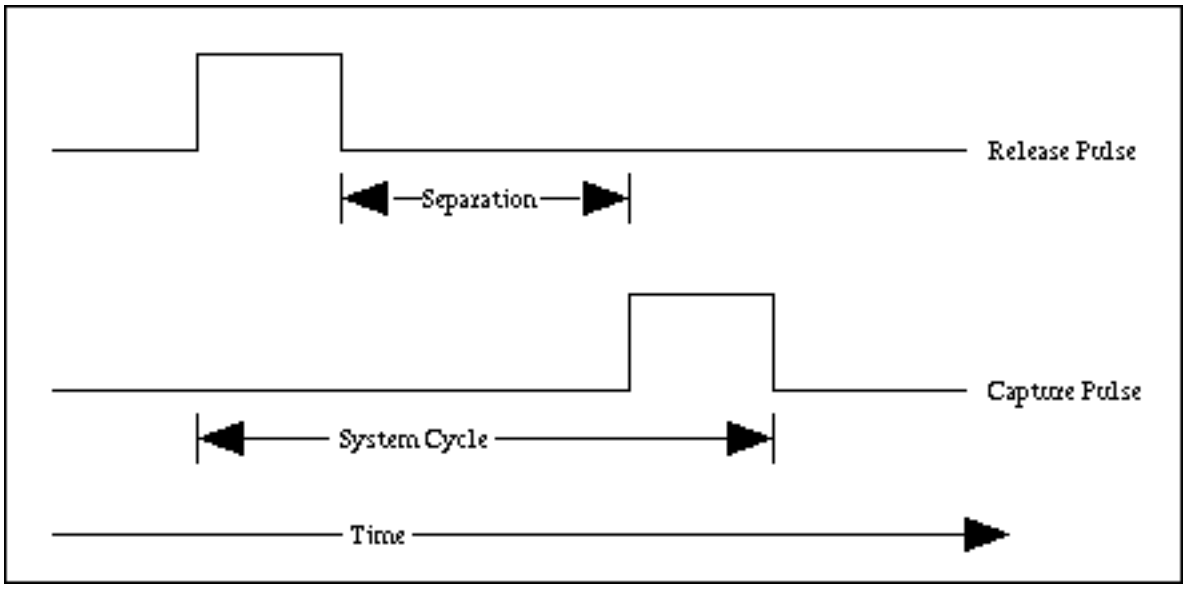
(SDF) as the input for all the delays in a chip. See "Timing Concepts" on page 96 for additional information.

The accuracy of the SDF drives the quality of the timings produced by Encounter Test. The level of SDF accuracy directly relates to the ability of Encounter Test to create timed test patterns that work on the tester after a single test generation iteration.

A Timed Test Pattern is comprised by 3 basic building blocks. The building blocks are:

1. The release pulse, used to launch captured data into the system logic.
2. Separation time, the calculated amount of time that is required between the release and capture pulses.
3. The capture, used to capture the design values in a measure point. See "General View Circuit Values Information" in the *Encounter Test: Reference: GUI* for related information.

Figure 4-24 Building Blocks of a Timed Test Pattern



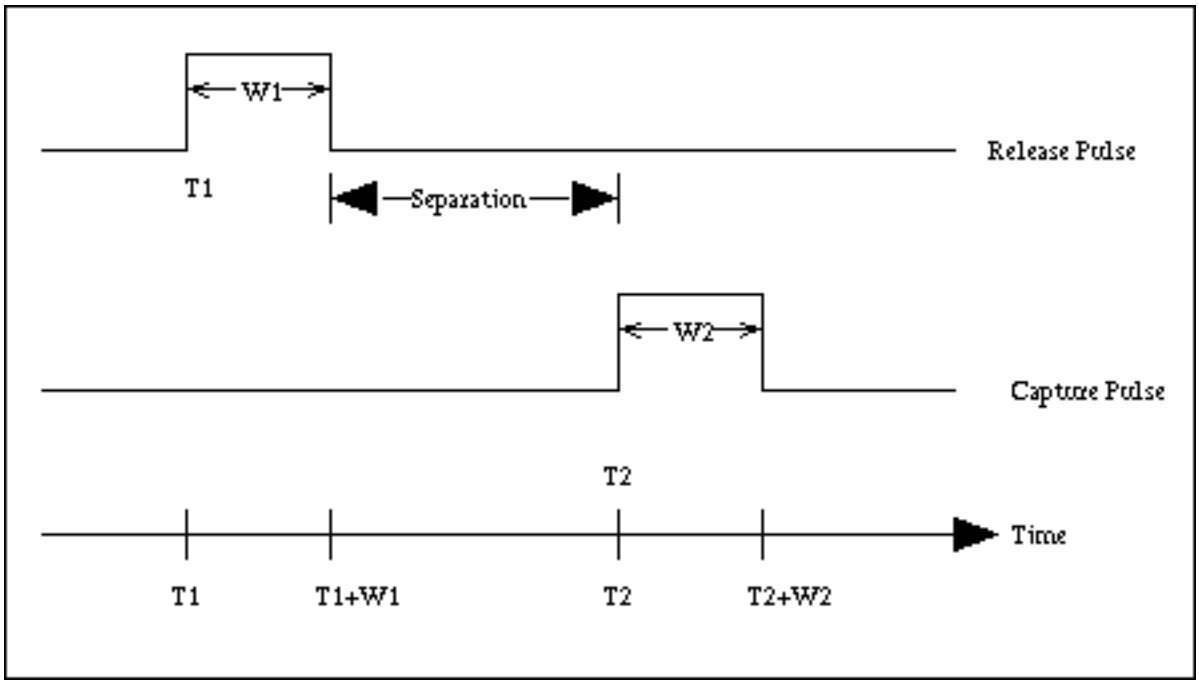
The release and capture pulses are comprised of the two following basic elements:

1. The time at which the pulses occur
2. The length of the pulses (pulse width)

The time and width of the pulses calculated by Encounter Test are based on the SDF.

Figure 4-25 shows the five elements that control the clocks at the tester:

Figure 4-25 Elements that Control Clocks at the Tester



1. Time of Release Pulse = T1
2. Width of Release Pulse = W1
3. Separation Time = S1
4. Time of Capture Pulse = T2 = T1+W1+S1
5. Width of Capture Pulse = W2

A WGL example for a generated Timed Test:

```
timeplate "parallel_cycle_lssd_dynamic_2010" period 80 ns
    "CE0_TEST" := input[ 0ns:S ];
    "D1" := input[ 0ns:S ];
    "D2" := input[ 0ns:S ];
    "D3" := input[ 0ns:S ];
    "BCLK" := input[ 0ns:D, 8ns:S, 16ns:D ];
    "SI1" := input[ 0ns:S ];
    "SI2" := input[ 0ns:S ];
    "CCLK" := input[ 0ns:D, 24ns:S, 32ns:D ];
```

Time of Release Pulse = BCLK = 8ns
Width of Release Pulse = 16ns - 8ns = 8ns
Separation Time = 24ns - 16ns = 8ns
Time of Capture Pulse = CCLK = 24ns
Width of Capture Pulse = 32 ns - 24ns = 8ns

When Timed Test Patterns fail at the tester, it is key to understand why and where these failures are occurring. The first step in determining why Timed Test Patterns fail is to slow the patterns down to verify that they pass at system speed similar to static tests. Static Tests are identical to a Timed Test Patterns except that a static pattern is not run at faster tester speeds.

The speed of a Timed Test Pattern can be slowed to the speed of a Static Test Pattern by performing the following steps.

1. Widen the separation time between the clock pulses. If the separation time between the release and capture clocks is 5ns, consider widening the separation to 200 ns.
2. Increase the widths of the release and capture pulses. Encounter Test creates the Timed Test Patterns with the minimum pulse widths required to release or capture a value at a latch. These calculated values (from the SDF) might not be large enough for the real product to correctly perform. Widening the pulses by 20x or 30x from the original values helps ensure that the latches have enough time to release or capture the values. The basic concept is to increase all the times associated with the timeplates to a big value.

If viewing the modified times associated with the 5 elements that control the clocks at the tester we would see:

Time of Release Pulse = T1
Width of Release Pulse = W1*30
Separation Time = P1*200
Time of Capture Pulse = T2 = T1+W1*30+P1*200
Width of Capture Pulse = W2*30

These values can be changed at the Tester using software provided by the Tester Company or can be modified by changing the timeplates inside of the WGL files.

A modified WGL example for a generated Timed Test:

```
timeplate "parallel_cycle_lssd_dynamic_2010" period 2400 ns
    "CE0_TEST" := input[ 0ns:S ];
    "D1" := input[ 0ns:S ];
    "D2" := input[ 0ns:S ];
    "D3" := input[ 0ns:S ];
    "BCLK" := input[ 0ns:D, 8ns:S, 248ns:D ];
    "SI1" := input[ 0ns:S ];
    "SI2" := input[ 0ns:S ];
    "CCLK" := input[ 0ns:D, 1848ns:S, 2088ns:D ];
```

Time of Release Pulse = BCLK = 8ns
Width of Release Pulse = 8ns * 30 = 240ns
Separation Time = 8ns * 200 = 1600ns
Time of Capture Pulse = CCLK = 8ns + 240ns + 1600ns = 1848ns
Width of Capture Pulse = 8ns * 30 = 240ns

Applying this new *slow* Test Pattern at the tester verifies that the patterns work. Once the patterns are successful at slow speeds, the timings can be moved closer to the originally computed values. Reduce pulse widths to their point of failure before changing the separation time. When the pulse widths are satisfactory, the separation between the clock pulses can be lowered. Determine which latch is failing by lowering the values until some failures start to occur.

After identifying some latches that miscompare at faster tester cycles that pass at slower test cycles, consider the following:

Are the timings associated with the logic feeding into the latch correct in the SDF? Should this latch be ignored because it will never make its timings? Do all chips fail at the same latch?

If the identified latch should be ignored or will not make the timings in the SDF, there are two options:

a. Change the SDF so that it contains the correct values. Rerun Build Delay Model to recreate or re-time the pattern set.

b. Use an ignore latch file to instruct Encounter Test to ignore this latch during simulation. If the delay information appears correct, consider analyzing the failure information using the Encounter Diagnostics tools to ascertain whether Diagnostics can determine the point of failure. Refer to "Encounter Diagnostics" in the *Encounter Test: Guide 7: Diagnostics* for additional information.

Delay Test can also be used to print out the longest paths it finds for a certain latch.

The primary goal at the tester is to achieve passing timed test patterns. If the calculated timings from Encounter Test do not work at the tester, verify the patterns work at a slow speed then increase speed to a point of failure.

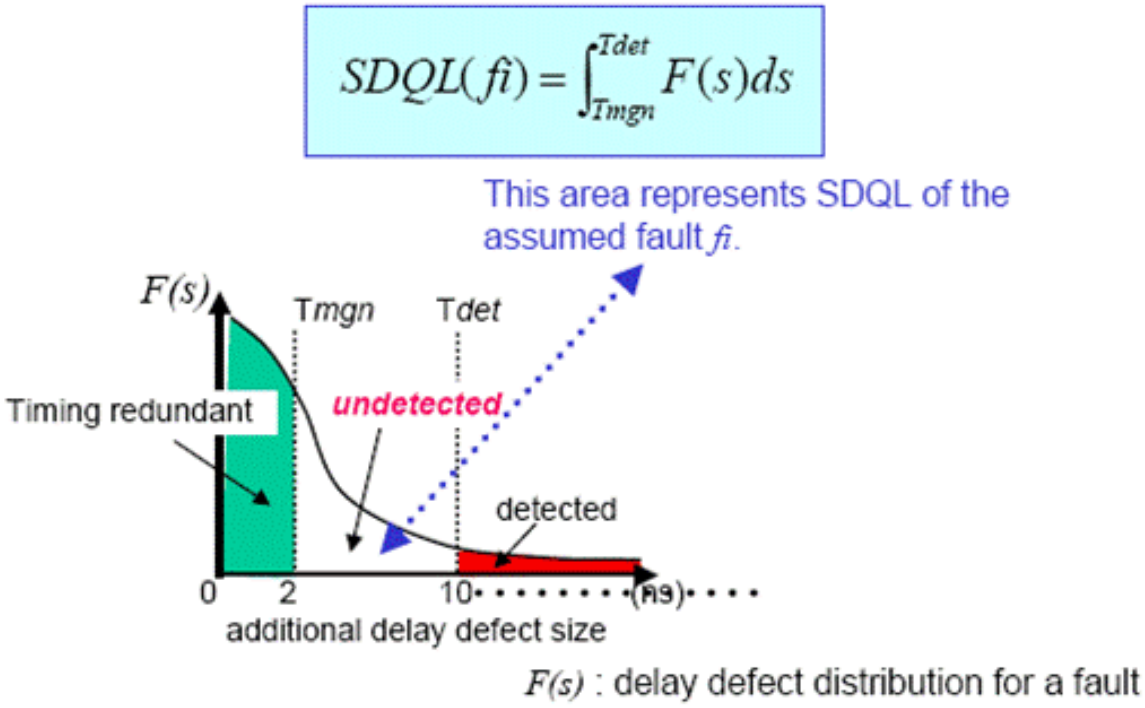
Performing and Reporting Small Delay Simulation

Encounter Test enables you to perform test generation that targets small delay defects. Based on the Statistical Delay Quality Model (SDQM), Encounter Test uses the Statistical (or Small) Delay Quality Level (SDQL) of a transition fault as the metric for measuring the effectiveness of Small Delay ATPG. When accumulated for all faults tested by a set of test patterns, SDQL measures the number of small delay defects that may escape the set of test patterns in parts per million (ppm).

SDQL: An Overview

To compute SDQL, a delay defect distribution function F(s) is used to determine how likely any one defect is to occur as a function of its size (as shown in Figure 4-26 on page 159). Typically this is provided by the manufacturing facility. Encounter Test uses a default delay defect distribution function if one is not provided.

Figure 4-26 Delay Defect Distribution Function



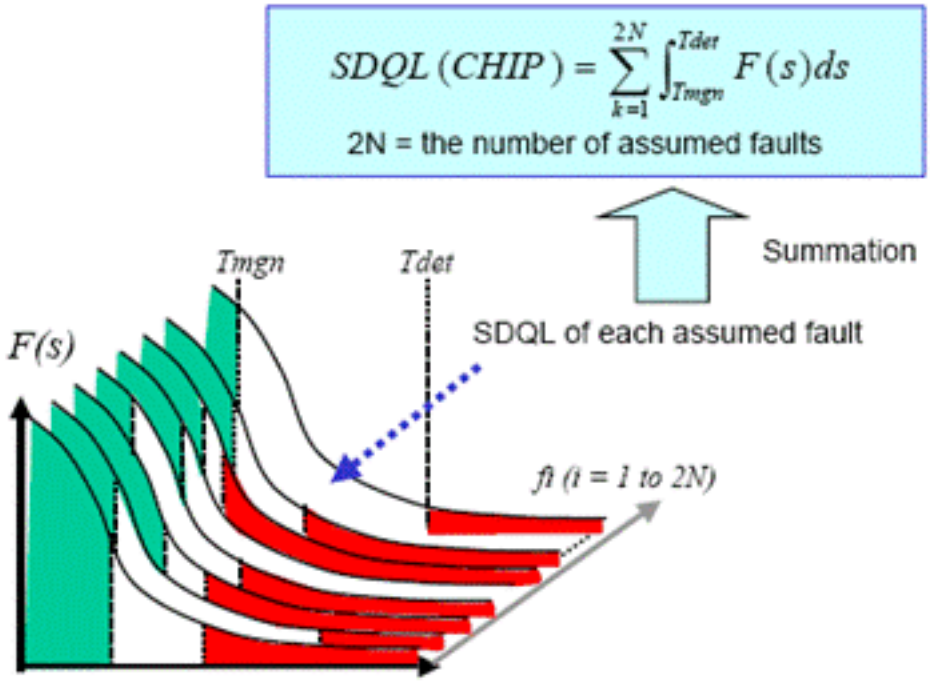
For any given fault, defects smaller than a certain size cannot be observed due to slack in the fault's longest path. This is often referred to as Tmgn or the timing redundant area.

When a fault is detected, the longest sensitized path (LSP) is determined. LSP is the longest path along which the fault can be observed by the test patterns. The detection time (or Tdet) is computed as the difference between the test clock period (Ttc) and the LSP. Defects larger than this are detected by the test.

The area under the curve that remains undetected by the test reflects the probability that a defect will occur for the given fault. This is referred to as the SDQL (or untested SDQL) for a fault.

The SDQL for the design is computed by accumulating the SDQL for all individual faults.

Figure 4-27 Computing Cumulative SDQL



SDQL Effectiveness

SDQL effectiveness is calculated as:

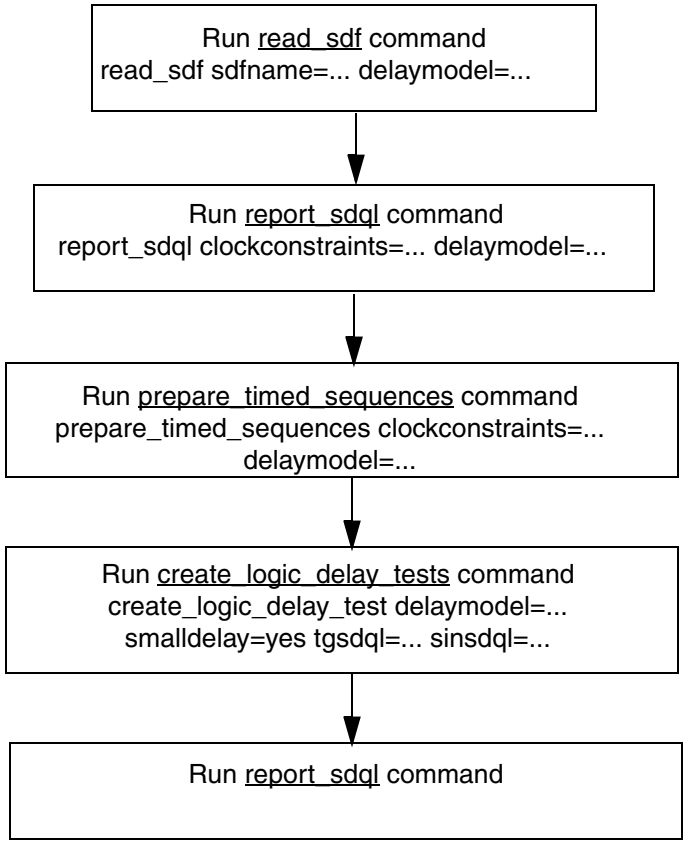
```
Tested SDQL
------------------------------ x  100%
Untested SDQL + Tested SDQL
```

Similar to test coverage, this measurement is independent of the number of faults (or in this case ppm) in the design.

Small Delay ATPG Flow

Small delay test generation is performed using the `create_logic_delay_tests` command:

Figure 4-28 Small Delay ATPG Flow



As small delay testing tends to run longer and generates increased pattern count as compared to traditional delay testing, a keyword (`tgsdql=`*`n.nnn`*) is provided to select only the most critical faults for small delay test generation. Any fault with a potential SDQL greater than the `tgsdql` value is selected for small delay test generation. All other faults use traditional delay test generation.

During ATPG, many patterns may detect a single transition fault. Typically, a transition fault is marked tested as soon as a test pattern detects the fault. To identify the longest path that detects a fault, full small delay simulation requires that a transition fault be simulated for every test pattern, which is a time consuming process. To reduce the run time, a keyword (`simsdql=`*`m.mmm`*) is provided to allow a fault to be marked tested as soon as it is detected

along a path long enough so that the untested SDQL is less than *m.mmm*. After the command is executed, any detected faults not meeting the threshold are marked tested.

Prerequisite Tasks

The following prerequisite tasks are required for small delay simulation:

1. Small delay simulation uses delay data provided in the form of an SDF. Therefore, run the `read_sdf` command before performing small delay simulation. Refer to "Performing Build Delay Model (Read SDF)" on page 92 for more information.
2. Since delay data is provided only to the technology cell boundaries, SDQL for faults internal to a technology cell cannot be computed accurately. Therefore, you should perform small delay simulation using a cell fault model. If `industrycompatible=yes` is specified for the `build_model` command, a cell fault model will be created by default. Otherwise, create a cell fault model by using the `build_faultmodel` command with `cellfaults=yes`. Refer to the *Encounter Test: Guide 1: Models* for more information.
3. Create test patterns using `create_logic_delay_tests` or `prepare_timed_sequences` with release-capture timing specified using a clockconstraints file. For patterns without release-capture timing, a clockconstraints file may be specified when performing small delay simulation (`analyze_vectors`). Refer to "Create Logic Delay Tests" on page 126 for more information.
4. To determine the appropriate values for `tgsdql` and `simsdql` keywords, run `report_sdql` to report a histogram of the potential SDQL without specifying the `experiment` keyword. Specify the `clockconstraints` keyword instead. The `report_sdql` command reports a histogram showing the distribution of the total potential SDQL among the transition faults. Recommendations for `tgsdql` and `simsdql` keywords are also reported, or you may select your own value.

Performing Small Delay Simulation

To perform small delay simulation run the `create_logic_delay_tests` command with the following keywords:

- `smalldelay=yes` - enables small delay simulation. When specified, the default value of the `writepatterns` keyword is set to `all`. The keyword `writepatterns=all` allows a pattern that may have detected a transition fault along a less than longest path to be kept.
- `delaymodel=<dm>` - the delay model used to compute path lengths

- `tgsdql=<n.nnn>` - faults with potential SDQL greater than this are chosen for small delay ATPG
- `simsdql=<m.mmm>` - faults with an untested SDQL less than this are marked tested

In addition to `simsdql`, the following threshold values are available to determine when a transition fault should be marked tested.

- `ndetect=<nn>` - mark a fault as tested after it has been detected the specified number of times (default is `10000`)
- `percentpath=<nn>` - mark a fault as tested if it is detected along a path that is the specified percentage of its longest possible path (default is `100`)

Note:

- When `simsdql`, `ndetect`, or `percentpath` is specified, faults are no longer simulated once they meet the detection threshold. As a result, the untested SDQL reported may be larger than the true untested SDQL for the pattern set.
- As a fault may never reach one of the specified thresholds, it may not be marked tested until the end of the test generation command when all faults that were detected but did not meet a threshold are marked tested. To prevent faults that did not meet a threshold from being marked tested, specify `marksdfaultstested=no`.

The following is an example of small delay simulation:

Example: Small Delay Test Generation Using report_sdql and create_logic_delay_tests

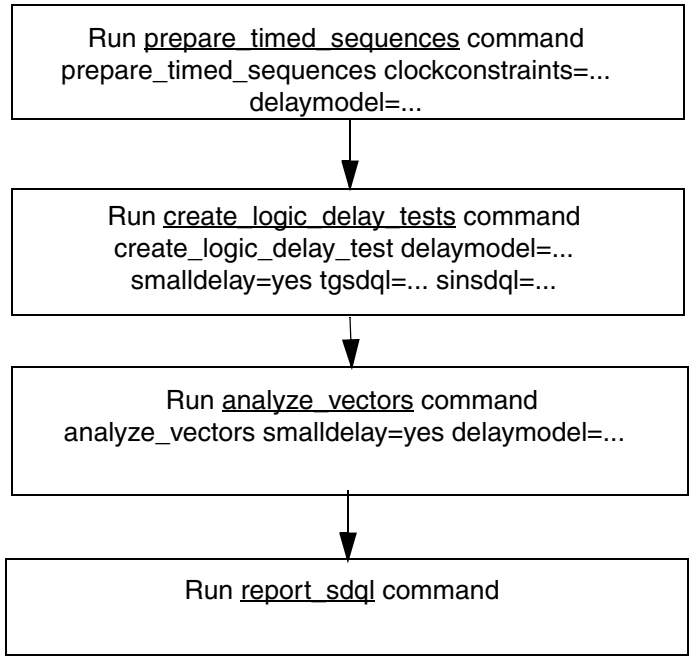
```
report_sdql delaymodel=<dm> testmode=<tm> clockconstraints=<ccfile>
(determine the value for tgsdql and simsdql from the report_sdql log)
create_logic_delay_tests .. delaymodel=<dm> smalldelay=yes simsdql=<value>
tgsdql=<value>…
```

In this example, faults with a potential SDQL greater than the `tgsdql` value are processed using small delay ATPG. Other faults are processed with traditional ATPG. Faults are not marked tested until they are detected along a path with an untested SDQL less than the `simsdql` value.

Small Delay Simulation of Traditional ATPG Patterns

The SDQL effectiveness of test patterns created using traditional ATPG may be determined using small delay simulation:

Figure 4-29 Flow for Small Delay Simulation of Traditional ATPF Patterns



Small delay simulation after ATPG

Example 1: Small Delay Simulation of traditional ATPG patterns

```
prepare_timed_sequences delaymodel=<dm>  clockconstraints=<ccfile>…
create_logic_delay_tests  … delaymodel=<dm> experiment=<exp>
analyze_vectors … delaymodel=<dm> smalldelay=yes inexperiment=<exp>
experiment=<exp_resim>
report_sdql testmode=<tm> experiment=<exp_resim>
```

In this example, patterns are created using traditional ATPG. Small delay simulation is then performed for all faults during `analyze_vectors`. As `simsdql` is not specified, faults are not marked tested unless they are detected along their longest possible path. Any faults that were detected along a shorter path are marked tested after the simulation is complete. Setting `simsdql` to `0` (or not specifying `simsdql`) provides the most accurate SDQL report, but `analyze_vectors` may run longer. `report_sdql` is then run to report the effectiveness of the resimulated pattern set.

Example 2: Small Delay Simulation of Traditional ATPG Patterns

```
create_logic_delay_tests  … delaymodel=<dm> experiment=<exp> smalldelay=yes
clockconstraints=<ccfile>
report_sdql testmode=<tm> experiment=<exp>
```

In this example, small delay simulation is performed during traditional ATPG eliminating the `analyze_vectors` step. Traditional ATPG is used for all faults because the `tgsdql` keyword has not been specified. If `prepare_timed_sequences` is not run, a `clockconstraints` file is required.

Note: When small delay simulation is being performed during traditional ATPG, fewer faults will be marked tested during simulation due to the requirement to meet a simsdql or other threshold. This will increase the run time for ATPG. Any faults detected but not meeting the simulation threshold are marked tested after the test generation is complete.

Example 3: Small Delay Simulation during ATPG with a User-specified Probability Function

```
create_logic_delay_tests … delaymodel=<dm> smalldelay=yes probfile=<mydirectory>/
SDQL.pl
```

In this example, instead of the default probability function, a user-specified probability function is provided. Refer to `report_sdql` in the *Encounter Test: Reference: Commands* for more information on the `probfile` keyword.

Reporting SDQL Effectiveness

Use the `report_sdql` command to create reports about the SDQL for a pattern set. Refer to `report_sdql` in the *Encounter Test: Reference: Commands* for more information on the command.

The reports produced by `report_sdql` are described below:

Reporting SDQL for Experiment

```
report_sdql testmode=<tm> experiment=<exp>
```

The following report is produced.

```
INFO (TDL-062): SDQL calculations will use a maximum time of 6100 ps and a total
area of  0.0008. [end TDL_062]

INFO (TDL-050): Clock Domains Being Processed
-------------------------------------------------------------------------
Domain #     Faults     Release Clock     Capture Clock     Test Period
-------------------------------------------------------------------------
1            6316       CLK               CLK               6100 ps
```

```
-------------------------------------------------------------------------
[end TDL_050]
```

The first section lists the clock domains being processed and the number of faults associated with them. Faults on PIs, POs, RAMs, ROMs, latches and flops are not included in this count. Faults that exist in multiple domains are included in the counts for each domain. The maximum time for integration and area under the defect probability distribution are also listed.

The next section lists the SDQL by domain for the faults that were tested during small delay simulation (as well as those that would have been tested if they had met the detection threshold). For faults that are included in multiple domains, their SDQL is included in each domain, but only once in the Chip SDQL.

```
INFO (TDL-052): SDQL By Clock Domain - Tested Faults
-------------------------------------------------------------------------------
         Untestable      Untested        Tested         Total        Test
Domain #   (ppm)           (ppm)          (ppm)          (ppm)      Effectiveness
-------------------------------------------------------------------------------
1          0.0105          0.0066         0.1065         0.1236       94.66%
Chip       0.0105          0.0066         0.1065         0.1236       94.66%
-------------------------------------------------------------------------------
```

Note that there is a column for Untestable SDQL. During small delay ATPG, some faults may be determined to be Untestable along their longest possible paths (LPPs). Rather than reducing the LPPs, an Untestable SDQL is computed. This allows a better comparison to patterns created using traditional ATPG since the total SDQL should remain the same.

The Tested Fault SDQL section of the report is followed by similar section that includes untested faults as well as tested faults.

```
INFO (TDL-052): SDQL By Clock Domain - All Faults
-------------------------------------------------------------------------------
         Untestable      Untested        Tested         Total        Test
Domain #   (ppm)           (ppm)          (ppm)          (ppm)      Effectiveness
-------------------------------------------------------------------------------
1          0.0105          0.0125         0.1065         0.1294       90.35%
Chip       0.0105          0.0125         0.1065         0.1294       90.35%
-------------------------------------------------------------------------------
Dropped Faults:
TMGN <= 0  :  0
Redundant  :  0
Untestable :  1371
Clock Line :  416
Tested Untimed:725
Total         :2512
[end TDL_052]
```

A table listing untested faults that were not included in the SDQL calculations is also included to help explain any differences in total SDQL from one experiment to the next.

Reporting SDQL for Individual Faults

```
report_sdql testmode=<tm> experiment=<exp> defectprobability=0.002
```

A sample report is given below:

```
INFO (TDL-051): Defect Probability Report
-------------------------------------------------------------------------------
Defect     Fault    Defect     System   Longest Poss Timing  Test    Longest
Size             Probability  Period      Path    Redundant Period Sens Path
(ps)                          (ps)             (ps)   (ps)     (ps)
-------------------------------------------------------------------------------

8371     0.000303    6100       5633      467      6100      1342     4758
8372     0.000210    6100       5573      527      6100      4727     1373
9147     0.000281    6100       5765      335      6100      5108      992
9148     0.000272    6100       5693      407      6100      4825     1275
9622     0.000211    6100       5444      656      6100      1029     5071
9890     0.000271    6100       5573      527      6100       992     5108
10148    0.000343    6100       5693      407      6100      1030     5070
11382    0.000349    6100       5804      296      6100      4888     1212
-------------------------------------------------------------------------------
```

This report lists those faults whose untested SDQL (Defect Probability) remains above 0.002.

Reporting SDQL Thresholds

```
report_sdql testmode=<tm> delaymodel=<dm> clockconstraints=<ccfile>
reporthistogram=yes
```

The report shows how the potential SDQL (all of the area under the curve except for the timing redundant area) is distributed among the faults. This is useful when determining the simsdql threshold to use when running small delay simulation.

```
INFO (TDL-063): Histogram of Per Fault Untested SDQL (Potential)

Faults w/SDQL   # Faults     Cum.SDQL      SDQL
More Than         (Cum.)       (%)       Histogram
-------------    ----------  ----------  -----------
0.000420           68          21.40     ********************** 21.40%
0.000396           76          23.82     *** 2.42%
0.000358           146         42.86     ******************** 19.04%
0.000302           163         46.98     ***** 4.12%
0.000265           165         47.38     *  0.39%
0.000221           183         50.62     ****  3.25%
0.000176           199         52.78     ***  2.16%
0.000132           222         55.23     ***  2.45%
0.000088           323         63.87     *********  8.64%
0.000044           457         70.60     *******  6.73%
0.000000           6316        100.00

Recommended tgsdql: 0.000044
Recommended simsdql: 0.000044
[end TDL_063]
```

This report shows that 68 faults have a potential SDQL of 0.000420 or greater and comprise 21.40% of the design's total potential SDQL. These are the most critical faults to detect along long paths. In fact, only 457 faults account for 70.6% of the design's total SDQL.

The remaining 5859 faults have a potential SDQL less than 0.000044 and comprise only 29.4% of the total SDQL. Setting `simsdql=0.00044` during small delay simulation will get these faults marked off on the first detect, saving simulation run time.

Specifying the Defect Probability Distribution

```
report_sdql … probfile=<directory>/SDQL.pl …
```

The default defect probability distribution function may be overridden during `report_sdql`. Ideally this function would be provided by the manufacturing facility. The function must be defined in a perl script and stored in a file with the name `SDQL.pl`. It may be located in any directory. The sample function below is provided in the $Install_Dir/etc/contrib directory:

```
sub probfunc {
my $s  = @_[0]; # s is the input value.
my $Fs;         # Fs is the output value.
# Compute the probability distribution function
$Fs = 1.58e-3 * exp(-(2.1*($s))) + 4.94e-6;
return $Fs;
}
```

Creating a Plotfile for graph_sdql

```
report_sdql … experiment=<exp> plotfile=<filename> plotpoints=<nn> …
```

The `report_sdql` command will create a file recording the defect size versus the percentage of defects of that size (tested, untested and total). This file is used as input to the `graph_sdql` command which produces a graphical representation of how defects are distributed among various defect sizes.

Graphing SDQL

Use the `graph_sdql` command to create a graphical representation showing how potential defects are distributed across the various defect sizes. A plot of the total defects, tested defects and untested defects is drawn. One graph is displayed for each clock domain.

Jpeg files are also created for each of the graphs. They are stored in the directory specified in the `jpegdir` keyword (default: $WORKDIR/testresults). If not specified, `jpegdir` defaults to the current directory.

The `graph_sdql` command takes the `plotfile` keyword as input, which is the name of a fully qualified input file containing the numeric data used to plot the test effectiveness graphs. The plotfile is produced as an output of the `report_sdql` command.

Note: Specifying the `plotfile` keyword for `report_sdql` is a prerequisite for running the `graph_sdql` command.

The graphs are stored in jpeg files named as `sdql_domain_`*n*`.jpeg`, where *n* identifies the clock domain number listed in the `report_sdql` output.

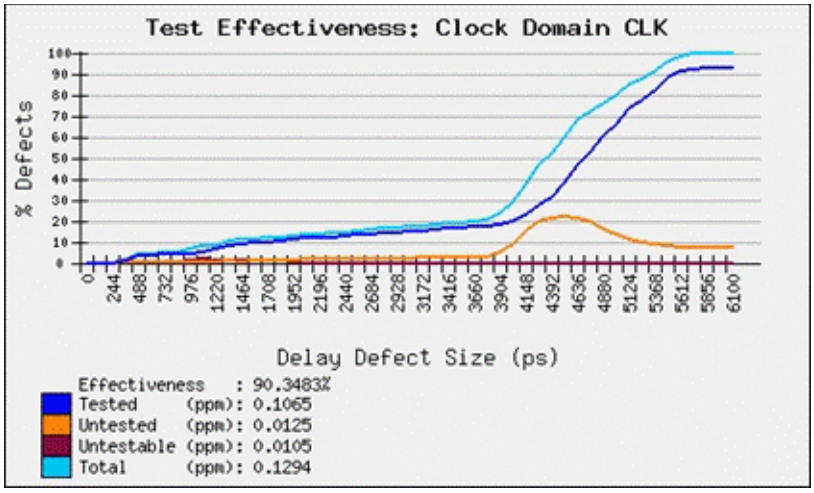
Refer to `graph_sdql` in the *Encounter Test: Reference: Commands* for more information on the command.

A sample usage of the `graph_sdql` command is shown below:

```
graph_sdql plotfile=<infilename> jpegdir=$WORKDIR/testresults
```

A sample SDQL graph is shown in the following figure:

Figure 4-30 SQDL Effectiveness Graph



Committing Tests

By default, Encounter Test makes all test generation runs as uncommitted tests in a test mode. Commit Tests moves the uncommitted test results into the committed vectors test data for a test mode. Refer to “Performing on Uncommitted Tests and Committing Test Data” on page 246 for more information on the test generation processing methodology.

Writing Test Vectors

Encounter Test writes the following vector formats to meet the manufacturing interface requirements of IC manufacturers:

- Standard Test Interface Language (STIL) - an ASCII format standardized by the IEEE.
- Waveform Generation Language (WGL) - an ASCII format from Fluence Technology, Inc.
- Verilog - an ASCII format from Cadence Design Systems., Inc.

Refer to "Writing and Reporting Test Data" on page 181 for more information.

5

Customizing Inputs for ATPG

Various tasks can be performed in preparation for Encounter Test test generation applications. This section describes concepts and techniques related to preparing special input to test applications. Refer to the following:

Linehold File

A linehold file provides a means whereby you can define statements that identify a set of nodes to be treated specially by Encounter Test applications that support lineholding. You may name this file any arbitrary name permitted by the operating system. Although special treatment of lineholds is application-specific, there is a set of global semantic rules common to all Encounter Test applications which govern the specification of linehold information. Anyone generating a linehold file must know the rules and syntax. When applying lineholds to a flop or a net sourced from a flop, the linehold value is not guaranteed to persist through the duration of the test, but will be set at the beginning of the test.

The linehold file also supports changing the frequency of oscillators that are connected to OSC pins and were started by the `Start_Osc` event in the mode initialization sequence.

Use the following linehold statement types to specify this test generation behavior:

- An `OSCILLATE` linehold overrides the frequency of a previously defined oscillating signal or disables the oscillator. `OSCILLATE` statements do not have a direct effect on the test generation or simulation step. However, the information is incorporated into the experiment output and used by processes such as `write_vectors`.
- A `HOLD` linehold is a *hard* linehold. Test patterns can never override this value. The `HOLD` value is always in effect.

- A DEFAULT linehold is a *soft* linehold. Test patterns can override this value if required to generate a test for a fault. Otherwise, the DEFAULT value is in effect.
  - A DEFAULT linehold is automatically generated for any SCAN_ENABLE, CLOCK_ISOLATION, or OUTPUT_INHIBIT test function pin.

The following is an example of a HOLD statement:

```
HOLD PIN "Pin.f.l.newpart.nl.lat1.LATCH_2$3.P01DATA"=1;
```

Create a file with all flops to be lineheld. hold. Specify LINEHOLD=*lineholdfilename* to apply linehold file content during test generation.

Important

During scan, opposite values are returned on these nets creating 0/1 hard contention. The test generator can ensure that the scanned in data will be 0+s for these flops.

Refer to "Linehold File Syntax" on page 173 and "General Semantic Rules" on page 177 for additional information.

The following criteria are used to determine the final linehold set:

- Test mode test function pin specifications. Refer to "Identifying Test Functions Pins in the *Encounter Test: Guide 2: Faults* for related information.
- An input linehold file. The values in this file override test function pin specifications.
- User-defined test sequences. These override the above two criteria. Refer to "Linehold Object - Defining a Test Sequence" on page 178.

Boolean constraints can also be described in linehold files. Refer to "Boolean Constraints" on page 116 for details.

General Lineholding Rules

In general any internal node or pin may be lineheld to a desired value. However, there are some exceptions when dealing with latches. One of the fundamental rules of a linehold set is that it must be possible to apply the desired values in a single load operation. Encounter Test does not support the application of a sequence of patterns with respect to lineholds. Therefore, any latch specifications must be consistent with respect to load (normal or skewed) consistency. Thus, it must be possible to justify internal nodes in a single time-frame.

Clock pins may not be held away from stability. This is the case whether a clock pin is held directly, or the justification of an internal node would require a clock pin held ON. Any test

function pins lineheld to the stability value may be taken out of stability for the scanning operation or to capture the effect of a fault in a latch.

If a linehold is specified on any pin or latch in a correlated set, then all the pins or latches in that set will be lineheld accordingly. It is recommended that you specify linehold values only for the representative pins or representative stim latches.

Whenever a cut point is lineheld (HOLD or DEFAULT), this applies to the associated Pseudo Primary Input (PPI), and thus to all the cut points associated with the PPI at their respective values. See "Linehold File Syntax" on page 173 for additional information.

When specifying lineholds by way of a Linehold object in a `Define_Sequence` statement (user-specified test sequence), each node must be either controllable in the target test mode, or in the case of Line Hold (LH) fixed-value latches, controllable in the parent test mode. Lineholds on other internal nodes are not permitted in Linehold objects. See "Define_Sequence" in the *Encounter Test: Reference: Test Pattern Formats* for related information.

Linehold File Syntax

The following defines the syntax for linehold file statements. See "General Semantic Rules" on page 177 for additional information.

`OSCILLATE` *`entity = [up] [down] [pulsespercycle];`* (using "=" is optional)

where:

- *`entity`* is the pin name optionally preceded by the keyword `PIN`.
- *`up`* is a decimal number specifying the duration in nanoseconds for which the signal is `1` on each oscillator cycle. There is no default value for this parameter and the specified value should be greater than `0`.
- *`down`* is a decimal number specifying the duration in nanoseconds for which the signal is `0` on each oscillator cycle. There is no default value for this parameter and the specified value should be greater than `0`.
- *`pulsespercycle`* is an integer specifying the number of oscillator cycles to be applied in each tester cycle. There is no default value for this parameter and the specified value should be greater than `0`.

Note: To turn off an oscillator for the experiment, specify the `STOPOSC` statement, as shown below:

`STOPOSC` *`entity;`*

Where *entity* is the pin name optionally preceded by the keyword PIN.

HOLD *entity = value* [*restriction*] [*action*]; (using "=" is optional)

where:

- *entity* is one of the following:
  - [PIN] *hier_pin_name*
  - NET *hier_net_name*
  - BLOCK *usage_block_name*
  - PPI *name*
- *value* is one of the following:
  - 0 - logic zero
  - 1 - logic one
  - Z - high impedance
  - X - logic X (unknown)
  - W - weak logic X
  - L - weak logic zero
  - H - weak logic one
- *restriction* is either:
  - all (default) - in effect for the entire sequence
  - dynamic - in effect only for the dynamic section of the sequence
- *action* is either:
  - ignore (default) - if a test pattern voilates a given constraint, then assign an unknown (X) value to the endpoint of the constraint. This improves the runtime performance as the test generation engine needs to protect only those constraints that would otherwise block (with an X) one of the faults it is targeting in the test pattern.
  - remove - consider a test pattern as invalid if a given constraint is violated. The simulator removes all such invalid patterns from the pattern set.

DEFAULT *entity* = *value*; (using "=" is optional)

where:

- *entity* is one of the following:
  - [PIN] *hier_pin_name*
  - NET *hier_net_name*
  - BLOCK *usage_block_name*
  - PPI *ppi_name*
- *value* is one of the following:
  - 0 - logic zero
  - 1 - logic one
  - Z - high impedance
  - X - logic X (unknown)
  - W - weak logic X
  - L - weak logic zero
  - H - weak logic one

RELEASE *entity*;

where:

- *entity* is one of the following:
  - [PIN] *hier_pin_name*
  - PPI *name*

TRANSITION *entity* = *value* [*restriction*] [*action*] [*propagation*]; (using "=" is optional)

where:

- *entity* is one of the following:
  - [PIN] *hier_pin_name*
  - NET *hier_net_name*
  - BLOCK *usage_block_name*
  - PPI *name*

- *value* is one of the following:
  - NONE - no transition permitted
- *restriction* is either:
  - all (default) - in effect for the entire sequence
  - dynamic - in effect only for the dynamic section of the sequence
- *action* is either:
  - ignore (default) - ignore all the scan bits fed by the last propagation entity in the propagation list
  - remove - discard the sequence if a constraint is violated
- *propagation* is one of the following:
  - none (default) - the constraint is considered violated even if not measured at an observe point
  - any - the constraint is considered violated if a value not equal to *value* occurs at the entity and its effects propagate to any observable point in the forward cone. The constraint is violated if *value* is violated and the effects of the transitions are propagated to one of the listed entities.
  - a comma-separated list of observe points as follows:

    entity, *entity*,...

    where each *entity* is one of the following:
    - [PIN] *hier_pin_name*
    - NET *hier_net_name*
    - BLOCK *usage_block_name*

Note: The `Transition` constraint is only checked by the simulator.

The following is a sample linehold file.

```
default in2c = 1 ;
hold in2 = 0 ;
hold in2c = 0 ;
```

where `in2c` and `in2` are pin names.

The syntax rules governing linehold parameter specification are as follows:

- A statement may occupy only a single line.

- Multiple statements per line are permitted (each ending with a semi-colon).
- Only a single *entity* is permitted per statement.
- Only a single *value* is permitted per HOLD or DEFAULT statement.
- Keywords and entity identifiers must be immediately followed by at least one space.
- There is no case sensitivity except for entity names.
- Comments delimited by `'//'` or `'/* */'` may be used freely.

General Semantic Rules

The following are general semantic rules for the specification of *entity*.

- The entity name must be the proper form unless it exists only in the top-level of the hierarchy (i.e., simple names may be specified for primary inputs).
- The entity name is assumed to correspond to a pin if no entity type (pin, block, or net) keyword is specified.
- If the entity type keyword is block, this block may only have a single output pin (to avoid ambiguity).
- The corresponding node must be active in the current test mode.
- The corresponding node should appear only ONCE in the linehold file. If conflicting usages of a node are detected, the last valid statement containing the node takes precedence.
- The corresponding node may NOT be one of the following:
  - Test Inhibit Primary Input, Test Inhibit pseudo primary input (PPI), or Test Inhibit fixed-value latch, or a block, pin, or net controlled by such (to conflicting value).
  - Test Constraint Primary Input, Test Constraint pseudo primary input (PPI), or Test Constraint fixed-value latch, or a block, pin, or net controlled by such (to conflicting value).
  - RAM or ROM
  - output of a chop block (CHOPL, CHOPT, NCHOPL, NCHOPT)
  - tie block
  - load clear latch
  - load preset latch

- floating latch
- latch which is only randomly controllable

Special Rules for the HOLD and DEFAULT statements

The following are special rules associated with *entity*.

- If the associated node corresponds to a controllable latch, the value may only be 0 or 1
- If the associated node is a primary input not contacted by the tester, the value may only be Z or X
- If the value is a weak value (W, L, H), the corresponding net must be sourced by a resistor, nfet, or pfet.
- If the associated node corresponds to a non-controllable point in the design, the justification of this (together with all other held non-controllable points), must be able to be done in a single time-frame. Otherwise this point will not be held.

Special Rules for the RELEASE Statement

The following are special rules associated with *entity*.

The entity must correspond to one of the following:

- A Linehold (LH) flagged PI
- A Linehold (LH) flagged PPI
- A correlated Primary Input whose Representative PI is a Linehold (LH flagged pin).

Special Rules for the OSCILLATE Statement

The *entity* must be a pin that has the `OSC` test function and is referenced by the `Start_Osc` event in the mode initialization sequence.

Linehold Object - Defining a Test Sequence

The following is an example test sequence definition containing a linehold object. Refer to "Lineholds" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

```
TBDpatt_Format (mode=node , model_entity_form=name);
[ Define_Sequence mle1 (test, user_defined);
```

```
 [ Lineholds ():
     Block.f.l.mle1.nl.mle1.slave = 0
     Block.f.l.mle1.nl.mle2.slave = 0
     Block.f.l.mle1.nl.mle3.slave = 0
     Block.f.l.mle1.nl.mle4.slave = 0
     Block.f.l.mle1.nl.mle5.slave = 0
     Block.f.l.mle1.nl.mle6.slave = 0
     Block.f.l.mle1.nl.mle7.slave = 0
     Block.f.l.mle1.nl.mle8.slave = 0
     ;
 ] Lineholds;
[ Pattern 1 ( pattern_type = static );
  . . .
  ] Pattern 1;
] Define_Sequence mle1;
```

The following rules apply to the specification of linehold objects in user-defined test sequences. Lineholds from user-defined test sequences override those from all other sources.

These rules are enforced when importing test sequences.

- No more than one Linehold object may be specified in a `Define_Sequence` statement. Refer to "Define_Sequence" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.
- A Linehold object may be specified only a user sequence of type test. Refer to the definition for test in the "Define_Sequence" section of the *Encounter Test: Reference: Test Pattern Formats*.
- Entries within the Linehold object must be consistent (no conflicting linehold values on a node).
- Entries within the Linehold object must be consistent with any stimulus events supplied with the sequence definition
- The sequence definition must include stimulus events required to apply specified lineholds.
- The node associated with a Linehold object entry must be either:
  - controllable in the target test mode(s) - those the test sequence is intended for.

    or
  - a Line Hold fixed-value latch, controllable in the parent mode of the target mode(s)

The following is an example sequence definition containing linehold information in keyed data. Refer to "Keyed Data" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

```
[ Define_Sequence Test1 1 (test);
  [ Keyed_Data;
    "CONSTRAINTS" = "transition A1 = T01; transition A2 = ANY dynamic ignore A1,
pin A2; equiv + A1, - A2;"
  ] Keyed_Data;
  [ Pattern 1.1 (pattern_type = static);
    Event 1.1.1  Stim_PI ():
"Pin.f.l.tstatesq.nl.D"=0;
  ] Pattern 1.1 ;
] Define_Sequence ;
```

Coding Test Sequences

Introduction to Test Sequences

In the present context, test sequence refers to a template to be used for a set of tests. The template typically specifies certain clock primary inputs to be pulsed in a given order for each test as well as certain non-clock primary input (control pin) stimuli. The template, or test sequence, is imported as a sequence definition and stored in the `TBDseq.testmode` file.

For WRPT and LBIST use, this is the exact sequence of stimuli that eventually get put in the `TBDbin` (Vectors) file output from the test generation applications, and finally get applied at the tester. In a similar fashion, test sequences for stored-pattern tests can also be user-defined. User specification of sequences for stored-pattern test is required for on-product generation or BIST controller support (referred to generically as OPC), but there may be instances other than OPC where it is desired to constrain the test sequences for stored patterns.

For stored patterns, the user-defined test sequence is augmented by the automatically generated input (and scannable latch) vectors for each test. See section "Stored Pattern Processing of User Sequences" on page 183 for a description of this process.

Common to all user-defined test sequence scenarios are the basic coding process and the process of importing the sequence definitions. On the other hand, the detailed contents of the sequence definition depend on its intended use (stored patterns vs. WRPT or LBIST). There are three basic steps in creating a sequence definition:

1. Know the names of the primary inputs, pseudo PIs, and latches or other internal nets that will be referenced in the test sequence.
2. Use the text editor of your choice to code the test sequence in `TBDpatt` format.
3. Import the test sequence to `TBDseq`.

The next subsection tells you how to get started. This is followed by an explanation of sequence coding for stored patterns, then an explanation of sequence coding for WRPT or LBIST, and then a brief explanation of the import process.

Getting Started with Test Sequences

Sequence definitions are coded in the format of `TBDpatt`, which is the ASCII version of `TBDbin` and `TBDseq`. You will need to know the names of the primary inputs and internal blocks that you will be setting values on. Mostly these are clocks; you may want to specify explicit static values on certain other inputs too. If you are not sure of the exact names by which Encounter Test recognizes the pins, then you can copy the names from the vector correspondence file (`TBDvect.`*`testmode`*). This file can be created any time after the design is imported and the test mode is built by running Create Vector Correspondence from the *Tools* pull-down. If the vector correspondence file already exists, then it is not necessary to rebuild it. If you choose to use the vector correspondence file, then you can copy the pertinent names directly from it into the `TBDpatt` file that you are creating.

The sequences are coded in an ASCII file, which is created using the text editor of your choice. Of course, if you can find an existing test sequence, it is usually quicker to copy and edit it instead of entering one from scratch. The name of the file is arbitrary, as long as it does not conflict with the name of some other file in the directory where it is located. *Encounter Test: Reference: Test Pattern Formats* contains detailed explanations of the `TBDpatt` format.

A `TBDpatt` file always starts with a `TBDpatt_Format` statement, which gives information about the formatting options that are used. There are two `TBDpatt` format modes, denoted by `mode=node` or `mode=vector`. In writing sequence definitions, you usually have no need to specify values on many inputs or latches all at once, so all examples in this section use `mode=node`. `Mode=node` is a list format that specifies only the pins that are to be set, and the value of each.

`TBDpatt` statements that refer to specific pins, blocks, or nets can use either their names or their indexes to identify them. The second `TBDpatt_Format` parameter is `model_entity_form`, which specifies either name, index, or `flat_index`.

Comments can be placed in the `TBDpatt` file to record any explanatory information desired. Comments can be placed in any white space in the file, but whatever follows the comment must begin on a new line. Comments are not copied into the `TBDseq` and `TBDbin` (Vector) files. Refer to "TBDpatt and TBDseqPatt Format" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

Following the `TBDpatt_Format` statement can be any number of sequence definitions. Each sequence definition starts with a `[Define_Sequence` statement and ends with a

]Define_Sequence statement. A sequence contains patterns which contain events. The events specify exactly what is to be done and what it is to be done to, for example: apply a negative-going pulse on clock pin Aclk (Pulse), set primary input DATA5 to 1 (Stim_PI), or scan out and look for a 0 in latch XYZ (Scan_Unload). The grouping of events into patterns should coincide with tester cycles; that is, each pattern contains the events to be applied in one tester cycle.

The events that go inside each pattern depend upon the type of test the sequence definition will be used for.

Stored Pattern Test Sequences

A typical test sequence for logic that conforms to the Encounter Test scan (LSSD or GSD) guidelines starts by loading the scannable latches, then applies the primary input test vector, measures primary outputs, then pulses a “system” clock, and then unloads the values captured in the scannable latches by the system clock pulse. Figure 5-1shows this simple test in the form of Encounter Test's TBDpatt.

Figure 5-1 A Simple Typical Stored-Pattern Test Sequence Definition

```
 TBDpatt_Format (mode=node,model_entity_form=name);
[ Define_Sequence spseq1 (test);
[ Pattern 1;
    Event 1.1.1   Scan_Load (): ;
] Pattern 1;
[ Pattern 2;
    Event 1.2.1   Put_Stim_PI (): ;

] Pattern 2;
[ Pattern 3;
    Event 1.3.1   Pulse (): "Pin.f.l.samplckt.nl.C1"=-;
    Event 1.2.2   Measure_PO (): ;
] Pattern 3;
[ Pattern 4;
    Event 1.4.1   Scan_Unload (): ;
] Pattern 4;
] Define_Sequence ;
```

In the example of Figure 5-1, “spseq1” is an arbitrary sequence name assigned by the user. “(test)” denotes this as a test sequence, as opposed to a modeinit sequence or a scan operation sequence. The numbers following the keywords “Pattern” and “Event” are arbitrary

identification numbers; they are shown in the example exactly as Encounter Test would construct them when exporting the sequence definition to an ASCII file.
"`Pin.f.l.samplckt.n1.C1`" is the fully-qualified name by which Encounter Test knows the C1 clock primary input pin. The short form of the name, just "C1" could be used in place of the longer name shown. The "-" following the clock name signifies a negative-going pulse (the quiescent state of this clock is a 1).

"`Scan_Load`" and "`Scan_Unload`" are the names of events which represent the operations of loading and unloading the scan chains, respectively. Note that these particular events may not be imported during test mode creation. If the sequence includes these events, create the test mode first, then import the sequence definition.

No latch state vectors are specified, because as test generation is performed using this sequence definition, Encounter Test will fill in these values. "`Put_Stim_PI`" is a placeholder for another event, called "`Stim_PI`". `Stim_PIs` are the events that hold the primary input state vectors for the test. The `Put_Stim_PI` event was devised specifically for use in a test sequence definition, and is not used any other place. In the context of a test sequence definition, `Put_Stim_PI` is a placeholder for the generated primary input vector, while `Stim_PI` is used only for manipulating control primary inputs and cannot be modified by the test pattern generator. (In this simple example, no such control signals are used, therefore no Stim PI events are shown). When the test vector is inserted into the sequence, the `Put_Stim_PI` event is changed to a `Stim_PI` event. Note that there is no such distinction made for the latch vectors because by its nature a `Scan_Load` event is not suitable for applying control signals to a design, so its appearance in a sequence definition always denotes where the latch test vector is to be placed.

For both the `Put_Stim_PI` and the `Scan_Load` events, PI or latch values may be specified. When any PI or latch value is thus specified, it takes precedence over any value specified on that PI or latch by the test generator; the automatically generated vectors are used only to fill in the unspecified positions of the vectors in the `Put_Stim_PI` and `Scan_Load` events.

Note: When writing sequences for an OPMISR test mode, each OPMISR test sequence should begin with a `scan_load` event and end with a `channel_scan` event.

When writing sequences for a test mode which has scan control pipelines, each test sequence must begin with a `scan_load` event and end with a either a `scan_unload` or `channel_scan` event.

Stored Pattern Processing of User Sequences

The `testsequence` option specifies which testsequence(s) the test generator can use for creating tests for faults and how the user-defined test sequence(s) are used by the ATPG process.

The first step is to write the test sequence template(s) be used by ATPG. The sequence definition is then imported into the test mode. Note that when non-scan flush sequences (data pipes) are applicable, then the non-scan flush patterns should be included in the user-defined test sequence (these patterns should include the non-scan flush attribute i.e. `nonscanflush`).

Processing via Test Sequence

The `testsequence` keyword is used to specify a list of imported test sequences which are available to ATPG for fault detection. Test generation progresses as usual with the tests being created for faults independent of any user-defined test sequences. A *best fit* process is applied to match the ATPG sequence to user-defined test sequence specified via `testsequence`. The `testsequence` method provides special event types (for example, Put_Stim_PI and keyed data) to allow variability in the test sequence definition and test sequence matching. A description of the sequence matching process and guidelines is described later in this chapter. Note that it is possible to have ATPG generate tests which are discarded due to the ATPG test sequence not fitting with any specified user-defined sequences.

Tip

Test Sequence processing can be invoked from the GUI as follows:

a. Click the *Advanced* button on a test generation form

b. Click the *General* tab

c. Select *Use manual sequences*

d. Select *To fit clocking sequences*

e. Enter a range of test sequences in the *Sequence name* field or enter a file that contains a range of test sequences in the *Read sequence from file* field.

Mapping Automatic Vectors into User Sequences

When user sequences are specified for stored pattern tests, Encounter Test takes the automatically generated test vectors (the primary input and scannable latch values) and places these vectors into the user-supplied sequence. The resulting test is then simulated and written to the `TBDbin` output file for eventual use at the tester. This seemingly simple process is made more complicated by the fact that sometimes the automatic test consists of a sequence of vectors, as in the case of a partial scan design or a design containing embedded RAM.

Each primary input vector and each latch vector must be inserted some place in the user-defined sequence. This has greater significance in the specification of user test sequences when the automatic test sequence contains more than one primary input vector or more than one latch vector. Latch vectors are inserted in the user-defined sequence at each `Scan_Load` (or `Skewed_Scan_Load`) event. If the automatic test sequence contains the same number of these events as the user sequence, then the first latch vector found is copied into the first `Scan_Load` (or `Skewed_Scan_Load`) event of the user sequence, the second is copied into the second stim latch event of the user sequence, and so on. If the number of stim latch events in the user sequence is more or less than the number of stim latch events (latch vectors) in the automatic sequence, then Encounter Test cannot map this automatic test into the user-defined sequence. Either some latch vectors from the automatic sequence would have to be discarded, or some stim latch events from the user sequence would have to be discarded. Instead of doing that, Encounter Test discards the entire automatic sequence, and it is not used at all.

With primary input vectors, the mapping is from `Stim_PI` events in the automatic sequence to `Put_Stim_PI` events in the user-defined sequence. `Stim_PI` events in the user-defined sequence (along with all other events except `Scan_Load`, `Skewed_Scan_Load`, and `Put_Stim_PI`) are kept exactly as specified by the user. Thus, the user sequence can stim a control primary input at any appropriate point in the sequence without any consideration of whether the test generator will be generating a corresponding primary input vector.

The same considerations explained above for latch vectors apply equally to primary input vectors. For each primary input vector (`Stim_PI` event) of the automatic test sequence there must be a corresponding `Put_Stim_PI` event in the user-defined sequence to map the vector into. If the number of `Stim_PI` events in the automatic sequence does not exactly match the number of `Put_Stim_PI` events in the user sequence, then the mapping fails and the automatic test is discarded.

Choosing from a List of User Sequences

For most designs, the automatic test generator will produce tests which vary with respect to the number of primary input and latch vectors contained therein. For a full-scan design having no embedded RAM, most tests will contain a single latch vector and a single primary input vector, but when non-scan sequential elements are present, the test generator will produce some tests with multiple input vectors to be applied in sequence with other activity such as clock pulses.

If your test protocol can support a variety of test templates, and especially if the test generator will produce tests with variable numbers of latch or primary input vectors, it is important to introduce some flexibility to accommodate this variety of test templates. This flexibility is supported in two ways. One way is to define each different sequence, or template, as a unique test sequence and provide a list of test sequences to the test generation run.

When multiple test sequences are specified for a stored-pattern test generation run, each generated test will be mapped, if possible, into one of the specified sequences. Encounter Test has a two-tiered algorithm for choosing which user sequence to map the test into.

The first step of the algorithm determines whether each user sequence has the requisite numbers of latch stims and `Put_Stim_PI` events, in accordance with the criteria explained in the preceding section, "Mapping Automatic Vectors into User Sequences" on page 184. Only those sequences which "match" the automatic test with respect to the number of latch stims and the number of primary input stims (`Put_Stim_PI` vs. `Stim_PI`) are considered in the second step.

The second step of the matching algorithm selects from those user sequences which can be mapped into, the one which most closely resembles the automatic test, considering all other factors, such as which clocks are being pulsed, the order of the clock pulses, type of scan (skewed load, skewed unload), number and placement of `Measure_PO` events, etc. In this selection, a high weight is given to the comparison of the measure latch events between the user sequence and the test sequence. If the test generator produced a `Skewed_Scan_Unload` event for an LSSD design, it is reasonable to assume that it has stored some meaningful test information in the L1 latches; if the user sequence has a `Scan_Unload` event (no preceding B clock) then the scan out will begin with a shift A clock, destroying the test information in the L1 latches. In this case, a user sequence with a `Skewed_Scan_Unload` event would get a much better "score" for matching than a user sequence with a `Scan_Unload` event.

Control Statements in Sequence Definitions

Besides allowing multiple user sequences to be specified for a test generation run, Encounter Test has additional flexibility in the sequence matching by means of "keyed data" statements that can be placed in the sequence definition. See "Keyed Data" in the *Encounter Test: Reference: Test Pattern Formats* for additional information.

There are three unique types of sequence matching control statements, and one of these has two variations.

STIM=DELETE

This statement is placed on a `Scan_Load`, `Skewed_Scan_Load`, or `Put_Stim_PI` statement in a sequence definition and causes the vector to be removed from the sequence. The purpose of such a statement is to allow the sequence to be matched with an automatic test and yet eliminate the event from the sequence. This is best explained by an example.

In the hypothetical example of Figure 5-2, there is a choice of two system clocks, C2 and C3. The automatic test generator picks clock C2 but the user wants to use C3. The C2 clock signal

is gated by a control primary input. If we assume that the test generator is sophisticated enough to figure this out and place the gating signal in a separate `Stim_PI` event prior to pulsing the C2 clock, the automatic test could have two `Stim_PI` events. The sequence matching and mapping algorithms count each `Stim_PI` event in the automatic test as a separate vector, and hence the user sequence definition must have two `Put_Stim_PI` events. But let us say the user wants to leave the C2 clock gating primary input at its original state from the first vector. This can be accomplished as shown in the figure, using the second `Put_Stim_PI` event to “match” the second `Stim_PI` event in the automatic sequence, and the `STIM=DELETE` statement to throw away this “vector” after the sequence matching has been done.

Figure 5-2 Illustration of the STIM=DELETE control statement

```
[ Test_Sequence ;
 [ Pattern 1;
   Event 1   Stim_PI ():    1011001010001;
   Event 2   Measure_PO (): ;
 ] Pattern 1;
 [ Pattern 2;
   Event 1   Stim_PI ():    .....1.......;
   Event 2   Pulse ():      "C2"=-;
   Event 3   Measure_PO (): ;
 ] Pattern 2;
] Test_Sequence ;


[ Define_Sequence Jones (test) ;
 [ Pattern 1;
   Event 1   Put_Stim_PI (): ;
   Event 2   Measure_PO (): ;
 ] Pattern 1;
 [ Pattern 2;
   Event 1   Put_Stim_PI (): ;
   [ Keyed_Data ;
     STIM=DELETE
   ] Keyed_Data ;
   Event 2   Pulse ():      "C3"=-;
   Event 3   Measure_PO (): ;
 ] Pattern 2;
] Define_Sequence Jones ;
```

`STIM=DELETE` statements are valid on `Put_Stim_PI`, `Scan_Load`, and `Skewed_Scan_Load` events, and are coded as shown in the example of Figure 5-2. As has been explained, `STIM=DELETE` is used to cause the event to be deleted from the sequence after the event has been used for purposes of sequence matching.

PI_STIMS/LATCH_STIMS=n

This is the second type of sequence matching control statement, and there are two statements of this type:

```
PI_STIMS=n
LATCH_STIMS=n
```

where `n` is a positive integer.

These statements are interpreted and applied before sequence matching takes place. Their primary purpose is to allow the same user test sequence definition to be “matched” with different automatic tests that contain different numbers of primary input and/or latch vectors. For example, an event to which a `PI_STIMS=3` statement is attached is effectively removed from the sequence definition if the automatic test sequence does not have at least three `STIM_PI` events. Similarly, an event with an attached `LATCH_STIMS=2` event is effectively removed from the sequence definition if the automatic sequence does not have at least two `Scan_Load` and/or `Skewed_Scan_Load` events.

Again using a hypothetical example, Figure 5-2 shows a possible use for the `PI_STIMS=n` statement. The user's sequence definition “matches” both automatic sequences 1 and 2, and so can be used for both of these automatic tests. In the case of test sequence 1, the sequence contains one `Stim_PI` event, so the second `Put_Stim_PI` event of the sequence definition is removed for this usage as it specifies `PI_STIMS=2`. With the second Put_Stim_PI event removed, the number of `Put_Stim_PI` events remaining (one) matches the number of primary input vectors in the first test sequence.

In the case of test sequence 2, the sequence contains two `Stim_PI` events, so the `Put_Stim_PI` event with the `PI_STIMS=2` statement is used for the purpose of matching with the automatic test sequence. The number of `Put_Stim_PI` events (two) matches the number of primary input vectors in the second test sequence.

Figure 5-3 Illustration of the PI_STIMS=n control statement

```
[ Test_Sequence 1 ;
  [ Pattern 1;
    Event 1   Stim_PI ():    1001001000011;
    Event 2   Measure_PO (): ;
  ] Pattern 1;
] Test_Sequence 1 ;

[ Test_Sequence 2 ;
  [ Pattern 1;
    Event 1   Stim_PI ():    0100101011100;
    Event 2   Measure_PO (): ;
  ] Pattern 1;
  [ Pattern 2;
    Event 1   Stim_PI ():    .....1.......;
    Event 2   Pulse ():      "C2"=-;
    Event 3   Measure_PO (): ;
  ] Pattern 2;
] Test_Sequence 2 ;


[ Define_Sequence Hamilton (test) ;
  [ Pattern 1;
    Event 1   Put_Stim_PI (): ;
    Event 2   Measure_PO (): ;
  ] Pattern 1;
  [ Pattern 2;
    Event 1   Put_Stim_PI (): ;
    [ Keyed_Data ;
      PI_STIMS=2
      STIM=DELETE
    ] Keyed_Data ;
    Event 2   Pulse ():      "C3"=-;
    Event 3   Measure_PO (): ;
  ] Pattern 2;
  ] Define_Sequence Hamilton ;
```

In the example of Figure 5-3, the conditional `Put_Stim_PI` event is discarded after sequence matching, but the conditional (i.e., the `PI_STIMS=n` statement) is useful also in cases where the vector is to be kept, as illustrated in the next example.

The `PI_STIMS=n` and `LATCH_STIMS=n` statements can be attached to any event, and are not limited to `Stim_PI` and stim latch events. In the example of Figure 5-4, an entire chunk of events consisting of a stim, a measure and a pulse is made conditional based upon whether the automatic test sequence contains one, two, or three `Stim_PI` events.

Figure 5-4 Illustration of the PI_STIMS=n control statement

```
        [ Define_Sequence McPhee (test);
         [ Pattern 1 (pattern_type = static);
           Event 1.1   Scan_Load (): ;
         ] Pattern 1;
         [ Pattern 2 (pattern_type = static);
           Event 2.1   Put_Stim_PI (): ;
           [ Keyed_Data ;
             PI_Stims=3
           ] Keyed_Data ;
           Event 2.2   Measure_PO (): ;
           [ Keyed_Data ;
             PI_STIMS=3
           ] Keyed_Data ;
         ] Pattern 2;
         [ Pattern 3 (pattern_type = static);
           Event 3.1   Pulse ():
"Pin.f.l.ptnmatch.nl.C4"=+ ;
           [ Keyed_Data ;
             PI_STIMS=3
           ] Keyed_Data ;
       ] Pattern 3;
       [ Pattern 4 (pattern_type = static);
          Event 4.1   Put_Stim_PI (): ;
          [ Keyed_Data ;
            PI_Stims=2
          ] Keyed_Data ;
          Event 4.2   Measure_PO (): ;
          [ Keyed_Data ;
            PI_STIMS=2
          ] Keyed_Data ;
       ] Pattern 4;
       [ Pattern 5 (pattern_type = static);
            Event 5.1   Pulse ():
"Pin.f.l.ptnmatch.nl.C5"=+ ;
            [ Keyed_Data ;
              PI_STIMS=2
            ] Keyed_Data ;
       ] Pattern 5;
            [ Pattern 6 (pattern_type = static);
         Event 6.1   Put_Stim_PI (): ;
         Event 6.2   Measure_PO (): ;
       ] Pattern 6;
       [ Pattern 7 (pattern_type = static);
         Event 7.1   Pulse ():
"Pin.f.l.ptnmatch.nl.C6"=+ ;
       ] Pattern 7;
       [ Pattern 8 (pattern_type = static);
         Event 8.1   Skewed_Scan_Unload (): ;
       ] Pattern 8;
     ] Define_Sequence McPhee ;
```

TG=keyed data

This is another statement to control which pattern will be ignored by the test generator, and there are three statements of this type:

```
TG=IGNORE
TG=IGNORE_FIRST
TG=IGNORE_LAST
```

The statement `TG=IGNORE` is used for a single pattern, while `TG=IGNORE_FIRST` and `TG=IGNORE_LAST` statements allow multiple patterns to be ignored. To ignore multiple patterns, specify `TG=IGNORE_FIRST` for the first pattern to be ignored, add other patterns after this pattern, and specify `TG=IGNORE_LAST` for the last pattern to be ignored. All the patterns starting from the one with `TG=IGNORE_FIRST` through the one with `TG=IGNORE_LAST` will be ignored by the test generator.

Using a hypothetical example, the following snippet shows the possible use of the `TG=IGNORE` statement in a user sequence:

```
[ Define_Sequence Jones (test);
    [ Pattern 1;
      Event  1 Scan_Load :
   ] Pattern 1;
   [ Pattern 2;
      Event  1 Put_Stim_PI ():  ;
   ] Pattern 2;
   [ Pattern 3;
      [Keyed_Data ;
          TG=IGNORE
      ] Keyed_Data ;
      Event 1 Pulse () : "TCK"=-;
  ] Pattern 3;
  [ Pattern 4;
      Event 1  Pulse  () : "C3"=-;
      Event 2 Measure_PO () : ;
   ] Pattern 4;
] Define_Sequence Jones ;
```

The test generator discards Event 1 in Pattern3 because it has the `TG=IGNORE` statement specified for it and considers the user sequence as follows:

```
[ Define_Sequence Jones (test) ;
   [ Pattern 1;
      Event  1 Scan_Load :
   ] Pattern 1;
```

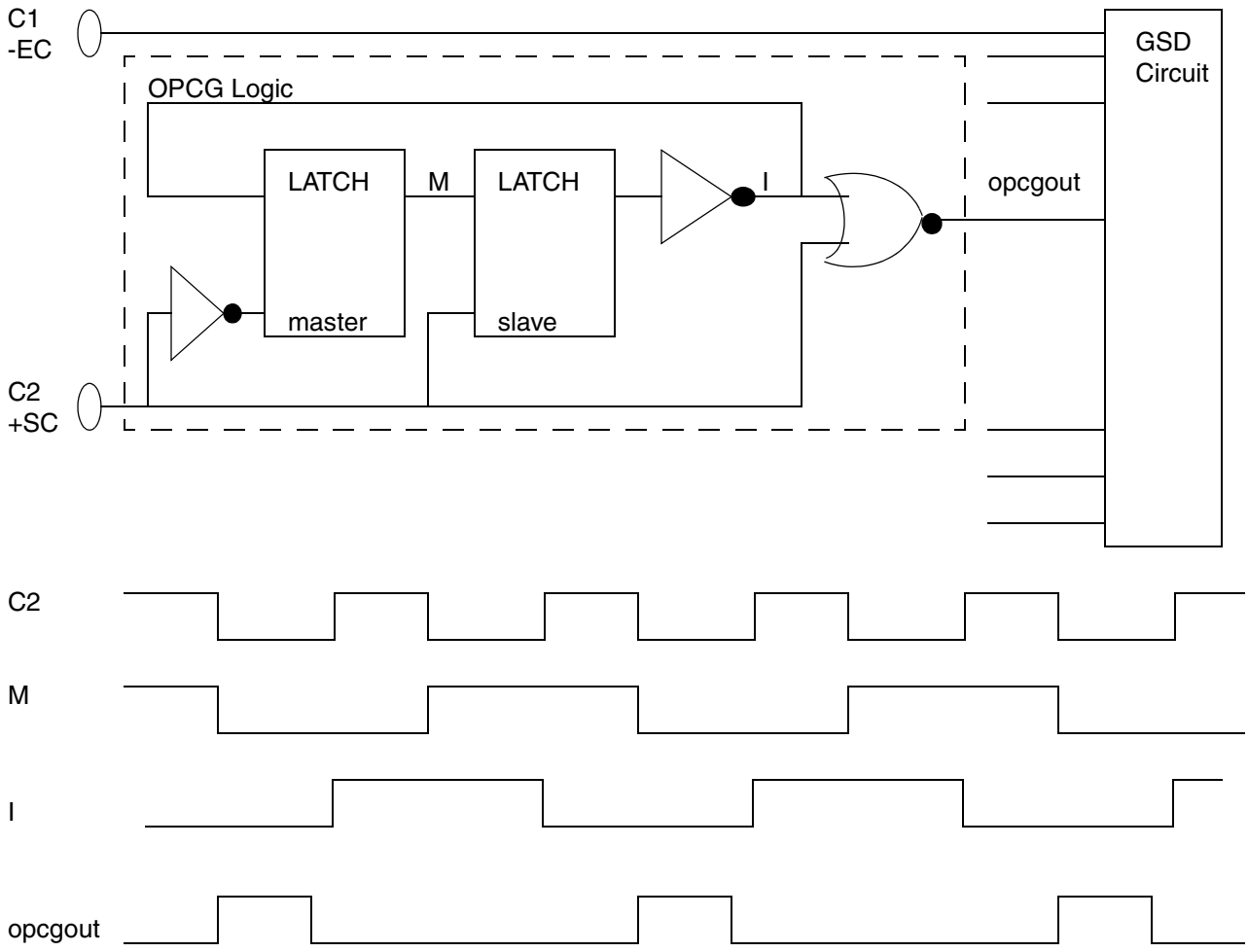
```
   [ Pattern 2;
      Event  1 Put_Stim_PI ():  ;
   ] Pattern 2;
   [ Pattern 4;
      Event 1  Pulse  () : "C3"=-;
      Event 2 Measure_PO () : ;
   ] Pattern 4;
] Define_Sequence Jones ;
```

Sequences with On-Product Clock Generation

There are some unique considerations involved when writing a test sequence if you have sectioned off some logic, such as clock generation or BIST control, with cut points. Although WRPT and BIST may be applied to a design with OPC logic (cut points), the treatment of pseudo primary inputs is explained in terms of a stored-pattern test sequence definition.

The following figure shows a design with clock generated by OPC logic.

Figure 5-5 GSD Circuit with Clock Generated by OPC Logic



An example of a static stored-pattern test sequence definition for the design of Figure 5-5 is shown in Figure 5-6. For ease of reference the standard Encounter Test numbering convention for patterns and events has not been followed here. Test data import ignores them anyway, so the example is valid.

Figure 5-6 A stored-pattern test sequence definition for a design with OPC logic

```
        TBDpatt_Format (mode=node,model_entity_form=name);
        # Mode Test sequence for opcgckt
        # Created by TDA   11/08/96
        # Normal Load and Unload
        [ Define_Sequence TDA_opcgckt_test1 (test);
        [ Pattern 1;
        Event 1   Force ():
        "Block.f.l.opcgckt.nl.opcg1.master"=0 ;
        ] Pattern 1;
        [ Pattern 2;
        Event 2   Scan_Load (): ;
        ] Pattern 2;
        [ Pattern 3;
Event 3   Put_Stim_PI (): ;
Event 4   Measure_PO (): ;
] Pattern 3;
[ Pattern 4;
Event 5   Pulse ():
"Pin.f.l.opcgckt.nl.C2"=-;
] Pattern 4;
[ Pattern 5;
Event 6   Pulse ():
"Pin.f.l.opcgckt.nl.C2"=-;
Event 7   Pulse_Pseudo_PI ():
"opcgoutPPI"=+ ;
] Pattern 5;
[ Pattern 6;
Event 8   Scan_Unload (): ;
] Pattern 6;
] Define_Sequence TDA_opcgckt_test1;
```

Event 1 is a Force event which tells a simulator that the given net (usually a latch, as in the present case) should be at the specified state. This event is needed here because most simulators start out by assuming an unknown initial state in all latches, and this example design does not have a homing sequence. In the present case, the lack of a homing sequence is not a concern, since the static behavior of the design is the same no matter which state the frequency divider starts in. The initial state affects the phase relationship between the input clock pulses and the derived clock pulses on net opcgout. For static logic tests this does not matter. No Force event is needed to initialize the other OPCG logic latch, because it is flushed in the stability state and therefore will obtain its initial value through normal simulation.

The Force event could have been placed in a setup sequence, but since no setup sequence was defined, it is more convenient, and does no harm to put it inside the test sequence as shown.

`Scan_Load` (Event 2) is usually the first event in an automatic test sequence for a scan design. If you have an LSSD design with A and B scan clocks, you will do well to define a second sequence identical to this one except with a `Skewed_Scan_Load` in place of this `Scan_Load` event, and supply both sequences to the test generator. Encounter Test will use

the sequence definition that most closely matches the generated test. The latch values from the generated sequence are moved into the event in the sequence definition; if the `Scan_Load` types do not match, the test coverage could be adversely affected.

Events 2 and 3 together comprise what is commonly thought of as the “test vector,” the latch and primary input values required for the test. Event 3 is actually a placeholder telling Encounter Test where to put the `Stim_PI` portion of the test vector. Besides acting as a placeholder, the `Put_Stim_PI` event allows the specification of primary input values that will override any values specified by the test generator. When the test sequence is constructed from this sequence definition and the generated test vector, the `Put_Stim_PI` event will be replaced by the `Stim_PI` event.

Event 4 is an instruction to measure the primary outputs. When creating its own test sequences, Encounter Test has algorithms that determine the sequential relationship between clocks, primary input stims, primary output measures, and scan operations. However, when the user specifies the sequence, Encounter Test must be told where the stims and measures go.

Events 5 and 6 produce two cycles of the clock primary input, which cause the frequency divider (the OPC logic in this design) to produce a single pulse on the derived clock, opcgout, which is identified as a “cut point” in the design source or mode definition file.

Event 7 is the pulse on the derived clock, which is referred to by the name of its corresponding pseudo PI. For this example the name opcgoutPPI was chosen. This event is used when verifying the sequence, and for Encounter Test simulation of the tests. Note that the automatically generated sequences will be in terms of the pseudo PIs, so this event is also used for sequence matching when multiple sequence definitions are given to the test generation application.

Event 8, the final step in this sequence, tells Encounter Test to scan out the latch values. As with the `Scan_Load` event, when you are using LSSD with A and B scan clocks, there are two forms of this event, the other one being `Skewed_Scan_Unload`. The choice of which measure latch event to use is related to the clocking, so if the wrong event is used, the test coverage is likely to be dramatically reduced. As with the `Scan_Load` event (see the discussion of Event 2 above), you may want to define two sequences, one with each measure latch event. Combined with the two different stim latch events, you will often have four similar sequence definitions for an LSSD design. In contrast, a single clock edge-triggered design may require only a single static test sequence for good fault coverage.

Setup Sequences

Encounter Test automatically creates a setup sequence for WRPT and LBIST to hold the weight information (for WRPT), to apply the optional initializing channel scan from PRPG (for LBIST), and to apply any lineheld primary input values that are specified.

If you code your own setup sequence, Encounter Test will augment it with the above information if needed. You would need to code a setup sequence only if there are additional stimuli that need to be applied, or some of the “lineholds” are to pseudo primary inputs. Neither of these reasons is likely to exist unless you are processing in a test mode with cut points specified.

It should be helpful to think of the setup sequence as an initialization step for the WRPT or LBIST test loop.

In some cases of processing with cut points, especially with an on-board BIST controller, it may also be necessary to code a pseudo PI event (`Stim_PPI`) in the setup sequence just to remind Encounter Test that the corresponding cut point nets were initialized to that state in the `modeinit` sequence. For example, it is possible, especially with an on-product BIST controller, that some control signal represented as a PPI is set in the mode initialization sequence and then left in that state to begin the test sequence. If this PPI does not have a stability value (that is, it is not attributed as a clock, TI, nor TC), Encounter Test will assume the test sequence starts with the PPI at X unless told otherwise.

When a setup sequence is used, it is defined as type `setup` and then referenced in the test sequence definition by the following construct:

```
[ Define_Sequence setup_seq_name (setup) ;
        .
        .
        .
] Define_Sequence setup_seq_name;
[ Define_Sequence test_seq_name (test) ;
  [ SetupSeq=setup_seq_name ];
        .
        .
        .
```

If you import the setup sequence definition, but do not refer to it by the `SetupSeq` object within a test sequence, Encounter Test will ignore it. The `SetupSeq` object is required within the test sequence to tell Encounter Test to use the named sequence definition as the base for the setup sequence that will precede the loop test sequence in the test generation output file for WRPT or LBIST.

Endup Sequences

At the end of a BIST test procedure, it may be necessary to de-activate a phase-locked-loop or cycle the BIST controller into some special state, or quiesce the design in some other fashion. If this is the case, you may be able to provide the stimuli to do this in the signature observation sequence. On the other hand, you may choose to separate this operation from the MISR observation sequence, or it may be necessary to apply this quiescing sequence between two concatenated test sequence loops, without an intervening signature observation.

The quiescing, or `endup` sequence can be coded as a separate sequence definition and imported along with your test sequence definitions. Encounter Test will use the `endup` sequence only if so directed by the presence of an `EndupSeq` object within a test sequence definition. This is specified in a similar manner as a setup sequence:

```
[ Define_Sequence endup_seq_name (endup) ;
    .
    .
    .
] Define_Sequence endup_seq_name;
[ Define_Sequence test_seq_name (test) ;
  [ EndupSeq=endup_seq_name ];
    .
    .
    .
```

This object appears before the first pattern of the test sequence definition, but it gets applied at the end of the test sequence by means of an `Apply:endup_seq_name;` event that Encounter Test will automatically insert after the loop.

Specifying Linehold Information in a Test Sequence

Linehold information is usually supplied to a test generation run through a linehold file, specified as one of the inputs to the run. There are some cases where it is important that a given test sequence always be used with some subset of invariant lineholds. A case in point is linehold overrides of a fixed-value latch which is part of an LBIST controller or clock generation logic. In this case, the latch is separated from the logic under test by cut points. Assuming this lineheld fixed-value latch influences the value of the cut point nets, the sequence verification program must be run with the same value in this latch as will be specified when the sequence is used in test generation. To ensure that the test generation run time override (linehold) value of this latch is the same as the value assumed by the sequence verifier, the linehold for such a latch should be placed inside the test sequence.

We note that in the case of a “lineheld” primary input, the value can be specified directly by means of a `Stim_PI` event in the sequence definition. Also, lineholds are specified because the verification and the test generation/simulation are performed in two separate steps, and

the need for the verification step is unique to the use of cut points and pseudo primary inputs. Therefore, it is recommended that the use linehold specification within a test sequence definition be limited to fixed value latches in OPC logic, but Encounter Test places no such restriction on its use, in case the user should have some reason to make more extensive use of it.

The linehold object is placed before the first pattern in the sequence definition, with the following `TBDpatt` syntax:

```
[ Define_Sequence Example_Name (test);
  [ Lineholds;
    Netname_A=1;
    Netname_B=0;
  ] Lineholds;
  [ Pattern ;
       .
       .
       .
```

Lineholds specified in this manner through a sequence definition are limited to primary inputs and FLH (fixed-value) latches.

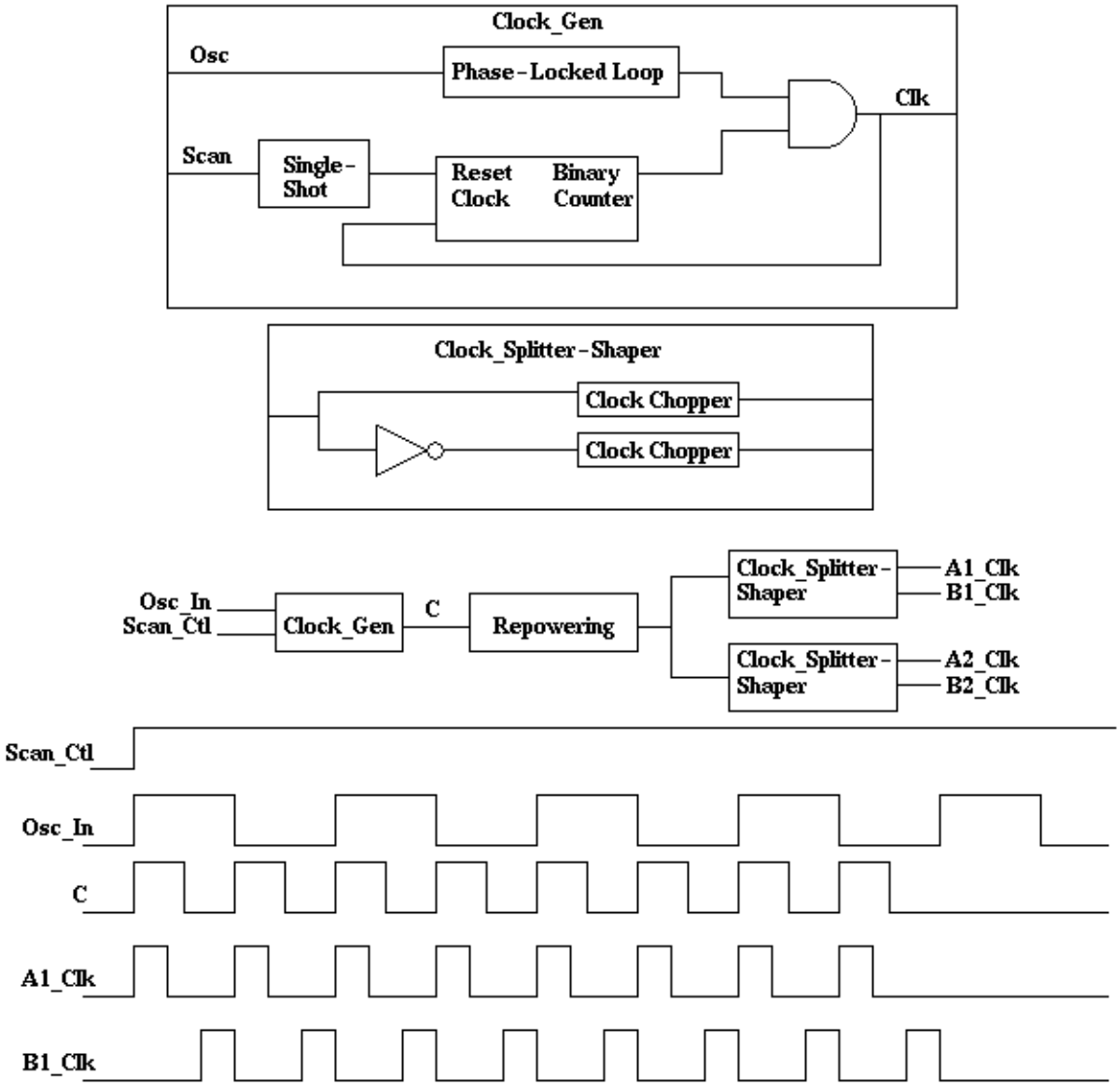
Lineholds specified in a sequence definition override any conflicting linehold specified in the model source, test mode definition, or linehold file.

This linehold object is valid in a test sequence only, not in the setup sequence.

Using Oscillator Pins in a Sequence

This will be explained by an example where the scan operation is under control of a free-running oscillator. We assume an LBIST design, so that no external signals have to be applied during the scan operation other than the oscillator waveform. The following clock generation design is assumed:

Figure 5-7 Clock Generation Logic



The scan sequence definitions, which must be provided manually, are shown in Figure 5-8 on page 203.

The cut point nets, A1_Clk, A2_Clk, B1_Clk, and B2_Clk of Figure 5-7 are defined by either the model source or mode definition statements as pseudo primary inputs `A_Clk` and `B_Clk`.

Thus, the sequence definitions in Figure 5-8 make reference to these PPI (pseudo primary input) names.

A shift gate signal named `Scan_Enable`, with a test function attribute of -SE, is assumed to be gating the scan data path. This signal is not part of the clock circuitry of Figure 5-7.

The free-running oscillator, "`Osc_In`" in Figure 5-7 , is assumed to have a test function of oTI, indicating that it is "tied" to an oscillator. The mode initialization sequence must have applied a `Start_Osc` event on this pin. Synchronization of the events in this sequence definition with the oscillator is by means of `Wait_Osc` events.

The `Wait_Osc` event specifies how many oscillator cycles must have elapsed since the previous `Wait_Osc` event. In our example, when the scan operation is not being executed, the clock generation circuitry of Figure 5-7 is dormant, and has no effect upon the application of the test patterns being applied between the scan operations. Thus, there need be no `Wait_Osc` events in that portion of the test data between scan operations.

When the scan operation is started, the `Scan_Enable` signal is first set to its scan state of 0. Then, in preparation for turning on the `Scan_Ctl` signal to initiate the scan operation, a `Wait_Osc` (Cycles=0) event is specified. This tells us that we must start paying attention to the oscillator. "Paying attention" means simply to start counting oscillator pulses so that subsequent events can be synchronized with the oscillator. Subsequent `Wait_Osc` events will tell how many oscillator cycles should have elapsed at the given point in the test data. Thus, the initial `Wait_Osc` event with Cycles=0 serves only as a signal to turn on one's stopwatch, so to speak.

The next pattern turns on the `Scan_Ctl` input. When an oscillator is active (meaning that a `Wait_Osc` has been encountered), any events following the `Wait_Osc` signal (even though possibly in a different "pattern") are assumed to be applied immediately within the first oscillator cycle following the `Wait_Osc` event.

The scansequence, OPCKTscanSeq, is executed next. In this example, the scansequence consists of a single pattern that defines two pseudo PI events. There are no external primary input signals applied here, since the entire scan operation is effected by the train of pulses on the oscillator after the `Scan_Ctl` signal was raised. This scansequence serves to tell Encounter Test programs how the scan operation works with respect to the pseudo PIs, which are treated like real primary inputs.

After the scansequence has completed (eight repetitions in our example), the scansectionexit sequence, OPCKTscanSectExit, is executed. This sequence serves two purposes:

- Provides a place to specify the `Wait_Osc` event which tells how many oscillator cycles should elapse before the tester can assume that the scan operation is complete.
- This is where the `Scan_Ctl` signal is returned to its normal state of 0.

Note from the timing diagram of Figure 5-7 that the scan operation automatically terminates and the clock generation logic goes dormant four oscillator cycles after the raising of the `Scan_Ctl` signal. Thus, the restoring of the `Scan_Ctl` signal to the 0 state does not have to be timed, and can be done at any time following the four oscillator pulses and before it is time for the next scan operation. The `Wait_Osc` (Cycles=4,Off) event not only causes the four oscillator cycles to elapse before proceeding, but the "Off" parameter also signifies that subsequent events are not being synchronized with the oscillator; the oscillator is considered to be inactive (i.e. ignored but still running) at this point, and will remain inactive until the next `Wait_Osc` event is encountered.

When an oscillator is active, the intervening events between `Wait_Osc` events are applied as follows:

- All external events (e.g., `Stim_PI`, `Pulse`, `Measure_PO`) are applied immediately.
- All pseudo PI events are assumed to occur in their order of appearance in the sequence, and be complete by the number of cycles in the following `Wait_Osc` event.
- No other external activity occurs until the time specified in the following `Wait_Osc` event.

Encounter Test will not rearrange the order of the events in the sequence definitions. This means that when composing the sequences, the user should adhere to the above guideline.

To understand how Encounter Test handles `Wait_Osc` events, it helps to think in terms of an oscillator signal being either non-existent, running, or active. An oscillator comes into existence when it is connected to a pin by a `Start_Osc` event. A `Stop_Osc` event effectively kills the oscillator, returning the pin to a static level. The oscillator is said to be running as long as it "exists"--that is, as long as it is connected to the pin. But connecting a pin to an oscillator does not automatically mean that the design is responding to the oscillator. There may be (and usually are) some enabling signals, such as `Scan_Ctl` in our example, that control the effect of the oscillator upon the logic. You can think of the oscillator as being "active" whenever it is causing design activity. While the oscillator is active, the design is usually running autonomously, perhaps communicating to the outside world through asynchronous interfaces.

Encounter Test does not understand asynchronous interfaces, and continually simulating a free-running oscillator signal would be prohibitively expensive. Therefore, Encounter Test supports oscillators only if they affect the design through cut points, and all Encounter Test programs except for Verify OPC Sequences treat those cut points as primary inputs, ignoring the oscillator completely. Still, consideration has to be given to the relationship between the oscillator and other primary input stimuli so that the tester knows how to apply the patterns and Verify On Product Clock Sequences knows how to simulate the sequence to assure that it actually produces predictable results and that the cut point (pseudo PI) information is correctly specified for the mainstream Encounter Test processes.

When the first `Wait_Osc` event is encountered, the specified pin must have previously been connected to an oscillator by a `Start_Osc` event. The first `Wait_Osc` is a sign that the oscillator is about to become active.

With the oscillator considered to be in an active condition, subsequent `Wait_Osc` events specify the number of oscillator cycles since the previous `Wait_Osc` event that must transpire before proceeding any further in the pattern sequence.

When the pattern sequence reaches a phase where the design will no longer be “listening” to the oscillator signal, and primary input stimuli can be applied without any regard to the oscillator, the `Wait_Osc` event provides a flag, Off. The off attribute on a `Wait_Osc` event is a sign that the oscillator is no longer causing any significant design activity, and is considered as being reverted to the should be in an inactive state after the specified number of cycles.

If two or more oscillators are active simultaneously, Encounter Test assumes that they are controlling independent portions of the design.

Figure 5-8 A scanop sequence definition using a free-running oscillator

```
TBDpatt_Format (mode=node, model_entity_form=name);

[ Define_Sequence OPCKTscanPre 1 (scanprecond);
  [ Pattern 1.1 (pattern_type = static);
    Event 1.1.1 Stim_PI (): "Pin.f.l.opckt.nl.Scan_Enable"=0;
    Event 1.1.2 Wait_Osc (Cycles=0): "Pin.f.l.opckt.n.l.Osc_In";
  ] Pattern 1.1;
  [ Pattern 1.2 (pattern_type = static);
    Event 1.2.1 Stim_PI (): "Pin.f.l.opckt.nl.Scan_Ctl"=1;
  ] Pattern 1.2;
] Define_Sequence OPCKTscanPre 1;

[ Define_Sequence OPCKTscanSeq 2 (scansequence) (repeat=8);
  [ Pattern 2.1 (pattern_type = static);
    Event 2.1.1 Pulse_PPI (): "A_Clk"=+;
    Event 2.1.2 Pulse_PPI (): "B_Clk"=+;
  ] Pattern 2.1;
] Define_Sequence OPCKTscanSeq 2;

[ Define_Sequence OPCKTscanSectExit 3 (scansectionexit);
  [ Pattern 3.1 (pattern_type = static);
    Event 3.1.1 Wait_Osc (Cycles=4,Off): "Pin.f.l.opckt.n.l.Osc_In";
  ] Pattern 3.1;
  [ Pattern 2.2 (pattern_type = static);
    Event 3.2.1 Stim_PI (): "Pin.f.l.opckt.nl.Scan_Ctl"=0;
  ] Pattern 3.2;
] Define_Sequence OPCKTscanSectExit 3;

[ Define_Sequence OPCKTscanSect 4 (scansection);
  [ Pattern 4.1 (pattern_type = static);
    Event 4.1.1 Apply (): OPCKTscanPre;
    Event 4.1.2 Apply (): OPCKTscanSeq;
    Event 4.1.3 Apply (): OPCKTscanSectExit;
  ]Pattern 4.1;
]Define_Sequence OPCKTscanSect 4;

[ Define_Sequence OPCKTscanOpSeq 5 (scanop);
  [ Pattern 5.1 (pattern_type = static);
    Event 5.1.1 Apply (): OPCKTscanSect;
  ] Pattern 5.1;
] Define_Sequence OPCKTscanOpSeq 5;
```

Importing Test Sequences

When specified by the user, test mode initialization and scan sequences must be provided as input to Build Test Mode, Test and setup sequences must be imported Read Sequence Definition after the test mode has been built. When running the import test data function, give it the name of the file which you created as described above. See “Reading Sequence Definition Data (TBDseq)” on page 265 for further details on this procedure.

Ignoremeasures File

An ignoremeasures file is used to specify measure points to be ignored during test generation or fault simulation. Measures are suppressed for all latches in the file by assigning a measure X value instead of measure 1 or measure 0. Specify the keyword value `ignoremeasures=`*`ignoremeasure_filename`* to utilize the ignoremeasures file.

The format of the file is a list of block or pin names, one name per line. The names may be specified using either full proper form (Pin.f.l.topcell.nl.hier_name) or short form (hier_name).

You can add comments in the ignoremeasures file using any of the following characters:

```
#   <white space followed by comment>

//  <white space followed by comment>

/*  <white space followed by comment block> */
```

The following is an example of an ignoremeasures file:

```
# Sample ignoremeasures file
Block.f.l.DLX_TOP.nl.DLX_CORE.C_REG.STORAGE.Q_N64_reg_0.I0.dff_primitive
DLX_CORE.C_REG.STORAGE.Q_N59_reg_5.I0.dff_primitive
Pin.f.l.DLX_TOP.DLX_CHIPTOP_DATA[31]
DLX_TOP.DLX_CHIPTOP_DATA[30]
```

The Scan_Unload and/or the Measure PO for the specified latches/POs will be set to X.

The following are potential results when resimulating with an ignoremeausure file:

- Miscompare messages are produced since the miscompare checking is done before the measures are X'd out.
- If a measure is X'd out, the fault coverage is adjusted to remove the fault(s) detected by that pattern.

Keepmeasures file

An keepmeasures file is used to specify measure points to be retained during test generation or fault simulation. This file is useful if the number of measures to be ignored is greater than the number to be kept.

All measures except those specified in the file are suppressed by assigning a measure X value to all latches. The keepmeasures file is utilized by specifying the keyword value `keepmeasures=`*`keepmeasure_filename`*.

The format of the file is a list of block or pin names, one name per line. The names may be specified using either full proper form (Pin.f.l.topcell.nl.hier_name) or short form (hier_name).

You can add comments in the keepmeasures file using any of the following characters:

```
#

//

/* <comment block> */
```

The following is an example of a keepmeasures file:

```
#sample keepmeasures file
Block.f.l.DLX_TOP.nl.DLX_CORE.C_REG.STORAGE.Q_N64_reg_0.I0.dff_primitive
DLX_CORE.C_REG.STORAGE.Q_N59_reg_5.I0.dff_primitive
Pin.f.l.DLX_TOP.DLX_CHIPTOP_DATA[31]
DLX_TOP.DLX_CHIPTOP_DATA[30]
```

The Scan_Unload and/or the Measure PO for the specified latches/POs are retained.

6

Advanced ATPG Tests

This chapter discusses the commands and techniques to generate advanced ATPG test patterns. These techniques are performed on designs at the same stage as in the basic ATPG tests. Some of these techniques require special fault models to be built.

The chapter covers the following topics:

Create IDDq Tests

IDDq testing exploits the fact that a fully complementary CMOS device has a very low leakage current when the device is in the quiescent (static) state. A defect that causes the CMOS device to have a high leakage current can be detected by measuring the current (Idd) into the Vdd bus. Many defects and faults are easier to test with IDDq than with conventional testing which measures only the signal outputs; some faults that cannot be detected by the conventional measures can be detected by IDDq testing. See "IDDq Test Status" in the *Encounter Test: Guide 4: Faults* for more information.

A few issues must be considered when attempting to integrate IDDq testing into a high-volume manufacturing test process:

- Measurement of the supply current takes longer than measuring signal voltages. The design has to be given time to settle into a quiescent state after the test is applied, and the current measurement has to be averaged over some time interval (maybe over tests) to eliminate noise in the measurement. Therefore, the test time per pattern is quite large and so the number of current sensing tests must be kept small (considerably less than one hundred for what most manufacturers would consider reasonable throughput in chip testing).
- Some design may contain high steadystate current conditions that are to be avoided.
- The techniques used to find the cutoff between good and faulty levels of IDDq current are often ad hoc and empirical, based on experience gained through extensive experimentation.

Despite the challenges associated with IDDq testing, the benefit of detecting faults which are not detected by conventional voltage sensing techniques (for example, gate-oxide shorts) has given IDDq testing a place in many chip manufacturers' final test processes.

A typical IDDq test is composed of the following sequence of events:

1. Apply the test stimuli - load scan chain latches and set primary inputs (PIs).
2. Apply the required stimuli to disable any current paths that could invalidate an IDDq measurement.
3. Wait until the design has stabilized.
4. Measure the IDDq current.

The Encounter Test stored pattern test generation application can automatically generate IDDq test patterns. Since application of IDDq patterns may take significantly longer that normal patterns due to having to wait for the design activity to quiesce, Encounter Test provides some options for helping to keep the size of the IDDq test pattern set small.

IDDq test vectors can be used to support extended voltage screen testing via the generation of a scan chain unload (`Scan_Unload` event) immediately after each `Measure_Current` event. The `Scan_Unload` event supports the existing Ignore Latch functions as found in the existing ATPG pattern generation.

To perform Create IDDq tests using the graphical user interface, refer to "Create Iddq Tests" in the *Encounter Test: Reference: GUI.*

To perform Create IDDq tests using the command line, refer to "create_iddq_tests" in the *Encounter Test: Reference: Commands.*

The syntax for the `create_iddq_tests` command is given below:

```
create_iddq_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated

The most commonly used keywords for the `create_iddq_tests` command are:

- `compactioneffort=<value>` - The amount of time and effort to reduce down the pattern size. Default is `medium` effort.
- `iddqeffort=<value>` - The amount of time and effort for the test generator to create tests for hard-to-test faults. Default is `low` effort.
- `iddqunload=no|yes` - Whether to enable the measure of the scan flops. Default is `no`.
- `ignoremeasures=<filename>` - List of flops to ignore during measures. Refer to "Ignoremeasures File" on page 204 for more information.
- `iddqmaxpatterns=#` - The default number of sequences as defined in the Tester Description Rule (TDR) used when creating the testmode. This can be overridden. Refer to Tester Description Rule (TDR) File Syntax in the *Encounter Test: Guide 2: Testmodes* for more information.

Refer to "create_iddq_tests" in the *Encounter Test: Reference: Commands* for information on these keywords.

Prerequisite Tasks

Complete the following tasks before executing Create IDDq Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

Output

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

Debugging Low Coverage

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

Note: Deterministic Faults Analysis will not be available based on random pattern generation.

Iddq Compaction Effort

After generating each Iddq pattern, Encounter Test randomly fills and simulates it multiple times to identify the best set of random filled data to save. This process is done for each generated pattern, thus resulting in high simulation time.

With `compactioneffort=medium`, the test generator compacts tests to a certain point and then randomly fills/simulates the patterns multiple times. With `compactioneffort=high`, more tests are compacted into the patterns, thus causing a higher test coverage for a given number of tests. If you do not set the pattern limit (which is 20 patterns by default), the end result with `compactioneffort=high` will most likely have a higher coverage with lesser number of patterns. This is because Iddq patterns are mostly driven by simulation time so less patterns simulated means less over time.

The test generator works harder but the way Iddq patterns are simulated and fault graded results in simulation time making up a majority of the Iddq pattern generation task.

Create Random Tests

Create Random Tests is used to generate and simulate random patterns. Random tests detect both static and dynamic faults.

To perform Create Random Tests using the graphical interface, refer to "Create Random Tests" in the *Encounter Test: Reference: GUI*.

To perform Create Random Tests using the command line, refer to create_random_tests in the *Encounter Test: Reference: Commands*.

The syntax for the `create_random_tests` command is given below:

```
create_random_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated

The most commonly used keywords for the `create_iddq_tests` command are:

- `maxrandpatterns=#` - Specify the maximum number of random patterns to generate.
- `minrandpatterns=#` - Specify the minimum number of random patterns to simulate.
- `detectthresholdstatic=#` - Specify a decimal number (such as `0.1`) as the minimum percentage of static faults that must be detected in a given interval. Simulation terminates for the current clocking sequence when this threshold exceeds.

Refer to "create_random_tests" in the *Encounter Test: Reference: Commands* for information on these keywords.

Prerequisite Tasks

Complete the following tasks before executing create IDDq tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.

3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

Output

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

A sample output log is given below:

```
************************************************************************
         ----Stored Pattern Test Generation Final Statistics----
                Testmode Statistics: FULLSCAN_TIMED
                        #Faults   #Tested   #Redund   #Untested   %TCov   %ATCov
Total Static             908       704       52        106         77.53   82.24
Total Dynamic            814       242       44        513         29.73   31.43


                            Global Statistics
                        #Faults   #Tested   #Redund   #Untested   %TCov   %ATCov
Total Static             1022       737       52      187          72.11   75.98
Total Dynamic            1014       242       44      713          23.87   24.95
************************************************************************
                    ----Final Pattern Statistics----
   Test Section Type                          # Test Sequences
   ------------------------------------------------------
      Logic                                   45
   ------------------------------------------------------
      Total                                   45
```

Debugging Low Coverage

Note: In case of random ATPG, if the design is not random testable, low test coverage can be expected.

If you do not achieve the desired ATPG coverage, check for the following problems:

- Contention in the design - Look for TSV-193 and TSV-093 messages from verify_test_structures to identify internal contention.
- Broken scan chains - Analyze the `verify_test_structures` log for broken scan chains.

Note: Deterministic Faults Analysis will not be available based on random pattern generation.

Create Exhaustive Tests

Create Exhaustive Tests applies test patterns to a design to identify faults that are resistant to random patterns. An uncommitted test fault model file is created which contains the status of the faults after simulation. Patterns are simulated until either a fault detection threshold is met or the specified pattern limit is reached. The Encounter Test Log window automatically displays the results when Random Pattern Fault Simulation is complete.

It is possible to control fault detection during Create Exhaustive Tests by inserting Keyed Data into the test sequence. Create Exhaustive Tests checks each event for the existence of keyed data, and if found, allows fault detection to begin with that event. The keyed data should only be placed on a single event within a test sequence. If the keyed data exists on more than one event, Create Exhaustive Tests begins fault detection on the last event having keyed data.

Additional information is available in:

- "Keyed Data" in the *Encounter Test: Reference: Test Pattern Formats*.
- "Coding Test Sequences" on page 180

This scenario does affect the measure points where Create Exhaustive Tests detects faults. For example, if the test sequence contains a `Scan_Unload` event, the slave latches are still the only latches used as the detect points.

You can perform Create Exhaustive Tests using only the command line. Refer to "create_exhaustive_tests" in the *Encounter Test: Reference: Commands* for more information.

The syntax for the `create_exhaustive_tests` command is given below:

```
create_exhaustive_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated

Prerequisite Tasks

Complete the following tasks before running Create Exhaustive Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

Output Files

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

Restrictions

Restrictions of Create Exhaustive Tests are:

- Invalid test data and reduced test coverage can result from designs that do not conform to Encounter Test LSSD guidelines or GSD guidelines. The Test Structure Verification application verifies that these guidelines are met.
- Dynamic faults must be included in the test mode definition and the fault model.
- The only supported values of the `TDR PMU` parameter are `PMU=1` and unlimited PMUs. A PMU value less than the number of data pins on the design is treated by Encounter Test as `PMU=1`. A PMU value equal to or greater than the number of data pins is equivalent to unlimited PMUs.
- IDDq tests are not generated.
- Driver and receiver faults are not processed.
- Random patterns are not generated for a test mode having onboard PRPGs or MISRs.
- Exhaustive simulation is supported for combinational designs with 24 or fewer input pins.

Create Core Tests

These tests are used to verify a macro device (for example, a RAM) embedded in the design. Encounter Test uses isolation requirements and test patterns written specifically for the macro device to map the test patterns for the macro to the tester contactable I/O of the design.

Macro tests may be static, dynamic, or dynamic timed. See "Test Vector Forms" on page 254 for more information on these test formats. Also refer to Properties for Embedded Core Tests in the *Encounter Test: Reference: Legacy Functions* for more information.

To perform Create Core Tests using the graphical interface, refer to "Create Core Tests" in the *Encounter Test: Reference: GUI*.

To perform Create Core Tests using the command line, refer to `"create_core_tests"` in the *Encounter Test: Reference: Commands*.

The syntax for the `create_core_tests` command is given below:

```
create_core_tests workdir=<directory> testmode=<modename> experiment=<name>
tdminput=<infilename> tdmpath=<path>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated
- `tdminput`= Indicates the name of the input TBDbin file containing pre-existing test data for the macro (core) being processed. This file specification is required only if you invoke TCTmain for migrating pre-existing core tests from the core boundary to the package boundary.
- `tdmpath`= Indicates the directory path of input TBDbin files containing pre-existing test data for the macro (core) being processed. The specification is a colon separated list of directories to be searched, from left to right, for the input TBDtdm files. You can also set this option using *Setup Window* in the graphical user interface.

Create Embedded Test - MBIST

This is used for pattern generation and configuration for memory built-in self test (MBIST) Refer to Create Embedded Test in the *Encounter Test: RAK: MBIST Pattern Generation* for more information.

To perform Create MBIST patterns using the graphical interface, refer to "Create Embedded Tests" in the *Encounter Test: Reference: GUI*.

To perform Create MBIST patterns using command line, refer to "create_embedded_test" in the *Encounter Test: Reference: Commands*.

The syntax for the `create_embedded_tests` command is given below:

```
create_embedded_tests workdir=<directory>
```

where `workdir` is the name of the working directory.

Create Parametric Tests

Parametric tests exercise the off-chip drivers and on-chip receivers. For each off-chip driver, objectives are added to the fault model for DRV1, DRV0, and if applicable, DRVZ.

For each on-chip receiver, objectives are added to the fault model for RCV1 and RCV0 at each latch fed by the receiver. These tests are typically used to validate that the driver produces the expected voltages and that the receiver responds at the expected thresholds.

These tests require the fault model including driver/receiver faults (`build_faultmodel includedrvrcvr=yes`)

To perform Create Parametric Tests using the graphical interface, refer to "Create Parametric Tests" in the *Encounter Test: Reference: GUI*.

To perform Create Parametric Tests using command lines, refer to "create_parametric_tests" in the *Encounter Test: Reference: Commands.*

The syntax for the `create_parameteric_tests` command is given below:

```
create_parametric_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG
- `experiment`= name of the test that will be generated

Prerequisite Tasks

Complete the following tasks before running Create Exhaustive Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

Output Files

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

Create IO Wrap Tests

These tests are produced to exercise the driver and receiver logic. The tests use the chip's internal logic to drive known values onto the pads and to observe these values through the pad's receivers. IO wrap tests may be static or dynamic. Static IO wrap tests produce a single steady value on the pad. Dynamic IO wrap tests produce a transition on the pad.

These tests require the fault model including stuck driver and shorted net objects (`build_faultmodel sdtsnt=yes`).

To perform Create I/O Wrap Tests using the command line, refer to "create_iowrap_tests" in the *Encounter Test: Reference: Commands*.

To perform Create I/O Wrap Tests using the graphical user interface, refer to Create IOWrap Tests in the *Encounter Test: Reference: GUI*.

The syntax for the `create_iowrap_tests` command is given below:

```
create_iowrap_tests workdir=<directory> testmode=<modename> experiment=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the testmode for dynamic ATPG

- `experiment=` name of the test that will be generated

Prerequisite Tasks

Complete the following tasks before running Create Exhaustive Tests:

1. Import a design into the Encounter Test model format. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Create a Test Mode. See "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.
3. Build a fault model for the design. See "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

Output Files

Encounter Test stores the test patterns in the experiment name.

Command Output

The output log contains information about testmode, global coverage, and the number of patterns used to generate those results.

IEEE 1149.1 Test Generation

Encounter Test supports two ways to efficiently do stored pattern test generation (SPTG) for a design which implements the 1149.1 Boundary standard. Both of these do not so much affect test generation itself as they do the test mode initialization steps and the scan protocol associated with test generation.

1. LSSD or GSD scan

   For 1149.1 LSSD or GSD scan SPTG the design is brought to the Test-Logic-Reset state of the TAP controller by the mode initialization sequence; all SPTG takes place with the controller held in that state. All scanning operations are performed using the defined LSSD or GSD scan chains, with the Test-Logic-Reset state maintained; the TAP port is not involved in scanning.

   When the TAP controller is in the Test-Logic-Reset state it is effectively disconnected from the chip's system logic and does not interfere with normal chip operation. SPTG can proceed without the sequential TG problem that would be incurred by having to

repeatedly manipulate the finite state machine of the TAP controller every time a test is being developed.

Figure 6-1 Overview of Scan Structure for 1149.1 LSSD/GSD Scan SPTG

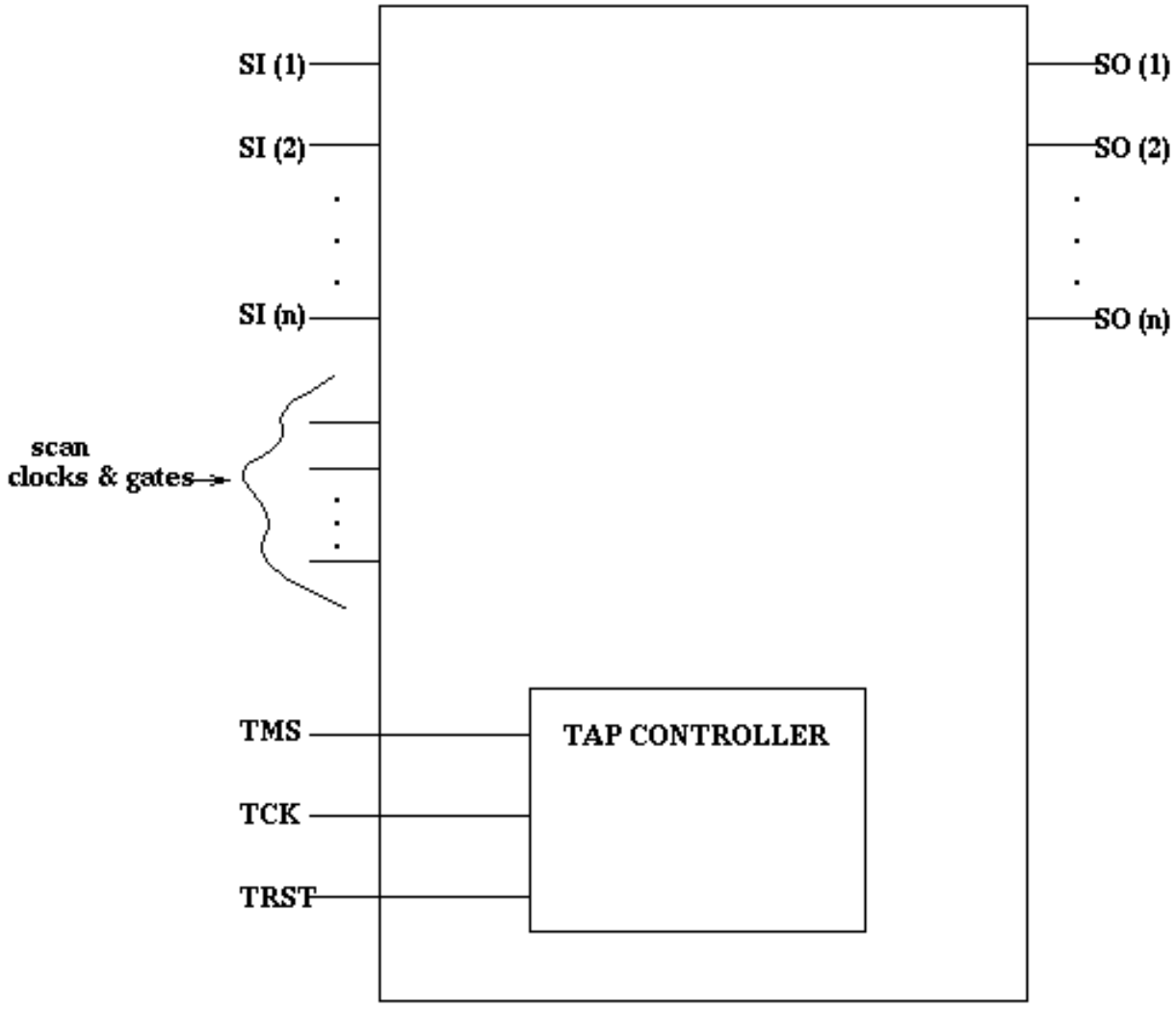


Figure 6-1 shows an overview of the scan structure used for 1149.1 LSSD or GSD scan SPTG. Note the following important points:

a. The TAP controller is in the Test-Logic-Reset state at all times, both for test generation and for scanning.

b. The scan chains used are all the defined LSSD or GSD scan chains; the 1149.1 TDI-to-TDO scan chains are not used for scanning.

c. The scan protocol, which is automatically generated by Encounter Test, is the usual A-E-B scan sequence, utilizing the LSSD or GSD scan clocks and gates.

2. TAP scan

For 1149.1 TAP scan SPTG the design is brought to a user-specified TAP controller state by the mode initialization sequence, and all SPTG takes place with the controller held in the specified state. This state is not maintained for the scan operation, however, which is done exclusively by way of the TAP and therefore necessitates TAP controller state changes. The test generation state desired is specified by the TAP_TG_STATE parameter of the SCAN TYPE mode definition statement. This state may be Test-Logic-Reset, the same as is automatically assumed for 1149.1 LSSD or GSD scan SPTG, or may be either Run/Test-Idle, Shift-DR or Pause-DR.

Figure 6-2 Overview of Scan Structure for 1149.1 TAP Scan SPTG

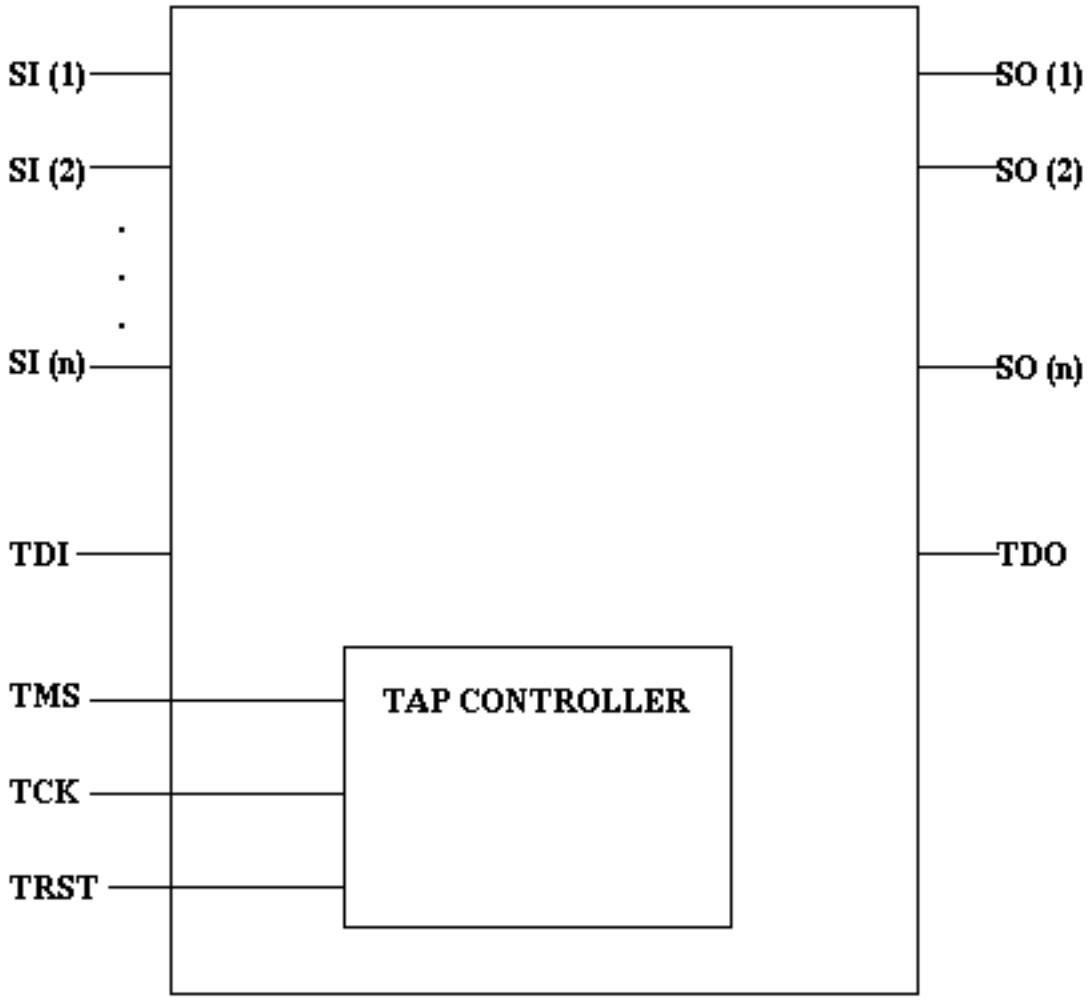


Figure 6-2 shows an overview of the scan structure used for 1149.1 TAP scan SPTG. In contrast with Figure 6-1 note the following points:

a. The TAP controller is in one of four supported states for test generation, selectable by the user: Test-Logic-Reset, Run-Test/Idle, Shift-DR or Pause DR. (The selected TG state does not hold for the scanning operation.)

b. The scan chain used is whatever test data register is enabled between TDI and TDO. Optionally, other defined scan chains (SI/SO) will also be used for scanning, but only if scannable via the TAP scan protocol (see next).

c. The scan protocol, automatically generated by Encounter Test, conforms to the 1149.1 standard. That is, all scanning takes place purely by TMS and TCK manipulation. No other defined scan clocks or scan enables are used.

Whether to do 1149.1 SPTG by LSSD or GSD scan or TAP scan is dependent on how your scan chains are configured. If the tester on which the chip is to be tested is not too severely pin limited then the best way to do stored pattern test generation is probably by LSSD or GSD scan, utilizing totally all the normal scan capabilities of the chip. If, on the other hand, you are dealing with a severely constrained tester pin interface, as might very well be the case, for instance, with burn-in test, then you might want to consider TAP scan SPTG. With TAP scan SPTG all scan operations are performed by the standard 1149.1 defined scan protocol. TCK and TMS are manipulated to cause scan data to move along the TDI-to-TDO scan chain, as well as possibly other defined SI-to-SO scan chains. So you are not limited to just one scan chain under TAP scan SPTG, but it is important to realize that all scan chains that are defined must be scannable in parallel and all by the same TCK-TMS scan protocol.

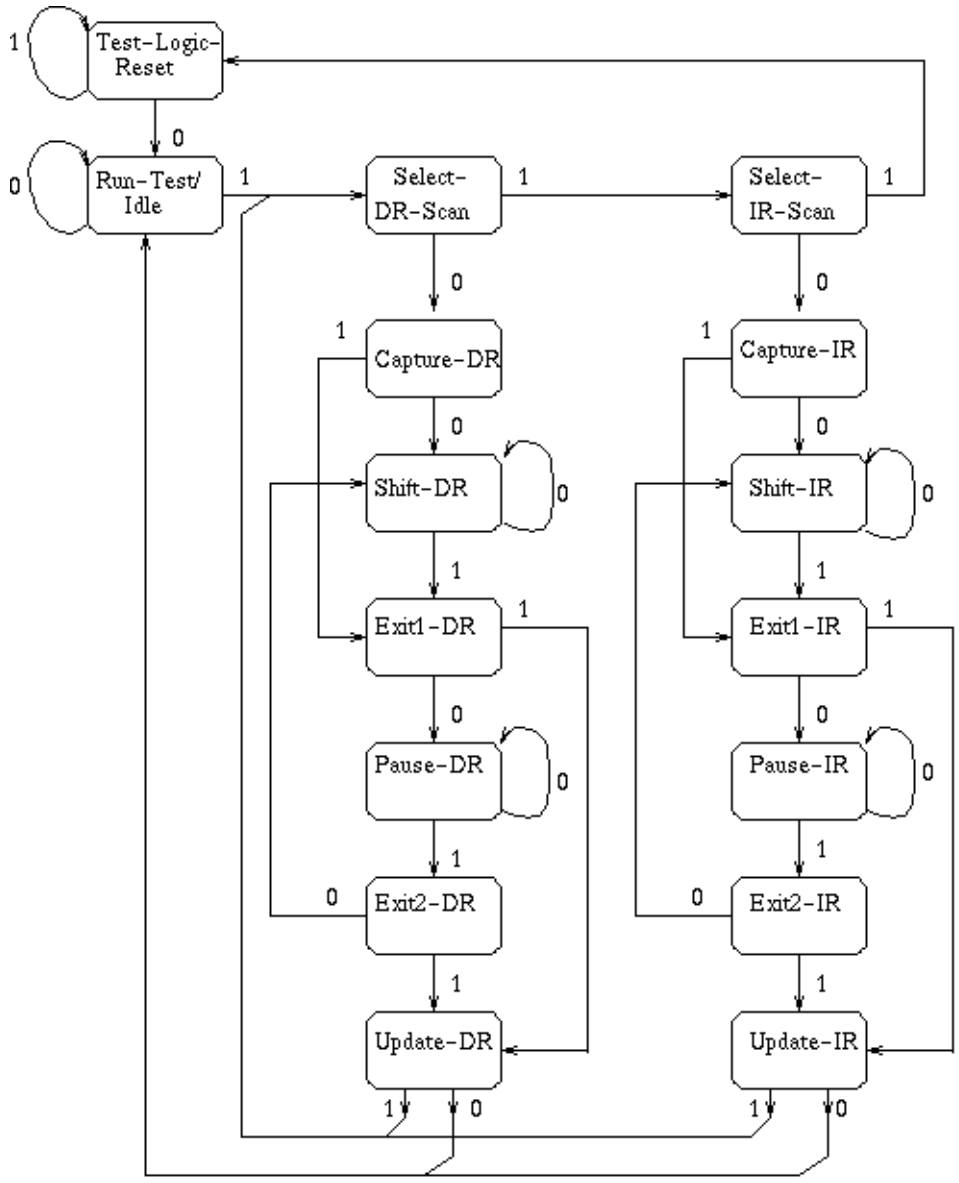
Configuring Scan Chains for TAP Scan SPTG

For this approach to 1149.1 SPTG it is necessary that your part's test aspects take two things into consideration:

1. TCK must be distributed as a scan clock to all memory elements which are to be scanned by the 1149.1 scan protocol.
2. For all those scannable memory elements that are to be scanned via the TDI-to-TDO path there must exist scan path gating to redirect scanning away from the defined SI-SO LSSD or GSD scan paths and to the TDI-TDO TAP scan path.

The way that these two things are accomplished is by defining a special instruction - call it TAPSCAN - that when loaded into the 1149.1 instruction register (IR) will bring the necessary clock and scan path gating into play. Refer to Figure 6-3, which describes TAP controller operation, will aid in understanding why clock gating is necessary.

Figure 6-3 TAP Controller State Diagram



This figure is a representation of the state diagram for the 1149.1 TAP controller contained in IEEE standard 1149.1-1990, IEEE Standard Test Access Port and Boundary Scan Architecture.

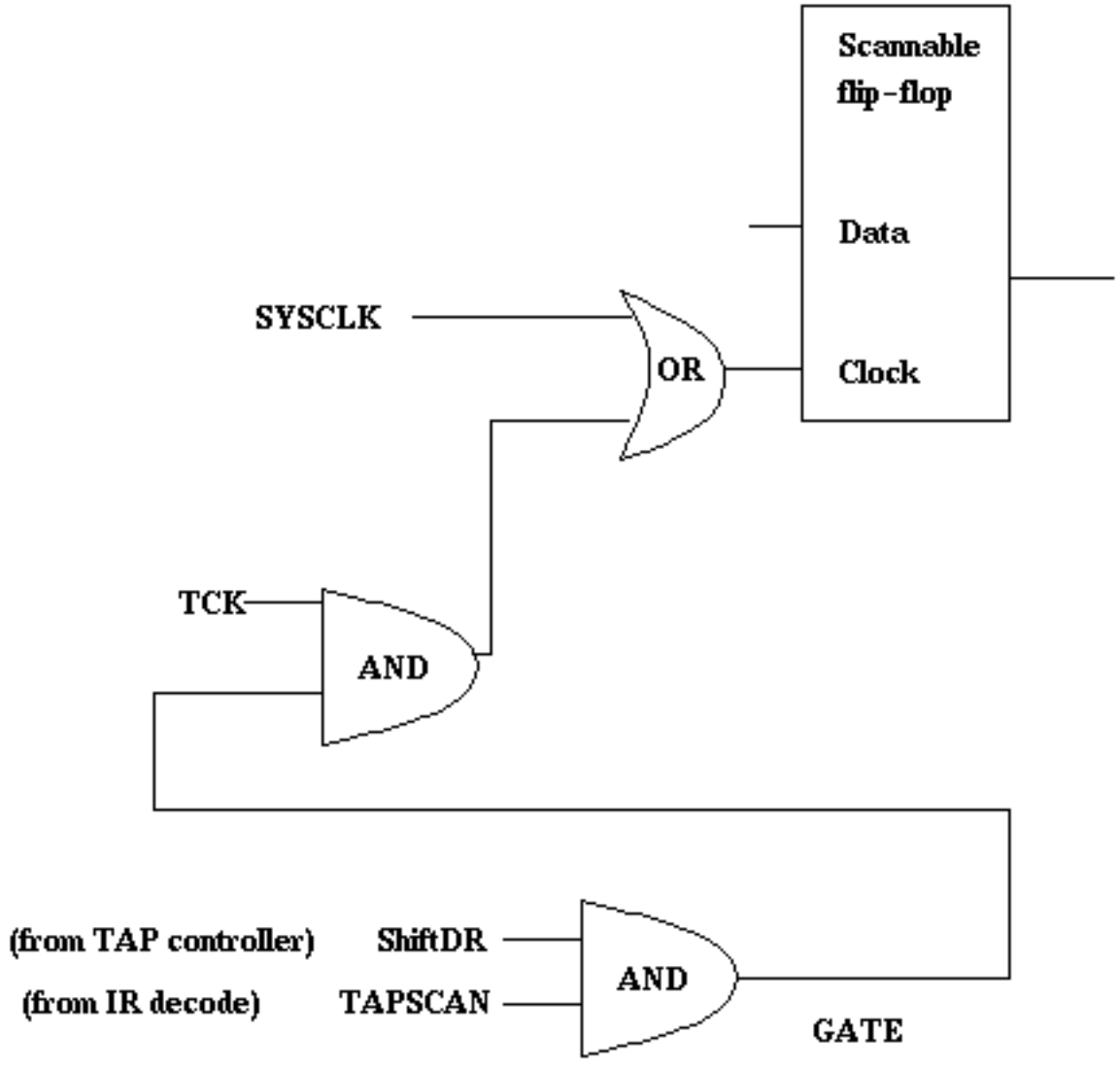
The TAP controller is a synchronous finite state machine that responds to changes at the TMS and TCK inputs to the TAP to control the operation of the circuitry defined by the 1149.1 standard.

NOTES:

1. The value shown adjacent to each state transition in this figure represents the signal present on TMS at the time of a rising edge of TCK

As seen from this figure, scanning of a test data register (TDR) must take place by exercising the state transitions in the Select-DR-Scan column. Thus we have the following sequence of state transitions for a TDR scan: Select-DR-Scan, Capture-DR, Shift-DR (repeat), Exit1-DR, Update-DR. For this TDR scan protocol it is necessary to pass through the Capture-DR state, in which data is parallel loaded into the TDR selected by the current instruction. Bearing in mind that SPTG will already have captured test response data into this TDR prior to scanning you must not allow the Capture-DR state to capture new data, thus destroying the test generation derived response. It is thus necessary that the use of TCK as a scan clock to the TDR be gated off except when in the Shift-DR state. Figure 6-4 shows an example of how this can be done for a MUXed scan design.

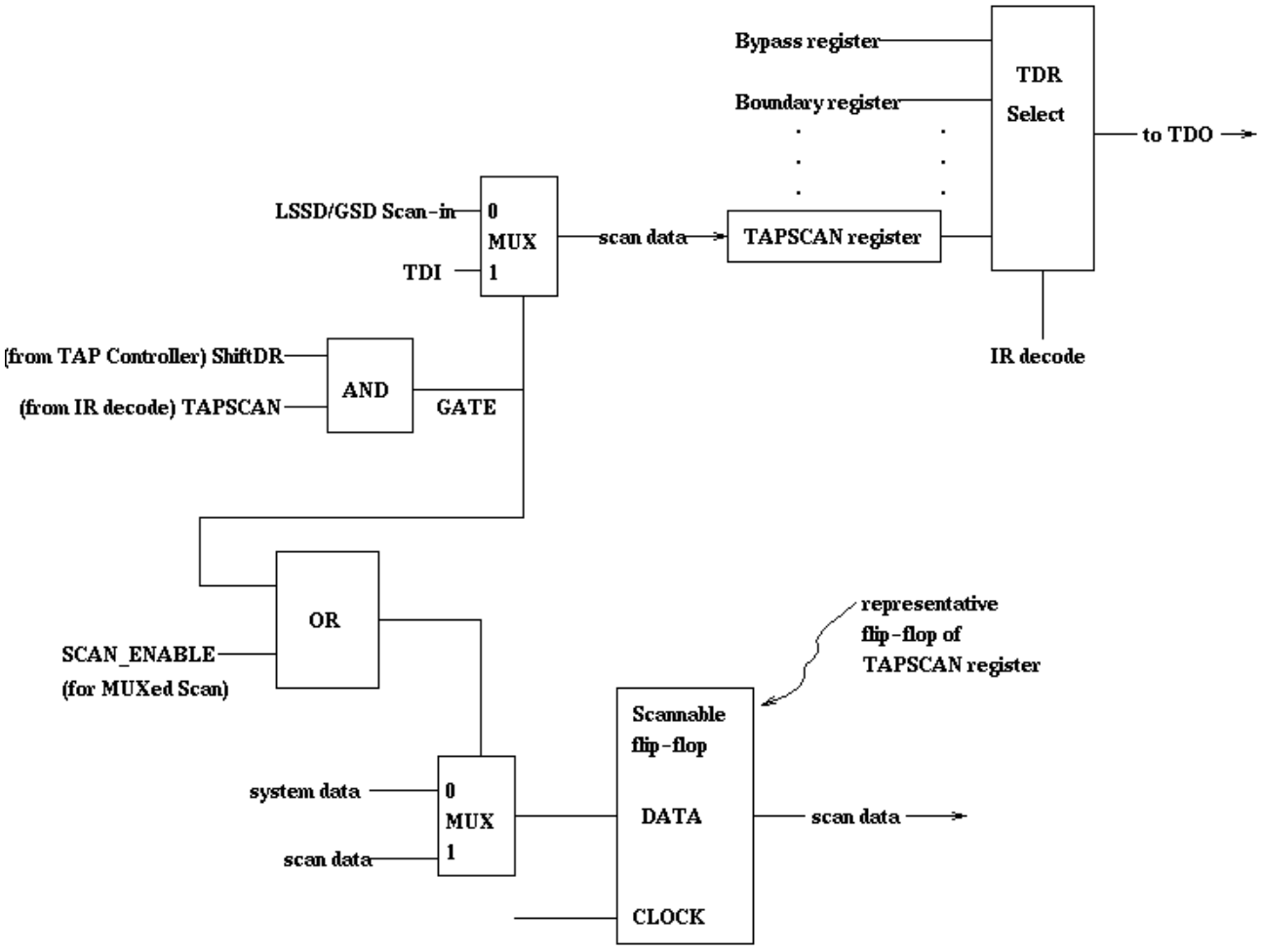
Figure 6-4 Example of TCK Clock Gating for TAP Scan SPTG



The TAPSCAN signal in this figure is hot (logic 1) when the instruction loaded into the IR is TAPSCAN, the name we have arbitrarily assigned to the special instruction implemented in support of TAP scan SPTG.

Gating of scan data must similarly use the ShiftDR TAP controller signal and the decode of the TAPSCAN IR state. Figure 6-5 shows an example of how this might be implemented for a MUXed scan design.

Figure 6-5 Example of scan chain Gating for TAP Scan SPTG



1149.1 SPTG Methodology

The preceding sections dealt with the concepts of implementing 1149.1 SPTG. We will now discuss the methodology considerations directly related to this form of SPTG.

The first thing necessary for 1149.1 SPTG is that the test mode of interest be recognizable as having 1149.1 characteristics. This means that it must have at least one each of the following pins:

- TEST_CLOCK Input
- TEST_MODE_SELECT Input
- TEST_DATA_INPUT
- TEST_DATA_OUTPUT

Refer to "1149.1 Boundary Controls" in the *Encounter Test: Guide 1: Models* for details on these pin types.

Optionally, there may also be a Test Reset Input -TRST. There are three ways of identifying these pins to Encounter Test - either in the model, in the BSDL or in Test Mode Define ASSIGN statements. Conflicts between these three possible sources of information are resolved by having the mode definition ASSIGN statement take precedence over the BSDL, which in turn takes precedence over the model.

Given that the test mode being processed is 1149.1, then there are two components to the test generation process:

1. Fault simulation of 1149.1 BSV verification sequences

   Algorithmic SPTG will likely have some difficulty generating tests for faults associated with the 1149.1 test logic (TAP controller, BYPASS register, etc.). The most efficient way to cover these faults is by invoking the General Purpose Simulator to simulate the verification sequences developed by 1149.1 Boundary Scan Verification (BSV). Here is a sample mode definition which will serve both for BSV and invocation of the General Purpose Simulator:

```
Tester_Description_Rule = tdr1
                ;

scan            type = 1149.1
                boundary=no
                in = PI
                out = PO
                ;

test_types      none
                ;

faults          static
                ;
```

2. Algorithmic SPTG

The determination of whether SPTG is to proceed using LSSD or GSD scan or TAP scan is made by interrogating the mode definition TEST_TYPES and SCAN TYPE statements:

- TEST_TYPES not NONE, SCAN TYPE = LSSD or GSD indicates LSSD or GSD scan
- TEST_TYPES not NONE, SCAN TYPE = 1149.1 indicates TAP scan

Following are two sample mode definitions for 1149.1 SPTG, one for LSSD or GSD scan and one for TAP scan:

```
/************************************************************/
/*
/*     Sample mode definition for 1149.1 LSSD or GSD scan SPTG
/*
/************************************************************/

Tester_Description_Rule = tdr1
              ;

scan          type = gsd
              boundary=no
              in = PI
              out = PO
              ;

test_types    static logic signatures no
              ;

faults        static
              ;

/*********************************************************/
/*
/*     Sample mode definition for 1149.1 TAP scan SPTG
/*
/*********************************************************/

Tester_Description_Rule = tdr1
              ;

scan          type = 1149.1 instruction=10 tap_tg_state=rti
              boundary=no
              in = PI
              out = PO
              ;

test_types    static logic signatures no
              ;
```

```
faults          static
                ;
```

Finite state machine (FSM) latches:

The mode initialization sequence for 1149.1 SPTG must bring the finite state machine latches of the TAP controller to a known state. If a TRST pin is present then an asynchronous reset accomplishes this quite readily. If, on the other hand, there is no TRST pin, then it is necessary to initialize these latches by a synchronous reset employing TCK and TMS. Standard three-valued simulation will not bring these latches to a known state from an initial undefined state. Special provisions must be made by Test Mode Define to identify these latches and then resort to a complex simulation process to get them to their home state. You can aid in this process by using the FSM attribute, either in the model source or the mode definition ASSIGN statement, to identify the finite state machine latches of the TAP controller. Otherwise Test Mode Define will attempt to automatically identify them. If you choose to identify these latches then the FSM attribute is placed either on the output pin of the latch primitive or the output pin of a cell definition or instance that contains the latch primitive. When placed on the cell definition or instance then the attribute is associated with the latch which drives the pin carrying the attribute.

One other consideration applies with respect to mode definition, but only if TAP scan is called for. The SCAN TYPE mode definition statement must specify both an INSTRUCTION and a TAP_TG_STATE.

- INSTRUCTION:

  Specify the instruction to be loaded into the IR to select a test data register (TDR) for scanning through the TAP. This instruction will configure the design under test so that SPTG can work effectively in the TAP controller state designated by TAP_TG_STATE. It not only gates the selected TDR for scanning but also causes the correct TCK gating to be brought into effect for all those memory elements to be scanned. (An example of the type of clock gating necessary is shown in Figure 6-4 on page 223.

  The instruction to be loaded is specified in one of two ways, either:

  - `bit_string`

    Specify the binary bit string to be loaded into the IR, with the bit closest to TDI being the left-most bit and the bit closest to TDO being the right-most bit.

  - `instruction_name`

    Specify the name of the instruction to be extracted from the BSDL.

- TAP_TG_STATE

This parameter is used to specify the TAP controller state in which test generation is to be performed. Acceptable values:

a. `RUN_TEST_IDLE` (RTI)

TG is to take place in the un-Test/Idle state of the TAP controller.

b. `TEST_LOGIC_RESET` (TLR)

TG is to take place in the Test-Logic-Reset state of the TAP controller.

c. `SHIFT_DR` (SDR)

TG is to take place in the Shift-DR state of the TAP controller.

d. `PAUSE_DR` (PDR)

TG is to take place in the Pause-DR state of the TAP controller.

e. `CAPTURE_DR` (CDR)

This option is intended for use only if you have implemented parallel scan capture clocking via the `CAPTURE_DR` state. This is not a recommended way to implement internal scan via an 1149.1 interface, but Encounter Test will support it in a limited fashion.

For `CAPTURE_DR` 1149.1 test modes, Encounter Test generates a test sequence definition that can be used when performing ATPG. When there are clocks defined beyond `TCK`, an additional sequence definition is generated. It is permissible to copy this additional sequence to use as a template for defining additional test sequences for use during ATPG. **Note** that all such test sequences must include a single `TCK` pulse and may optionally include as may pulses as desired of other clocks.

See TAP_TG_STATE, described under “SCAN” in the *Encounter Test: Guide 2: Testmodes* for additional information.

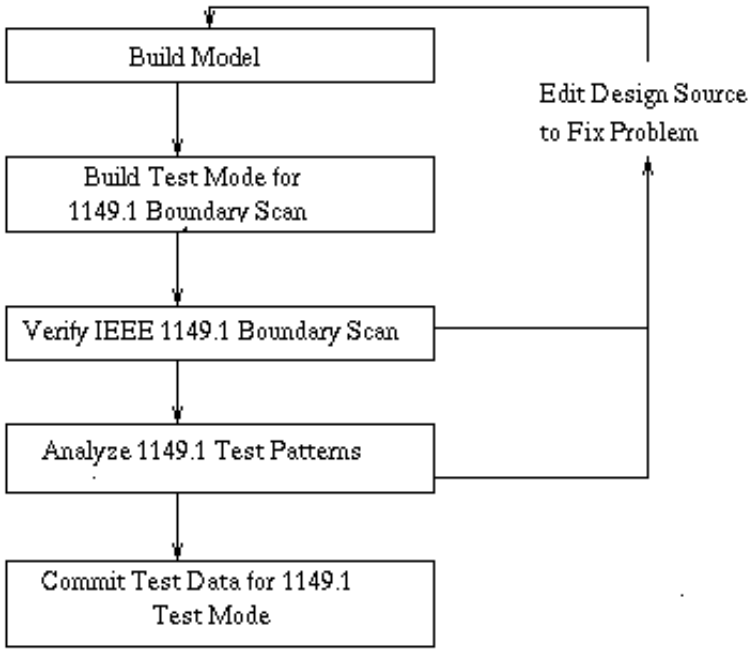
From this point on, for both LSSD or GSD scan and TAP scan, the usual SPTG methodology is followed for the 1149.1 test mode. Test Mode Define will automatically derive the mode initialization sequence and scan protocol necessary for test generation.

Refer to “IEEE 1149.1 Test Generation” on page 218 for a typical IEEE 1149.1 Test Generation task flow.

1149.1 Boundary Chain Test Generation

The following figure shows a typical processing flow for running Stored Pattern Test Generation with an 1149.1 Boundary chip.

Figure 6-6 Encounter Test 1149.1 Boundary Chip Processing Flow



1. Build Model

   For complete information on *Build Model*, refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models*.

2. Build Test Mode for 1149.1 Boundary

   A sample of a mode definition file for this methodology follows:

```
TDR=bs_tdr_name
SCAN TYPE=1149.1 IN=PI OUT=PO;
TEST_TYPE=NONE;
FAULTS=NONE;
.
.
.
```

   For complete information, refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.

3. Verify 1149.1 Boundary

   For complete information, refer to the "Verify 1149.1 Boundary" in the *Encounter Test: Guide 3: Test Structures*.

During the verification of the 1149.1 structures, Encounter Test creates an experiment called 1149. This experiment can be fault graded to get coverage.

4. Analyze 1149.1 patterns

   Fault grade the 1149 test patterns. Refer to "Analyzing Vectors" on page 272 for more information.

5. Commit Results

   Refer to "Committing Tests" on page 81 for more information.

Reducing the Cost of Chip Test in Manufacturing

As chip densities continue to grow, the number of test vectors required to test a chip also tends to grow. In addition, data volume required to represent a single test also grows since the number of scan bits grows along with chip complexity. This results in test data volumes that grow at a faster than linear rate as compared to the number of gates in the design. The increase in test data volume translates into an increase in the tester socket time required to test a chip due to both the increase in scan bits (scan chain length) and the increase in the number of tests.

Based on the preceding paragraph, the following assumptions are used to address the reduction of chip manufacturing cost:

- Tester time and test data volume are among the primary sources of test cost.
- Any decrease in the time to apply patterns required to detect a faulty design translates to cost savings.

Breaking the elements of cost into finer detail, it can also be assumed that a near-minimal set of test vectors will take too long to apply and require too much buffer resource, resulting in the conclusion that test vector cost should be reduced. The two aspects that contribute to the cost of a test vector are:

- stimulus data - the data stored on the tester which allows us to excite certain areas of the design.
- response data - data stored on the tester which allows us to detect when the design under test responds differently than expected.

For stored pattern testing, the stimulus and response data comprise about 98% of the test data for a large chip, with the remaining 2% attributed to clocking template and control overhead. The response data can be as much as twice as large as the stimulus data depending on the type of tester being used and how the data bits are encoded. However,

stimulus data can be more of a problem for highly sequential (partial scan) designs. For large chips, most of the stimulus and response data is located in the scan events.

The Encounter Test Stored Pattern test methodology includes a method, based on the creation and use of an On-Product MISR test mode, to reduce data volumes while supporting:

- Both stored pattern and WRP test methods
- Use of test reordering to improve fault coverage versus pattern counts
- Diagnostics
- Reduced pin count test methodology

Using On-Product MISR in Test Generation

A sample scenario for implementing the On-Product MISR test mode in the test generation process:

- Insert the MISR and associated muxing logic into the design.
- For stored pattern test generation with an on-product MISR it is necessary to define both an On-Product MISR test mode and a diagnostics test mode. The On-Product MISR test mode definition refers to the diagnostics test mode by the DIAGNOSTIC_MODE statement. See "DIAGNOSTIC_MODE" in the *Encounter Test: Guide 2: Testmodes* for more information.
- Process the design through Encounter Test to generate test patterns. Refer to "Static Test Generation" on page 57 for details.

The resulting test data from Stored Pattern Test Generation will contain a Channel_Scan event and optionally, Diagnostic_Scan_Unload events to represent the scan-out operation. There will also be Product_MISR_Signature events to denote the expect data for each test. See the "Event" section of the *Encounter Test: Reference: Test Pattern Formats* for details on these event types.

When a failing MISR signature is detected in the application of OP-MISR tests, diagnosis of the failure proceeds by switching to the diagnostics mode. In this mode the design is reconfigured so that channel latch values can be scanned out. Encounter Test automatically generates the `Diagnostics_Observe` and `Diagnostics_Return` sequences for purposes of getting to and from the diagnostic test mode. Refer to "Define_Sequence" types in the *Encounter Test: Reference: Test Pattern Formats* for details.

Using On-Product MISR, the test data file can realize a potential reduction of up to 90% for a large design. For tests that include `Diagnostic_Scan_Unload` events for diagnostics, there are corresponding increases to the test data volume. You can limit the amount of

diagnostic test data by specifying a fault coverage threshold (`diagnosticmeasures`) to limit the volume of diagnostic measure data to retain.

On-Product MISR Restrictions

Following restrictions apply to any use of on-product MISRs without masking to prevent X values from propagating into MISRs:

- On-Product MISR is a signature based test methodology similar to LBIST or WRPT. The designs must be configured to prevent X values from propagating into the signature register. Specific options on the test generation process may need to be used to ensure that X sources are prevented (for example, `globalterm=none` should not be specified if the test mode has active (visible to a scan-measurable latch) 3-state bidirectional pins).
- Multiple scan sections are not supported
- Macro test patterns are not supported.
- Interconnect Test is not supported.
- AC Path Delay Test is not supported.
- AC test constraint timings are not supported.

Parallel Processing

The ongoing quest to reduce test generation processing time has resulted in the introduction of the data parallelism approach to the Encounter Test Generation and Fault Simulation applications. Parallel Processing implements the concept of data parallelism, the simultaneous execution of test generation and simulation applications on partitioned groups of faults or patterns. This reduces the elapsed time of the application.

The data set is partitioned either dynamically (during the test generation phase of stored pattern test generation) or statically (at the beginning of the simulation phase).

The set of hosts used for parallel processing can be designated using one of these methods:

- Specifying a list of machines (available via either graphical user interface for the application being run or via command line).
- Using Load Sharing Facility (LSF)* (available command line only, see"Load Sharing Facility (LSF) Support" on page 233).

Encounter Test supports parallel processing for Stored Pattern Test, Weighted Random Pattern Test, Logic Built-In Self Test, and Test Simulation. Refer to the following for conceptual information on these applications:

- "Advanced ATPG Tests" on page 207
- "LBIST Concepts" on page 245
- "Test Simulation Concepts" on page 275
- "Stored Pattern Test Generation Scenario with Parallel Processing" on page 241 for a sample processing flow incorporating Parallel Processing Test Generation.

Refer to "Performing Test Generation/Fault Simulation Tasks Using Parallel Processing" on page 236 for additional details needed to set up and execute applications using parallel processing.

Load Sharing Facility (LSF) Support

Encounter Test supports the use of Load Sharing Facility (LSF) from Platform Computing Corporation** to pick machines for a parallel run. LSF support is available in command line mode only. Entering `application_command -h` on the command line displays the LSF options for the application. Refer to subsequent sections for additional information.

Parallel Stored Pattern Test Generation

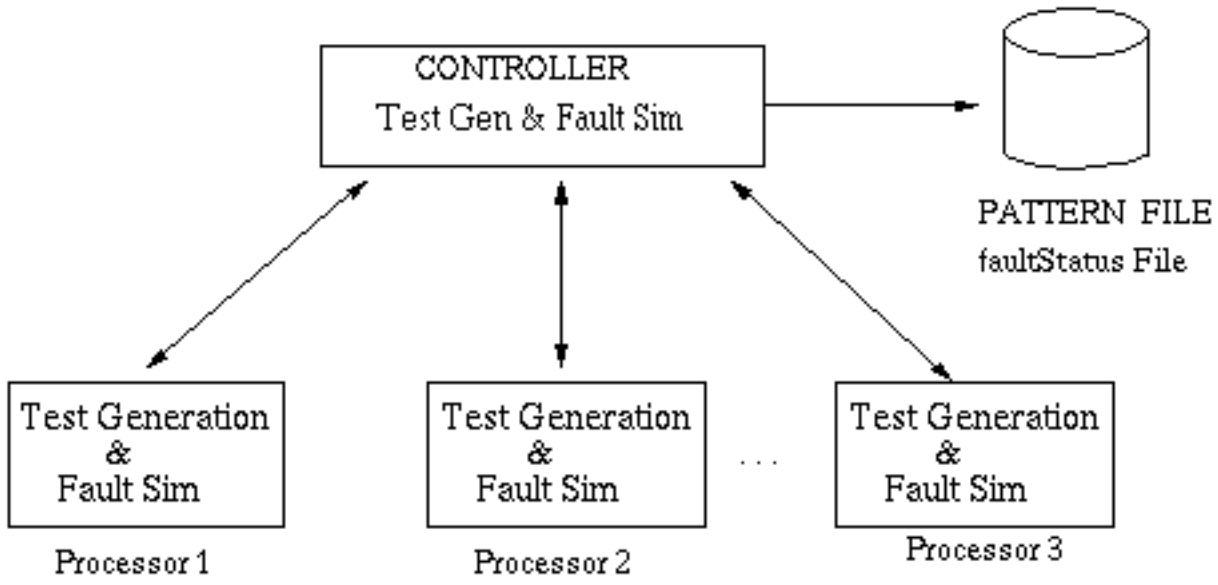
This function is currently available using command line only. Refer to "Parallel Processing Keywords" in the *Encounter Test: Reference: Commands*.

A controller process is started on the machine that the run is started from. Test generation and simulation processes are started on the machines identified by the user. At any given time, only one of the processes will be working. The controller process primarily performs test generation and simulation processes. It performs test generation when it is not busy with simulation and handling communication from the processes. The controller process is responsible for generating the patterns and fault status file.

Test Generation Parallel Processing Flow

An overview of how parallel processing works for stored pattern test generation:

Figure 6-7 Test Generation Parallel Processing



Parallel Simulation of Patterns

Fault Simulation can be performed in parallel mode via graphical user interface for Test Simulation and Manipulate Tests or in the command line environment.

For command line invocation, refer to "Parallel Processing Keywords" in the *Encounter Test: Reference: Commands*.

Parallel processing support is available for simulation of patterns. The following features are supported:

- High Speed Scan and General Purpose simulation (via command line `simulation=hsscan|gp`)
- Simulation of Scan Chain, Driver-Receiver, IDDq and Logic Tests.

To run parallel processing on these commands, add the following options:

- To directly call certain machines - `[hosts=hostname1,hostname2,hostname3...]`
- To call an LSF queue - `queuename=<string>`

An option for `queuename` is `Numslaves=#` that specifies the number of ATPG slave processors on which to run parallel processing.

Currently, it does not support signature-based patterns (WRPT and LBIST patterns). The Fault simulation phase of the algorithm is parallelized.

A controller process is started on the machine that the run is started from. Fault simulation processes are started on the machines identified by the user. The controller process performs good machine simulation and coordination of fault simulation processes. It also produces the patterns and fault status files. The host used to start the application is used to run the controller. The fault simulation phase is run in parallel on the hosts you select. Therefore, if *n* hosts are selected, there are *n* simulation jobs running.

The Good Machine Simulation (in order to write patterns) on the originating host is performed in parallel with the Fault Simulation running on the selected hosts. Therefore, for optimal results, do not select the host that you originate the run from as one of the hosts to run Fault Simulation on.

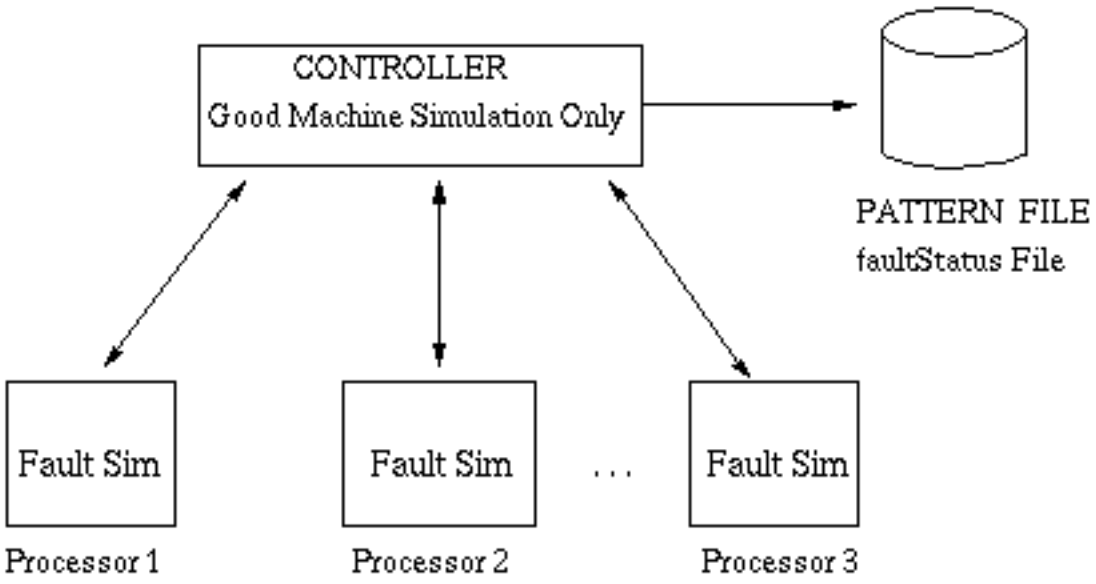
Optimum performance is achieved if the selected machines running in parallel have been dedicated to your run.

In a parallel run, a greater number of patterns may be found effective as compared to a serial run.

Fault Simulation Parallel Processing Flow

An overview of how parallel processing works for fault simulation:

Figure 6-8 Fault Simulation Parallel Processing Flow



Performing Test Generation/Fault Simulation Tasks Using Parallel Processing

Refer to "Parallel Processing" on page 232 for conceptual information.

To perform parallel processing using command line, refer to "Parallel Processing Keywords" in the *Encounter Test: Reference: Commands*.

Restrictions

For Logic Built-In Self Test restrictions, refer to "Restrictions" on page 252

Applicable to Parallel Processing:

- If you are performing simultaneous parallel processing runs, each set of hosts identified for the run can not include the same hosts. If a particular host is used for one parallel run, that host may not be used in another simultaneous parallel run by the same user.
- The resimulation phase of a parallel stored pattern test generation run is not parallelized.
- Multiple test sequences are not supported within a single run for parallel LBIST processing. However, multiple test sequences can be run in the parallel environment by

using the expert option `parallelpattern=no`. This option causes the algorithm to revert to the parallel processing support available with previous releases.

- The parallel processing architecture does not support dynamic distribution of faults This forces a user to run on dedicated machines in order to benefit from parallel processing. It may also produce more patterns.
- General Purpose Simulation is not supported for WRPT and LBIST when running in parallel mode. High Speed Scan Based Simulation should be used.
- LSF machines that access more than one job can cause parallel LSF to terminate. The workaround is to use the `lsfexclusive` option. If you specify `lsfexclusive`, ensure your queue is configured to accept exclusive jobs.

Prerequisite Tasks for LSF

Refer to the following for application-specific prerequisite tasks:

- For *Logic Built-In Self Test*, "Input Files" on page 252
- For *Test Simulation*, "Prerequisite Tasks" on page 271

Ensure the following conditions are met in your execution environment prior to using LSF:

- Ensure that the statements in your `.profile` or profile files for your shell (`.kshrc`, `.cshrc`) are error-free; otherwise you will not be able to run in the parallel environment. The statements for profile-related information must be syntactically correct, i.e. invocations that do not exist or are typed incorrectly will prevent successful setup of the parallel processing environment.
- Ensure that you can rsh to the machines selected for the parallel run from the machine that you originate your run from and be able to write to the directory that contains your part. The following command executed from the machine that you plan to originate your run from will validate this requirement:

  ```
  rsh name_of _machine_selected touch directory_containing_part/name_of
  some_file
  ```

  You should also be able to rsh to the selected host and be able to read the directory (and all it's subdirectories) where the code resides. Execute the following command from the machine you plan to originate your run from in order to verify this:

  ```
  rsh name_of_machine_selected ls directory_where_code_is_installed
  ```

  If these commands fail, please contact your system administrator
- If you are running across cells, either of the following is required:

- The version of rsh you use should inherit tokens obtained via the `klog` command (versus one obtained as a result of `login`).

Notes:

a. If you are starting the run on AIX, verify that your PATH environment variable ensures that the `rsh` command from the AFSRCMDS package will be picked up. This version of `rsh` inherits tokens obtained via klog. This is usually installed in `/usr/afsrcmds/bin`. In addition, ensure that the environment variable `KAUTH` is set to `yes`.

b. `KAUTH` is available only from AIX.

- The `.cshrc` (for C shell users) and `.profile` (for Korn shell users) should obtain tokens for all cells. A suggested method is the following:

a. Ensure that the directory containing your `.profile` or `.cshrc` have read and lookup access.

b. Create an executable file in `/tmp` on all machines you will using that obtains tokens using the `klog` command.

c. Ensure that the file can be read and executed only by you.

d. Put a statement in the `.profile` or `.cshrc` to execute the file.

- Ensure that any directories used by Encounter Test parallel processing (for example part identification parameters `WORKDIR/`) are accessible by all machines chosen to execute in parallel. For example:
  - Ensure the directory where the code is installed is accessible by all machines.
  - All machines selected for the parallel run should be able to access the code installation for their respective platforms.
  - The directory specified by the `TB_PERM_SPACE` and `TB_TEMP_SPACE` environment variables should be accessible by all machines chosen to execute in parallel. Refer to "Common Advanced Keywords" in the *Encounter Test: Reference: Commands* for additional information.
  - Any directories you have manually linked to the part directory should be accessible by all machines chosen to execute in parallel.
- Invoke the `xhost` command to enable machines participating in a parallel run to display on the originating host. Examples of using `xhost` are:
  - `xhost + machine_name` - enables the specified *machine_name* to display on the machine that issues the `xhost` command.

- `xhost + -` enables all machines to display on the machine that issues the `xhost` command.

Maintain awareness of the following:

- Before the parallel run is started, the `.profile` is executed. Environment variables set outside of the `.profile` are not inherited by the parallel run.
- echo statements in the `.profile`, `.kshrc`, `.cshrc` or other files associated with the execution of the shell will cause the parallel run to fail. If you are reading the TERM variable or echoing output in these files, do so conditionally by using the statement `if [tty -s]` around the echo or reading of the TERM variable.
- Any processes which expect a manual response during your logon will cause the parallel processing logon processes to hang.
- Processes such as auto klogs. These processes can cause problems during a parallel run.
- A process called `lamd` is created on every machine participating in the parallel run. Please do not kill this process.
- Do not remove these files in the `/tmp` directory:
  - `lam-userid@`*`hostname`*
  - `lam-userid@hostname -s`
  - `lam-userid@hostname -sio`
- If your parallel run hangs, you can execute the following commands on one of the participating machines to terminate the run:

  `et -c`

  then

  `TPChaltMPI hosts =` *`name1,name2,...`* where *`name1,name2,...`* are the names of the machines participating in the run.
- Run time improvements can be guaranteed only if all machines participating in the parallel run are dedicated to the parallel run. This is because the current algorithm does not perform load balancing.

In addition to the previously stated prerequisites for test generation/simulation applications, ensure the following conditions are met in your execution environment prior to using LSF for parallel processing:

- Ensure the machine you are launching from has an LSF license.

- The PATH environment variable should be set up to find the *bsub* command. Specify this setting in your `.login` file for `.cshrc` users and `.profile` for korn shell.
- VERY IMPORTANT: The user running the LSF job must be allowed to `rsh` into the machines being used while the LSF job is running.

Refer to "Parallel Processing Keywords" in the *Encounter Test: Reference: Commands* for additional information.

Also note that numerous repositories of LSF information exist on the WWW.

Input Files

Refer to the following for application-specific input files:

- For Logic Built-In Self Test, "Input Files" on page 252

Output Files

Refer to the following for application-specific output files:

- For Logic Built-In Self Test, "Output" on page 253
- For Test Simulation, "Output" on page 271

During parallel processing, interim output files are created for each process.

The naming conventions for these interim files are:

- `faultStatus.mode.`*`exper_name_prefix`*`__`*`n`* of *`N`* where *`N`* is the total number of hosts and *`n`* is 1 of *`N`*, 2 of *`N`*, etc.

When the parallel processing ends, these files are combined into individual output file for `faultStatus`.

Temporary Output Files

When using parallel processing, a file named `TPCgetMachine.`*`date.time`* is generated in the home directory during the parallel run. Please do not delete this file unless the GUI or command line executable terminates abnormally. If the event the GUI does terminate abnormally, this file should be manually deleted.

Additional files in the home directory:

- `remotetb.`*`pid.source_string`*
- *`pid`*`.myapp`
- *`pid`*`.mynodes`

These files appear in the `/tmp` directory:

- `lam-userid@`*`hostname`*
- `lam-userid@`*`hostname`* `-s`
- `lam-userid@`*`hostname`* `-sio`
- `./tmp/lamstdout.`*`pid.taskid`* `-` this file contains the output of the child process.

  `pid` is the process id of the child process

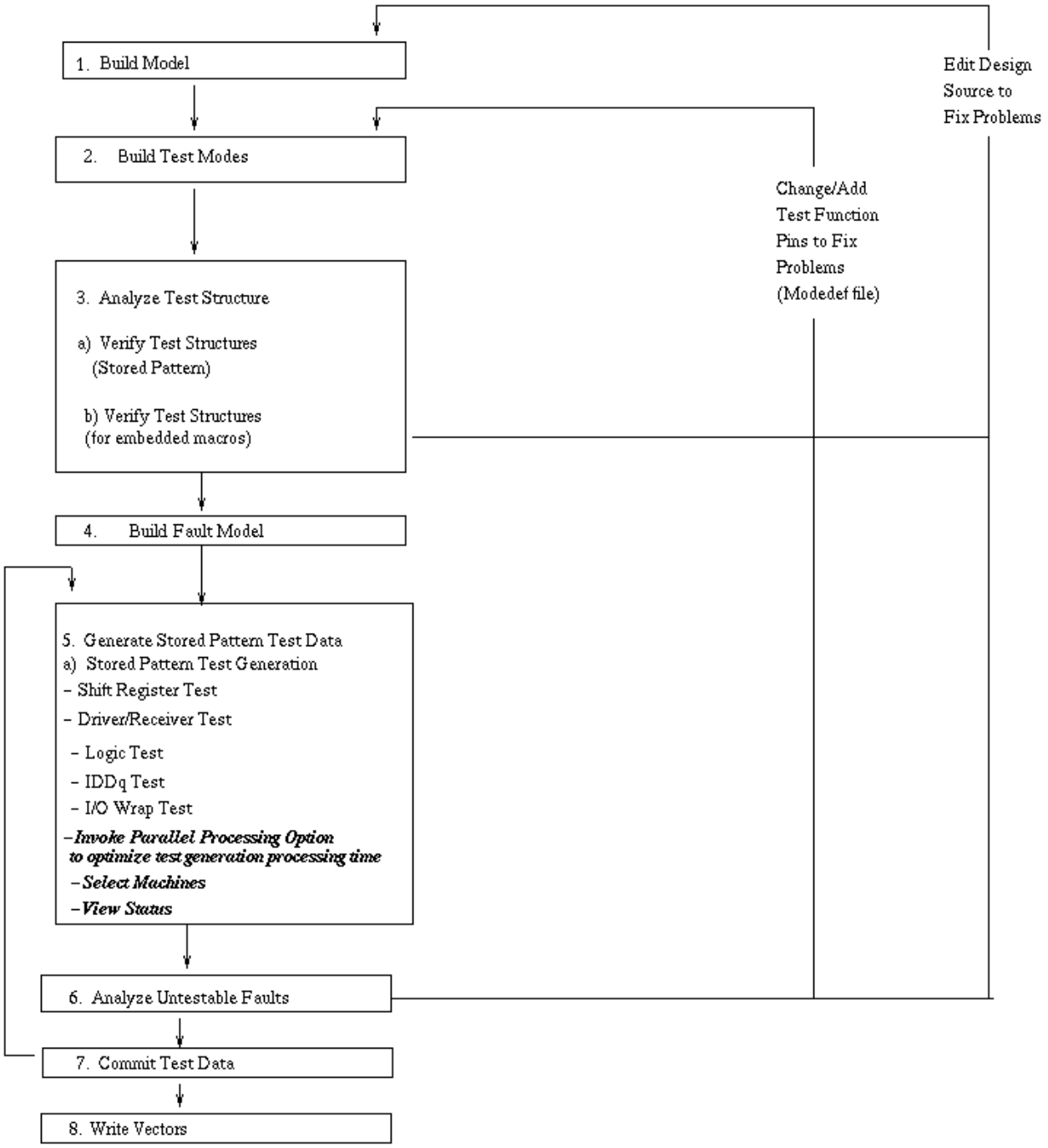
  `taskid` is a number from 1 to *`n`* where *`n`* is the number of hosts selected.

Note: If you manually kill the run, you may have to manually delete these files.

Stored Pattern Test Generation Scenario with Parallel Processing

The following figure shows a typical processing flow for Test Generation incorporating Parallel Processing.

Test Generation Process with Parallel Processing



1. Build Model

For complete information on *Build Model*, refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models.*

2. Build Test Mode

   A sample mode definition file for this methodology follows:

```
TDR=stored_pattern_tdr_name
SCAN TYPE=LSSD  (or SCAN TYPE=GSD) BOUNDARY=NO IN=PI OUT=PO;
TEST_TYPES=STATIC LOGIC SIGNATURES=NO,STATIC MACRO,
                  IDDQ PROPAGATE CELL_BOUNDARY,DRIVER_RECEIVER,SCAN_CHAIN;
FAULTS=DYNAMIC;
TEST_FUNCTION_PIN_ATTRIBUTES=MLE_FLAG;
ASSIGN PIN XYZ=-SC;
```

   For complete information, refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes.*

3. Analyze Test Structure

   a. Logic Test Structure Verification (TSV)

      For complete information, refer to "Logic Test Structure Verification (TSV)" in the *Encounter Test: Guide 3: Test Structures*.

   b. Verify Core Isolation

      For complete information, refer to "Verify Core Isolation" in the *Encounter Test: Guide 3: Test Structures*.

4. Build a Fault Model

   For complete information, refer to "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

5. Stored Pattern Test (Scan Chain, Logic, IDDq, Driver and Receiver, I/O Wrap)

   For complete information, refer to "Advanced ATPG Tests" on page 207.

6. Analyze Untestable Faults

   For complete information, refer to "Deterministic Fault Analysis" in *Encounter Test: Guide 4: Faults*.

7. Commit Tests

   For complete information, refer to "Utilities and Test Vector Data" on page 245.

8. Export Test Data.

   For complete information, refer to "Writing and Reporting Test Data" on page 181.

Note: To create the most compact pattern set for manufacturing, resimulate the final parallel patterns, or use uni-processor test generation for your final pass.

7

Logic Built-In Self Test (LBIST) Generation

This chapter discusses the Encounter Test support for Logic Built-in Self Test (LBIST).

LBIST: An Overview

Logic Built-In Self Test (LBIST) is a mechanism designed into the design which allows the design to effectively test itself. Encounter Test supports the design LBIST approach called STUMPS (Self Test Using MISR and Parallel SRSG). A STUMPS design contains linear feedback shift registers (LFSRs) which implement a Pseudo-Random Pattern Generator (PRPG), sometimes referred to a Shift Register Sequence Generator or SRSG, to generate the (random) test vectors to be applied to a scan design. It also includes an LFSR which implements a Multiple Input Signature Register (MISR) to collect the scanned response data.

Refer to "Modeling Self Test Structures" in the *Encounter Test: Reference: Legacy Functions* for additional information.

This application can generate tests for either/both of the following test types based on user controls:

- scan chain tests - refer to "Scan Chain Tests" on page 28
- logic tests - refer to "Logic Tests" on page 29
  - Logic tests are done using manually entered test sequences.

LBIST Concepts

Logic Built-In Self Test (LBIST) is a mechanism implemented into a design to allow a design to effectively test itself. LBIST is most often used after a design has been assembled into a product for power-on test or field diagnosis. However, some or all of the on-board LBIST circuitry may also be used when testing the design stand-alone using a general-purpose logic

tester. Design primary inputs and primary outputs may be connected to pseudo-random pattern generators and signature analyzers or controlled by a boundary scan technique. The tester may exercise considerable control over the execution of the test, unlike the situation in a “pure” LBIST environment where the design is in near-total control. This is being pointed out here to emphasize that:

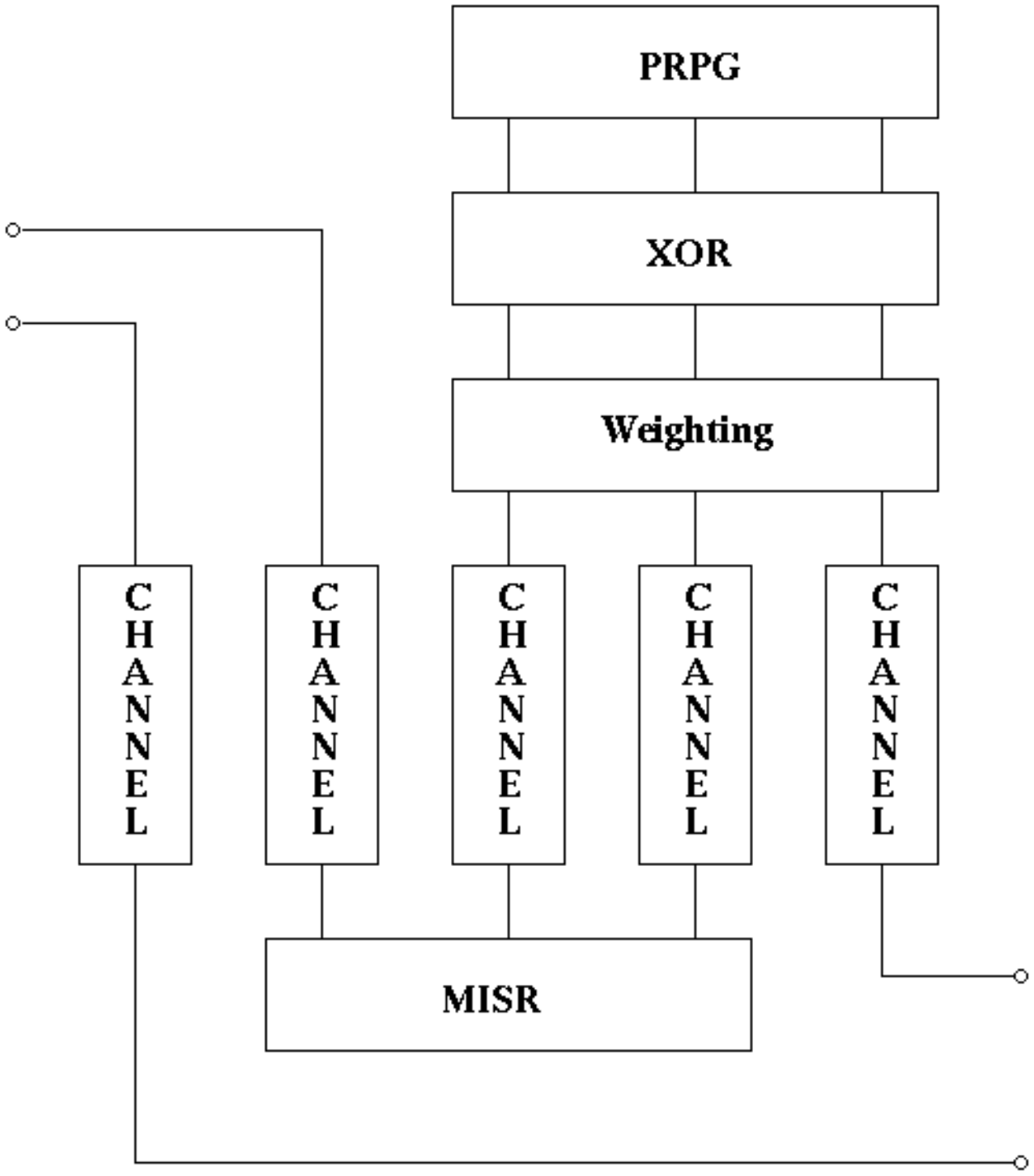
- Encounter Test does not require LBIST pattern generators and signature analyzers to reside in a design.
- Encounter Test assumes that the LBIST operations are controlled through primary input signals.

Encounter Test supports the STUMPS style of LBIST. STUMPS is a compound acronym for Self Test Using MISR and Parallel SRSG. MISR is an acronym for Multiple Input Signature Register. SRSG stands for Shift Register Sequence Generator. In Encounter Test documentation, we call an SRSG a pseudo-random pattern generator (PRPG).

STUMPS follows scan guidelines for either LSSD or GSD. STUMPS scan chains are often called channels. The basic layout of the STUMPS registers is illustrated in Figure 7-1.

During a scan operation, the channels are completely “unloaded” into the MISR and replenished with pseudo-random values from the PRPG. The scan operation consists of a number of scan cycles equal to or larger than the number of bits in the longest channel. Each scan cycle advances the PRPG by one step, the MISR by one step, and moves the data down the channels by one bit position. Between scan operations, a normal LSSD or GSD test is performed by pulsing the appropriate clocks that cause outputs of the combinational logic between channels to be observed by the channel latches (and possibly the primary outputs).

Figure 7-1 A Simplified Example STUMPS Configuration



STUMPS requires a means of initializing the LFSRs and a means of reading out the contents of the signature registers. The initializing sequence is supplied to Encounter Test by the customer. Refer to "SEQUENCE_DEFINITION" in the *Encounter Test: Guide 2: Testmodes* for more detailed information about initialization sequences. The only support for reading out the signature registers provided by Encounter Test is to verify signature registers are observable by a parent test mode's scan operation.

The specification of a parent mode for scanning also allows you to alter the initial state of some latches from one LBIST test generation run to another. Having once verified that the initialization sequence works, Encounter Test is able to re-specify the latch states within the initialization sequence when the sequence is copied into the test pattern data. No fixed-value latches are allowed to be changed in this manner, because altering their initial states would invalidate the TSI and TSV results.

The control logic that steers the LBIST operation must be modeled so that to Encounter Test it appears that all clock signals and any other dynamic signals emanate from primary inputs or pseudo primary inputs; control signals that are constant throughout the LBIST operation may be fed by either primary inputs or fixed-value latches.

Encounter Test provides limited support for the initialization of RAM by non-random patterns. The test mode initialization sequence may initialize embedded RAM either by explicit, non-randomized patterns or, if the RAM supports it, by the use of a reset signal. Encounter Test determines through simulation which RAMs are initialized; any uninitialized RAMs are treated as X-generators for LBIST, and therefore they must be blocked from affecting any observation point.

Additional features of STUMPS processing include:

- Fast Forward

  This feature speeds up the application of tests by skipping tests that do not detect any new faults. Since the determination of which tests detect new faults is based upon the cycling of the PRPG through all the tests, the fast forward operation must be able to preserve the original PRPG sequence while skipping some of the scan operations. To do this, it makes the PRPG state at the beginning of each scan operation to be one shift cycle advanced from the PRPG state at the beginning of the previous test. The LBIST implementation of this (which is slightly different from WRPT Fast Forward) saves the PRPG into a "shadow register" during the scan operation. At the initial fault simulation of the LBIST patterns, it is assumed that the PRPG is saved after one cycle of the scan operation. When the reduced test set is applied, the saving may actually occur after m+1 cycles, where m is the number of consecutive tests to be skipped. To apply the next "effective" test, the PRPG is restored from the shadow register prior to the scan operation. Thus, the state of the PRPG does not depend upon whether the previous originally simulated scan operation was applied.

In the general case, you supply two input pattern sequences in support of fast forward: A PRPGSAVE sequence that moves the PRPG seed into the shadow register and a PRPGRESTORE sequence that moves the seed from the shadow register into the PRPG.

Alternatively, Encounter Test supports a specific implementation of the PRPG save and restore operations whereby they occur concurrently with a channel scan cycle under control of PV (PRPG save) and PR (PRPG restore) signals. When this implementation is used, these signals must be identified by test function pin attributes and the PRPGSAVE and PRPGRESTORE sequences are implicitly defined. For convenience, we refer to the support of user-specified PRPGSAVE and PRPGRESTORE sequences as “fast forward with sequences” and we refer to the support of PV and PR pins as “fast forward with pins”. If the PV and PR pins are defined and user sequences are supplied also, then “fast forward with sequences” takes precedence; in this case the PV and PR pins are held to their inactive states throughout the application of test data, much like TIs.

“Fast forward with sequences” assumes that the PRPGSAVE sequence does not disturb any latches except the shadow register and that the PRPGRESTORE sequence does not disturb any latches except the PRPG. This of course implies that these operations can not be executed concurrently with a channel scan cycle, but must be inserted (between scan cycles in the case of the PRPGSAVE sequence).

There is no corresponding shadow register for MISRs. Skipping some scan cycles changes the signature, and there is no way to avoid this fact. When fast forward is used, two sets of MISR signatures are computed.

The PRPG shadow register must, like the PRPG itself, not be contaminated during self test operations. The PRPGs are checked to verify that in the test mode, they have no other function. To ensure that the PRPG shadow register is not corruptible, it must be modeled as fixed value latches, which means that the TI inputs for this test mode prohibit the shadow register latches from being changed.

The PRPGSAVE and PRPGRESTORE sequences will necessarily violate some TI inputs. This is okay because Encounter Test does not simulate these sequences except when checking to ensure that they work properly. TSV ensures that the PRPGSAVE and PRPGRESTORE sequences work and that they do not disrupt any latches other than the PRPG and its shadow latches.

In “fast forward with pins” the channels and MISR shift one cycle concurrently with the restore operation, and the restore is executed on the last cycle of the scan. This works only if the PRPG is never observed between scans; either the existence of explicit A, B, or E shift clocks in the test sequence or a path from the PRPG to anything other than a channel input would make it infeasible to use fast forward with pins. This is because it would then be impossible to predict the effect of some tests being skipped; the PRPG state would depend upon how many subsequent tests are to be skipped, and this

information is not available until a fault simulation is performed, but the fault simulation is not possible until the PRPG state is determined.

- Channel input signal weighting

  Channel input signal weighting improves the test coverage from LBIST. A multiple-input AND or OR design is allowed to exist between the PRPG and a channel input. The signal weight is determined by the selection of the logic function and the number of pseudo-random inputs to it. This selection is made by control signals fed from primary inputs. These controls are not allowed to change during the scan operation, and so every latch within a given channel will be weighted the same, varying only as to polarity which depends upon the number of inversions between the channel input and the latch. The primary inputs that are used to control the channel input weight selection must be flagged with the test function WS (for weight selection).

  Weighting logic also may be fed by Test Inhibit (TI) or Fixed Value Linehold (FLH) latches. FLH latches may be changed from one experiment to another, causing channels to be weighted differently. The FLH latch values may be changed by adding them as lineholds to the lineholds file.

- Channel scan that considers the presence of a pipelined PRPG spreading network in LBIST test modes

  The pipelined spreading PRPG network between PRPGs and channel latches enables simultaneous calculation of spreader function and weight function in one scan cycle. The sequential logic in the PRPG spreading network can be processed by placing a Channel Input (`CHI`) test function in the design source or the test mode definition file. Encounter Test uses the `CHI` to identify PRPG spreading pipeline latches during creation of the LBIST test mode.

Encounter Test LBIST does not require scan chains to be connected to primary pins or on-board PRPG and MISR. For example, you may have some scan chains connected to scan data primary inputs and scan data primary outputs, other scan chains connected to PRPG(s) and MISR(s), other scan chains connected to scan data primary inputs and MISR(s), and other scan chains connected to PRPG(s) and scan data primary outputs, all on the same part, all in the same test mode. The scan data primary inputs and outputs, if used, must connect to tester PRPGs and SISRs (single-input signature registers). Encounter Test assumes that all the scan chains are scannable simultaneously, in parallel.

See "Task Flow for Logic Built-In Self Test (LBIST)" on page 257 for the processing flow.

Performing Logic Built-In Self Test (LBIST) Generation

To perform Logic Built-In Self Test (LBIST) using the graphical interface, refer to "Create LBIST Tests" in the *Encounter Test: Reference: GUI*.

To perform Logic Built-In Self Test (LBIST) using command lines, refer to "create_lbist_tests" in the *Encounter Test: Reference: Commands*.

An Encounter True-Time Advanced license is required to run Create LBIST Tests. Refer to "Encounter Test and Diagnostics Product License Configuration" in *What's New for Encounter® Test and Diagnostics* for details on the licensing structure.

The syntax for the `create_lbist_tests` command is given below:

```
create_lbist_tests workdir=<directory> testmode=<modename> testsequence=<name>
```

where:

- `workdir` = name of the working directory
- `testmode`= name of the LBIST testmode
- `testsequence` = name of the test sequence being used (optional but generally specified)

The commonly used keywords for the `create_lbist_tests` command are given below:

- `extraprpgcycle=#`

  Indicates the number of times PRPGs are shifted by a clock or a pulse events in the test sequence. If using parallel simulation by setting `forceparallelsim=yes`, also set `extraprpgcycle` to accurately simulate the test sequence.

- `extramisrcycle=#`

  Indicates the number of times the MISRs are shifted by a clock or a pulse events in the test sequence. If using parallel simulation by setting `forceparallelsim=yes`, also set `extramisrcycle` to accurately simulate the test sequence.

- `prpginitchannel=no|yes`

  Specifies whether the channels need to be initialized by the PRPG before the first test sequence.

- `reportmisrmastersignatures=yes/reportprpgmastersignatures=yes`

  Reporting options to print signature and channel data at specific times.

Refer to "create_lbist_tests" in the *Encounter Test: Reference: Commands* for information on all the keywords available for the command.

Restrictions

- General Purpose Simulation is not supported for LBIST when running in parallel mode. High speed scan-based simulation should be used instead.
- If stored-pattern (or OPMISR) tests are coexistent on the same `TBDbin` file with WRPT or LBIST data, then resimulation of this `TBDbin` cannot be accomplished in a single Test Simulation (`analyze_vectors`) pass. The selection of test types being processed is controlled by the `channelsim` parameter. `Channelsim=no` (the default) will process all stored-pattern (and OPMISR) tests; `channelsim=yes` will process all WRPT and LBIST tests.
- Support for multiple oscillators works only in cases where the oscillators are controlling independent sections of logic that are not communicating with each other. In some cases, it may be possible to use Encounter Test support of multiple oscillators if the two domains are operating asynchronously and the communication is one-way only.
- Within a test sequence definition, lineholds can be specified only on primary inputs and fixed value latches (FLH).
- The `useppis=no` option in the Good Machine Delay Simulator is not guaranteed to work unless scan sequences have been fully expanded.

Input Files

User Input Files

- Linehold file

  This user-created file is an optional input to test generation. It specifies design PIs, latches, and nets to be held to specific values for each test that is generated. See "Linehold File" on page 171 in the for more information.

- `Testsequence`

  This file contains initialization sequences. This file needs to be read into Encounter Test using the read_sequence_definition command.

- Seed File

  This user-created file specifies unique starting seeds per run for on-board PRPGs and MISRs. For more information, refer to "Seed File" on page 253.

Output

The following is a sample output log displaying the coverage achieved for a specific clocking sequence:

```
---------------- Sequence Usage Statistics ---------------------
 Sequence_Name     EffCycle   TotCycle    DC_%    AC_%
================================================================
<userSequence01>   10         512         69.03   0.00
<< cross reference for user defined sequences >>  <userSequence01>:Test1
Starting Fault Imply processing... fault status may be updated.
----------------------------------------------------------------
                        LBIST Statistics
----------------------------------------------------------------
Logic Test Results
     Patterns simulated     :512
     Effective patterns     :10
Static fault coverage (DC)  :69.0323%
Dynamic fault coverage (AC)  :0%
```

Seed File

The purpose of this file is to allow the specification of unique starting seeds per run for on-board PRPGs and MISRs. This allows flexibility in specifying the initial values for floating latches and scannable latches.

The file uses `TBDpatt` format statements as follows:

```
TBDpatt_Format (mode=type, model_entity_form=nameform);

Event Scan_Load(): latchvalues;

[ SeqDef=(seqname," ")] SeqDef ;
```

where,

`type` is node or vector

`nameform` is name, `hier_index`, or `flat_index`

`latchvalues` represents the specification of the latch initial values in either `TBDpatt` node list format or vector format, as specified by the `TBDpatt_Format` statement.

The `[SeqDef` statement is optional, and is used primarily when multiple `Scan_Load` events (seeds) are specified. `seqname` is the name of a test sequence definition that has already been imported

For more information about `TBDpatt` syntax, refer to the *Encounter Test: Reference: Test Pattern Formats* manual.

The specification of an initial value to any latch that is not scannable in the parent mode or is a fixed value latch in the target mode is ignored. Encounter Test does not modify the initial value of any latches that are not scannable in the parent mode, and overriding the initial value of fixed values is not allowed because it invalidates processing done by Test Mode Build and Test Structure Verification.

If so desired, multiple seeds (`Scan_Load` events) can be specified. When there are multiple seeds, or when there is only one seed and it has an associated `[SeqDef` statement, each seed is used as the starting PRPG state for only one test sequence. At the end of the specified number of test iterations, a final signature is provided; if there are additional seeds, then the design is reinitialized with the new PRPG seed and additional test iterations are applied from that starting state using a different (or possibly the same) test sequence as indicated by the associated `[SeqDef`. When multiple seeds are specified, any `Scan_Load` event with no associated `[SeqDef` will be applied using the first test sequence named in the create_wrp_tests `testsequence` parameter list.

Following is an example:

Figure 7-2 Example of a Seed File

```
TBDpatt_Format (mode=node, model_entity_form=name);
Event Scan_Load():  "Block.f.l.lfsr.nl.prpg.014" = 0
    "Block.f.l.lfsr.nl.prpg.015" = 0
    "Block.f.l.lfsr.nl.prpg.016" = 0
    "Block.f.l.lfsr.nl.prpg.017" = 0
    "Block.f.l.lfsr.nl.prpg.018" = 0
    "Block.f.l.lfsr.nl.prpg.019" = 0
    "Block.f.l.lfsr.nl.prpg.020" = 0
    "Block.f.l.lfsr.nl.prpg.021" = 0
    "Block.f.l.lfsr.nl.prpg.022" = 0
    "Block.f.l.lfsr.nl.prpg.023" = 0
    "Block.f.l.lfsr.nl.prpg.024" = 0;
```

A seed file can also have multiple seeds, as shown in the following example.

Figure 7-3 Example of a Multiple Seed File

```
TBDpatt_Format (mode=node, model_entity_form=name);

Event          Scan_Load ():  "Block.f.l.BF.nl.PRPG.PRPG0"=0
                               "Block.f.l.BF.nl.PRPG.PRPG1"=0
                               "Block.f.l.BF.nl.PRPG.PRPG2"=1
                               "Block.f.l.BF.nl.CH2.L2"=0
                               "Block.f.l.BF.nl.CH10.L2"=1
                               "Block.f.l.BF.nl.CH11.L2"=1
                               "Block.f.l.BF.nl.CH12.L2"=0
                               "Block.f.l.BF.nl.CH13.L2"=1
                               "Block.f.l.BF.nl.CH1.L2"=0
                               "Block.f.l.BF.nl.MISR.misr.MISR1"=1
                               "Block.f.l.BF.nl.MISR.misr.MISRO"=1 ;
                                    [ SeqDef=(test1," ") &rbrl. SeqDef;
Event 1  Scan_Load ():   "Block.f.l.BF.nl.PRPG.PRPG0"=0
                               "Block.f.l.BF.nl.PRPG.PRPG1"=1
                               "Block.f.l.BF.nl.PRPG.PRPG2"=0
                               "Block.f.l.BF.nl.CH2.L2"=0
                               "Block.f.l.BF.nl.CH10.L2"=1
                               "Block.f.l.BF.nl.CH11.L2"=1
                               "Block.f.l.BF.nl.CH12.L2"=0
                               "Block.f.l.BF.nl.CH13.L2"=1
                               "Block.f.l.BF.nl.CH1.L2"=0
                               "Block.f.l.BF.nl.MISR.misr.MISR1"=1
                               "Block.f.l.BF.nl.MISR.misr.MISRO"=1 ;
                                     [ SeqDef=(test2," ") ] SeqDef;
```

Note that the " " associated with the `SeqDef` is enclose an optional time/date stamp. The quotation marks are required even if you choose to leave out the date/time stamp.

Parallel LBIST

This function is currently available using graphical user interface and command line. Optimum performance is achieved if the selected machines running in parallel have been dedicated to your run. In a parallel run, a greater number of patterns will be found effective as compared to a serial run.

For command line invocation, `TWTmainpar -h` displays the available options for specifying a list of machines.

Note: For LBIST, General Purpose Simulation is not supported when running in parallel mode. High Speed Scan Based Simulation should be used.

The host used to start the application acts as the coordinator for the parallel process. As a default, the run is made in two phases. In the first phase, the faults are partitioned across the selected hosts. The controller performs Good Machine Simulation only to write out the patterns.

If, for any given simulation interval, the Good Machine Simulation time exceeds the fault simulation time, the run is switched to the second phase. In the second phase, the patterns are partitioned across all of the processes (including the controller) and each process works on all of the faults.

Figure 7-4 on page 256 and Figure 7-5 on page 257 parallel process.

Figure 7-4 LBIST Parallel Processing Flow, Phase 1

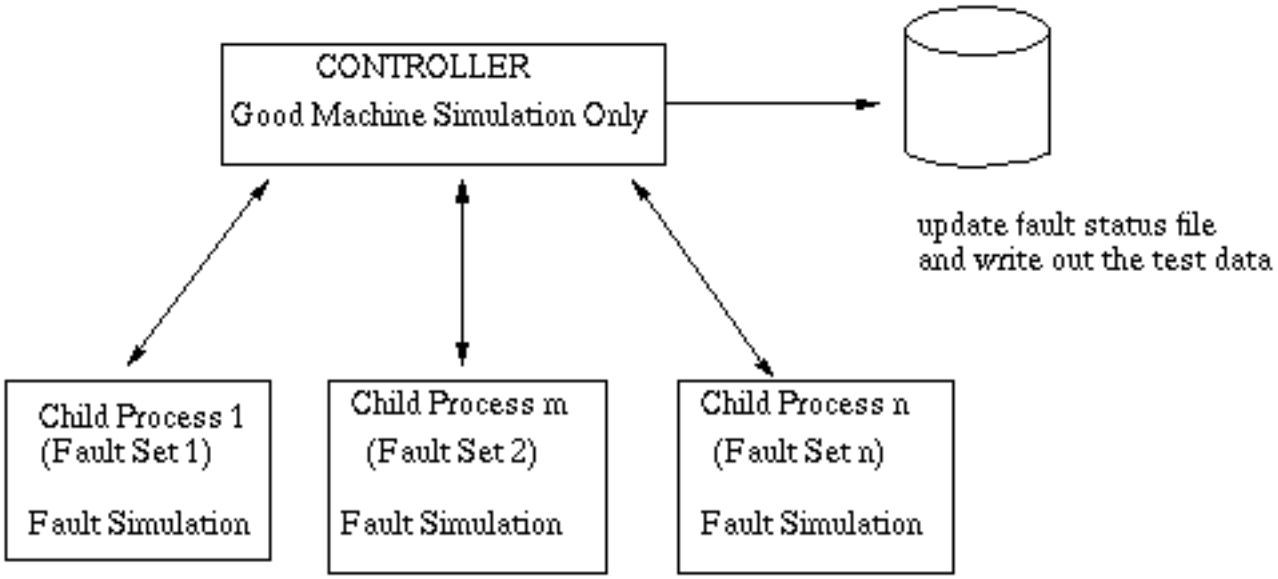
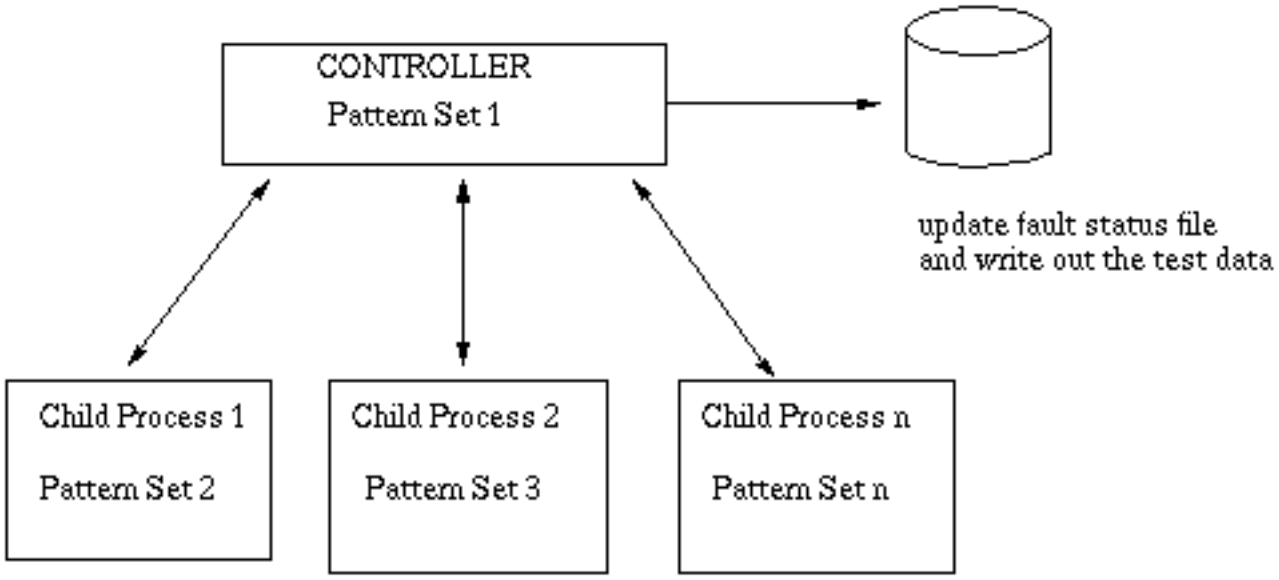


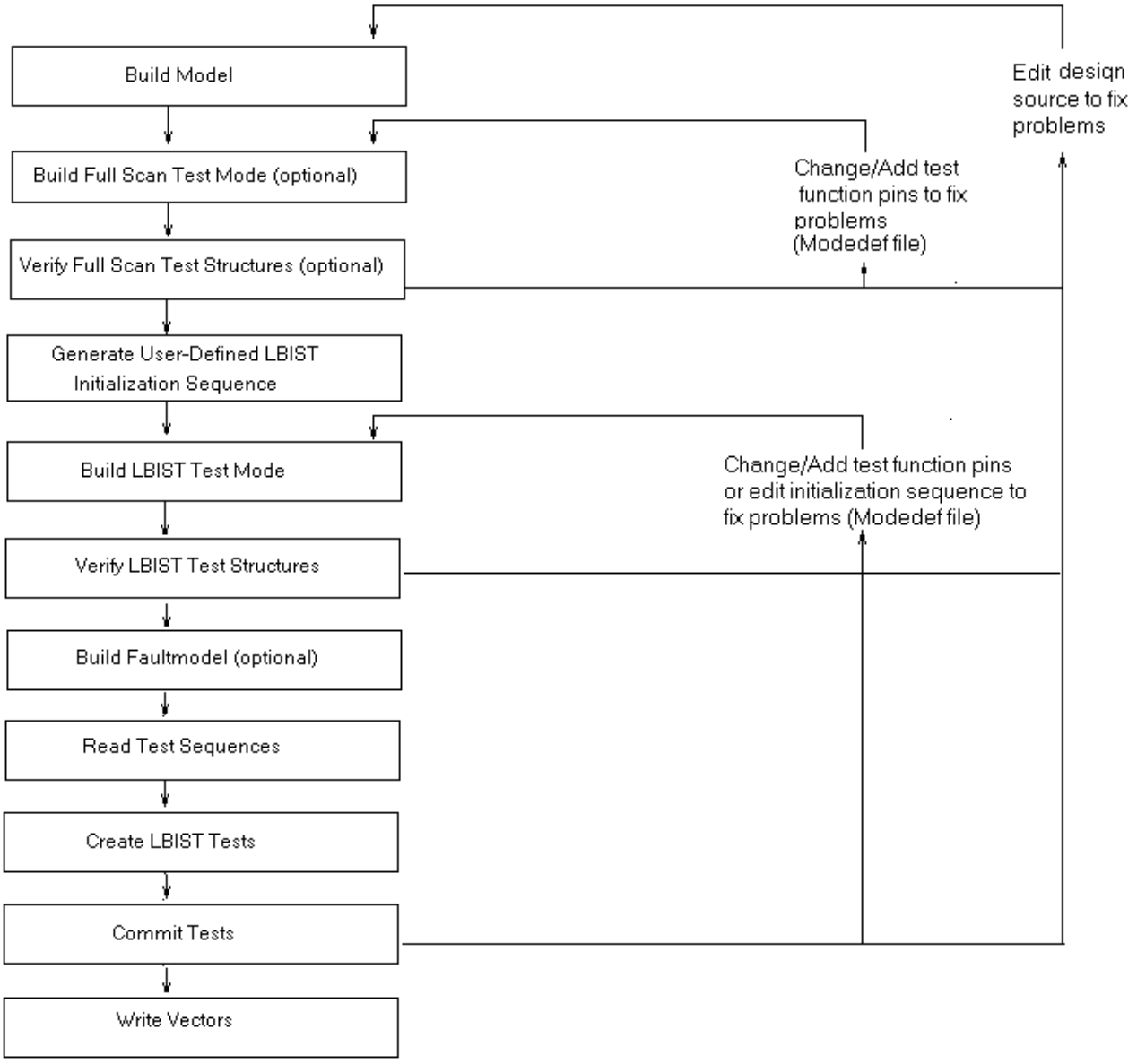
Figure 7-5 LBIST Parallel Processing Flow, Phase 2



Task Flow for Logic Built-In Self Test (LBIST)

The following figure shows a typical processing flow for running Logic Built-In Self Test.

Figure 7-6 Encounter Test Logic Built-In Self Test Processing Flow



1. Build an Encounter Test model. Refer to "Performing Build Model" in the *Encounter Test: Guide 1: Models* for more information.
2. Build a full scan test mode. Refer to "Performing Build Test Mode" in the *Encounter Test: Guide 2: Testmodes* for additional information.

   A sample mode definition file is given below:

```
TDR=lbist_tdr_name
SCAN_TYPE=LSSD BOUNDARY=NO IN=PI OUT=PO;
TEST_TYPES=STATIC LOGIC SIGNATURES=NO,STATIC MACRO,SCAN_CHAIN;
TEST_FUNCTION_PIN_ATTRIBUTES=TF_FLAG;
    .
    .
    .
```

3. It is recommended that you perform Test Structure Verification (TSV) to verify the conformance of a design to the Encounter Test LSSD guidelines or the GSD guidelines. Non conformance to these guidelines may result in poor test coverage or invalid test data. Refer to "Verify Test Structures" in the *Encounter Test: Guide 3: Test Structures*.

4. Generate initialization sequence.

   The LBIST methodology requires a user-defined initialization sequence for the test mode to initialize the on-board BIST elements such as PRPG, MISR, and fixed-value latches. Encounter Test does not automatically identify how to get these latches initialized; in fact, it is only the results of the initialization sequence that tell Encounter Test which state each fixed-value latch is supposed to be set to. Figure "An Example Test Mode Initialization Sequence in TBDpatt Format" in the *Encounter Test: Guide 2: Testmodes* shows an example mode initialization sequence definition for an LBIST test mode.

   If the LBIST testing uses fast forward with sequences, you must also generate the PRPGSAVE and PRPGRESTORE sequences at this time. Include these sequence definitions in the same file with the mode initialization sequence.

   LBIST processing also requires sequence definitions to specify the order of the clocks and other events which get applied for each test. Each test generation run may use a different sequence of clocks. These *test* sequence definitions may be included in the mode initialization sequence input file, or they may be imported separately. An example is given below.

5. Build LBIST test mode. The SCAN_TYPE must be LSSD or GSD.

   Following is a sample mode definition file for LBIST:

```
Tester_Description_Rule = dummy_tester;
scan type = gsd boundary=internal length=1024 in = on_board out = on_board;
test_types static logic signatures only shift_register;
faults static ;
comet opcglbist markoff LBIST stats_only ;
misr Net.f.l.top.nl.BIST_MODULE.BIST_MISR.MISRREG[1048]=(0,2,21,23,53);
misr Net.f.l.top.nl.BIST_MODULE.BIST_MISR.MISRREG[1089]=(0,2,21,23,41);
prpg Net.f.l.top.nl.BIST_MODULE.BIST_PRPG.PRPGREG[52]=(1,2,6,53);
prpg Net.f.l.top.nl.BIST_MODULE.BIST_PRPG.PRPGREG[105]=(1,2,6,53);
```

6. Perform the TSV checks for LBIST to verify conformance to the LBIST guidelines. Refer to "Performing Verify Test Structures" in the *Encounter Test: Guide 3: Test Structures* for more information.

7. Build a fault model for the design. Refer to the "Building a Fault Model" in the *Encounter Test: Guide 4: Faults* for more information.

8. To use user-defined clock sequences, read the test sequence definitions. See "Coding Test Sequences" on page 180 for an explanation of how to manually create test (clock) sequences.

   For complete information, refer to "Reading Test Data and Sequence Definitions" on page 257.

   The following example uses internal clocking (cutpoints) for the LBIST sequence:

```
TBDpatt_Format (mode=node, model_entity_form=name);
[Define_Sequence Universal_Test (test);
[ Pattern 1.1; # Set Test Constraints to pre-scan values    Event 1.1.1   Stim_PPI ():
BIST_MODULE.BIST_SCAN"=0;
] Pattern 1.1;
[ Pattern 1.2;
Event 1.2.1   Pulse_PPI ():"BIST_MODULE.BIST_CASCADE[7]"=+;
] Pattern 1.2;
[ Pattern 1.3;
Event 1.3.1   Pulse_PPI ():"BIST_MODULE.BIST_CASCADE[14]"=+;
] Pattern 1.3;
[ Pattern 1.4;   # Set Test Constraints to pre-scan values
Event 1.4.1   Stim_PPI ():"BIST_MODULE.BIST_SCAN"=1;
] Pattern 1.4;
[ Pattern 1.5;
Event Channel_Scan ();
] Pattern 1.5;
] Define_Sequence Universal_Test;
```

9. Create LBIST Tests

   For complete information, refer to "LBIST Concepts" on page 245.

10. Commit Tests

    For complete information, refer to "Utilities and Test Vector Data" on page 245.

11. Write Vectors

    For complete information, refer to "Writing and Reporting Test Data" on page 181.

Debugging LBIST Structures

To ensure correct implementation of BIST, it is a common practice to simulate the BIST controller and verify that the control signals for scanning and clocking are produced in the proper sequence. However, if the BIST controller has been automatically inserted by Encounter Test, you may consider this to be unnecessary. Regardless of the origin of the BIST controller design and its level of integrity, other things may go wrong in the BIST process. For example, some system clocks may be incorrectly generated or incorrectly wired to their associated memory elements.

Encounter Test's Verify Test Structures tool is designed to identify many such design problems so they can be eliminated before proceeding to test generation. Even so, it is advisable to use your logic simulator of choice to simulate the BIST operation on your design for at least a few test iterations (patterns) and compare the resulting signature with the signature produced by Encounter Test's Logic Built-In Self Test generation tool for the same number of test iterations. This simulation, along with the checking offered by Encounter Test tools, provides high confidence that the signature is correct and that the test coverage obtained from Encounter Test's fault simulator (if used) is valid.

When the signatures from a functional logic simulator and Encounter Test's LBIST tool do not match, the reason will not be apparent. It can be tedious and technically challenging to identify the corrective action required. The problem may be in the BIST logic, its interconnection with the user logic, or in the Encounter Test controls. The purpose of this section is to explain the use of signature debug features provided with Encounter Test's Logic Built-In Self Test generation tool.

Prepare the Design

Follow the normal steps for running Logic Built-In Self Test generation, up to, but not including the test generation step. Refer to "Task Flow for Logic Built-In Self Test (LBIST)" on page 257.

Check for Matching Signatures

It is not necessary to run the full number of test iterations to attain a high confidence that your LBIST design is implemented properly and Encounter Test is processing it correctly. In fact, the functional logic simulation run, against which you will compare Encounter Test's signature, might be prohibitively expensive if you were to compare the final signatures after several thousand test iterations. It is recommended that you run a few hundred or a few thousand test iterations, or whatever amount is feasible with your functional logic simulator.

Submit a Logic Built-In Self Test generation run, specifying the chosen number of test iterations (called ".patterns". in the control parameters for the tool). You will need to obtain the MISR signatures; this can be done in any of three ways:

1. Request ".scope ". data from the test generation run: `simulation=gp watchpatterns=range watchnets=misrnetlist` where range is one of the valid `watchpatterns` options and `misrnetlist` is any valid `watchnets` option that includes all the MISR positions.
2. Specify `reportmisrsignatures=yes|no` in the test generation run, or
3. After the test generation run, export the test data and look at the `TBDpatt` file.

In the first method, you will use *View Vectors* to look at the test generation results as signal waveforms. Refer to "Test Data Display" in the *Encounter Test: Reference: GUI* for details on viewing signal waveforms. This may seem the most natural if you are used to this common technique for debugging logic. However, you may find it more convenient to have the MISR states in the form of bit strings when comparing the results with your functional logic simulator.

In both cases, MISR signatures are produced at every "`.detection interval.`". Signatures are printed in hexadecimal, and are read from left to right. The leftmost bit in the signature is the state of MISR register position 1. (The direction of the MISR shift is from low- to high-numbered bits with feedback from the high-numbered bit(s) to the low-numbered bits.) Signatures are padded on the right with zeroes to a four-byte boundary, so there are trailing zeroes in most signatures which should be ignored.

The MISR latch values found in the signatures are manually compared with the results of the functional logic simulator, often by reading a timing chart.

Find the First Failing Test

If the final signatures do not match, it is necessary to find the first mismatching, or "failing" signature. Make another debug run with the signature interval to set to 1, so as to narrow down the problem to the exact failing test iteration. If you find the output log is becoming inconveniently large, you can restrict the number of "patterns"; (test iterations) to stop after the first known failing signature. Results of this run should narrow the problem down to the first failing test iteration.

Diagnosing the Problem

The next step in finding the cause of mismatching signatures between the functional logic simulator and Encounter Test depends upon the symptoms.

Verify Primary Input Waveforms

If the signatures match for the first signature interval and then fail later on, several potential causes can be eliminated, and the problem is likely to be found in the system clocking, or "release-capture" phase of the test. You may have to figure out which channel latch is failing and backtrace from there to find the problem, but first you should look carefully at the clock sequence that Encounter Test is simulating to make sure it agrees with the functional logic simulation input. If there is no obvious discrepancy from looking at the `TBDpatt` form of the sequence definition, use the waveform display to compare the waveforms with the timing chart from the functional logic simulator. Start by comparing the waveforms at the clock primary inputs. If this does not provide any clues, select some representative clock splitter design and compare the waveforms at the clock splitter outputs.

Find a Failing Latch

As Encounter Test signatures can be observed either by looking at the signal waveforms or by having them printed in the output log, so the channel latch states can also be observed by either of these methods. Using the response compaction of the MISR, you can find the failing channel latch by means of a binary search instead of having to compare the simulation states of thousands upon thousands of channel latches. The binary search technique requires that the Encounter Test simulation of the channel scan be expanded in some fashion.

Scoping the Channel Scan

Using a waveform display is not the recommended way of finding the failing channel latch, but some users may be more comfortable doing it this way. Once the failing channel latch is identified, the waveform display may prove invaluable in the next step of the diagnosis of a failing signature.

Another debug test generation run must be made to generate the waveform data for the channel scan. From the command line, you would use `create_lbist_tests simulation=gp watchpatterns=n watchnets=scan watchscan=yes` along with any other `create_lbist_tests` parameters necessary for your design (such as `testsequence`), where *`n`* is the number of the failing test iteration. Note that when a single test iteration is specified on the `watchpatterns` parameter, `create_lbist_tests` will expand the channel scan for that test iteration so the detailed waveforms can be generated. Refer to "Test Data Display" in the *Encounter Test: Reference: GUI* for details on viewing signal waveforms.

Using the LBIST Debug Reporting Options

You may find it more convenient, when locating the failing latch, to use the debug reporting options instead of the waveform display. Instead of, or in addition to, the BIST parameters listed in the previous section, specify `create_lbist_tests reportlatches= n1:n2` to obtain MISR signatures for each iteration of the test from iteration *n1* through iteration *n2*. Note that it is not necessary to specify `simulation=gp` to use the `printlatches` option. This reporting option also produces the states of all the channel latches in addition to the MISR values. The information is printed to the log.

Once the failing test iteration is identified, if there appears to be a mismatch in the scan operation, similar debug printout can be obtained for each scan cycle by specifying only a single test iteration: `lbist reportlatches=`*n*.

A

Three-State Contention Processing

This appendix describes the uses of Encounter Test test generation and simulation keywords that control processing and reporting of three-state contention.

The following keywords are directly related to three-state contention:

- `contentionreport` identifies the specified type of the contention of interest and also instructs the simulator as to the type of contention to report via error messages.
- `contentionprevent` indicates whether the test generator is to try to prevent the type of the contention of interest (based on `contentionreport`) from occurring in the generated patterns. If `contentionprevent=yes` and the test generator produces a pattern and is unable to protect it, it will try to generate a different pattern. With `contentionprevent=yes`, there are fewer patterns (theoretically zero) discarded out by the simulator thus achieving higher coverage without having to perform additional passes.
- `contentionremove` indicates whether the simulator is to discard patterns that cause the type of contention of interest (based on `contentionreport`).
- `contentionmessage` indicates how many contention messages are to be printed.

The following are some combinations of the keywords and resulting actions:

- `contentionreport=soft contentionmessage=all contentionremove=yes contentionprevent=yes` (all the defaults)

  The test generator will prevent soft (known vs. X) and hard (0 vs. 1)contention from occurring in the patterns it creates. If it cannot create a pattern for the fault without causing contention, it will not create a pattern for that fault. There are no "contention messages" issued from this process. Theoretically, the simulator will not find any patterns to remove; BUT, just in case a burnable pattern gets through, the simulator is there to ensure it doesn't get into the final set of vectors. If the test generator successfully eliminates contention from all patterns, no contention messages are produced.

- `contentionreport=soft contentionmessage=all contentionremove=yes contentionprevent=no`

When specifying `contentionprevent=no`, the test generator will not ensure that the pattern does not cause contention. The simulator will remove any patterns that would cause soft or hard contention. Messages are produced AND, you will see messages for patterns discarded due to contention. The final vectors will not include burnable patterns.

- `contentionreport=soft contentionmessage=all contentionremove=no contentionprevent=yes`

  The test generator will prevent hard and soft contention from occurring in the patterns it creates. If it cannot create a pattern for the fault without causing contention, it will not create a pattern for that fault. There are no "contention messages" issued from this process. Theoretically, the simulator will not find any patterns to report; however, if a burnable pattern gets through, the simulator will report it; but it will not remove it since `contentionremove=no` was specified. If the test generator allows a burnable pattern to get through, contention messages are produced and the final vectors will include the burnable vectors.

- `contentionreport=soft contentionmessage=all contentionremove=no contentionprevent=no`

  When specifying `contentionprevent=no`, the test generator will not ensure that the pattern doesn't cause contention. The simulator will report the patterns that cause hard or soft contention, however, because `contentionremove=no` was specified, the simulator will not discard them; if you see any contention messages with these settings, the final vectors include patterns that may burn.

The following table lists *contention* keyword combinations and their respective results.

contentionreport	contentionremove	contentionprevent	contentionmessage	Log Messages	Sim Removes Contention	TG Removes Contention	Contention in Final Vectors
hard, soft, all	yes	yes	greater than 0	only if sim removes contention	should not have anything to remove	yes	no

contentionreport	contentionremove	contentionprevent	contentionmessage	Log Messages	Sim Removes Contention	TG Removes Contention	Contention in Final Vectors
hard, soft, all	yes	no	greater than 0	yes	yes	no	no
hard, soft, all	no	yes	greater than 0	no	no	yes	Should not be
hard, soft, all	no	no	greater than 0	yes	no	no	yes if reported
none	yes	yes	greater than 0	no	no	hard only (contention prevent= yes wins over contentionr eport =none)	Should not be hard contention
none	no	no	greater than 0	no	no	no	Could be: not reported, prevented or removed.

Note: If `contentionmessage=0` then no messages will be printed; but the rest of the processing is the same.

Index

Numerics

C

D

E

F

H

I

T

U

V

W