

Encounter Test: Flow: MBIST Analysis

Product Version 12.1.101
February 2013

© 2003–2012 Cadence Design Systems, Inc. All rights reserved.

Portions © IBM Corporation, the Trustees of Indiana University, University of Notre Dame, the Ohio State University, Larry Wall. Used by permission.

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.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

Contents

<u>Preface</u>	7
<u>About This Manual</u>	7
<u>Additional References</u>	7
<u>Encounter Test Documentation Roadmap</u>	8
<u>Getting Help for Encounter Test and Diagnostics</u>	9
<u>Contacting Customer Service</u>	9
<u>Encounter Test And Diagnostics Licenses</u>	10
<u>Using Encounter Test Contrib Scripts</u>	10
<u>Typographic and Syntax Conventions</u>	10
<u>What We Changed for This Edition</u>	10

1

<u>Validation After Memory BIST Insertion</u>	11
<u>MBIST Insertion Messages</u>	11
<u>MBIST Post-insertion Analysis Interface</u>	11
<u>Static Timing Analysis</u>	11
<u>Boolean Equivalence Checking</u>	12
<u>Design Verification</u>	12
<u>Encounter Test 1149 Structure Verification</u>	12
<u>Manufacturing Test Results Analysis</u>	13

2

<u>Static Timing Analysis</u>	15
<u>How MBIST Affects Static Timing in a Design</u>	15
<u>Guiding Timing Constraint Generation for MBIST</u>	17
<u>MBIST Module-level Timing</u>	18
<u>Design Timing with MBIST</u>	18
<u>Functional Timing Constraints when Timed in Functional Mode</u>	19

Encounter Test: Flow: MBIST Analysis

<u>MBIST Timing Constraints when Timed in MBIST-specific Mode</u>	19
<u>Non-MBIST Timing Constraints when Timed in MBIST-specific Mode</u>	20

3

Boolean Equivalence Checking

<u>Methodology</u>	21
<u>Block Level Insertion Flow</u>	22
<u>Chip Level Insertion Flows</u>	22
<u>Multiple Block Merging Flows</u>	23

4

Design Verification

<u>Getting to Encounter Test MBIST Pattern Simulation</u>	27
<u>Simulation Scripts</u>	29
<u>MBIST Patterns</u>	30
<u>ROM Data Load Files</u>	30
<u>Design and Technology Libraries</u>	31
<u>Analyzing Pattern Class Simulation Results</u>	31
<u>irun to analyze_embedded_test Flow</u>	32
<u>irun to CPP to analyze_embedded_test Flow</u>	33
<u>irun to FDS to CPP to analyze_embedded_test Flow</u>	34
<u>Special Handling of 4 pin TAP controller in Simulation</u>	35
<u>Special Handling of Split Retention Patterns in Simulation</u>	36
<u>Simulation Miscompares</u>	36
<u>Injecting Memory Faults</u>	38
<u>Methodology</u>	38
<u>Changes in the Verilog Memory Models</u>	38
<u>Fault Injection Control Files</u>	39
<u>Using analyze_embedded_test to Analyze Simulation Results with Fault Injection</u> ..	41
<u>Understanding analyze_embedded_test Results</u>	43
<u>Executing the Command</u>	44
<u>Working with Simulation Logs</u>	44
<u>Working with CPP Data</u>	48
<u>Analyzing and Debugging MBIST Simulation Results</u>	52
<u>Simulation Environment</u>	52

Encounter Test: Flow: MBIST Analysis

<u>MBIST Startup</u>	56
<u>Memory Connections and Activity</u>	68
<u>MBIST Comparators</u>	73
<u>MBIST Completion</u>	75
<u>Determining Miscomparing Registers from Simulation Logs</u>	78

5

<u>Manufacturing Test Support</u>	81
<u>FDS to CPP to analyze embedded test Flow</u>	81
<u>Using the prepare_memory_failset Command</u>	82
<u>Basic Keywords</u>	83
<u>Commands invoked by prepare_memory_failset</u>	86
<u>Outputs</u>	87
<u>prepare_memory_failset command line examples</u>	91
<u>Common messages and solutions</u>	91
.....	92

Encounter Test: Flow: MBIST Analysis

Preface

About This Manual

This manual describes the necessary analyses after Encounter® memory built-in self-test has been inserted within a design using Design For Test in Encounter® RTL Compiler. The description includes the interface and action required to support static timing analysis, boolean equivalence checking, and design verification through simulation of testbenches generated using Encounter® Test. The manual also covers manipulation of manufacturing test data results relative to memory built-in self-test using Encounter Test applications.

Additional References

This manual is the third in a set which apply to Encounter memory built-in self-test. The manuals, available within the Cadence product documentation, are listed in the order in which they most naturally would be referenced.

- Design For Test in Encounter RTL Compiler, Inserting Memory Built-In Self-Test Logic chapter

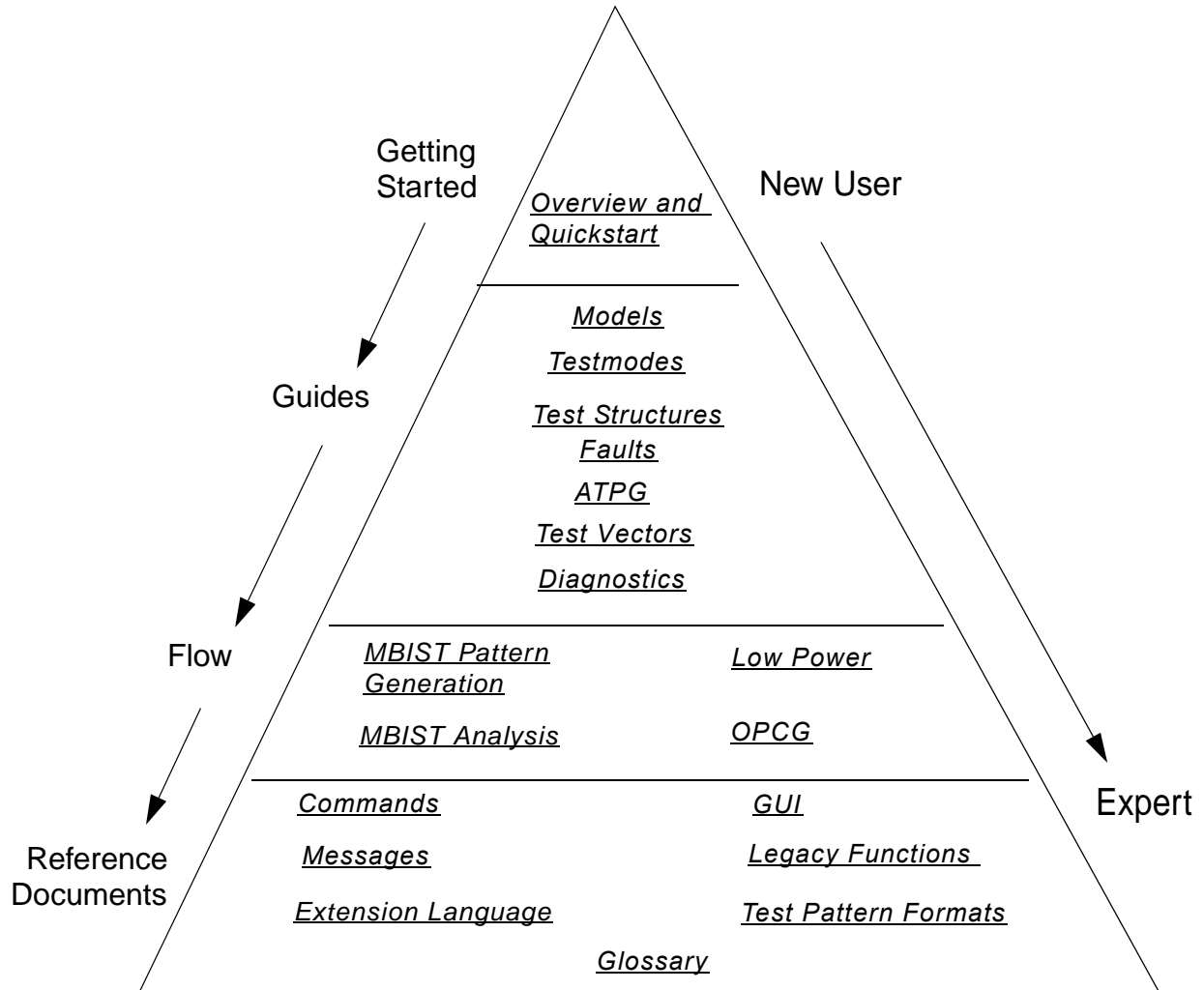
Analyzing the design and planning for memory test, along with insertion of the memory built-in self-test structures are covered in this manual.
- Encounter Test Memory Built-In Self-Test Pattern Generation Guide

All aspects of pattern generation for design verification and manufacturing test are included in this user guide.
- Encounter Memory Built-In Self-Test Analysis Guide

After insertion of the memory built-in self-test logic, accuracy of the function and its impacts on the design must be considered. This manual describes the analysis required for timing closure, boolean equivalence checking, design verification, and post-pricessing of manufacturing test results.

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` and press `Enter`. The installed Cadence documentation set is displayed.
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 *Graphical User Interface Reference* 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.

Encounter Test And Diagnostics Licenses

Refer to “Encounter Test and Diagnostics Product License Configuration” in *What’s New for Encounter Test and Diagnostics* 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.

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 `analyze_embedded_test -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 results files are stored.

- 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 for the model.

What We Changed for This Edition

There are no significant changes for this version of the document.

Validation After Memory BIST Insertion

MBIST Insertion Messages

Memory built-in self-test (MBIST) logic is inserted into a design within an RTL Compiler session. During the insertion command, `insert_dft mbist`, errors and warnings may occur. Errors generally result in early termination of the command. Warnings are issued when the insertion process can continue, but unusual or unexpected conditions may be encountered which may have an adverse effect on the subsequent processing steps. After insertion is complete, a set of rules are checked to ensure operational control of the inserted MBIST logic. This is typically done by running `check_mbist_rules` as part of the `insert_dft mbist` command. However, you can also run `check_mbist_rules` standalone after `insert_dft mbist` completion. Warning messages may also be issued by `check_mbist_rules`. All MBIST related warning messages should be examined the first time MBIST is inserted, but not all of them are equally important.

MBIST Post-insertion Analysis Interface

Static Timing Analysis

Static timing is potentially handled at two levels, MBIST module and design.

MBIST Modules

If `dont_map` is not specified on the `insert_dft mbist` command line, the command attempts to close timing at the required clock frequencies at the MBIST module level. Constraints are automatically created and applied with MBIST clock period information coming from the relevant `define_dft mbist_clock` data.

These constraints are discarded after use unless `insert_dft mbist -debug` is specified, in which case the command writes a constraints file for each synthesized MBIST module to the `directory` specified on the command line.

Encounter Test: Flow: MBIST Analysis

Validation After Memory BIST Insertion

Target Design

MBIST logic can be timed in a functional or specific MBIST timing mode. If timing is performed in a functional mode, `insert_dft mbist` should not create additional constraints: `no mode` and `no notmode` are specified on the `insert_dft mbist` command line. If `insert_dft mbist -mode` is specified, constraints are placed into the specified timing mode for MBIST. If `notmode` is specified, `insert_dft mbist` updates the specified timing mode(s) to hide the MBIST logic from static timing analysis.

`write_sdc -mode name` can be used to write the content of the MBIST timing mode to file. `read_sdc -mode name` is used to restore the timing constraints for this mode in an RTL Compiler session.

Boolean Equivalence Checking

For all the MBIST insertion flows, the various inserted MBIST features, and the variations in memory module ports, boolean equivalence constraints are automatically created during `insert_dft mbist` execution using mostly `define_dft formal_verification_constraint` commands. These constraints hide the MBIST logic in the post-insertion design during the formal verification process to ensure equivalence with the pre-insertion design.

The `write_do_lec` command is used to write the design boolean equivalence constraints to a file containing Encounter Conformal Logic Equivalence Checking commands.

Design Verification

MBIST verification patterns are created within Encounter Test using the `create_embedded_test` command and `write_vectors` to write Verilog testbench files embedding the design in which MBIST was inserted as the unit targeted for test.

The shortest path to generate these testbenches uses the RTL compiler command `write_mbist_testbench`. This command calls Encounter Test under the covers to generate the simulation testbenches and create one or more scripts to invoke Incisive Enterprise simulation to perform the verification. The `irun.simscrip.*` simulation scripts are found in the directory specified with `write_mbist_testbench`.

Encounter Test 1149 Structure Verification

The implementation of the IEEE1149 test data registers within the MBIST logic relies upon sharing the functional registers and use of a Shift-DR enabled path to both load and unload

Encounter Test: Flow: MBIST Analysis

Validation After Memory BIST Insertion

information within these registers. Consequently, the IEEE1149 clock (TCK) is logically or'ed with the functional register clocks within the MBIST engine using appropriate clock gating logic and this creates a situation where the IEEE1149 specification is not strictly followed. A mode initialization sequence to reset the functional clocks is a required input to the test mode built for the Encounter Test `verify_11491_boundary` command when testing the 1149 instructions used to access MBIST.

The `write_et_mbist -bsv` command is used to create the scripts required to test IEEE1149 logic when MBIST is inserted into the design.

Manufacturing Test Results Analysis

Manufacturing test results usually take the form of chip pin miscompares from expected values at particular tester cycles. This memory test failure information must be translated to a common intermediate form called Chip-Pad-Pattern (CPP) data prior to analysis by Encounter Test command `analyze_embedded_test`.

Encounter Test command `FDSlog2CPP` translates certain tester failure logs to CPP data formats for further processing. Encounter Test command `prepare_memory_failset` can accept memory test failure logs directly and populate a directory structure amenable to visualization software used to handle memory failure analysis.

Encounter Test: Flow: MBIST Analysis

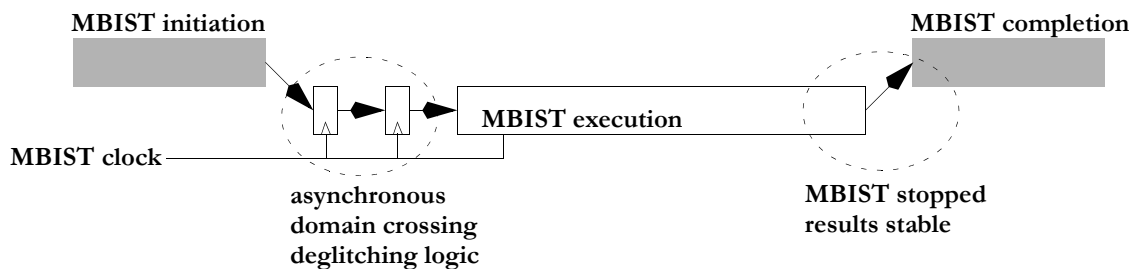
Validation After Memory BIST Insertion

Static Timing Analysis

How MBIST Affects Static Timing in a Design

Once MBIST is started, it operates in isolation as a set of closed-loop systems. Each MBIST engine drives controls, addresses, and data to its target memories which feed memory data to their MBIST comparators that in turn send results to their MBIST engine. The initiation of the MBIST operation utilizes asynchronous domain crossing hardware to avoid the need to statically time the control transfer across domains. This applies to the JTAG initiation sequence as well as the direct access initiation sequence. Upon completion of MBIST execution, results are again transferred safely back to the clock domain which initiated the operation, JTAG or direct access.

Figure 2-1 MBIST Timing Domain Crossings



During MBIST initiation, an asynchronous reset of the targeted MBIST logic occurs. When MBIST is controlled by JTAG, `create_embedded_test` builds this reset into the generated patterns either using the TRST port, if available, or forcing a synchronous reset of the TAP controller through manipulation of the JTAG finite state machine. In either case, the `jtag_reset` signal into the MBIST logic is used for this purpose. If the direct access interface is used to initiate MBIST, the asynchronous reset pulse is created within the MBIST logic. This occurs during the initiation sequence between the second and third clock pulses

Encounter Test: Flow: MBIST Analysis

Static Timing Analysis

after activating the direct access signal, which can either be the `mbist_poweron_run` or `mbist_burnin_run` input to the MBIST engine.

The domain crossing in [Figure 2-1](#) on page 15 occurs after this asynchronous reset within the MBIST logic. For JTAG-initiated operations, the domain crossing occurs when the MBIST run instruction is loaded into the TAP and the Run-Idle state is entered, asserting the `jtag_runidle` input on the MBIST engine. For direct access controlled MBIST, it occurs at the third clock pulse of the `mbist_burnin_run` or `mbist_poweron_run` signal being asserted.

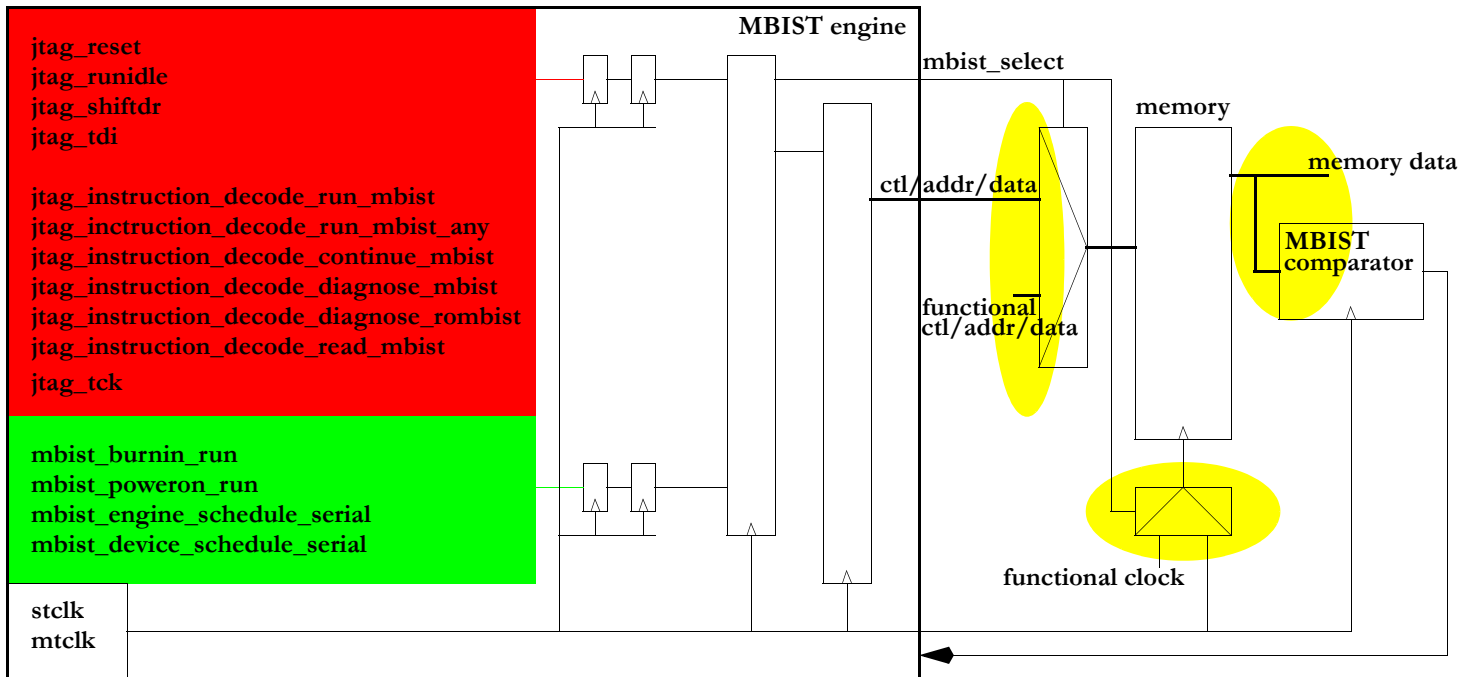
Once the frequency domain control transfer has occurred, the `mbist_select` and `mbist_selectn` (inverted `mbist_select`) lines are asserted from the activated MBIST engines. This causes the MBIST control, address, and data paths to the target memories to be selected, whether through external multiplexors or multiplexors internal to the memory devices themselves. At this point, the internal MBIST clock gates and the memory clock gates are also enabled. The MBIST finite state machine is started and four MBIST clock periods later the first MBIST operation is sent to the target memories. As a result, the `mbist_select[n]` signals need not be timed.

The static timing problem for MBIST is therefore reduced to one of closing timing within the MBIST closed loop systems and avoiding any undesired *interference* across the boundaries with the functional logic. At completion of MBIST execution, the engines are idled and results remain stable for inspection by the initiating control mechanism.

Encounter Test: Flow: MBIST Analysis

Static Timing Analysis

Figure 2-2 MBIST Timing Cross-Domain Interference



Areas of potential interference between functional timing and MBIST timing occur all around the memory devices. These include the control, address, and data input paths to the memories, possibly the clock input path(s), and the data propagation paths from the memories. After establishing the MBIST operational mode, the functional inputs to the target memories terminate at the input of the multiplexors as the `mbist_select[n]` signals are stable. The same claim can be made regarding the clock input path to the memories whether the clock is multiplexed with an MBIST derived clock or simply gated via an MBIST controlled clock gating cell. The memory outputs, however, fanout to functional paths as well as MBIST comparators and may show up in MBIST timing analysis.

Guiding Timing Constraint Generation for MBIST

When the MBIST logic is inserted, the `insert_dft mbist` command generates timing constraints for the design. If MBIST is to be timed in the same mode as the functional logic, no timing constraints are generated. It is assumed that all MBIST logic is timed using functional clocks which drive the MBIST.

Encounter Test: Flow: MBIST Analysis

Static Timing Analysis

If MBIST is to be timed in a separate timing mode, you should first create a timing mode using `create_mode -name mbist_mode`. Then, supply this timing mode to the MBIST insertion command as `insert_dft mbist -mode mbist_mode...` so that the command will create the necessary constraints within the timing mode. It is also possible to isolate the MBIST logic such that it has no effect on non-MBIST timing modes. You can do this automatically at the time of MBIST insertion by specifying the “non-MBIST modes” as `insert_dft mbist -mode mbist_mode -notmode {non-MBIST timing mode list}...`

MBIST Module-level Timing

As each MBIST module is synthesized during `insert_dft mbist`, the timing constraints are established by the command based on the frequency specified for the clock driving that MBIST logic. The designer has no control at this level. If the user selects RTL insertion or chooses to write out MBIST logic that is not mapped by `insert_dft mbist`, then he may apply constraints during later synthesis steps to the previously generated MBIST logic.

`insert_dft mbist` produces a summary of the synthesis timing step for each module indicating whether timing closed at the specified frequency. The following message in the RTL Compiler log indicates that synthesis is starting for the specified module:

```
Info      : Embedded test macro targeted to run at specified frequency. [MBIST-20]
           : Macro: temmbist1plrwd_a0
           : Frequency: 100.0 MHz
```

At completion of the synthesis step for the module, the following message appears in the RTL Compiler log if timing closed successfully:

```
Info      : Synthesis/Timing completed successfully for module. [MBIST-16]
           : Module: temmbist1plrwd_a0
           : Clock frequency: 100.0 MHz
```

If timing closure was not successful during the module synthesis step, a message is issued in the RTL Compiler log indicating this status along with the frequency at which the module could close timing.

Design Timing with MBIST

Design-level timing constraints created by `insert_dft mbist` are added to a timing mode based on various sources of the information:

- Clocks are created in the MBIST timing mode based upon the `define_dft mbist_clock` commands.

Encounter Test: Flow: MBIST Analysis

Static Timing Analysis

- MBIST DFT configuration mode signals are set to values in the MBIST timing mode based upon the applicable `define_dft dft_configuration_mode`, `define_dft test_mode`, and `define_dft shift_enable` commands.
- The balance of the constraints are applied at the boundaries of the inserted MBIST engines in the MBIST timing mode and notmodes.

This approach allows the generation of the timing constraints by `insert_dft mbist` to be independent of the MBIST insertion flow chosen: RTL or gate level; top-down or bottom-up; block or chip. MBIST timing constraint generation is independent of all other timing modes and makes no attempts to constrain them in any way.

The MBIST impact to various constraints may vary and can be classified as:

- Functional timing constraints when timed in functional mode
- MBIST timing constraints when timed in MBIST-specific mode
- Non-MBIST timing constraints when timed in MBIST-specific mode

Functional Timing Constraints when Timed in Functional Mode

`insert_dft mbist` must be executed without specification of the `mode` keyword.

No constraints are added by MBIST insertion when MBIST is timed as a part of the functional mode. It is assumed that all clocks and DFT configuration signals are established by the user and MBIST timing paths are viewed simply as additional functional paths to and from the targeted memories.

MBIST Timing Constraints when Timed in MBIST-specific Mode

`insert_dft mbist -mode mbist_mode...` must be executed. The `mbist_mode` must have been defined previously by a `create_mode -name mbist_mode` command.

The constraints are added to the `mbist_mode` using the following guidelines.

- `create_clock` for each `define_dft mbist_clock` command, using the name, `hookup_period`, and `hookup_pin` values

`set_clock_uncertainty -setup (.05 * hookup_period)` for each `define_dft mbist_clock` command

`set_clock_uncertainty -hold (.03 * hookup_period)` for each `define_dft mbist_clock` command

Encounter Test: Flow: MBIST Analysis

Static Timing Analysis

- `set_case_analysis` to values necessary to enable MBIST operations for each of the relevant test signals defined by `define_dft test_mode`, `define_dft shift_enable`, and used by `insert_dft mbist -dft_configuration_mode mode...`

- For each MBIST engine inserted:

```
set_case_analysis 0 [get_pins mbist_engine_instance/jtag_tck]
set_case_analysis 0 [get_pins mbist_engine_instance/jtag_reset]
set_case_analysis 1 [get_pins mbist_engine_instance/mbist_select]
set_case_analysis 0 [get_pins mbist_engine_instance/mbist_selectn]
```

Non-MBIST Timing Constraints when Timed in MBIST-specific Mode

`insert_dft mbist -mode mbist_mode -notmode mode_name_list...` must be executed. The `mbist_mode` and all modes in the `mode_name_list` must have been defined previously by a `create_mode` command.

The constraints are added to the modes of the `mode_name_list` using the following guidelines, effectively causing all MBIST related logic to be ignored in each specified timing mode.

- For each MBIST engine inserted:

```
set_case_analysis 0 [get_pins mbist_engine_instance/stclk]
set_case_analysis 0 [get_pins mbist_engine_instance/mtclk]
```

Boolean Equivalence Checking

Various MBIST insertion flows exist within RTL Compiler. Memory devices may have variations in ports. Some include ports for access exclusively by MBIST and test-related scan chains. The selection of the original design for the basis of comparison and the amount of DFT logic inserted may vary as one performs equivalence checking under different circumstances. Each of these factors can cause variations in the constraints necessary to satisfy boolean equivalence checking. The standard flows and implicit support within `insert_dft mbist` are documented in this chapter.

Methodology

The philosophy in all flows is to make the inserted MBIST logic hidden to the boolean equivalence checking. This effort includes the MBIST logic itself, the extra design hierarchy pins added as a result of propagating connections with MBIST, and the change in functionality resulting from MBIST connections at the memory boundaries. The approach taken by the `insert_dft mbist` command assumes that:

- All relevant DFT structures are inserted into the design prior to any checking.
- All relevant DFT structures are inserted successfully.
- The original design, prior to any DFT insertion in this RTL Compiler session, is used as the base (golden) design for comparison. Note that within the [Multiple Block Merging Flows](#) on page 23 the merged blocks either contain the previously inserted memory built-in self-test structures and ports in the original (golden) design or are blackboxed in the original design with the previously inserted memory built-in self-test ports.

Designer modifications to the constraints may be necessary when deviating from this approach. The details of the `insert_dft mbist` generated constraints are included below for insertion into various design levels.

Encounter Test: Flow: MBIST Analysis

Boolean Equivalence Checking

Block Level Insertion Flow

Within RTL compiler, `insert_dft mbist` defines a test mode for the `jtag_reset` port at the block level.

Resulting LEC dofile contribution is:

```
add pin constraints 1 jtag_reset -revised
```

If direct access to MBIST is implemented, additional constraints are necessary for the ports defined with `define_dft mbist_direct_access`.

Resulting LEC dofile potential contributions are as follows:

```
add pin constraints inactive-state burnin_run-port -golden
add pin constraints inactive-state poweron_run-port -golden
add pin constraints inactive-state engine_schedule_serial-port -golden
add pin constraints inactive-state device_schedule_serial-port -golden

add pin constraints inactive-state burnin_run-port -revised
add pin constraints inactive-state poweron_run-port -revised
add pin constraints inactive-state engine_schedule_serial-port -revised
add pin constraints inactive-state device_schedule_serial-port -revised

// when monitor connected to previously unconnected port
add ignored output monitor-port-hookup -golden
add ignored output monitor-port-hookup -revised
```

Test-wrapped memories are those memory devices which have ports specifically used by MBIST applications. The set of memory ports is described in *Design For Test in Encounter RTL Compiler, Customizing a Configuration File, port_alias Specification*. A pair of constraints for each scalar port and vector port are required. For each port the * wildcard is used in the statement to cover all bits of any associated bus.

Resulting LEC dofile potential contributions:

```
add ignored input memory-module-test-port -module memory-module -golden
add ignored input memory-module-test-port -module memory-module -revised
```

Chip Level Insertion Flows

If direct access to MBIST is implemented, additional constraints are necessary for the pins or ports defined with `define_dft mbist_direct_access`.

Resulting LEC dofile potential contributions are as follows:

```
add pin constraints inactive-state burnin_run-pin-or-port -golden
add pin constraints inactive-state poweron_run-pin-or-port -golden
add pin constraints inactive-state engine_schedule_serial-pin-or-port -golden
```

Encounter Test: Flow: MBIST Analysis

Boolean Equivalence Checking

```
add pin constraints inactive-state device_schedule_serial-pin-or-port -golden
add pin constraints inactive-state burnin_run-pin-or-port -revised
add pin constraints inactive-state poweron_run-pin-or-port -revised
add pin constraints inactive-state engine_schedule_serial-pin-or-port -revised
add pin constraints inactive-state device_schedule_serial-pin-or-port -revised

// when monitor port output enable unconnected at pad cell
add pin constraints inactive-state monitor-port-output-enable -golden
// when monitor connected to previously unconnected port or internal pin
add ignored output monitor-pin-or-port-hookup -golden
add ignored output monitor-pin-or-port-hookup -revised
```

If bitmap pattern generation is required of MBIST, additional constraints may be necessary for the ports specified with `insert_dft mbist -measure_ports`.

Resulting LEC dofile potential contributions are as follows:

```
// when measure_port output enable unconnected at pad cell
add pin constraints inactive-state measure-port-output-enable -golden
// when measure_port connected to previously unconnected port
add ignored output measure-port-hookup -golden
add ignored output measure-port-hookup -revised
```

Test-wrapped memories are those memory devices which have ports specifically used by MBIST applications. The set of memory ports is described in *Design For Test in Encounter RTL Compiler, Customizing a Configuration File, port_alias Specification*. A pair of constraints for each scalar port and vector port are required. For each port the * wildcard is used in the statement to cover all bits of any associated bus.

Resulting LEC dofile potential contributions:

```
add ignored input memory-module-test-port -module memory-module -golden
add ignored input memory-module-test-port -module memory-module -revised
```

Multiple Block Merging Flows

For each block which is not a blackbox and which previously had MBIST inserted that is merged into this design, the following actions are taken. The test mode and JTAG ports are added to the block during `insert_dft mbist` by default. In this case, additional constraints are necessary for the associated merged block ports.

Resulting LEC dofile contribution is as follows:

```
// when the test mode signal is unconnected in the original design
add primary input block-instance-name/test-mode-signal -golden
add pin constraints inactive-state block-instance-name/test-mode-signal -golden

add primary input block-instance-name/jtag_reset -golden
```

Encounter Test: Flow: MBIST Analysis

Boolean Equivalence Checking

```
add pin constraints 1 block-instance-name/jtag_reset -golden
add primary input block-instance-name/jtag_tck -golden
add pin constraints 0 block-instance-name/jtag_tck -golden
add primary input block-instance-name/jtag_runidle -golden
add pin constraints 0 block-instance-name/jtag_runidle -golden
add primary input block-instance-name/jtag_shiftdr -golden
add pin constraints 0 block-instance-name/jtag_shiftdr -golden
```

If direct access to MBIST is implemented, additional constraints are necessary for the associated merged block ports. Only those present on the merged block are required in the golden design.

Resulting LEC dofile potential contributions are:

```
add primary input block-instance-name/burnin_run -golden
add pin constraints inactive-state block-instance-name/burnin_run -golden
add primary input block-instance-name/poweron_run -golden
add pin constraints inactive-state block-instance-name/poweron_run -golden
add primary input block-instance-name/engine_schedule_serial -golden
add pin constraints inactive-state block-instance-name/engine_schedule_serial -
golden
add primary input block-instance-name/device_schedule_serial -golden
add pin constraints inactive-state block-instance-name/device_schedule_serial -
golden
```

For each block which is a blackbox and which previously had MBIST inserted that is merged into this design, the following actions are taken. The test mode and JTAG ports are added to the block during `insert_dft mbist` by default. Additional constraints are necessary for the associated merged blackbox ports.

Resulting LEC dofile contributions are as follows:

```
add ignored input test-mode-signal -module blackbox-module-name -golden
add ignored input test-mode-signal -module blackbox-module-name -revised

add ignored input jtag_reset -module blackbox-module-name -golden
add ignored input jtag_tck -module blackbox-module-name -golden
add ignored input jtag_runidle -module blackbox-module-name -golden
add ignored input jtag_shiftdr -module blackbox-module-name -golden

add ignored input jtag_reset -module blackbox-module-name -revised
add ignored input jtag_tck -module blackbox-module-name -revised
add ignored input jtag_runidle -module blackbox-module-name -revised
add ignored input jtag_shiftdr -module blackbox-module-name -revised
```

If direct access to MBIST is implemented, additional constraints are necessary for the associated merged blackbox ports. Only those present on the merged blackbox are required in the design.

Resulting LEC dofile potential contributions are:

Encounter Test: Flow: MBIST Analysis

Boolean Equivalence Checking

```
add ignored input burnin_run -module blackbox-module-name -golden
add ignored input poweron_run -module blackbox-module-name -golden
add ignored input engine_schedule_serial -module blackbox-module-name -golden
add ignored input device_schedule_serial -module blackbox-module-name -golden

add ignored input burnin_run -module blackbox-module-name -revised
add ignored input poweron_run -module blackbox-module-name -revised
add ignored input engine_schedule_serial -module blackbox-module-name -revised
add ignored input device_schedule_serial -module blackbox-module-name -revised
```

For the actions to be taken for the top-level design, refer to “[Block Level Insertion Flow](#)” on page 22 if it is a block and refer to “[Chip Level Insertion Flows](#)” on page 22 if it is a chip.

Encounter Test: Flow: MBIST Analysis

Boolean Equivalence Checking

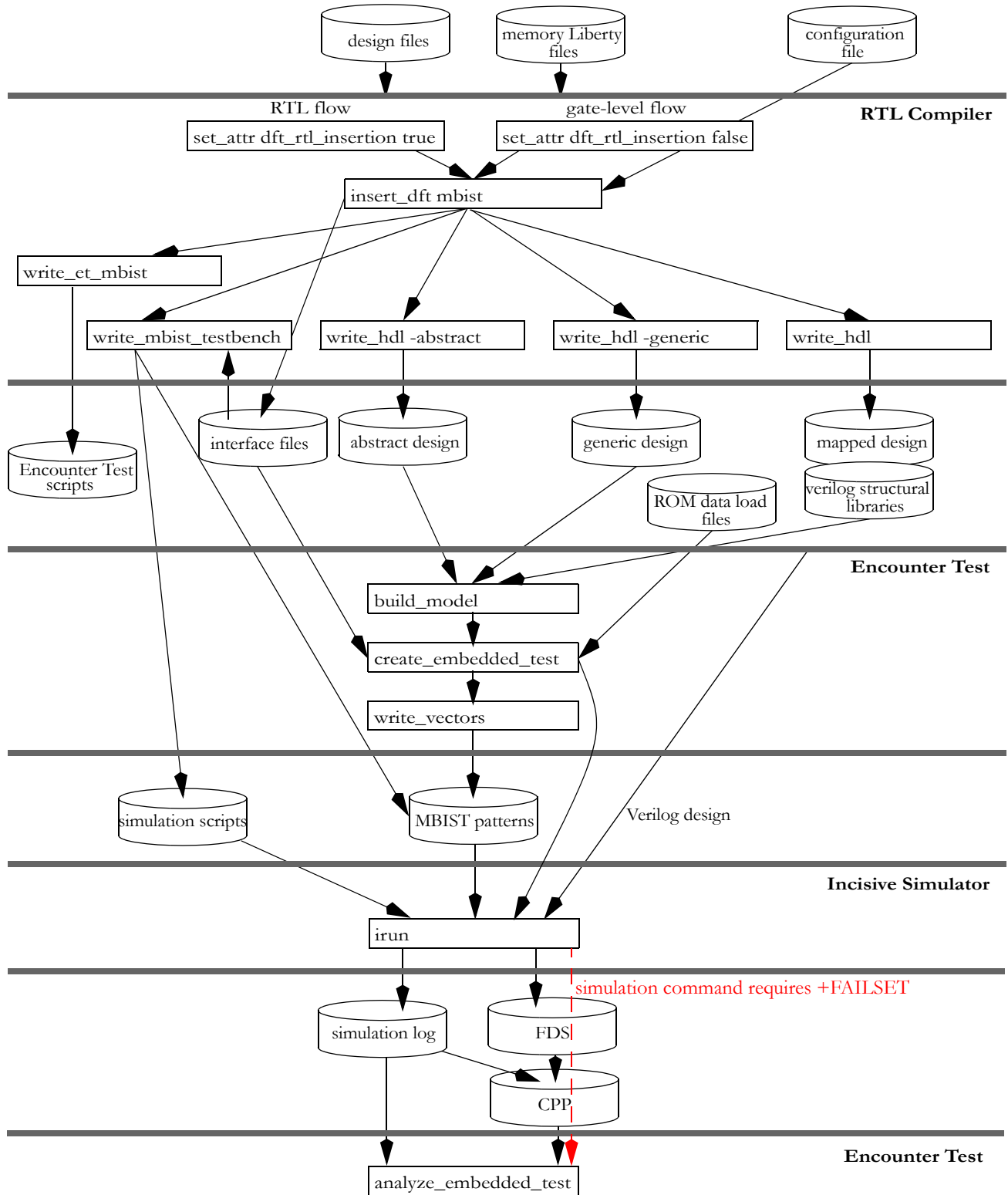
Design Verification

Getting to Encounter Test MBIST Pattern Simulation

To verify the operation of the memory built-in self-test logic inserted by RTL Compiler, patterns can be created as described in the *Encounter Test Memory Built-in Self-test Pattern Generation Guide*. The following figure shows the process flow through entry into the Incisive Simulator. The inputs in the process flow are described in the subsequent sections.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-1 Memory Built-In Self-Test Process Flow



Encounter Test: Flow: MBIST Analysis

Design Verification

Simulation Scripts

The simulation script generated by the RTL Compiler `write_mbist_testbench` command has the following structure:

```
#!/bin/ksh

## Template script for running Cadence Incisive simulator (NCSIM) on Cadence Encounter Test generated test vectors.
## Vectors are MBIST direct access.

## Generated by Encounter(R) RTL Compiler 10.1.300

cd write_mbist_testbench-testbench_directory

irun -libext .v \
    -define MEM_CHECK_OFF \
    -define no_warning \
    -define no_macro_msg \
    -delay_mode zero \
    -seq_udp_delay 10ps \
    -pathpulse \
    -pulse_r 0 \
    -pulse_e 0 \
    -transport_int_delays \
    -multisource_int_delays \
    -nowarn TFNPC \
    -licqueue \
    -mindelays \
    -define comp_ksc \
    -incdir write_mbist_testbench-testbench_directory \
    -access r \
    -nospecify \
    write_mbist_testbench-testbench_directory/VER.testmode-name.experiment-name.mainsim.v \
    +TESTFILE1=write_mbist_testbench-testbench_directory/VER.testmode-name.experiment-name.data.logic.ex1.ts1.verilog \
    -y write_mbist_testbench-ncsim_library-directory \
    -v write_mbist_testbench-ncsim_library-file \
    write_mbist_testbench-directory/design-files \
    -l write_mbist_testbench-testbench_directory/sim.experiment-name.log
```

The script executes memory built-in self-test pattern simulations in zero delay mode. To avoid race conditions associated with sequential elements in the design, including registers and latches, `seq_udp_delay 10ps` is included on the command line. This should be sufficient to ensure proper propagation of values as clocks pulse for most modern technology libraries.

The `+TESTFILE1` line and the preceding line in the file above are described in “[MBIST Patterns](#)” on page 30. The `-y` and `-v` lines in the given file reference the simulation libraries passed through the `write_mbist_testbench -ncsim_library` keyword. The line following these library specifications contains the reference to the actual design description being tested. Such a script is generated for each experiment created by `write_mbist_testbench`.

If RTL Compiler `write_mbist_testbench` is not the path taken to simulation of the memory built-in self-test patterns, a comparable script must be created by the designer. If the righthand path from `irun` through `FDS data` as displayed in [Figure 4-1](#) on page 28 is followed, the simulation script command line must include the “`+FAILSET \`” option in the line preceding the `+TESTFILE1` option.

MBIST Patterns

The Encounter Test Verilog output consists of two files for each experiment with default names using the given formats.

■ `VER.testmode-name.experiment-name.mainsim.v`

This file contains the Verilog module, which instantiates the design under test. It creates connections to all ports on the design, allowing it to stimulate inputs and monitor outputs. The stimulation of inputs and measurement of outputs occur within a test cycle. The set of test cycles comprises an experiment.

Test cycles may vary in their period and more than one may be used in the experiment. In a typical memory built-in self-test experiment, there exists a static test cycle in which JTAG operates to load and unload test data registers. This test cycle period is based on the JTAG TCK clock period. The following relationship must be satisfied within the generated pattern for proper execution:

`inputs stimulated < outputs measured < pulse JTAG TCK`

The actual execution of the memory built-in self-test algorithms usually occurs within a set of dynamic test cycles whose period is based on the clock driving the MBIST logic. In these dynamic test cycles, only the MBIST clock(s) are pulsed.

The generic test application driver within this module accepts instructions, input data values, and expected output values from the given data file.

■ `VER.testmode-name.experiment-name.data.logic.ex1.ts1.verilog`

This file contains the information that the generic test application driver within the `*mainsim.v` file used to control simulation of the patterns, applying values, and expecting results during the required test cycles. The file is organized as a set of lines each containing a three-digit instruction field followed by instruction-dependent data.

ROM Data Load Files

These files are necessary to load the contents of the ROMs during simulation of the read-only memory built-in self-test pattern execution. They must be placed into the directory containing the simulation script. Their formats are described in the [Basic Keywords](#) section in the *Encounter Test Memory Built-in Self-test Pattern Generation Guide*.

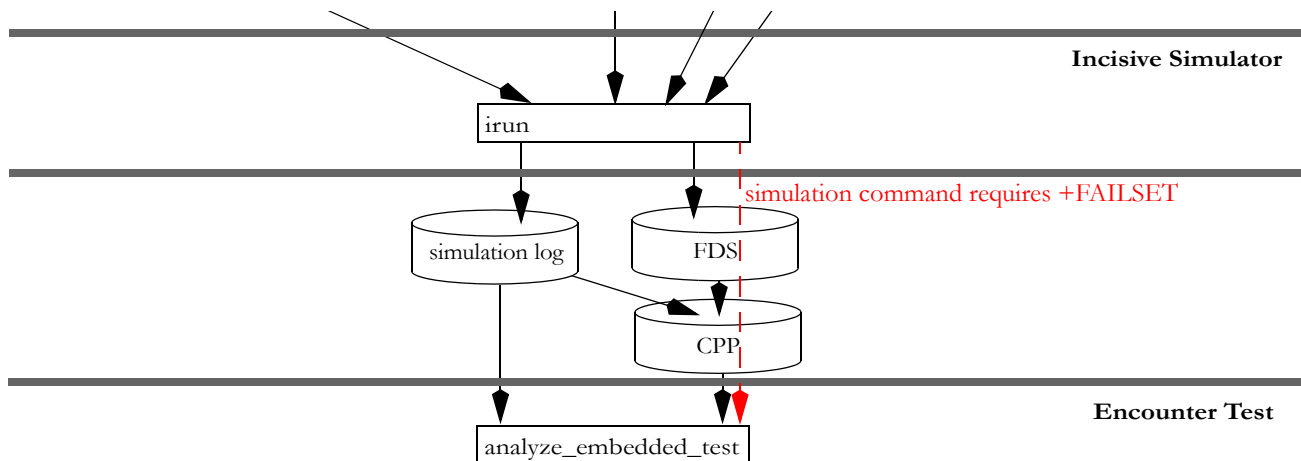
Design and Technology Libraries

Verilog technology description libraries and files are referenced on the `-y` and `-v` lines in the simulation script. These may be necessary when simulating a mapped design. In other cases, they may be necessary to model higher level functions or analog devices. The file(s) describing the design under test must be included in the simulation script. The number and format are dependent on the output generated and selected from RTL Compiler.

Analyzing Pattern Class Simulation Results

The Verilog format of the pattern classes which can be generated as part of memory built-in self-test may require some modification to support analysis in the various design verification flows. These modifications, if necessary, are noted in the table below. The design verification portion of [Figure 4-1](#) on page 28 is repeated in the following figure for easier cross-reference with [Table 4-1](#) on page 32.

Figure 4-2 Memory Built-In Self-Test Design Verification Flows



Three design verification flows are supported from the Incisive Simulator command, `irun`, to Encounter Test command `analyze_embedded_test`.

Encounter Test: Flow: MBIST Analysis

Design Verification

Table 4-1 Design Verification Pattern Class Support

Pattern Class	irun to analyze_embedded _test		irun to CPP to analyze_embedded _test		irun to FDS to CPP to analyze_embedded _test	
	block	chip	block	chip	block	chip
bypass (connectivity)	yes	yes	no	yes ¹	no	yes ^{1,3}
production	yes	yes	no	yes ¹	no	yes ^{1,3}
retention (production)	yes	yes	no	yes ¹	no	yes ^{1,3}
diagnostic	yes	yes	no	yes ²	no	yes ^{2,3}
bitmap	no	no	no	yes ¹	no	yes ^{1,3}
retention (bitmap)	no	no	no	yes ¹	no	yes ^{1,3}
direct access	yes	yes	no	yes ¹	no	yes ^{1,3}
repair	no	yes	no	yes ¹	no	yes ^{1,3}

¹ The Encounter Test contrib sed script, `$Install_Dir/etc/tb/contrib/add_eawoffset.sed`, should be executed against the Verilog `*mainsim.v` pattern file prior to executing the modified `*mainsim.v` in simulation.

² The Encounter Test contrib sed script, `$Install_Dir/etc/tb/contrib/add_eawoffset_diag.sed`, should be executed against the Verilog `*mainsim.v` pattern file prior to executing the modified `*mainsim.v` in simulation.

³ The Encounter Test contrib sed script, `$Install_Dir/etc/tb/contrib/add_tds.sed`, should be executed against the Verilog `*mainsim.v` pattern file after the appropriate `add_eawoffset*` script and prior to executing the modified `*mainsim.v` in simulation.

irun to analyze_embedded_test Flow

In this flow, `analyze_embedded_test` interprets the results of the `irun` simulation log directly. This flow depends upon the message format inserted by Encounter Test Verilog pattern generation. In most cases, block- and chip-level memory built-in self-test pattern generation and simulation are supported. Valid `analyze_embedded_test` command line templates for supported pattern classes are:

- pattern class: bypass, production, retention (production)

Encounter Test: Flow: MBIST Analysis

Design Verification

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=production  
simlog=simulation-log-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: diagnostic**

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=diagnostic  
simlog=simulation-log-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: diagnostic with software repair analysis**

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=repair  
simlog=simulation-log-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: direct access**

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=poweron  
simlog=simulation-log-file
```

irun to CPP to analyze_embedded_test Flow

The output of the `irun` simulation log can be translated to Chip-Pad-Pattern (CPP) format prior to processing with `analyze_embedded_test`. This CPP format is the same memory built-in self-test format into which manufacturing automated test equipment logs must be translated prior to using `analyze_embedded_test`. This permits the simulator to generate and validate this manufacturing test flow. The flow depends upon the message format inserted by Encounter Test Verilog pattern generation. Only chip-level memory built-in self-test pattern generation and simulation are supported.

The CPP file must be generated from the simulation log file.

```
$Install_Dir/etc/tb/contrib/simlog2CPP -simlog simulation-log-file -cppfile  
generated-cpp-file
```

Valid `analyze_embedded_test` command line templates for the supported pattern classes are:

- **pattern class: bypass, production, retention (production)**

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=production cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: diagnostic**

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=diagnostic cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: diagnostic with software repair analysis**

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=repair  
cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- **pattern class: bitmap, retention (bitmap)**

Encounter Test: Flow: MBIST Analysis

Design Verification

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=bitmap  
cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
stclkstep=stclk-scheduling-step
```

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=bitmap  
cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
mtclkstep=mtclk-scheduling-step
```

■ pattern class: direct access

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=poweron cpp=generated-cpp-file
```

irun to FDS to CPP to analyze_embedded_test Flow

The final option supports the direct generation of `irun` simulation output failing data sets (FDS) which can be translated to CPP format prior to processing with `analyze_embedded_test`. This permits the simulator to generate a form of manufacturing automated test equipment log and validate this manufacturing test flow. The flow depends upon the message format inserted by Encounter Test Verilog pattern generation and the generation of either STIL or WGL output patterns to match the simulated Verilog patterns. Only chip-level memory built-in self-test pattern generation and simulation are supported.

By executing `report_vectors` with the *Include vector correspondence* button selected in the Encounter Test GUI for a memory built-in self-test pattern, the output file generated contains a numbered list of the pins on the part. A portion of this file can be used to build a pin correlation file for use in the `FDSlog2CPP` command. The output pin portion of the file is displayed in the following snippet. The highlighted line is used during JTAG-controlled memory built-in self-test patterns to monitor the output and determine correct operation. If bitmap patterns are generated using measure ports beyond the JTAG TDO port, they must be included in the pin correlation file as well.

Figure 4-3 Portion of Vector Correspondence

```
# PO:  
# (PO 1 = "Pin.f.l.TopBlock.nl.BOUNDARY_REG_OUT_DUMMY", # index = 26  
# PO 2 = "Pin.f.l.TopBlock.nl.MMP1", # index = 27  
# PO 3 = "Pin.f.l.TopBlock.nl.MMP2", # index = 28  
# PO 4 = "Pin.f.l.TopBlock.nl.MONITOR", # index = 29  
# PO 5 = "Pin.f.l.TopBlock.nl.SO", # index = 30  
# PO 6 = "Pin.f.l.TopBlock.nl.SO1", # index = 31  
# PO 7 = "Pin.f.l.TopBlock.nl.SO2", # index = 32  
# PO 8 = "Pin.f.l.TopBlock.nl.TDO", # index = 33 tf = TDO
```

This information can be used to create a file containing the following column headers and data extracted from the vector correspondence for use in the `FDSlog2CPP` script.

Encounter Test: Flow: MBIST Analysis

Design Verification

Figure 4-4 Pin Correlation File Example

PAD_PIN	PIN_NAME
8	TDO

The CPP file must be generated from the simulation FDS file.

```
$Install_Dir/etc/tb/contrib/FDSlog2CPP -fds_log_file simulation-fds-file  
-pattern_file STIL.testmode-name.experiment-name.logic.ex1.ts1.stil  
-pin_correlation_file pin-correlation-file -cpp_file generated-cpp-file
```

```
$Install_Dir/etc/tb/contrib/FDSlog2CPP -fds_log_file simulation-fds-file  
-pattern_file WGL.testmode-name.experiment-name.logic.ex1.ts1.wgl  
-pin_correlation_file pin-correlation-file -cpp_file generated-cpp-file
```

Valid `analyze_embedded_test` command line templates for the supported pattern classes are shown below:

- pattern class: bypass, production, retention (production)

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=production cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: diagnostic

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=diagnostic cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: diagnostic with software repair analysis

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=repair  
cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: bitmap, retention (bitmap)

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=bitmap  
cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
stclkstep=stclk-scheduling-step
```

```
analyze_embedded_test patterncontrolfile=pattern-control-file analysis=bitmap  
cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
mtclkstep=mtclk-scheduling-step
```

- pattern class: direct access

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=poweron cpp=generated-cpp-file
```

Special Handling of 4 pin TAP controller in Simulation

In the case where a 4 pin TAP controller is used, `create_embedded_test` will create a testmode for pattern generation where the mode initialization sequence can not use the reset pin. In these situations, the mode initialization sequence consists of holding the TMS signal

Encounter Test: Flow: MBIST Analysis

Design Verification

to the active state, pulsing TCK for 10 cycles, and setting TMS to the inactive state. However, this is not enough to initialize all of the FSM latches inside the TAP controller.

In order to initialize the FSM latches, it is necessary to manually assert the `JTAG_TRST` signal just after TMS is active by using a force event, run for a small amount of cycles, and then deassert the `JTAG_TRST` signal using another force event. When using the `irun` command to perform simulation, it is necessary to use the `-access rw` option to allow force events to occur. An example of these steps is:

```
run 150ns
force path_to_tap_controller_instance.tap_instance.JTAG_TRST = `b0
run 10ns
force path_to_tap_controller_instance.tap_instance.JTAG_TRST = `b1
run
```

Special Handling of Split Retention Patterns in Simulation

In cases where the test engineer wishes to control the time period between writes and subsequent reads in the retention test patterns, or alter environmental or voltage values, `create_embedded_test splitretentionpatterns=yes...` must be executed. This is described in the [Retention](#) section in the *Encounter Test Memory Built-In Self-Test Pattern Generation Guide*. If there is a need to simulate the split retention patterns, they must first be merged as described in the [Production Retention Pattern](#) and [Bitmap Retention Pattern](#) sections in the *Encounter Test Memory Built-In Self-Test Pattern Generation Guide*.

When merging Verilog patterns, the data values files (default naming is `VER.testmode-name.experiment-name.data.logic.ex1.ts1.verilog`) must be merged into a single file to ensure proper operation. Attempts to reference multiple `TESTFILE[1-99]` files in the simulation script results in the introduction of X values on all design input pins between test file executions as the testbench assumes the test files are independent and they are not in these cases.

Proper verification of the merged bitmap retention patterns in simulation can only be accomplished without fault injection, correctly resulting in no simulation mismatches. Fault detection cannot be verified directly using simulation logs with the merged patterns due to the changes in the Encounter Test odometer values when the pattern is split into segments.

Simulation Mismatches

Due to the structure of the memory built-in self-test hardware and the Verilog patterns, it is possible for some pattern classes to show mismatches in the simulation log file even when no faults are injected in the memories. These cases are described below.

Diagnostic

The following diagnostic mismatches only appear when using RC version 10.1.301 and older. Newer versions of RC will not show these mismatches.

ROM device patterns use the starting and ending address registers within the hardware to determine the failing address during diagnostic pattern execution. The expected values are all zeroes to allow analysis software to determine the address read when a data mismatch occurs. These appear as mismatches against the `start_address` and `end_address` bit positions in the `design_rombistdiag_tdr_map.txt` fields.

`analyze_embedded_test` filters these mismatches only using the addresses when an actual ROM data mismatch requires analysis.

`mrnw` device patterns may indicate failures on the active write port register(s) during diagnostic pattern execution. The expected values are zero and may mismatch as one. These appear as mismatches against the `wport` bit positions in the `design_mbistdiag_tdr_map.txt` fields.

`analyze_embedded_test` filters these mismatches only using the values when an actual memory data mismatch requires analysis.

Diagnostic - Retention

`1rw`, `mrnw`, and `2rw` device patterns use the MBIST engine `done` bits to indicate completion of the engine operations. When retention patterns pause during the test, the mechanism used to stop engine execution is the same as that used when a memory data failure is detected. As such the stoppage due to pause cannot be distinguished from that due to a failure, and the pattern expects the engine `done` bit to be set to one as it would at completion. These appear as mismatches for each active engine `done` bit position in the `design_mbistdiag_tdr_map.txt` fields: two mismatches for `1rw` engines; 2n mismatches for `mrnw` engines; four mismatches for `2rw` engines.

`mrnw` device patterns may indicate failures on the active write port register(s) during diagnostic pattern execution. The expected values are zero and may mismatch as one. These appear as mismatches against the `wport` bit positions in the `design_mbistdiag_tdr_map.txt` fields.

`2rw` device patterns may indicate failures on the active read port register during diagnostic pattern execution. The expected value is zero and may mismatch as one. These appear as two mismatches against the `rport` bit position per active `2rw` engine in the `design_mbistdiag_tdr_map.txt` fields.

`analyze_embedded_test` filters all these mismatches, only using the values when an actual memory data mismatch requires analysis.

All mismatches not related to these pattern classes and particular registers must be investigated further as *true* failure indications.

Injecting Memory Faults

Methodology

Many possible methods exist to inject faults within and around memories to verify detection occurs during memory built-in self-test. The recommended methodology relies upon changes in the Verilog memory models used in simulation. By making the changes indicated below, `analyze_embedded_test` can be used to determine that proper faults were injected and detected during the various simulation flows as displayed in [Figure 4-2](#) on page 31.

- Simulation memory models must be modified to support reading fault injection control files, injecting memory errors on memory write operations, and recording the error injection action in the simulation log file.
- Fault injection control files must identify the desired stuck-at fault set. Fault sets are bound to memory modules and not memory instances.
- `analyze_embedded_test` can be used to determine if the MBIST engines properly detect the faults injected.

Changes in the Verilog Memory Models

Modified code snippets of a Verilog memory model for an Artisan memory are shown below. The boldface and italicized text indicates additions to the original memory model which support the stuck-at fault injection.

Figure 4-5 Verilog Memory Model Changes Supporting Fault Injection

```
module RF32X32 (  
    Q,  
    CLK,  
    CEN,  
    WEN,  
    A,  
    D  
);  
...
```

Encounter Test: Flow: MBIST Analysis

Design Verification

```
parameter UPM_WIDTH = 2;
parameter insert_SAF=1; // enable error injection when 1
parameter num_SAF=16;

output [31:0]          Q;
...
// store tmpdata in memory
    if (rren === 1'b1)
        mem[a]=fi_d(a,tmpdata);
...
reg [15:0] SAF_row[0:num_SAF-1];
reg [7:0]  SAF_col[0:num_SAF-1];
reg      SAF_st[0:num_SAF-1];

initial $readmemb("RF32X32.row", SAF_row);
initial $readmemb("RF32X32.col", SAF_col);
initial $readmemb("RF32X32.st", SAF_st);
initial active_flag = 0;

function [BITS+1:0] fi_d;
input [ADDR_WIDTH-1:0] a;
input [BITS+1:0] d;

integer i;

begin

fi_d = d;
if (insert_SAF)
    begin
        for (i=0; i<num_SAF; i=i+1)
            begin
                if((a == SAF_row[i]) &&
                    (SAF_st[i] !== 1'bx))
                    begin
                        fi_d[SAF_col[i]] = SAF_st[i];
                        $display("insert fault in RF32X32 at address %d, column %d, state %d at Time %t",
                            a, SAF_col[i], SAF_st[i], $realtime);
                    end
                end
            end
        end
    end
endfunction
...
```

Fault Injection Control Files

From the previous example, it is evident that the `$display` statement identifies the filename of the memory fault injection control files. Three fault injection control files are required:

Encounter Test: Flow: MBIST Analysis

Design Verification

- *memory_module_name.row*

Used to identify the address at which to insert a stuck-at fault in the memory.

- *memory_module_name.col*

Used to identify the bit at which to insert a stuck-at fault in the memory.

- *memory_module_name.st*

Used to identify the state creating a stuck-at fault in the memory: 0 or 1. A value of x indicates no stuck-at fault should be inserted for this entry.

These files are correlated by lines, the information at the same line number in each file is used to create a stuck-at fault. The files are each expected to be `num_SAF` entries in size, one entry per line in the file. Unused entries should be filled with zeroes in the row and col files, and x in the state file.

The following sample files inject two failures into the RF32X32 memory model.

Table 4-2 Memory Model Fault Injection Control Files

RF32X32.row	RF32X32.col	RF32X32.st	Injected Fault
0000000000000000	00000000	0	address 0, bit 0, SA0
0000000000001000	00011111	1	address 16, bit 31, SA1
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none
0000000000000000	00000000	x	none

Using `analyze_embedded_test` to Analyze Simulation Results with Fault Injection

As previously noted, `analyze_embedded_test` can process simulation log files and CPP data for post-execution analysis. When processing simulation log files, if properly formatted messages are displayed to indicate where faults are injected `analyze_embedded_test` can determine based on the algorithms executed if the proper number of failures are detected throughout MBIST execution. The message format is displayed in [Figure 4-6](#) on page 42 with RF32X32 indicating the memory module name. This module name is used by `analyze_embedded_test` to associate injected module faults with the failing memory characteristics from the corresponding `design_pattern_control.txt` file.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-6 Simulation Log File Fault Injection Messages

```
insert fault in RF32X32 at address 0, column 0, state 0 at Time      3165250000
insert fault in RF32X32 at address 16, column 31, state 1 at Time   3205250000
```

The following table shows the corresponding detailed miscompare analysis of the diagnostic simulation log generated by `analyze_embedded_test` when the inserted failures are known to `analyze_embedded_test`.

Table 4-3 `analyze_embedded_test` Simulation Log Detailed Failure Summary

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)
0	0 11m1/sram	0	checkerboard	16	4	(31,0)
0	0 11m1/sram	0	checkerboard	16	4	(31,0)
0	0 11m1/sram	0	checkerboard	0	0	(0,0)
0	0 11m1/sram	0	checkerboard	0	0	(0,0)
0	0 11m1/sram	0	checkerboard	0	0	(0,0)
0	0 11m1/sram	0	checkerboard	16	4	(31,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)
0	0 11m1/sram	0	march_lr	0	0	(0,0)
0	0 11m1/sram	0	march_lr	16	4	(31,0)

Finally, `analyze_embedded_test` reports a simulation log failure analysis summary per memory instance and indicates whether the expected number of failures were actually detected. Refer to [Figure 4-7](#) on page 43 to complete the example.

The TEM-506 message indicates the completion status for each engine. The value in the table indicates the value of the loop (specified by the `failurelimit` keyword to `create_embedded_test`) for which the engine completed execution.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-7 analyze_embedded_test Simulation Log Memory Instance Failure Results

```
INFO (TEM-302): Analyzing data for:
                  MBIST Engine: 0
                  Target Memory: 0 [end TEM_302]

INFO (TEM-303): Expected stuck-at-0 faults: 3
                  Expected stuck-at-1 faults: 3
                  Total Expected Failures: 6 [end TEM_303]

Algorithm          Expected Total    Actual Total
-----
checkerboard        6                6
galloping_ones      0                0
march_c             0                0
march_lr            0                0
port_interaction     0                0
pseudo_random_address 0                0
wordline_stripe     0                0

INFO (TEM-318): The address and bit locations where faults were manually inserted match
                  the address and bit locations determined by the simulation patterns for engine 0 target memory 0. [end TEM_318]

INFO (TEM-317): Actual failures identified by the simulation results match the expected number of fails. [end TEM_317]

INFO (TEM-506): Engine completion summary: [end TEM_506]

-----
Engine    Last Active failurelimit Loop Index
-----
0          7

INFO (TEM-314): Successfully analyzed failures.
                  Output file is: './testde/testresults/logs/log_analyze_embedded_test_061311141857-471327000'. [end TEM_314]
```

Understanding analyze_embedded_test Results

The information presented by `analyze_embedded_test` after processing MBIST execution results, either simulation logs or CPP data, is limited by the amount of information available within the pattern class:

- direct access poweron patterns yield passing die indications;
- bypass, production, and production retention patterns yield passing memory instance indications;
- diagnostic patterns yield memory instance failing address and bit (column) information and possible software-based repair solutions if selected;
- bitmap and bitmap retention patterns yield memory instance failing address, bit (column) and data value information.

When processing properly annotated simulation log files, `analyze_embedded_test` can perform some additional consistency checks when injecting faults according to the methodology described in [Injecting Memory Faults](#) on page 38. When processing CPP data, such checks are not possible.

Executing the Command

Examples of `analyze_embedded_test` commands for different pattern classes can be found at the end of the descriptions of the various simulation flows:

- [irun to analyze_embedded_test Flow](#) on page 32
- [irun to CPP to analyze_embedded_test Flow](#) on page 33
- [irun to FDS to CPP to analyze_embedded_test Flow](#) on page 34

Working with Simulation Logs

For most of the pattern classes, sample `analyze_embedded_test` output is included with a brief explanation interpreting the output. The examples have two stuck-at faults injected in a single memory device running only the checkerboard algorithm.

Direct Access

The *Test Status* indicates a failing test for the die due to the injected stuck-at faults. Information is limited due to the direct access of the memory built-in self-test.

```
INFO (TEM-301): Parsing the test data register mapping file 'test_mult_lrw/testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file 'test_mult_lrw/testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-311): Parsing the simulation log file 'test_mult_lrw/testde/logs/simMBIST_ATE_PROD_1_ref_clkaPatterns.Output'. [end TEM_311]
INFO (TEM-304): Total Failures Analyzed: 1 [end TEM_304]
```

```
-----
Engine      Target RAM      Test Status      Bits
-----
0           0 1l1m1/sram    Failed           .
```

```
INFO (TEM-314): Successfully analyzed failures.
Output file is: 'test_mult_lrw/testde/testresults/logs/log_analyze_embedded_test_030611104253-215512000'. [end TEM_314]
```

Production

The *Test Status* indicates a failing test for the engine 0, target 0 memory device due to the injected stuck-at faults. This information appears in the JTAG-accessible MBISTDIAG test data register.

Encounter Test: Flow: MBIST Analysis

Design Verification

The Test Status indicates a failing test for the engine 1, target 0 and engine 2, target 0 memory devices due to the test not running long enough or the clock not being active. It is also possible that the clock ratio between the clock port and the hookup pin may be bad, or the clock may not be propagating to the engine. The TEM-329 warning message indicates that the test possibly needs to execute for a longer duration.

```
INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-311): Parsing the simulation log file './testde/logs/simMBIST_ATE_PROD_1_ref_clkPatterns.Output'. [end TEM_311]
INFO (TEM-304): Total Failures Analyzed: 3 [end TEM_304]

-----
Engine      Target RAM      Test Status      Bits
-----
0           0 1lm1/sram      Failed           .
1           0 1lm2/sram      Failed-Incomplete .
2           0 1lm3/sram      Failed-Incomplete .

WARNING (TEM-329): One or more engines did not finish executing the selected algorithms.
                  Engines: 1 2.
                  Rerun create_embedded_test with the failurelimit keyword set to a higher value for diagnostic patterns.
                  Also check to make sure the clock is properly connected and operational at the engines. [end TEM_329]

INFO (TEM-314): Successfully analyzed failures.
                  Output file is: './testde/testresults/logs/log_analyze_embedded_test_061311144442-171971000'. [end TEM_314]
```

Diagnostic

During JTAG-initiated memory built-in self-test, sufficient information exists in the MBISTDIAG test data register to determine the details shown in the table below. In this test, the two stuck-at faults were each detected by the checkerboard algorithm three times as shown in the TEM-304 table. Following the methodology of Injecting Memory Faults on page 38, extra information is available in the simulation log to support message TEM-303, indicating the proper number of faults that were analyzed.

The TEM-506 message is an engine completion summary table indicating the value of the failurelimit loop index value when the engine completed. In this example a value of 5 for engine 0 indicates that any additional loops greater than 5 are unnecessary. The create_embedded_test command used to generate the patterns being analyzed in this situation can be executed with the keyword failurelimit=5. This allows for the proper execution time while avoiding wasted simulation execution time.

Encounter Test: Flow: MBIST Analysis Design Verification

```
INFO (TEM-301): Parsing the test data register mapping file './testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-311): Parsing the simulation log file './testde/logs/simMBIST_ATE_DIAG_CONTROLLER_ref_clka_0_0Patterns.Output'. [end TEM_311]
INFO (TEM-304): Total Failures Analyzed: 6 [end TEM_304]
```

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)
0	0 1l1m1/sram	0	checkerboard	16	4	(31,0)
0	0 1l1m1/sram	0	checkerboard	16	4	(31,0)
0	0 1l1m1/sram	0	checkerboard	0	0	(0,0)
0	0 1l1m1/sram	0	checkerboard	0	0	(0,0)
0	0 1l1m1/sram	0	checkerboard	0	0	(0,0)
0	0 1l1m1/sram	0	checkerboard	16	4	(31,0)

```
INFO (TEM-302): Analyzing data for:
                MBIST Engine: 0
                Target Memory: 0 [end TEM_302]
```

```
INFO (TEM-303): Expected stuck-at-0 faults: 3
                Expected stuck-at-1 faults: 3
                Total Expected Failures:    6 [end TEM_303]
```

Algorithm	Expected Total	Actual Total
checkerboard	6	6
galloping_ones	0	0
march_c	0	0
march_lr	0	0
port_interaction	0	0
pseudo_random_address	0	0
wordline_stripe	0	0

```
INFO (TEM-318): The address and bit locations where faults were manually inserted match
                the address and bit locations determined by the simulation patterns for engine 0 target memory 0. [end TEM_318]
```

```
INFO (TEM-317): Actual failures identified by the simulation results match the expected number of fails. [end TEM_317]
```

```
INFO (TEM-506): Engine completion summary: [end TEM_506]
```

Engine	Last Active failure	limit	Loop	Index
0	5			

```
INFO (TEM-314): Successfully analyzed failures.
                Output file is: './testde/testresults/logs/log_analyze_embedded_test_061311141857-471327000'. [end TEM_314]
```

Diagnostic with Software Repair Analysis

During JTAG-initiated memory built-in self-test, sufficient information exists in the MBISTDIAG test data register to determine the details shown in the table below. In this test, the two stuck-at faults were each detected by the checkerboard algorithm three times as shown in the TEM-304 table. Following the methodology of [Injecting Memory Faults](#) on page 38, extra information is available in the simulation log to support message TEM-303, indicating the proper number of faults were analyzed.

The repair analysis for the particular memory device begins with message TEM-302 and ends with TEM-307. The results of the software repair analysis for the die are summarized in the table of message TEM-505.

Encounter Test: Flow: MBIST Analysis Design Verification

INFO (TEM-301): Parsing the test data register mapping file './testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]

INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]

INFO (TEM-311): Parsing the simulation log file './testde/logs/simMBIST_ATE_DIAG_CONTROLLER_ref_clk_0_0Patterns.Output'. [end TEM_311]

INFO (TEM-304): Total Failures Analyzed: 6 [end TEM_304]

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)
0	0 llml/sram	0	checkerboard	16	4	(31,0)
0	0 llml/sram	0	checkerboard	16	4	(31,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	16	4	(31,0)

INFO (TEM-302): Analyzing data for:
MBIST Engine: 0
Target Memory: 0 [end TEM_302]

INFO (TEM-303): Expected stuck-at-0 faults: 3
Expected stuck-at-1 faults: 3
Total Expected Failures: 6 [end TEM_303]

Algorithm	Expected Total	Actual Total
checkerboard	6	6
galloping_ones	0	0
march_c	0	0
march_lr	0	0
port_interaction	0	0
pseudo_random_address	0	0
wordline_stripe	0	0

INFO (TEM-318): The address and bit locations where faults were manually inserted match the address and bit locations determined by the simulation patterns for engine 0 target memory 0. [end TEM_318]

INFO (TEM-317): Actual failures identified by the simulation results match the expected number of fails. [end TEM_317]

INFO (TEM-506): Engine completion summary: [end TEM_506]

Engine	Last Active failure	limit	Loop Index	Loop Value
0	5			

INFO (TEM-314): Successfully analyzed failures.
Output file is: './testde/testresults/logs/log_analyze_embedded_test_061311141857-471327000'. [end TEM_314]

INFO (TEM-504): Beginning repair analysis. [end TEM_504]

INFO (TEM-302): Analyzing data for:
MBIST Engine: 0
Target Memory: 0 [end TEM_302]

INFO (TEM-307): Redundancy analysis summary: repairable
Recommended rows/columns to fix: [rows: 0,4, columns:]
Redundancy capability defined: [rows: 2, columns: 2]
Redundancy capability unused: [rows: 0, columns: 2] [end TEM_307]

INFO (TEM-505): Repair summary: [end TEM_505]

Engine	Target	Failures Detected	Recommended To Fix	Redundancy Capability	Repair Status
		{row bit} or {row (bit,col)}	Rows Cols	Defined Cols Unused Rows Cols	
0	0	{0 (0,0)}	0	2 2	Repairable
		{16 (31,0)}	4	- 2	

Encounter Test: Flow: MBIST Analysis

Design Verification

Working with CPP Data

For most of the pattern classes, sample `analyze_embedded_test` output is included with a brief explanation interpreting the output. The examples have two stuck-at faults injected in a single memory device running only the checkerboard algorithm.

Direct Access

The *Test Status* indicates a failing test for the die due to the injected stuck-at faults. Information is limited due to the direct access of the memory built-in self-test.

```
INFO (TEM-500): Parsing the pattern_control file 'test_mult_lrw/testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-309): Parsing the chip pad pattern (cpp) file 'test_mult_lrw/testde/logs/cpp.MBIST_ATE_POWERON_1'. [end TEM_309]
INFO (TEM-319): Chip: serialno 23_022_030.
                  Total Failures Analyzed: 1. [end TEM_319]

Lot: 000000104C01291-01
Wafer: 23
Wafer Position: 022,030
Testblockname: Tblknam
Testcondition: RSFN1

-----
Test Status
-----
Failed

INFO (TEM-314): Successfully analyzed failures.
                  Output file is: 'test_mult_lrw/testde/testresults/logs/log_analyze_embedded_test_030611104420-870505000'. [end TEM_314]
```

Production

The *Test Status* indicates a failing test for the engine 0, target 0 memory device due to the injected stuck-at faults. This information appears in the JTAG-accessible MBISTDIAG test data register.

The Test Status indicates a failing test for the engine 1, target 0 and engine 2, target 0 memory devices due to the test not running long enough or the clock not being active. The TEM-329 warning message indicates that the test possibly needs to execute for a longer duration.

Encounter Test: Flow: MBIST Analysis Design Verification

```
INFO (TEM-301): Parsing the test data register mapping file './testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-309): Parsing the chip pad pattern (cpp) file './testde/logs/cpp.MBIST_ATE_PROD_1_ref_clka'. [end TEM_309]
INFO (TEM-319): Chip: serialno_23_022_030.
                  Total Failures Analyzed: 3. [end TEM_319]

Lot: 000000104C01291-01
Wafer: 23
Wafer Position: 022,030
Testblockname: Tblknam
Testcondition: RSFN1

-----
Engine          Target RAM          Test Status      Bits
-----
0               0 1lm1/sram          Failed           .
1               0 1lm2/sram          Failed-Incomplete .
2               0 1lm3/sram          Failed-Incomplete .

WARNING (TEM-329): One or more engines did not finish executing the selected algorithms.
                  Engines: 1 2.
                  Rerun create_embedded_test with the failurelimit keyword set to a higher value for diagnostic patterns.
                  Also check to make sure the clock is properly connected and operational at the engines. [end TEM_329]

INFO (TEM-314): Successfully analyzed failures.
                  Output file is: './testde/testresults/logs/log_analyze_embedded_test_061411070223-419615000'. [end TEM_314]
```

Diagnostic

During JTAG-initiated memory built-in self-test, sufficient information exists in the MBISTDIAG test data register to determine the details shown in the table below. In this test, the two stuck-at faults were each detected by the checkerboard algorithm three times as shown in the TEM-319 table.

The TEM-506 message is an engine completion summary table indicating the value of the `failurelimit` loop index value when the engine completed. In this example a value of 5 for engine 0 indicates that any additional loops greater than 5 are unnecessary. The `create_embedded_test` command used to generate the patterns being analyzed in this situation can be executed with the keyword `failurelimit=5`. This allows for the proper execution time while avoiding wasted simulation execution time.

Encounter Test: Flow: MBIST Analysis Design Verification

```
INFO (TEM-301): Parsing the test data register mapping file './testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-309): Parsing the chip pad pattern (cpp) file './testde/logs/cpp.MBIST_ATE_DIAG_CONTROLLER_ref_clk_a_0_0'. [end TEM_309]
INFO (TEM-319): Chip: serialno_23_022_030.
                  Total Failures Analyzed: 6. [end TEM_319]

Lot: 000000104C01291-01
Wafer: 23
Wafer Position: 022,030
Testblockname: Tblknam
Testcondition: RSFN1
```

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)
0	0 1lml/sram	0	checkerboard	16	4	(31,0)
0	0 1lml/sram	0	checkerboard	16	4	(31,0)
0	0 1lml/sram	0	checkerboard	0	0	(0,0)
0	0 1lml/sram	0	checkerboard	0	0	(0,0)
0	0 1lml/sram	0	checkerboard	0	0	(0,0)
0	0 1lml/sram	0	checkerboard	16	4	(31,0)

```
INFO (TEM-506): Engine completion summary: [end TEM_506]
```

Engine	Last Active failure	limit	Loop Index	Loop Value
0	5			

```
INFO (TEM-314): Successfully analyzed failures.
                  Output file is: './testde/testresults/logs/log_analyze_embedded_test_061411070913-407848000'. [end TEM_314]
```

Diagnostic with Software Repair Analysis

During JTAG-initiated memory built-in self-test, sufficient information exists in the MBISTDIAG test data register to determine the details shown in the table below. In this test, the two stuck-at faults were each detected by the checkerboard algorithm three times as shown in the TEM-319 table.

The repair analysis for the particular memory device begins with message TEM-302 and ends with TEM-307. The results of the software repair analysis for the die are summarized in the table of message TEM-505.

Encounter Test: Flow: MBIST Analysis Design Verification

```
INFO (TEM-301): Parsing the test data register mapping file './testde/mbist/TopBlock_mbistdiag_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file './testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-311): Parsing the simulation log file './testde/logs/simMBIST_ATE_DIAG_CONTROLLER_ref_clk0_0Patterns.Output'. [end TEM_311]
INFO (TEM-304): Total Failures Analyzed: 6 [end TEM_304]
```

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)
0	0 llml/sram	0	checkerboard	16	4	(31,0)
0	0 llml/sram	0	checkerboard	16	4	(31,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	0	0	(0,0)
0	0 llml/sram	0	checkerboard	16	4	(31,0)

```
INFO (TEM-302): Analyzing data for:
                MBIST Engine: 0
                Target Memory: 0 [end TEM_302]
```

```
INFO (TEM-303): Expected stuck-at-0 faults: 3
                Expected stuck-at-1 faults: 3
                Total Expected Failures:    6 [end TEM_303]
```

Algorithm	Expected Total	Actual Total
checkerboard	6	6
galloping_ones	0	0
march_c	0	0
march_lr	0	0
port_interaction	0	0
pseudo_random_address	0	0
wordline_stripe	0	0

```
INFO (TEM-318): The address and bit locations where faults were manually inserted match
                the address and bit locations determined by the simulation patterns for engine 0 target memory 0. [end TEM_318]
```

```
INFO (TEM-317): Actual failures identified by the simulation results match the expected number of fails. [end TEM_317]
```

```
INFO (TEM-506): Engine completion summary: [end TEM_506]
```

Engine	Last Active failure	limit	Loop Index	Loop Value
0	5			

```
INFO (TEM-314): Successfully analyzed failures.
                Output file is: './testde/testresults/logs/log_analyze_embedded_test_061311141857-471327000'. [end TEM_314]
```

Bitmap

During JTAG-initiated memory built-in self-test, sufficient information exists in the MBISTREAD test data register to determine the details shown in the table below. In this test, the two stuck-at faults were each detected by the checkerboard algorithm three times as shown in the TEM-319 table.

Encounter Test: Flow: MBIST Analysis Design Verification

INFO (TEM-301): Parsing the test data register mapping file 'test_mult_lrw/testde/mbist/TopBlock_mbistread_tdr_map.txt'. [end TEM_301]
INFO (TEM-500): Parsing the pattern_control file 'test_mult_lrw/testde/mbist/TopBlock_pattern_control.txt'. [end TEM_500]
INFO (TEM-309): Parsing the chip pad pattern (cpp) file 'test_mult_lrw/testde/logs/cpp.MBIST_ATE_BITMAP_1_ref_clka'. [end TEM_309]
INFO (TEM-319): Chip: serialno.
Total Failures Analyzed: 6. [end TEM_319]

Lot: 000000104C01291-01
Wafer: 23
Wafer Position: 022,030
Testblockname: Tblknam
Testcondition: RSFN1

Engine	Target RAM	Port	Algorithm	Logical Address	Row	Bit (Column)	Value Read
0	0 1lml/sram	0	checkerboard	16	4	(31,0)	1
0	0 1lml/sram	0	checkerboard	16	4	(31,0)	1
0	0 1lml/sram	0	checkerboard	0	0	(0,0)	0
0	0 1lml/sram	0	checkerboard	0	0	(0,0)	0
0	0 1lml/sram	0	checkerboard	0	0	(0,0)	0
0	0 1lml/sram	0	checkerboard	16	4	(31,0)	1

INFO (TEM-314): Successfully analyzed failures.
Output file is: 'test_mult_lrw/testde/testresults/logs/log_analyze_embedded_test_030711122103-481628000'. [end TEM_314]

Analyzing and Debugging MBIST Simulation Results

This section deals with user analysis and debug of memory built-in self-test simulation results. It is intended to serve as a guide into the more commonly encountered issues when initially validating MBIST operations. Nearly all integration and execution problems can be diagnosed at the boundaries of the inserted MBIST modules and their associated memories. Incisive Simulator Simvision Waveform screen snapshots are used extensively to identify MBIST module and memory module ports which are examined to determine correct operation.

The content of this section is divided into several topics with related items grouped under those headings:

- [Simulation Environment](#) on page 52
- [MBIST Startup](#) on page 56
- [Memory Connections and Activity](#) on page 68
- [MBIST Comparators](#) on page 73
- [MBIST Completion](#) on page 75
- [Determining Miscomparing Registers from Simulation Logs](#) on page 78

Simulation Environment

- 'timescale

Encounter Test: Flow: MBIST Analysis

Design Verification

The `VER.testmode-name.experiment-name.mainsim.v` testbenches set the simulation timescale compiler directive to:

```
`timescale 1 ps / 1 fs
```

supporting simulation time precision down to 1fs and a time unit of 1ps to avoid limiting the precision of the design under test.

- simulation delay mode: zero, unit, Standard Delay Format (SDF)

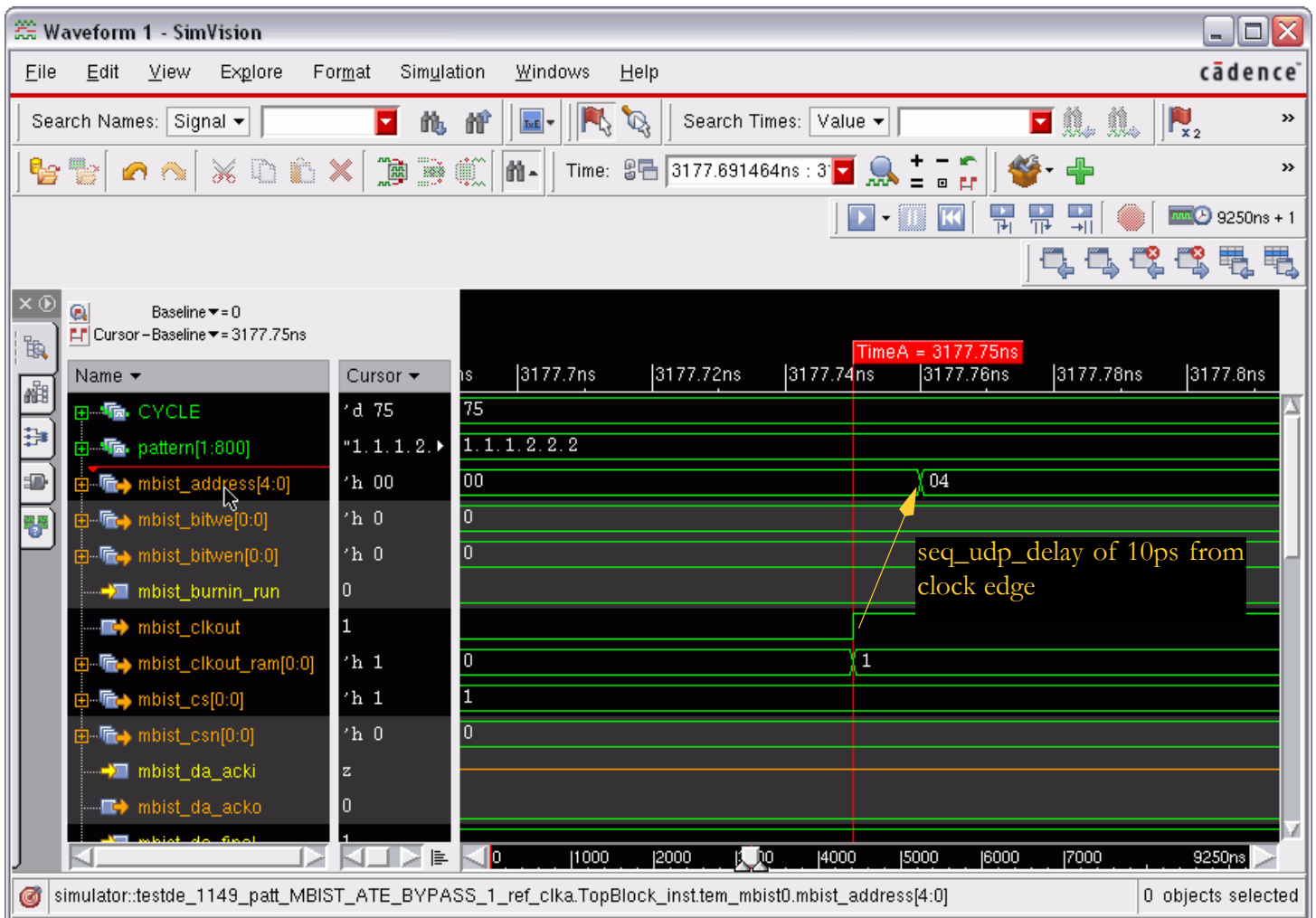
The simulation scripts generated by the RTL Compiler `write_mbist_testbench` command utilize `delay_mode zero`. Although the simulations will generally run in unit delay mode as well, any logic in the clock paths may cause false race conditions. Zero delay mode simulation is recommended. In such situations, the `seq_udp_delay` value of 10ps is used to ensure delays occur from data input to data output on sequential devices. Care must be taken to ensure the time unit of individual modules in the design and libraries support this value in their timescale directives. The proper application of `seq_udp_delay` can be inspected on any registered signal in the simulation waveforms, as displayed in [Figure 4-8](#) on page 54, where the `mbist_address` output on the MBIST engine changes value.

Back-annotated SDF simulation is also possible. Changes to the simulation scripts generated by `write_mbist_testbench` are necessary to support this properly.

Encounter Test: Flow: MBIST Analysis

Design Verification

Figure 4-8 Verifying Proper seq_udp_delay Behavior



■ Encounter Test patterns test cycle number and odometer readings

Test cycles may vary in their period and more than one may be used in the experiment. In a typical memory built-in self-test experiment, there exists a static test cycle in which JTAG operates to load and unload test data registers. This test cycle period is based on the JTAG TCK clock period.

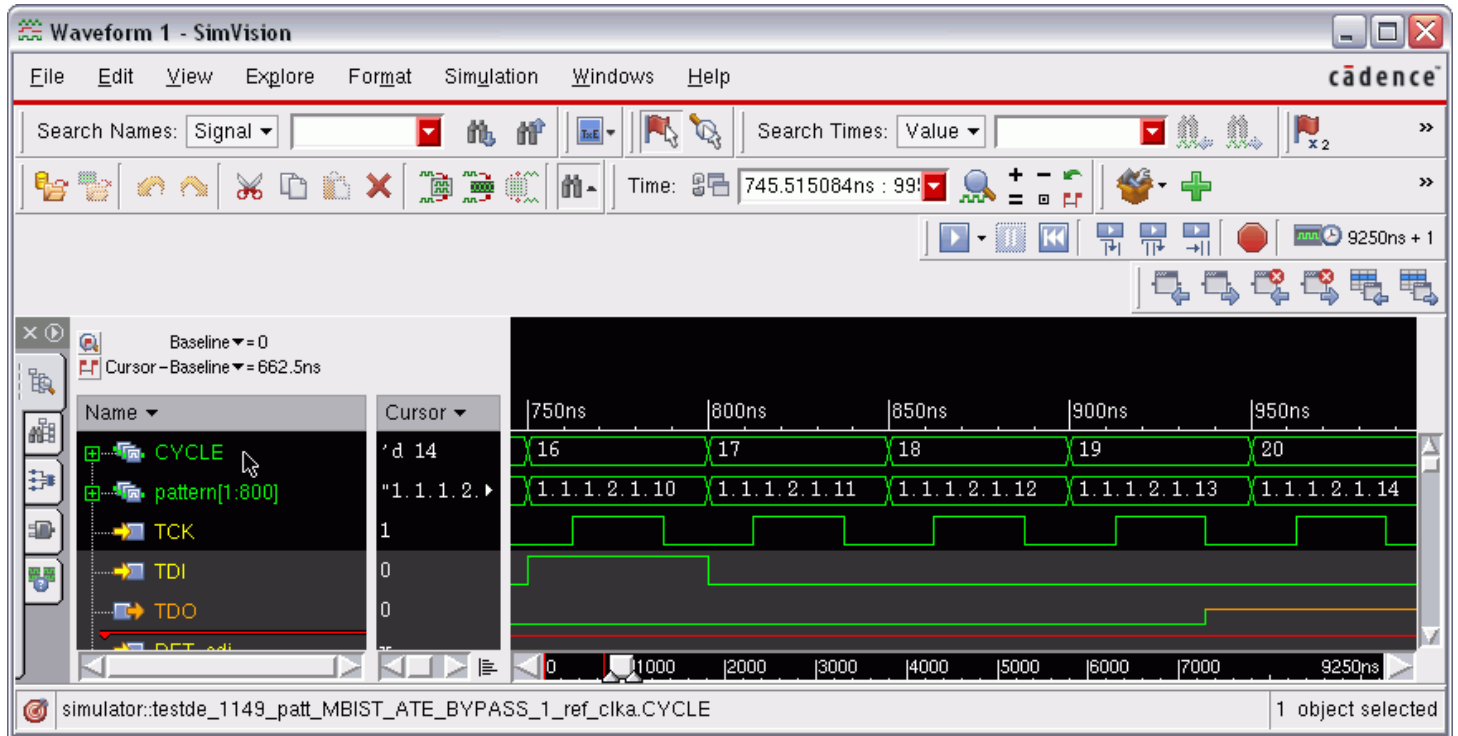
The static test cycle boundaries (*CYCLE* in the waveform) and the Encounter Test identifier used to mark them uniquely, the odometer (*pattern* in the waveform), are visible in the simulation waveforms for correlation with the pattern activity as displayed in Figure 4-9 on page 55.

Encounter Test: Flow: MBIST Analysis

Design Verification

The actual execution of the memory built-in self-test algorithms usually occurs within a set of dynamic test cycles whose period is based on the clock driving the MBIST logic. In these dynamic test cycles, only the MBIST clock(s) are pulsed.

Figure 4-9 Viewing test cycle and pattern



■ Encounter Test patterns test cycle event order

Memory built-in self-test pattern creation relies upon certain relationships between the applied stimulus, pulsed clocks, and measurements within a test cycle. The following inequality must be satisfied using the `write_vectors` keywords for proper pattern generation:

`testpioffset=testbidioffset < teststrobeoffset < testpioffset` for JTAG TCK pin

This can be verified through examination of the header in the `VER.testmode-name.experiment-name.mainsim.v` after generating the Verilog testbenches. From [Figure 4-10](#) on page 56, the extracted values are shown in the inequality.

Encounter Test: Flow: MBIST Analysis

Design Verification

Figure 4-10 mainsim.v Header Sample

```
0ps=testpioffset=testbidioffset < 5000ps=teststroboffset < 12500ps=testpioffset
// PROJECT NAME.....testde //
// TESTMODE.....1149_patt //
// INEXPERIMENT.....MBIST_ATE_BYPASS_1_ref_clka //
// TDR.....A6672m.mbist //
// TEST PERIOD.....50000.000TEST TIME UNITS.....ps //
// TEST PULSE WIDTH.....25000.000 //
// TEST STROBE OFFSET.....5000.000TEST STROBE TYPE.....edge //
// TEST BIDI OFFSET.....0.000 //
// TEST PI OFFSET.....0.000 X VALUE.....X //
// TEST PI OFFSET for pin "TCK" (PI # 13) is .....12500.000 //
// TEST PI OFFSET for pin "ref_clka" (PI # 25) is .....25000.000 //
//
```

■ ROM data load accessible

If ROMs are being tested by MBIST the data load files must be accessible in the simulation environment. Ensure \$readmem error messages like the given example do not exist in the simulation log.

```
Warning! $readmem error: open failed on file "v2ssl_256x8cm16.hex"
File: ./v2ssl_256x8cm16.v, line = 133, pos = 49
Scope: testde_mda_MBIST_ATE_BURNIN_1.TopBlock_inst.l1b1.l2m3.sram.uut
Time: 0 FS + 0
```

MBIST Startup

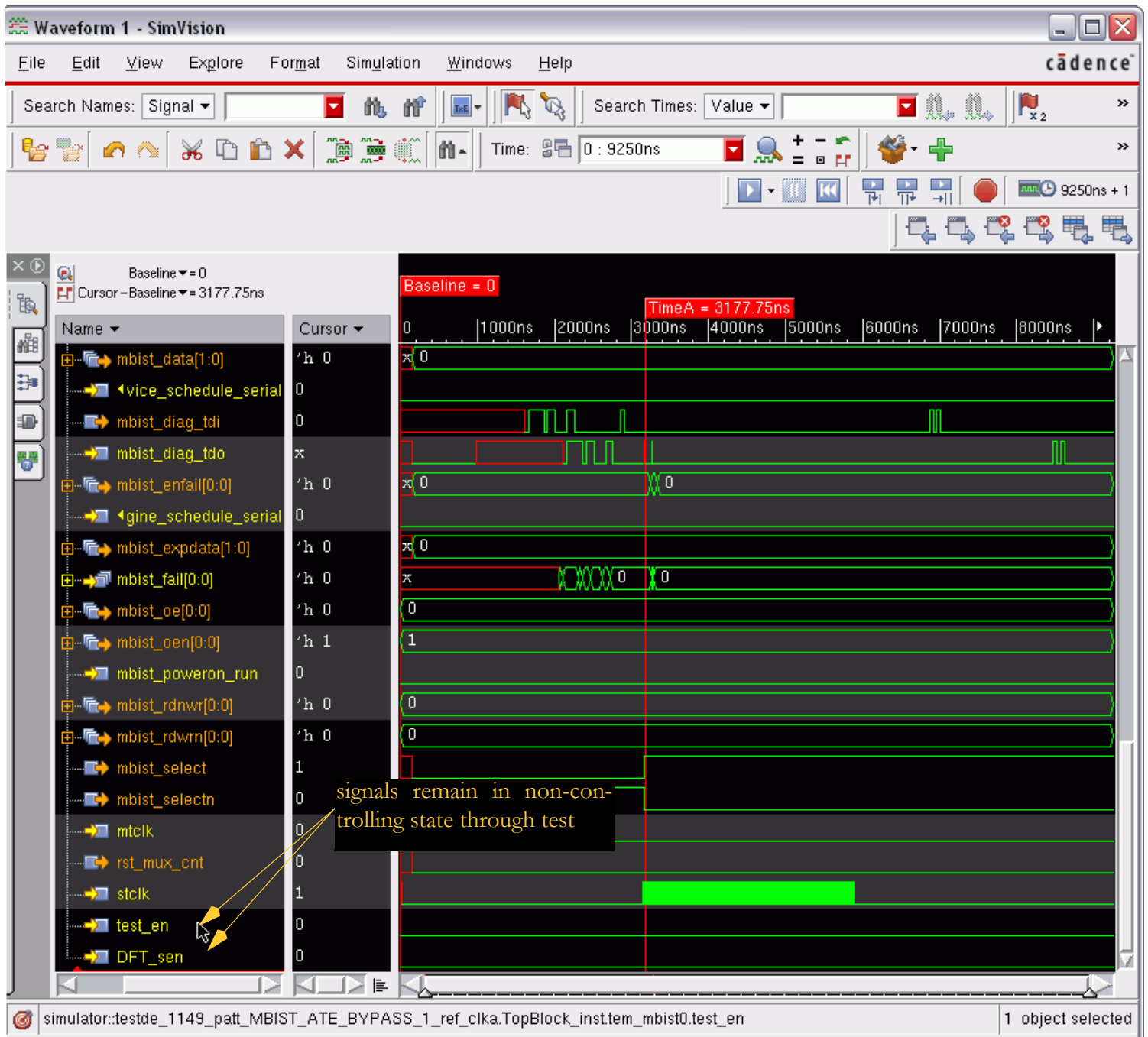
There are a number of simple checks covering the more common problems to verify that MBIST logic is operating properly at initiation.

■ test_enable/dft configuration mode stability

Whether an MBIST DFT configuration mode was established or simply a test signal used, the test_en input and any ATPG shift enables to all MBIST engines must be in non-controlling states for MBIST to operate. Refer to [Figure 4-11](#) on page 57.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-11 Test Control Signal Stability Check

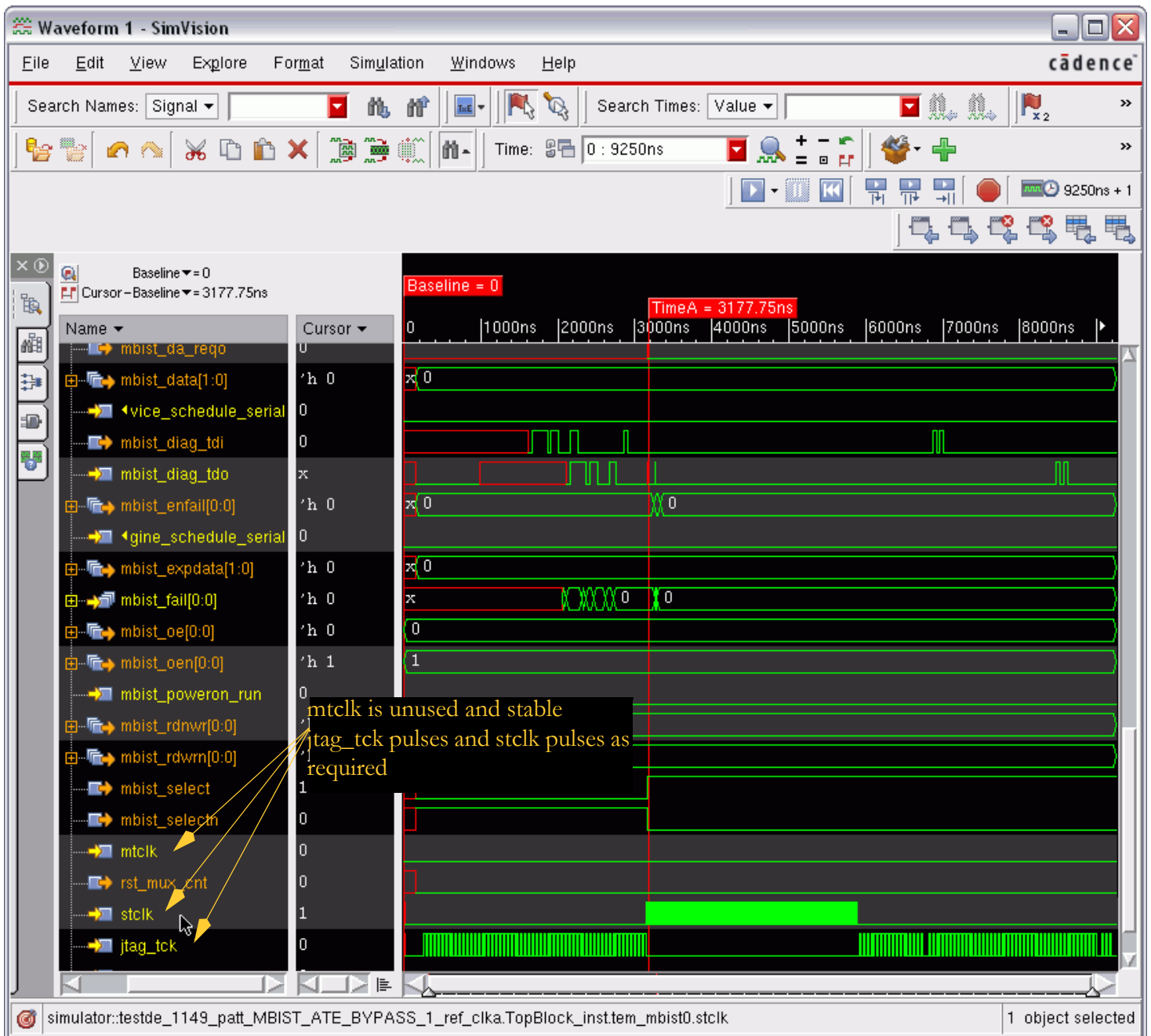


■ Clock Controllability

All clocks connected to all MBIST engines must be controlled throughout the MBIST execution. If the `mtclk` input to the MBIST engine is not used, the RTL Compiler `insert_dft mbist` command ties the input inactive at insertion. If the MBIST is controlled only by the direct access interface, the RTL Compiler `insert_dft mbist` command ties the `jtag_tck` input inactive at insertion. All other situations are the responsibility of the design to ensure properly controlled clocks to the MBIST engines for `jtag_tck`, `stclk`, and `mtclk` inputs. Refer to [Figure 4-12](#) on page 59.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-12 Clock Controllability



Encounter Test: Flow: MBIST Analysis

Design Verification

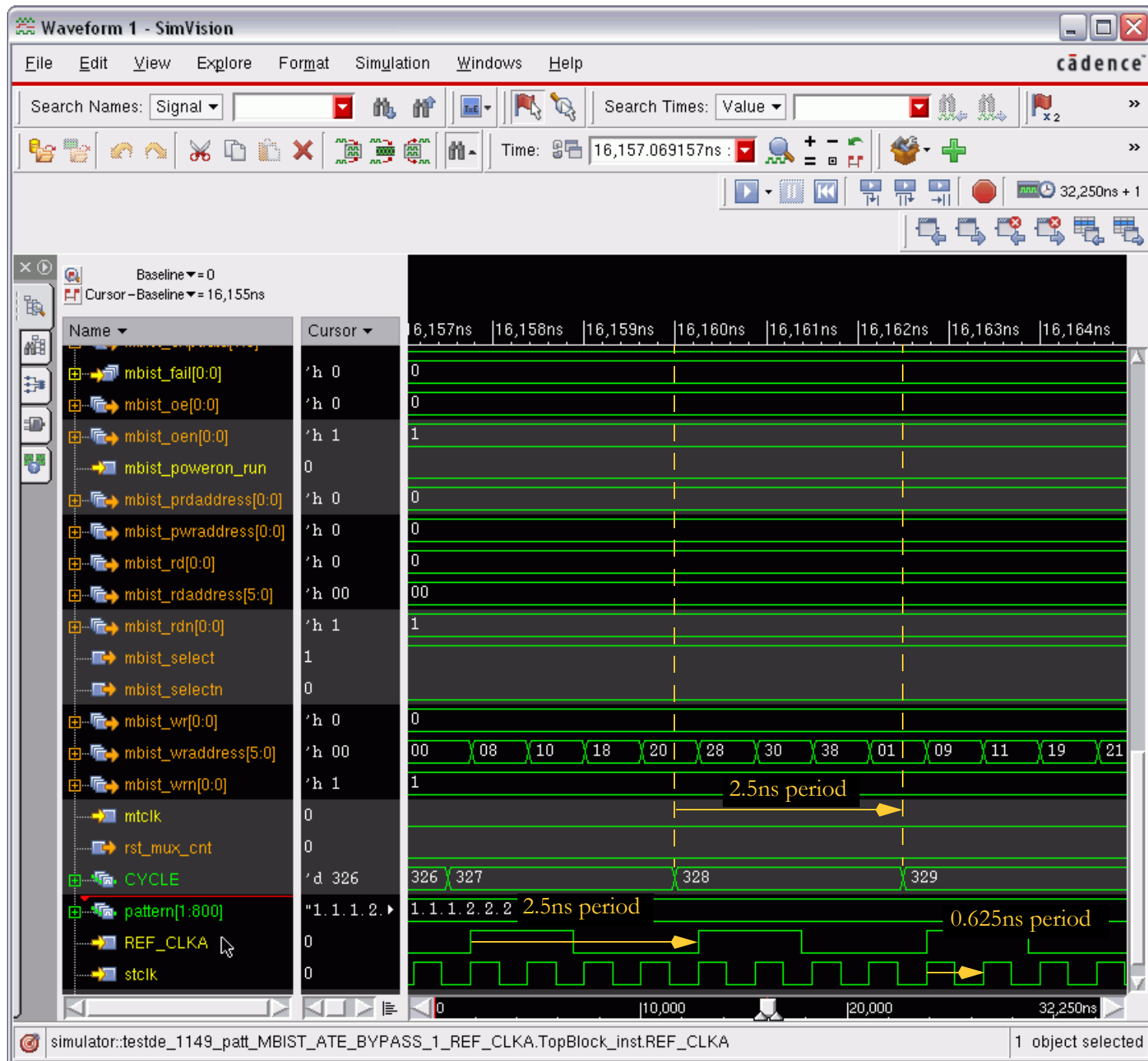
■ Clock Frequency Checks

Each MBIST clock frequency is defined prior to insertion in RTL Compiler using `define_dft mbist_clock` commands. Verifying the accuracy during pattern execution is essential as these definitions are used in the calculation of MBIST runtimes in the patterns run on the tester. The given RTL Compiler command defines a design port REF_CLKA running at 400MHz connected to a PLL with output frequency four times that value, 1600MHz. This is verified by inspection of the CYCLE, REF_CLKA and stclk waveforms in [Figure 4-13](#) on page 61.

```
define_dft mbist_clock -name REF_CLKA -period 2500 -hookup_period 625 -hookup_pin\  
    ref_clka_pll/pll_outa REF_CLKA
```

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-13 Clock Frequency Checks



Encounter Test: Flow: MBIST Analysis

Design Verification

■ JTAG Concerns

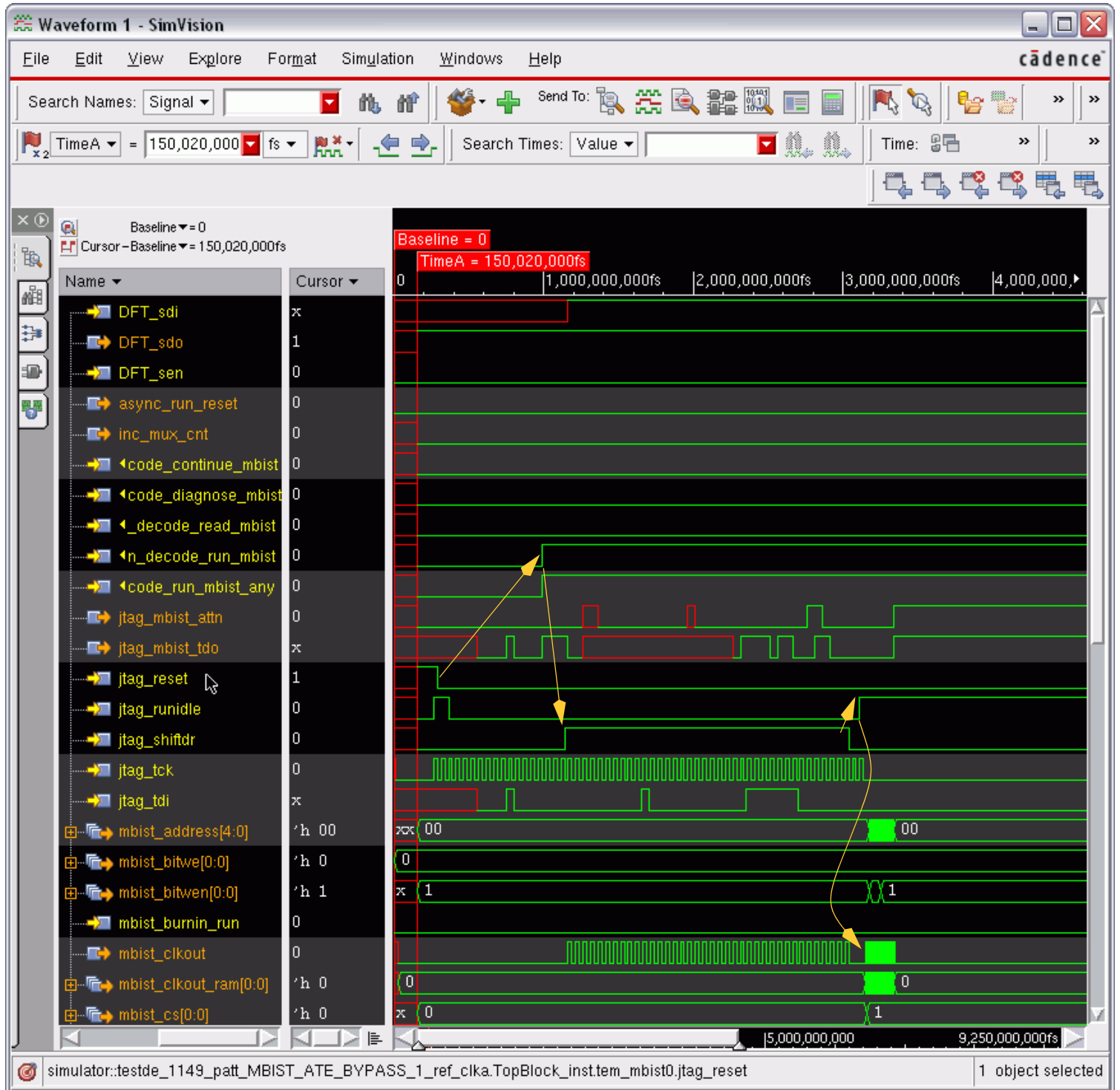
When the 1149.x Test Access Port is used to control MBIST, verifying it is operating properly at the MBIST engine ports is relatively simple. The following activity should be observed on the MBIST engine ports:

- ❑ `jtag_reset` should be pulsed high indicating the TAP was reset
- ❑ `jtag_instruction_decode_run_mbist` should be asserted to start the MBIST operation
- ❑ `jtag_shiftdr` should be asserted while `jtag_tck` is pulsed to load the MBIST test data register
- ❑ after `jtag_shiftdr` drops, `jtag_runidle` is asserted, signaling the transfer of control to the MBIST engine `stclk` or `mtclk` domain is commencing.

For details, refer to the following figure.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-14 JTAG Concerns



Encounter Test: Flow: MBIST Analysis

Design Verification

■ Direct Access Concerns

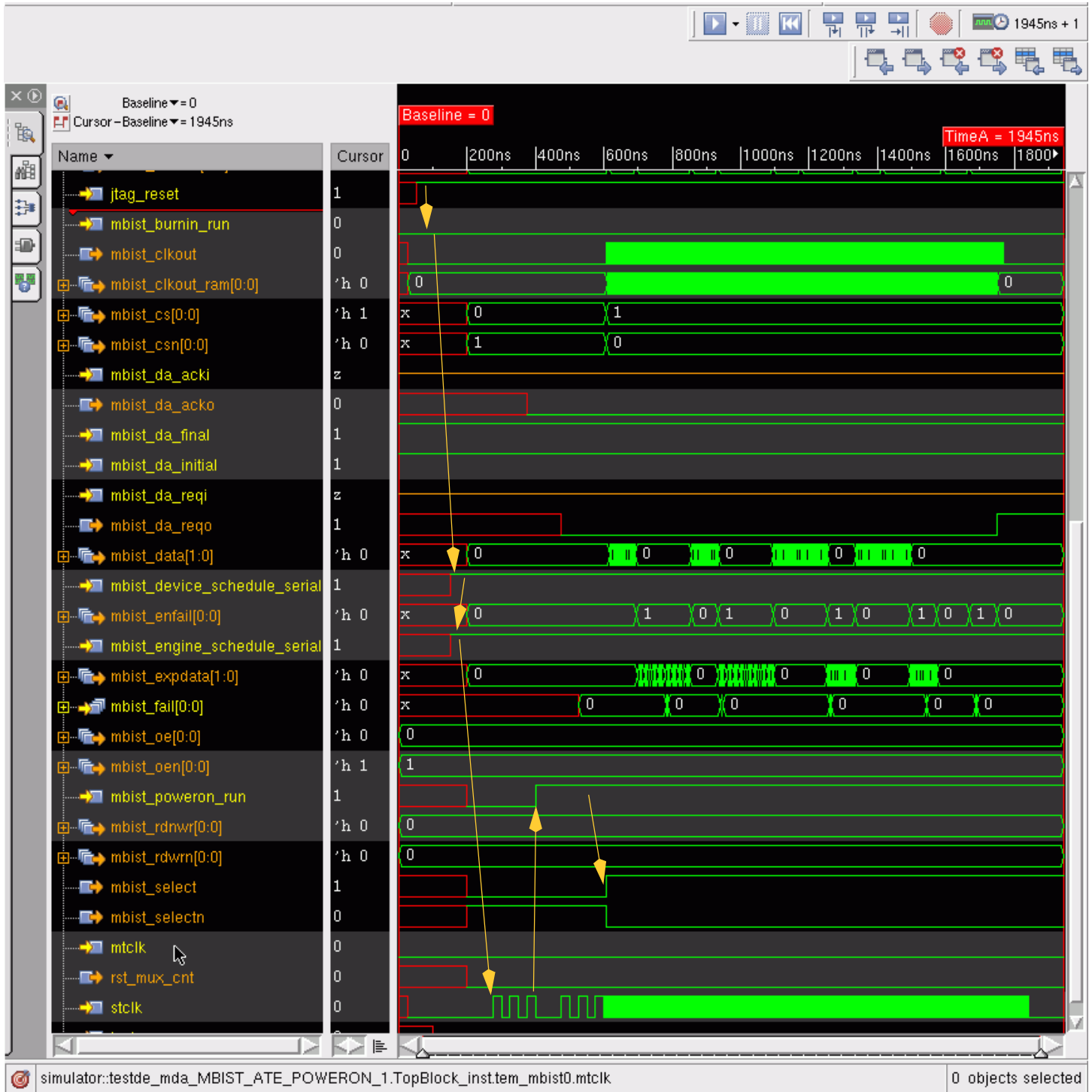
When the direct access mechanism is used to control MBIST, verifying it is operating properly at the MBIST engine ports requires observing the following activity on these ports:

- ❑ `jtag_reset` is held active or all jtag MBIST related instruction signals to the MBIST engine remain in their non-controlling states
- ❑ `mbist_device_schedule_serial` and `mbist_engine_schedule_serial` should be static and controlled high or low throughout the test sequence with their states dependent upon the scheduling criteria selected by the designer
- ❑ one of `mbist_burnin_run` and `mbist_poweron_run` is held inactive while the other is used to initiate the MBIST operations; the controlling signal should be observed inactive for a minimum of three clock pulses at the MBIST engine then be asserted
- ❑ after a minimum of three additional clock cycles transfer of control to the MBIST engine `stclk` or `mtclk` domain commences.

For details, refer to the following figure.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-15 Direct Access Concerns



Encounter Test: Flow: MBIST Analysis

Design Verification

■ Proper Startup Markers

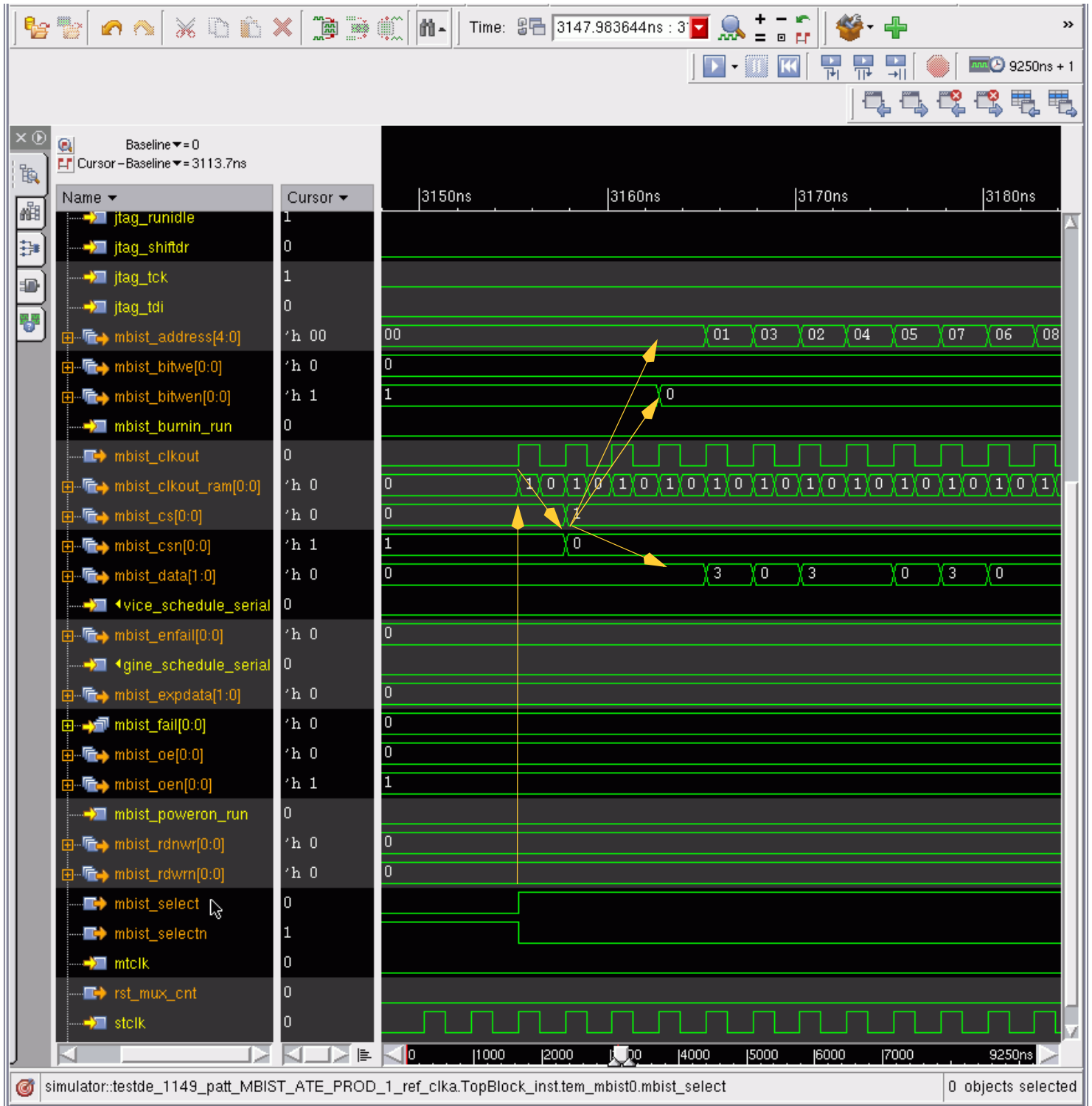
Whether the JTAG or direct access mechanism is used to control MBIST, the same actions occur at transfer of control to the MBIST engine stclk or mtclk domain. The following activity should be observed on the MBIST engine ports:

- ❑ `mbist_select` is asserted high at the time the first `mbist_clkout` pulse occurs and the set of scheduled memory devices associated with this BIST engine have their respective `mbist_clkout_ram[]` port activated, either delivering clock pulses or an asserted MBIST clock gate enable signal
- ❑ `mbist_cs` and `mbist_csn` then activate to enable the associated set of memories
- ❑ `mbist_bitwe[]` and `mbist_bitwen[]` activate as required to indicate the initial memory write operations of the algorithm are commencing along with changes to the `mbist_address` and `mbist_data[]` ports

Refer to [Figure 4-16](#) on page 67 for details in the waveforms.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-16 Proper Startup Markers



Memory Connections and Activity

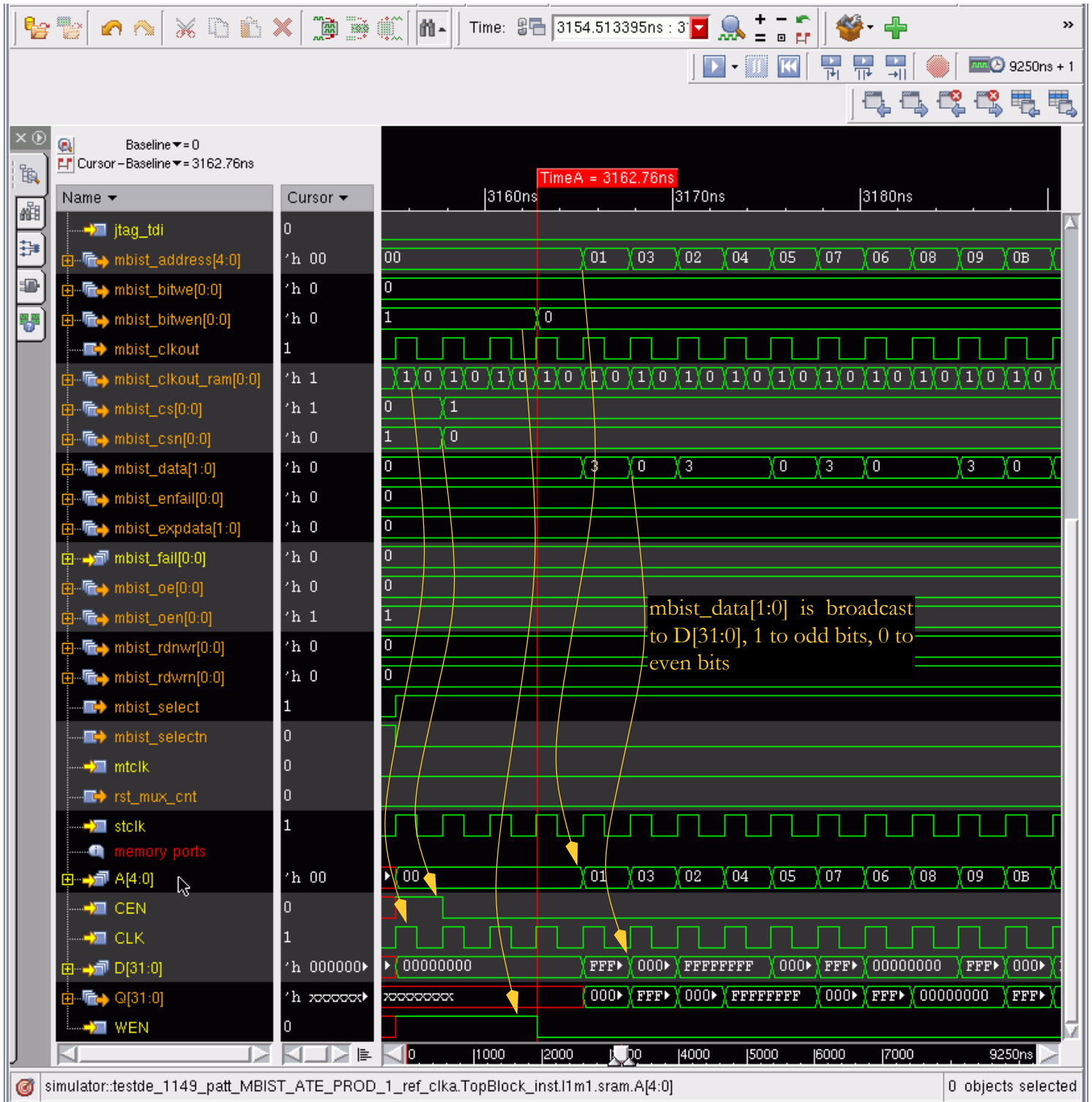
This section deals with rudimentary checks concerning properly controlled memory devices associated with MBIST engines during algorithm execution. The checks are performed at both MBIST and memory device ports to ensure the proper transfer of information.

■ Write Operation Markers

Write controls, address, and data are supplied directly from the MBIST engine to the target devices in the same clock cycle. Controls must be in the proper state to enable the write operations: CEN low and WEN low is this example. For more details, refer to [Figure 4-17](#) on page 69. Note the first write operation is being transferred to the memory device starting with the clock edge at the red cursor.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-17 Write Operation Markers



■ Read Operation Markers

Read controls, address, and data are supplied directly from the MBIST engine to the target devices in the same clock cycle. Controls must be in the proper state to enable the read operations: CEN low and WEN high is this example. Refer to [Figure 4-18](#) on page 71 for more details. Note the first read operation is being transferred to the memory device starting with the clock edge at the red cursor.

In this case, the memory device has a `read_delay 2` value specified in the memory BIST configuration file, meaning it takes one clock cycle to transfer the read request to the memory and a second clock cycle to access memory and transfer data from the memory to the MBIST comparator. This should be verified in the waveforms or MBIST execution failures will result. Also note in this algorithm that two reads are occurring for each memory address in consecutive cycles.

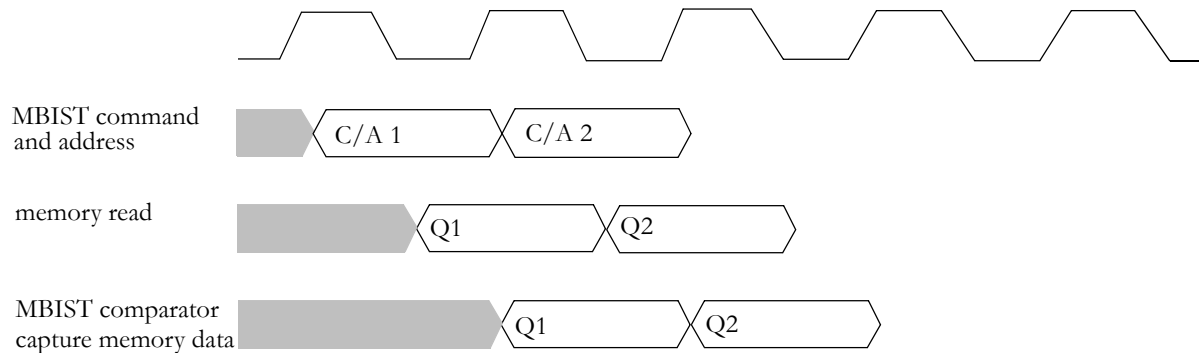
Encounter Test: Flow: MBIST Analysis

Design Verification

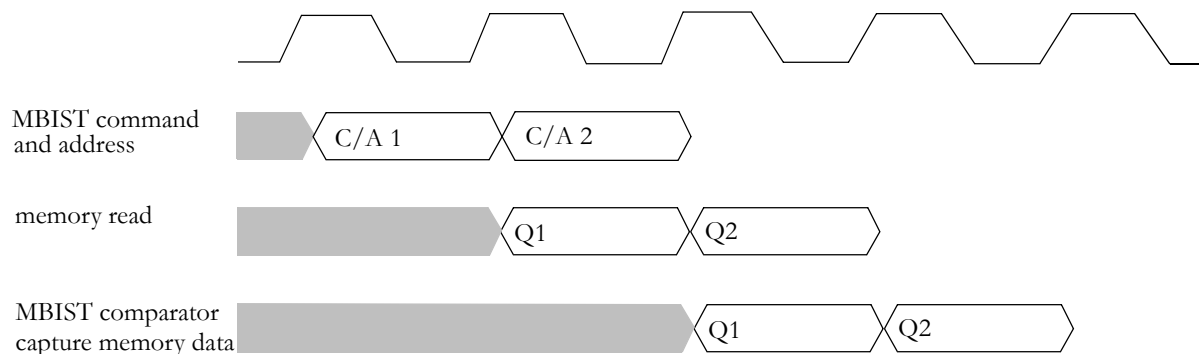
■ Memory Read Data Cycle Timing Variations

The number of clock cycles required to read memory data from presentation of the command and address on the MBIST engine signal interface to the memory device must be identified in the memory module definition within the configuration file supplied to the RTL Compiler `insert_dft mbist` command. Three likely scenarios are shown in the following examples, distinguished by their corresponding MBIST configuration file `read_delay` values.

`read_delay 1` - Asynchronous Memory Read Timing



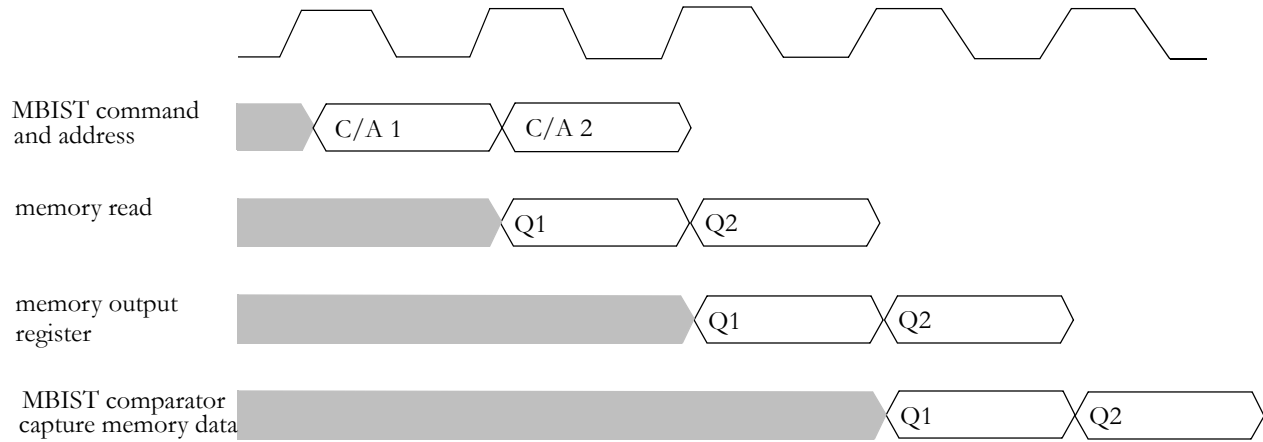
`read_delay 2` - Synchronous Memory Read Timing (Default Case)



`read_delay 3` - Memory Output Data Pipeline Register Timing

Encounter Test: Flow: MBIST Analysis

Design Verification



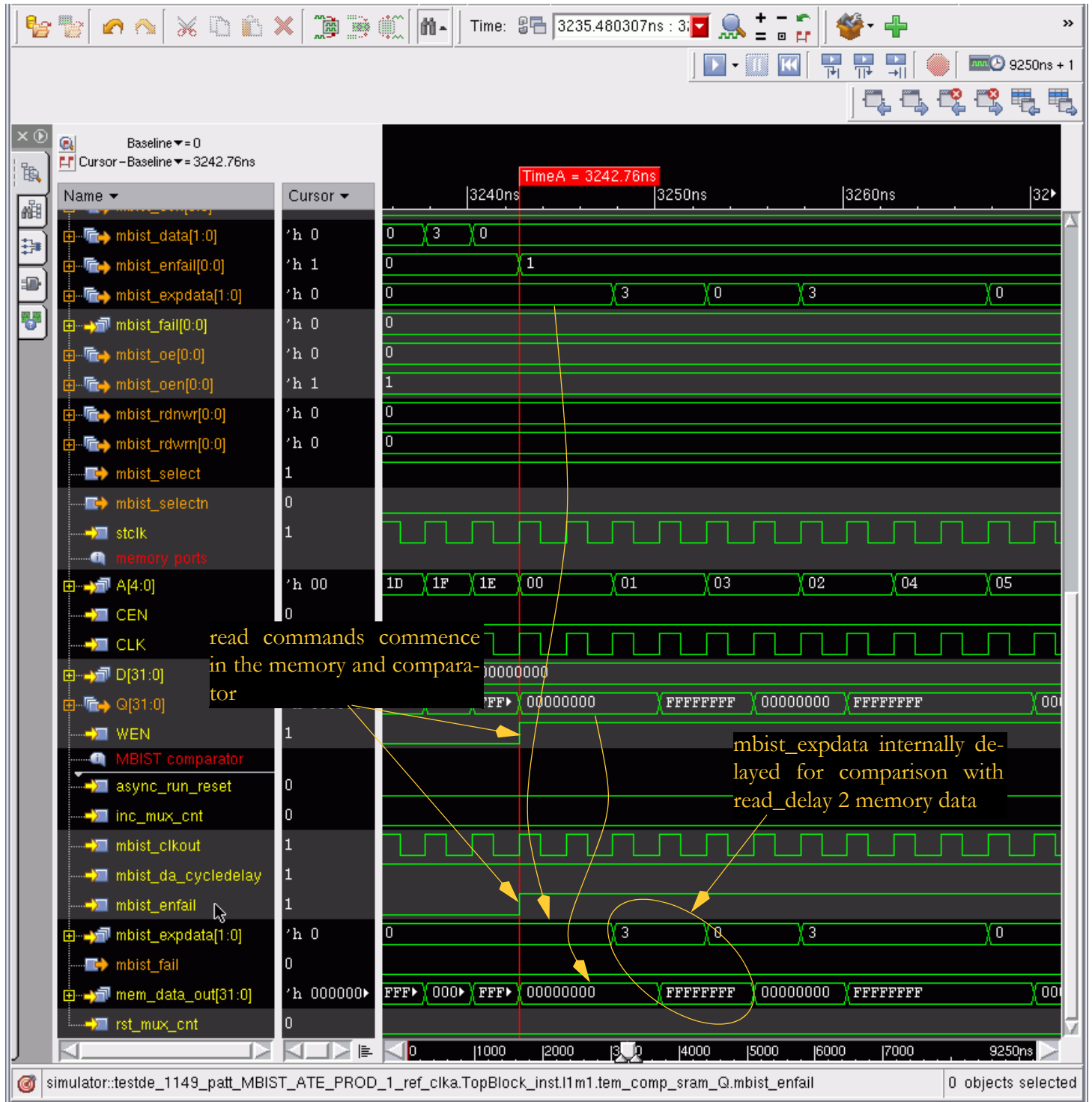
MBIST Comparators

■ Actual and Expect Data Comparison

Comparison of data read from memory with expected values from the algorithms is how MBIST determines failures. Expected data values, `mbist_expdata[]`, and a read enable, `mbist_enfail[]`, are supplied by the MBIST engines to the comparators in the same cycle as the read command and address are presented to the associated memories. Variations in memory read timings are managed in the individual comparators to allow memory devices of different characteristics to be tested simultaneously. The `mbist_fail` signal from the comparator indicates when a miscompare has occurred. Refer to the following figure for details.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-19 Actual and Expect Data Comparison



■ Failure Indication Variations

For SRAMs, `mbist_fail` asserted indicates a miscompare has been detected within the MBIST comparator on a given memory port in a given cycle. It can be asserted only as a result of a valid read operation in which a miscompare is detected, otherwise it is zero. The relationship between a particular memory read operation and the `mbist_fail` signal is a function of the `read_delay` of the memory and whether or not the memory read data is registered within the comparator prior to analysis. [Table 4-4](#) on page 75 summarizes this relationship relative to the clock cycle in which the read command is presented on the interface between the MBIST engine and the associated memory device.

For ROMs, `mbist_fail` is asserted from the beginning of the test, only deasserting at the end of the test if an all zeroes signature results in the MISR.

In some cases it is possible for a failure indication to occur at the output of an MBIST comparator but not be valid for capture within the corresponding engine. This situation may arise for the unselected memory devices when a subset of the memories associated with a MBIST engine are scheduled for execution.

Table 4-4 MBIST SRAM Failure Indication Timing

memory read_delay	memory data registered in comparator	comparator mbist_fail signal active in cycle
1	no	2
1	yes	3
2	no	3
2	yes	4

MBIST Completion

■ Done Register and Summary Failure Register Inspection

Once a MBIST engine has completed execution, its done register is asserted and remains asserted until reset by external controls. For JTAG-initiated MBIST, the done state is observed on the `jtag_mbist_tdo` engine port and reset during shifting of the MBISTDIAG test data register when the `jtag_instruction_decode_diagnose_mbist` input is active. Refer to [Figure 4-20](#) on page 77.

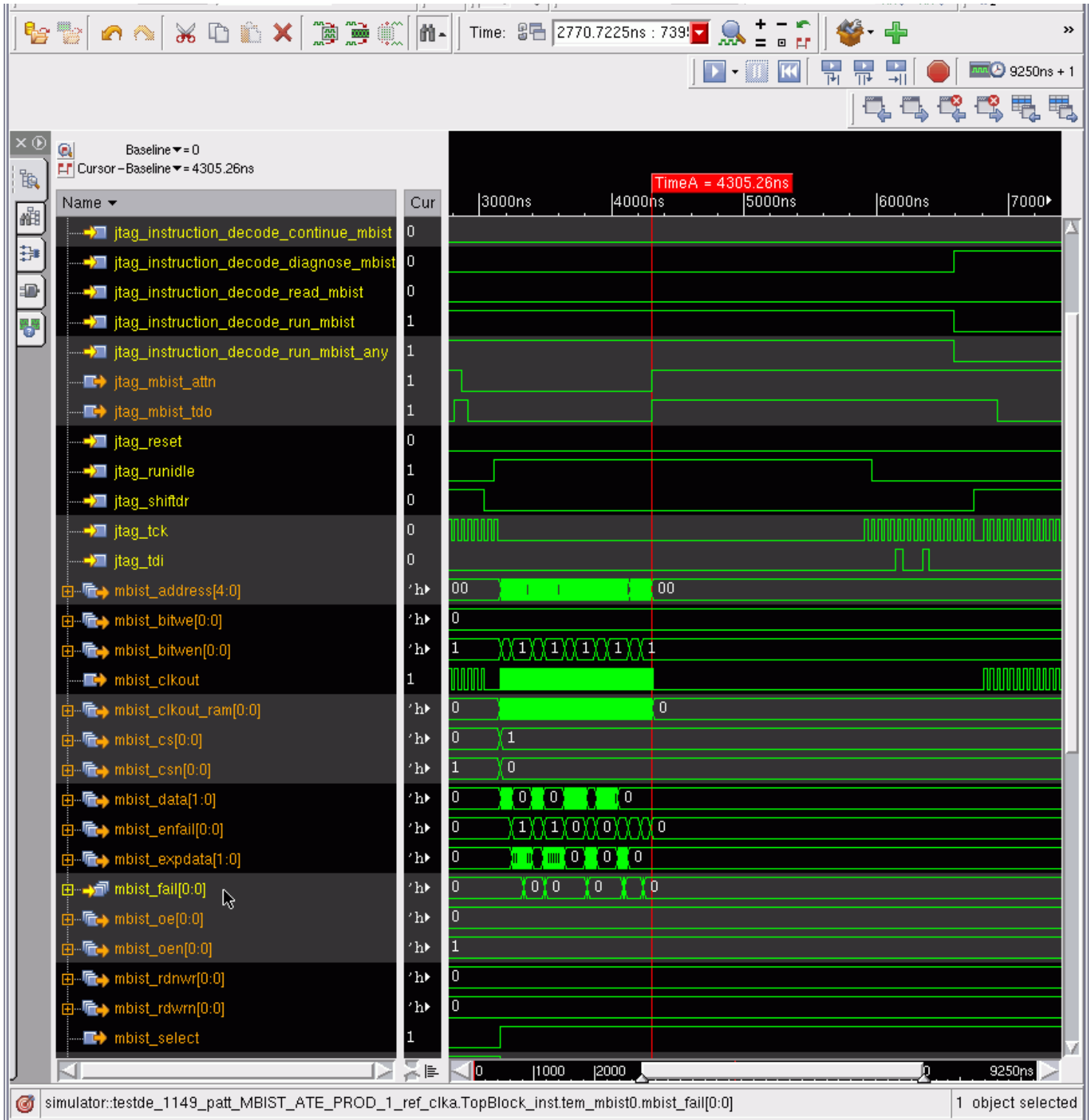
Encounter Test: Flow: MBIST Analysis

Design Verification

The `jtag_mbist_attn` port also indicates the done register status but it is sticky, not being reset during the `jtag_instruction_decode_diagnose_mbist` instruction. This same MBIST engine port indicates when a miscompare is detected during diagnostic execution or a retention test pause has occurred. Both situations force the MBIST engine to suspend execution.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-20 Done Register and Summary Failure Register Inspection



Encounter Test: Flow: MBIST Analysis

Design Verification

■ Direct Access Monitor Inspection

The direct access monitor is the logical AND of all MBIST engine `jtag_mbist_attn` signals in the design. Within each engine for poweron patterns, the done register is logically ANDed with the inverted summary failure register which drives the `jtag_mbist_attn` port, allowing for a passing/failing die indication to be reported at the monitor.

Determining Miscomparing Registers from Simulation Logs

The Encounter Test `analyze_embedded_test` command has the capability to process simulation log files to interpret failure data. This capability includes the generation of an auxiliary detailed register miscompare simulation log file named as the original with a “.translated” suffix appended and containing a line for each miscompare cross-referenced to the original simulation log.

The simulation log file snippet for the production class test

```
WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.20 at Time: 6855000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.21 at Time: 6905000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.56 at Time: 8655000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.58 at Time: 8755000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

INFO (TVE-201): Simulation complete on vector file:
test_mult_1rw/testde/testresults/embedded_macros/dv/verilog/VER.1149_patt.MBIST_ATE_PROD_1_ref_clk.

INFO (TVE-206): The number of good comparing vectors for the file just completed is 41

INFO (TVE-205): The number of miscomparing vectors for the file just completed is 4
```

is processed by `analyze_embedded_test` to yield a failing test status due to injected faults

```
INFO (TEM-301): Parsing the test data register mapping file 'test_mult_1rw/testde/mbist/TopBlock_mbi
INFO (TEM-500): Parsing the pattern_control file 'test_mult_1rw/testde/mbist/TopBlock_pattern_contro
INFO (TEM-311): Parsing the simulation log file 'test_mult_1rw/testde/logs/simMBIST_ATE_PROD_1_ref_c
INFO (TEM-304): Total Failures Analyzed: 1 [end TEM_304]

-----
Engine      Target RAM      Test Status      Bits
-----
0           0 1l1m1/sram    Failed           .

INFO (TEM-314): Successfully analyzed failures.
Output file is: 'test_mult_1rw/testde/testresults/logs/log_analyze_embedded_test_031
```

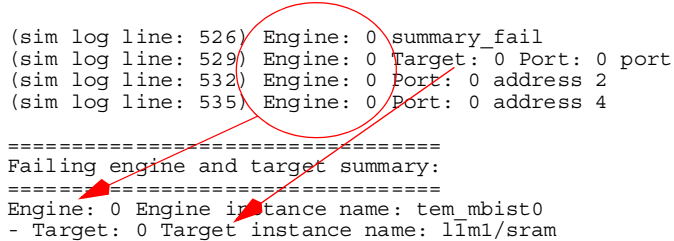
Encounter Test: Flow: MBIST Analysis

Design Verification

and further analyzed for failing register data written into the corresponding *.translated file.

```
(sim log line: 526) Engine: 0 summary_fail
(sim log line: 529) Engine: 0 Target: 0 Port: 0 port
(sim log line: 532) Engine: 0 Port: 0 address 2
(sim log line: 535) Engine: 0 Port: 0 address 4

=====
Failing engine and target summary:
=====
Engine: 0 Engine instance name: tem_mbist0
- Target: 0 Target instance name: l1m1/sram
```



This *.translated file also identifies within the design the MBIST block(s) and targeted memory devices associated with the detected mismatches.

An alternative involves manual lookup using the files referenced by `analyze_embedded_test`. [Figure 4-21](#) on page 80 contains an example of this process.

Encounter Test: Flow: MBIST Analysis Design Verification

Figure 4-21 Analyzing Simulation Log Failing Registers

From the *design_pattern_control.txt* file find the first measure for the production class pattern.

```
## TBD Odometer Values
production_first_measure=1.1.1.2.3.19.1
diagnostic_first_measure=1.1.1.2.3.19.1
bitmap_first_measure=1.1.1.2.3.43.2
poweron_first_measure=1.1.1.2.4.1.1
```

Then, subtracting this value from the miscomparing pattern value in the simulation log file, $20 - 19 = 1$ is the index into the *design_mbistdiag_tdr_map.txt* file.

```
WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.20 at Time: 6855000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.21 at Time: 6905000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.56 at Time: 8655000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

WARNING (TVE-650): PO miscompare at pattern: 1.1.1.2.3.58 at Time: 8755000000 at EAWoffset: -1
Expected: 0 Simulated: 1 On PO: TDO

INFO (TVE-201): Simulation complete on vector file:
test_mult_lrw/testde/testresults/embedded_macros/dv/verilog/VER.1149_patt.MBIST_ATE_PROD_1_ref_clka.data.logic.ex1.t

INFO (TVE-206): The number of good comparing vectors for the file just completed is 41

INFO (TVE-205): The number of miscomparing vectors for the file just completed is 4
```

This identifies the summary fail register in MBIST engine *tem_mbist0* within the design.

```
## Version information
RC generated version 10.1.300 TopBlock_mbistdiag_tdr_map.txt
```

engine	target	port	tdr bit index	function	instance name	redundancy capability
0	.	.	0	done	tem_mbist0	.
0	.	.	1	summary fail	.	.
0	0	0	2	port	l1m1/sram	b:1[r:2,c:2,d:0]m:2
0	0	0	3	slice	.	.
0	0	0	4	slice	.	.
0	0	0	5	slice	.	.
0	0	0	6	slice	.	.
0	0	0	7	slice	.	.
0	0	0	8	slice	.	.
0	0	0	9	slice	.	.
0	0	0	10	slice	.	.
0	0	0	11	slice	.	.
0	0	0	12	slice	.	.
0	0	0	13	slice	.	.
0	0	0	14	slice	.	.
0	0	0	15	slice	.	.
0	0	0	16	slice	.	.
0	0	0	17	slice	.	.
0	0	0	18	slice	.	.
0	0	0	19	slice	.	.
0	0	0	20	slice	.	.
0	0	0	21	slice	.	.

Manufacturing Test Support

FDS to CPP to analyze_embedded_test Flow

The manufacturing automated test equipment generates failing data sets (FDS) which can be translated to CPP format prior to processing with `analyze_embedded_test`. Only chip-level memory built-in self-test pattern generation and simulation are supported.

By executing `report_vectors` with the *Include vector correspondence* button selected in the Encounter Test GUI (`vectorcor=yes`) for a memory built-in self-test pattern, the output file generated contains a numbered list of the pins on the part. A portion of this file can be used to build a pin correlation file for use in the `FDSlog2CPP` command. The output pin portion of the file is displayed in the following snippet. The highlighted line is used during JTAG-controlled memory built-in self-test patterns to monitor the output and determine correct operation. If bitmap patterns are generated using measure ports beyond the JTAG TDO port, they must be included in the pin correlation file as well.

Figure 5-1 Portion of Vector Correspondence

```
# PO:
# (PO 1 = "Pin.f.l.TopBlock.nl.BOUNDARY_REG_OUT_DUMMY", # index = 26
# PO 2 = "Pin.f.l.TopBlock.nl.MMP1", # index = 27
# PO 3 = "Pin.f.l.TopBlock.nl.MMP2", # index = 28
# PO 4 = "Pin.f.l.TopBlock.nl.MONITOR", # index = 29
# PO 5 = "Pin.f.l.TopBlock.nl.SO", # index = 30
# PO 6 = "Pin.f.l.TopBlock.nl.SO1", # index = 31
# PO 7 = "Pin.f.l.TopBlock.nl.SO2", # index = 32
# PO 8 = "Pin.f.l.TopBlock.nl.TDO", # index = 33 tf = TDO
```

This information can be used to create a file containing the following column headers and data extracted from the vector correspondence for use in the `FDSlog2CPP` script.

Figure 5-2 Pin Correlation File Example

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

PAD_PIN	PIN_NAME
8	TDO

The CPP file must be generated from the simulation FDS file.

```
$Install_Dir/etc/tb/contrib/FDSlog2CPP -fds_log_file simulation-fds-file  
-pattern_file STIL.testmode-name.experiment-name.logic.ex1.ts1.stil  
-pin_correlation_file pin-correlation-file -cpp_file generated-cpp-file
```

```
$Install_Dir/etc/tb/contrib/FDSlog2CPP -fds_log_file simulation-fds-file  
-pattern_file WGL.testmode-name.experiment-name.logic.ex1.ts1.wgl  
-pin_correlation_file pin-correlation-file -cpp_file generated-cpp-file
```

Valid `analyze_embedded_test` command line templates for the supported pattern classes are shown below:

- pattern class: bypass, production, retention (production)

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=production cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: diagnostic

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=diagnostic cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: diagnostic with software repair analysis

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=repair cpp=generated-cpp-file diagtdr=design_mbistdiag_tdr.map
```

- pattern class: bitmap, retention (bitmap)

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=bitmap cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
stclkstep=stclk-scheduling-step
```

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=bitmap cpp=generated-cpp-file bitmaptdr=design_mbistread_tdr.map  
mtclkstep=mtclk-scheduling-step
```

- pattern class: direct access

```
analyze_embedded_test patterncontrolfile=pattern-control-file  
analysis=poweron cpp=generated-cpp-file
```

Using the `prepare_memory_failset` Command

The `prepare_memory_failset` command uses the failing data sets (FDS) created by the manufacturing test equipment, the STIL or WGL patterns used to generate the failing data sets and the interface files generated by the RTL Compiler `insert_dft mbist` command

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

that was used to perform MBIST insertion to populate a manufacturing memory test database.

Note: Note: This script is not formally supported.

Basic Keywords

The working directory for Encounter Test is indicated to `prepare_memory_failset` through the `WORKDIR=work_directory` keyword. Each invocation of `prepare_memory_failset` creates a logfile which is saved in the `work_directory/testresults/logs` directory.

The location of the manufacturing database to filled by `prepare_memory_failset` is specified using the `rootdirectory=root_directory` keyword. The last directory name found in `root_directory` is required to be the *design* name being processed. Below the `root_directory` there must exist the TANSI.csv file, CDNS_info and FDS-DATA directories. The TANSI.csv file is a pin correlation file as described [Figure 5-2](#) on page 81. An example of the entire directory can be seen here: [Example 5-10](#) on page 90.

The CDNS_info directory contains one or more sub-directories, where each sub-directory name is specified as `design_instruction-set-index`. Located in each of these directories are the STIL or WGL patterns named as `design_instruction-set-index.[wgl|stil].testblockname` and the interface files. The FDS-DATA directory contains output generated by `prepare_memory_failset`.

Example 5-1 Sample contents of CDNS_info directory

```
root_directory/CDNS_info/
  design_1/
    design_1.wgl.DIAG001
    design_1.wgl.PROD001
    design_1_mbist_tdr_map.txt
    design_1_mbistdiag_tdr_map.txt
    design_1_pattern_control.txt
    design_1_rombistdiag_tdr_map.txt
  design_2/
    design_2.wgl.BIT01
    design_2.wgl.BIT01A
    design_2.wgl.BYP03
    design_2_mbist_tdr_map.txt
    design_2_mbistdiag_tdr_map.txt
    design_2_mbistread_tdr_map.txt
    design_2_pattern_control.txt
```

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

`design_2_rombistdiag_tdr_map.txt`

The failing data sets can be specified using one of two keywords. The `faildataloglist=comma_separated_filenames` keyword analyzes the fully qualified input files and is useful when a relatively few number of failing data sets require analysis. When many failing data sets are present, it is recommended to use the `faildatalogfile=faildata_filename` keyword. The file identified by this keyword contains one failing data set file on each line. Lines beginning with the "#" character are considered comments.

Example 5-2 Contents of file specified using the faildatalogfile keyword

```
/workdir/failure_data/set1/fds_log_file.1
/workdir/failure_data/set1/fds_log_file.2
# This is a comment
/workdir/failure_data/set2/fds_log_file.1
```

The failing data set file is generated by the manufacturing test equipment and contains information specific to the set of tests being executed. Other information included in the file gives details about the physical characteristics of the device under test and the test equipment used.

Example 5-3 Contents of failing data set file

```
#####
PROCESS000000003DE9
DATAREV.00000000
MBNO....000000009TopBlock
OPERCODE000000007831176
LOTNO...0000000105A10055-22
|MASKNO...000000007AB5C456
TSTRNAME000000008D3343-G2
STARTTIM0000000162011/02/07-11:00
MANPRNAM000000044ce96eyyzzx_73ab1_7815.tdl
USRSBNAM00000000
LIMITNAM00000000
PATNAM1.000000004EFT99
PATNAM2.000000026D-6D17634,21.44
PATNAM3.00000000
PATNAM4.00000000
PATNAM5.000000023B-3D87614,32.00
MANPREV0000000861394-72357,3.05
USRSBREV000000034T-XZ26-73494,21
LIMITREV00000000
PLANT...000000003FED
FLANG...000000006BOTTOM
```

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

```
LIFETIME00000002M07
LOGATTR.00000008FDS-DATA
LOGATTR200000000
F.LIMIT.0000000800001024
ENDTIM..000000162015/04/12-12:00
FDS-DATA00000018CI23022030serialno
FDS-DATA00000019TIPROD01 CCDI1 ?
FDS-DATA00000015FD00008e1d0008H
FDS-DATA00000015FD00008e1e0008H
FDS-DATA00000015FD00008e390008H
FDS-DATA00000015FD00008e3f0008H
FDS-DATA00000015FD00008e400008H
FDS-DATA00000015FD00008e440008H
FDS-DATA00000015FD00008e450008H
FDS-DATA00000015FD00008e470008H
```

A description of each processed line in the failing data set file follows:

■ TPROCESS00000003DE9

The *test-information* DE9 is extracted from this line. This is placed in the *fail-data-set.idx* file.

■ MBNO...00000009TopBlock

The *chip-name* TopBlock extracted from this line. This is placed in the *fail-data-set.idx* file.

■ LOTNO...000000105A10055-22

The *lot-number* 5A10055-22 is extracted from this line. This is placed in the *fail-data-set.idx* file and the *FABIAN.idx* file.

■ MASKNO..00000007AB5C456

The *mask-number* AB5C456 is extracted from this line. This is placed in the *fail-data-set.idx* file.

■ TSTRNAME00000008D3343-G2

The *tester* D3343 and *equipment* G2 is extracted from this line. This is placed in the *fail-data-set.idx* file.

■ PLANT...00000003FED

The *plant* FED is extracted from this line. This is placed in the *fail-data-set.idx* file.

■ FLANG...00000006BOTTOM

The *facet* BOTTOM is extracted from this line. This is placed in the *fail-data-set.idx* file.

■ F.LIMIT.0000000800001024

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

The *fail-limit* 1024 is extracted from this line. This is placed in the *fail-data-set.idx* file.

- ENDTIM..000000162015/04/12-12:00

The *day-time* 2015/04/12-12:00 is extracted from this line. This is placed in the *fail-data-set.idx* file.

- FDS-DATA00000018CI23022030serialno

The *wafer-number* 23, *chip-x-coordinate* 022 and *chip-y-coordinate* 030 is extracted from this line. This is placed in the *FABIAN.idx* file.

- FDS-DATA00000019TIPROD01 CCDI1 ?

The *testblockname* PROD01 and *test-condition* CCDI1 are extracted from this line. The *testblockname* is placed in the *FABIAN.idx* file, and the *test-condition* is used as part of the directory name created for each failing data set processed. The directory name is *root_directory/FDS_data/fail-data-set-log_test-condition.fdd*.

- FDS-DATA00000015FD00008e1d0008H

The *failing-tester-cycle* 00008e1d, the *pin-index* 0008 and the measured-value H are extracted from this line. Both hex and decimal values are allowed for the failing-tester-cycle. Valid measure values are H (high or 1) and L (low or 0). The *TANSI.csv* pin correlation file is used to translate the pin index to a pin name where the measure occurred.

If during translation of the STIL or WGL pattern files to the test equipment language additional cycles are introduced, or additional setup time is required for the part at the tester, then the *patternoffset=integer* keyword can be used to synchronize the data. This offset is passed to the *FDSlog2CPP* command being executed by *prepare_memory_failset* to generate CPP files with the correct data.

Commands invoked by *prepare_memory_failset*

After all of the interface files, pattern files and failing data sets have been processed, *prepare_memory_failset* invokes the *FDSlog2CPP* command to generate CPP files which are then processed by the *analyze_embedded_test* command to create tables indicating passing and failing devices. Data from these tables are gathered and placed in several summary files created by *prepare_memory_failset*.

Outputs

The FBI file

One of the output files created by `prepare_memory_failset` is the FBI file. This file summarizes the failing and passing devices for a particular failing data set, test condition, wafer, chip x coordinate, chip y coordinate and testblock. Detailed information is provided about each memory instance.

Example 5-4 Location of FBI file

```
root_directory/FDS_data/fail-data-set-log_test-condition.fdd/  
  wchip/  
    cchip-x-coordinchip-y-coordinate/  
      testblockname.FBI
```

Example 5-5 Contents of FBI file

```
NAME:TopBlock.11b1.12m3.sram;  
PARTIAL:ON;  
BIT:2;  
WORD:17;  
COLUMN:1;  
PORT:1;  
WNAM:23;  
CNAM:022030;  
PFFLG:F;  
  
FKIND:MARCH;  
FDATACON:2;  
DATA:2,0,0,A;  
DATA:2,0,0,A;  
END;  
NAME:TopBlock.11m2.sram;  
PARTIAL:ON;  
BIT:64;  
WORD:1024;  
COLUMN:0;  
PORT:2;  
WNAM:23;  
CNAM:022030;  
PFFLG:P;  
FKIND:MARCH;  
FDATACON:0;  
END;
```

A description of each line in the FBI follows:

■ NAME

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

The full hierarchical memory instance name.

- PARTIAL

This value is always set to ON.

- BIT

Data width (number of bits) for the device.

- WORD

Number of words for the device.

- COLUMN

Column multiplexing factor for the device.

- WNAM

The wafer number.

- CNAM

Chip coordinates, given as a 3 digit x coordinate followed by a 3 digit y coordinate.

- PFFLG

A passing or failing flag indication. A “P” indicates the device has passed and there are no failures, an “F” indicates the device has failed. If the device has failed, the failures will be identified by the FKIND and FDATACON entries.

- FKIND

The type test pattern or algorithm that detected the failure. Possible values are:

MARCH - march_c or march_lr

CHECKER - checkerboard

RETENTION - checkerboard_retention

WL-STRIPE - wordline_stripe

GALLOP - galloping_ones

RANDOM - pseudo_random

PORT-INT - port_interaction

ROMTEST - ROM failure

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

■ FDATACON

A failing data count. Following this entry will be an entry for each failure. For example, if there are 3 failures, then there will be 3 entries following this entry indicating the failing locations.

■ DATA

This field is only present if the preceding FDATACON value is non zero. This indicates the failing location in the format:

failing_word, failing_bit, expected_value, failing_ports. A value of 100,10,1,AB would indicate a failure at word 100, bit 10 with an expected value of 1 and detected on ports A and B (or 0 and 1).

■ END

End of data for the device.

The FABIAN.idx file

Another output file created by `prepare_memory_failset` is the `FABIAN.idx` file. This file summarizes the failing and passing devices for a particular failing data set and test condition. Each line of this file contains:

lot-number, wafer-number, chip-x-coordinate, chip-y-coordinate, testblockname, memory-instance, pass-fail-status, fail-count

Example 5-6 Location of FABIAN.idx file

```
root_directory/FDS_data/fail-data-set-log_test-condition.fdd/  
FABIAN.idx
```

Example 5-7 Contents of FABIAN.idx file

```
4C01291-01,23,022030,BIT01,TopBlock.11b1.12m3.sram,F,2  
4C01291-01,23,022030,BIT01,TopBlock.11b1.12m4.sram,P,0  
4C01291-01,23,022030,BIT01,TopBlock.11m1.sram,F,1  
4C01291-01,23,022030,BIT01,TopBlock.11m2.sram,P,0  
4C01291-01,23,022030,BIT01A,TopBlock.11b1.12m3.sram,P,0  
4C01291-01,23,022030,BIT01A,TopBlock.11b1.12m4.sram,F,2  
4C01291-01,23,022030,BIT01A,TopBlock.11m1.sram,P,0  
4C01291-01,23,022030,BIT01A,TopBlock.11m2.sram,F,2
```

The *fail-data-set.idx* file

Another output file created by `prepare_memory_failset` is the `fail-data-set.idx` file. Each line of this file contains:

```
lot-number, chip-name, test-  
information, mask, tester, equipment, day-time, plant, facet, fail-  
limit
```

Example 5-8 Location of *fail-data-set.idx* file

```
root_directory/FDS_data/fail-data-set-log_test-condition.fdd/  
fail-data-set-log.idx
```

Example 5-9 Contents of *fail-data-set.idx* file

```
4C01291-01, TopBlock, PT1, TP0A123, T6673, F3, 2525/03/15-12:00, MIE, BOTTOM, 00001024
```

Internal files created by `prepare_memory_failset`

When `prepare_memory_failset` is executed, the `FDSlog2CPP` and `analyze_embedded_test` commands are executed under the covers. The CPP files created are stored in the `root_directory/pmf` directory. Log files for these commands are stored in the `root_directory/pmf/logs` directory.

Example 5-10 Complete directory structure for `prepare_memory_failset`

```
root_directory/  
  CDNS_info/  
    design_1/  
      design_1.wgl.DIAG001  
      design_1.wgl.PROD001  
      design_1_mbist_tdr_map.txt  
      design_1_mbistdiag_tdr_map.txt  
      design_1_pattern_control.txt  
      design_1_rombistdiag_tdr_map.txt  
    design_2/  
      design_2.wgl.BIT01  
      design_2.wgl.BIT01A  
      design_2.wgl.BYP03  
      design_2_mbist_tdr_map.txt  
      design_2_mbistdiag_tdr_map.txt  
      design_2_mbistread_tdr_map.txt  
      design_2_pattern_control.txt
```

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

```
design_2_rombistdiag_tdr_map.txt
FDS_data/
  fail-data-set-log_test-condition.fdd/
    FABIAN.idx
    fail-data-set-log.idx
  wchip/
    cchip-x-coordinatechip-y-coordinate/
      testblockname.FBI
pmf/
  logs/
TANSI.cvs
```

prepare_memory_failset command line examples

Valid `prepare_memory_failset` command lines are:

- Using the `faildataloglist` keyword

```
prepare_memory_failset rootdirectory=root_directory
faildataloglist=failing-data-set-file1,failing-data-set-file2,failing-
data-set-file3
```

- Using the `faildatalogfile` keyword

```
prepare_memory_failset rootdirectory=root_directory
faildatalogfile=failing-data-file
```

- Using the `patternoffset` keyword

```
prepare_memory_failset rootdirectory=root_directory
faildatalogfile=failing-data-file patternoffset=1000
```

Common messages and solutions

There are a number of warning and error messages issued by `prepare_memory_failset`. This section explains the message and potential solutions.

- **ERROR (TEM-752):** Keyed_Data could not be found while analyzing the pattern file.

This error occurs when the STIL or WGL patterns found in the manufacturing memory test database do not contain the Keyed_Data necessary to correlate the patterns to the rest of the files. The most common cause of the problem is the patterns were not generated with `write_vectors keyeddata=yes` option. Create the patterns with the `keyed_data=yes` option and rerun `prepare_memory_failset`.

- **ERROR (TEM-763):** Missing data from the failure data log file containing the chip information.

Encounter Test: Flow: MBIST Analysis

Manufacturing Test Support

This error occurs when the data lines containing the chip information are missing from the failing data set file. This line looks something like `FDS-DATA00000018CI23022030serialno` see [Example 5-3](#) on page 86 for more information.

- **WARNING (TEM-744):** A pattern offset value of `offset_cycles` cycles has been specified on the command line.

This warning occurs when the `patternoffset=offset_cycles` keyword has been specified on the command line. This keyword is used to increase the cycle counts calculated from the STIL or WGL patterns. This option is useful if the failing data set was produced from a tester which had a pll or oscillator which introduced additional cycles that were not originally present when the patterns were created.