Encounter Test: Flow: MBIST Pattern Generation

Product Version 12.1.101
February 2013

Contents

List of Figures

Preface

About This Manual

This manual describes the Encounter® Test memory built-in self-test pattern generation. The pattern generation capability applies specifically to the memory built-in self-test logic inserted within a design using Design For Test in Encounter® RTL Compiler, most prominently the insert_dft mbist command. The description includes the inputs required to ensure proper pattern generation considering the target test vehicle, the memory test scheduling options available for the types of patterns which can be generated based on the structures inserted into the design, and the types of patterns which can be written for subsequent use in the target test environment.

Additional References

This manual is the second 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

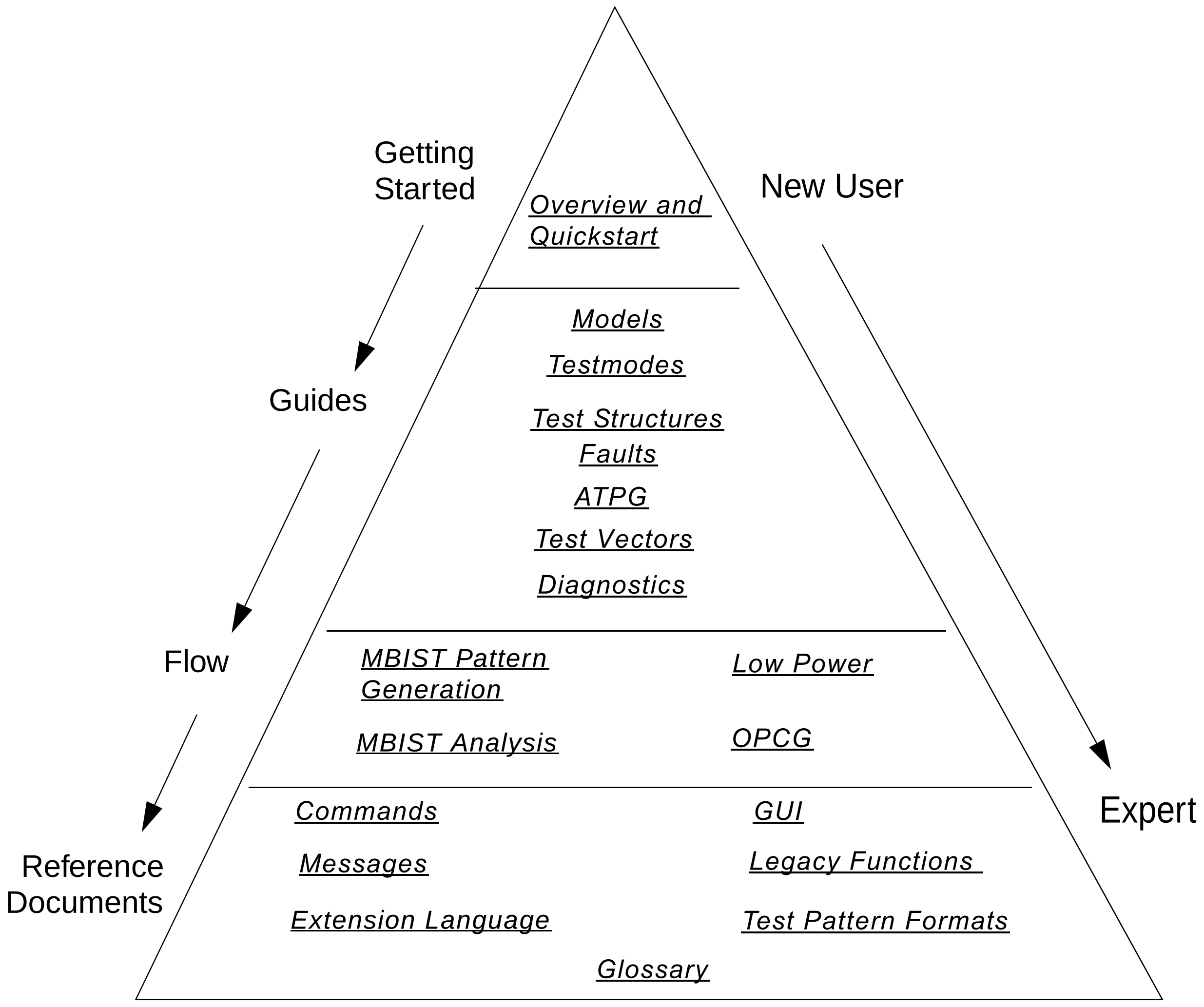
  All aspects of pattern generation for design verification and manufacturing test are included in this user guide.

- Encounter Test 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 equivalency checking, design verification, and post-processing 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 `create_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.

1 Introduction

This manual describes the Encounter® Test memory built-in self-test pattern generation. The pattern generation capability applies specifically to the memory built-in self-test logic inserted within a design using Design For Test in Encounter® RTL Compiler. Topics range from transferring the design from RTL Compiler into Encounter Test, memory built-in self-test pattern generation, structure and export, and supporting design verification and manufacturing test.

“Design Flow into Encounter Test” on page 15 describes the possible paths to move from insertion of memory built-in self-test to pattern generation. This includes the necessary inputs from the designer and outputs generated for subsequent use by the designer.

“Tester Description Rule” on page 19 contains the default target tester capabilities used to constrain the generation of patterns within Encounter Test. In general, this default definition is likely applicable to most patterns and testers.

“Create Embedded Test” on page 21 describes the command line and graphical invocation of the `create_embedded_test` command, initiating the Encounter Test memory built-in self-test scheduling and pattern generation application. The inputs, supported pattern options, and outputs of the command are described in detail.

“Write Vectors” on page 47 is the command necessary to write patterns in a form useable by other applications, either design verification or manufacturing test. The required options of write_vectors for the various memory built-in self-test pattern classes are documented.

“Pattern Structure” on page 49 includes information related to the memory built-in self-test patterns based on access methods used, the general structure of the patterns, the clocking and the flow of control within them.

“Design Verification Support” on page 63 includes descriptions of advanced verification features and how to enable them as well as an introduction to the use of `analyze_embedded_test` in the verification process.

“Manufacturing Test Support” on page 65 describes the features which help ensure the consistent use of related files spanning the flow from generation of test patterns through manufacturing test to failure data gathering and analysis. An introduction to chip-pad-pattern data and `prepare_memory_failset` is included.

2

Design Flow into Encounter Test

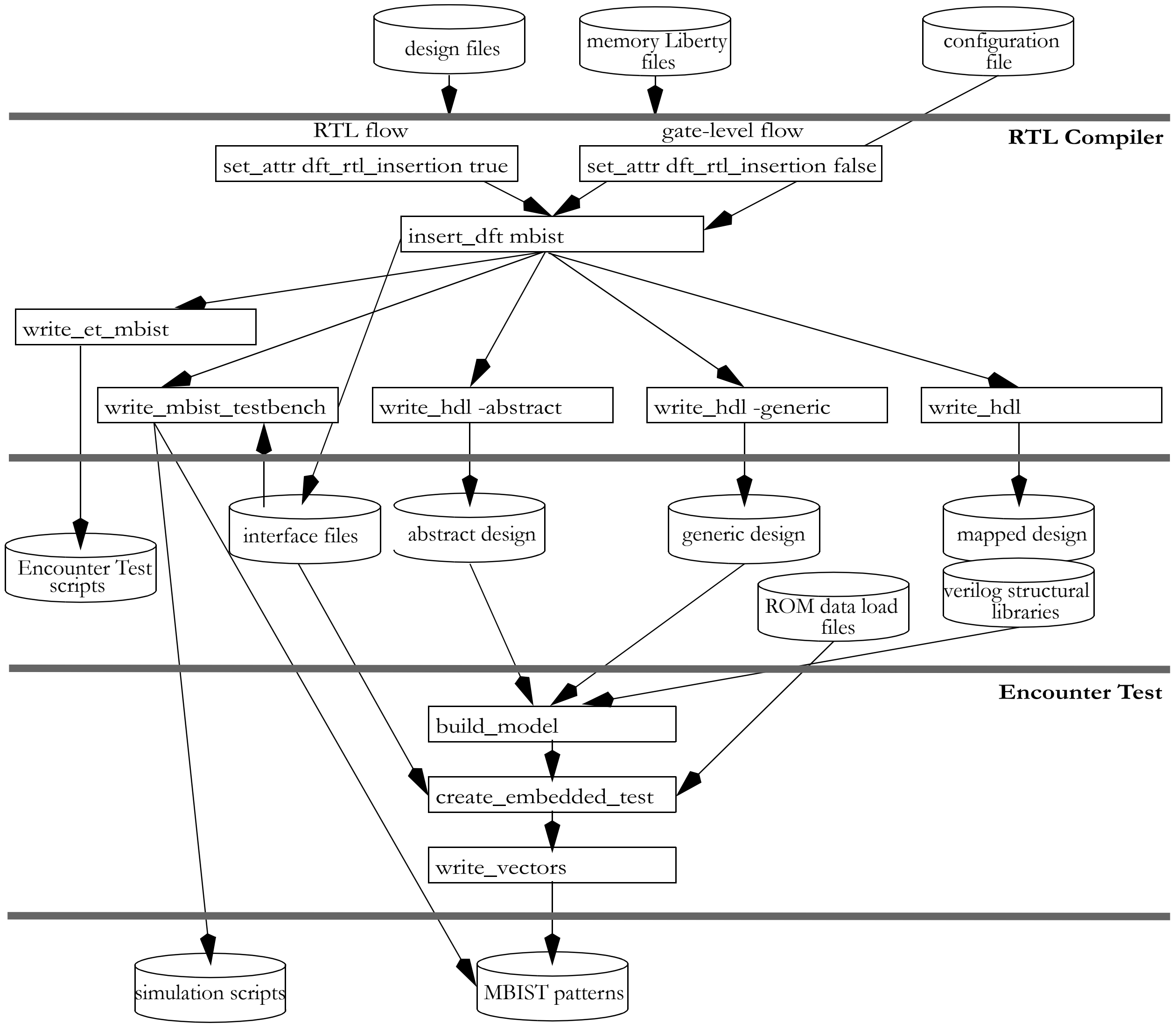
Memory built-in self-test can be implemented in several flows: a register-transfer-level (RTL) or gate level (generic or technology mapped) insertion methodology may be used, coupled with a top-down (full chip, single insertion) or bottom-up (one or more blocks with inserted memory test integrated into a design for the chip) approach. Pattern generation is supported in most cases for the combinations of these flows. Outlined in this chapter are the relevant commands executed on the RTL Compiler side and the required subsequent steps within Encounter Test.

After inserting memory built-in self-test within RTL Compiler, `write_mbist_testbench` can be used within the same RTL Compiler session to execute the Encounter Test processing steps necessary to generate the patterns to execute the memory built-in self-test algorithms using the simulation scripts created for Incisive Enterprise simulation. In this case, the `build_model` command is embedded within the `write_mbist_testbench` command and uses an equation level model as input.

An alternative approach uses the RTL Compiler command `write_et_mbist`. This command generates a script which can be executed within Encounter Test to generate memory built-in self-test patterns for simulation as well as tester application. Another script can be generated optionally to enable execution of Encounter Test boundary scan verification, including validation of the memory built-in self-test IEEE1149.1 instructions and registers.

For the remaining cases which rely upon the `write_hdl` command, the transition to pattern generation relies upon entry into Encounter Test at the `build_model` step. Several options are available.

Figure 2-1 Memory Built-In Self-Test Process Flow



■ `write_hdl` *`design`* -abstract > *`design`*`_abstract.v`

In this case, the design is an abstract representation, including the top level ports only. The user must manually make a connection from some input to some output in the Verilog description for Encounter Test to handle this appropriately. Also, no reference to internal design points in the interface files can be supported in this approach. If these conditions are satisfied, `create_embedded_test` can successfully generate patterns using the abstract model.

This option is useful to support an RTL flow as Encounter Test does not accept RTL as input to `build_model`.

```
build_model cellname=design designsource=design_abstract.v WORKDIR=directory
```

- `write_hdl` *`design`* -generic > *`design`*`_generic.v`

  This approach writes a design containing Verilog primitives, allows for references to internal design points in the interface files and does not require any technology libraries to create a functionally correct model.

  This option is useful in supporting either an RTL flow or gate-level flow as it does not require Verilog structural libraries as input to `build_model`. However, if technology cells exist in the design, e.g. pad cells, the process of unmapping them may result in blackboxes appearing in the Encounter Test model.

  ```
  build_model cellname=design designsource=design_generic.v WORKDIR=directory
  ```

- `write_hdl` *`design`* > *`design`*`.v`

  A gate-level model with structural Verilog libraries is input to `build_model` in this option. This supports not only memory built-in self-test pattern generation but permits detailed logic fault modeling as well.

  ```
  build_model cellname=design designsource=design.v
  TECHLIB=verilog_cell_libraries WORKDIR=directory
  ```

3 Tester Description Rule

In order to ensure the memory test data generated by Encounter Test will adhere to the constraints of the target tester on which the design is to be tested, Encounter Test takes as input a description of the tester's capabilities. This is called a tester description rule (TDR).

For memory built-in self-test, a default TDR is shipped with the product that contains a description that supports generation of memory test patterns that may exceed one or more parameters of the target tester. Although most pattern groups are constrained by a single clock source within `create_embedded_test`, this is not the default behavior for power-on and burnin-in patterns and parameters related to clocking capabilities should be constrained by the engineer. Test engineers should ensure the target tester capabilities are not exceeded prior to generating memory test patterns for manufacturing test.

The default memory built-in self-test TDR can be found in *`$Install_Dir`*`/defaults/rules/tdr/A6672m.mbist`. Relevant excerpts are listed below. This TDR allows for a maximum of 32 clocks with a maximum of 32 pulses per clock per tester cycle and a maximum clock frequency of 2GHz.

```
TEST_PINS
        CLOCK_PINS = 32
        SCAN_IN_PINS = 32
        OSCILLATORS = 32
;

OSCILLATOR_PULSES_PER_TESTER_CYCLE = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 ;

PIN_TIMING
        TIMING_RESOURCE = PER_PIN
        MAX_PULSES = 32
        MAX_STIMS = 1
        MAX_MEASURES = 1
        MAX_CYCLE_TIME = 1 MS
        MIN_CYCLE_TIME = 500 PS
        MAX_PULSE_WIDTH = 1 MS
        MIN_PULSE_WIDTH = 250 PS
        AC_MIN_PULSE_WIDTH  = 250 PS
        HF_MIN_PULSE_WIDTH  = 250 PS
        MIN_PULSE_OFF = 0 NS
        MIN_TIME_LEADING_TO_LEADING = 500 PS
        MIN_TIME_TRAILING_TO_TRAILING = 500 PS
        MIN_STIM_STABLE = 0 NS
        ACCURACY = 0 PS
        RESOLUTION = 1 PS
        LEADING_PADDING = 0 PS
        TRAILING_PADDING = 0 PS
        TERM_TIME_TO_1 = 125 NS
        TERM_TIME_TO_0 = 125 NS
        STROBE_SETUP = 0 NS
        STROBE_HOLD = 100 PS
        MIN_SCAN_CYCLE_TIME = 13250 PS
        MIN_SCAN_PULSE_WIDTH = 2500 PS
        DC_CYCLE_TIME = 100 NS
        DC_PULSE_WIDTH = 20 NS
        ;
```

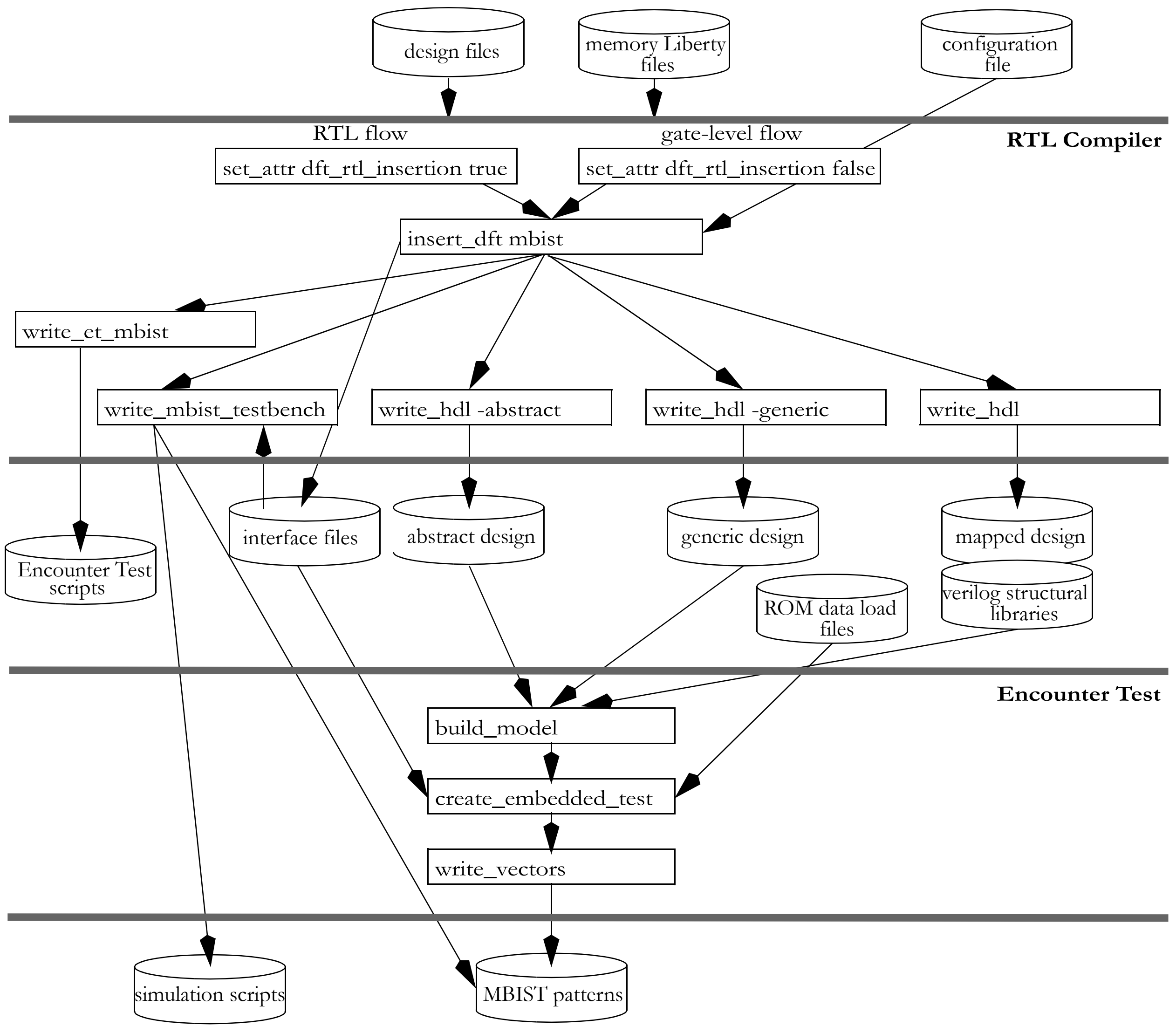
For further information regarding tester description rules, refer to the Tester Description Rule (TDR) File Syntax in the *Encounter Test: Guide 2: Testmodes*.

4

Create Embedded Test

`create_embedded_test` is the command which generates the memory built-in self-test patterns within the Encounter Test session.

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



Inputs

`create_embedded_test` takes as necessary inputs the Encounter Test design model from `build_model`, the interface file set written by `insert_dft mbist` in the RTL Compiler session, and the ROM data load files for any ROMs in the design. Other command options are selected as necessary for the pattern generation desired.

Basic Keywords

The working directory for Encounter Test is indicated to `create_embedded_test` through the `WORKDIR=`*`directory`* keyword.

The design is identified to `create_embedded_test` through the `block=design` and `topshell=`*`design`* keywords. This identifies the top level module in the design. When `block` is specified, it indicates to `create_embedded_test` that this is not chip level processing and patterns are generated assuming any JTAG controller is *not* present in the design. When `topshell` is specified, `create_embedded_test` infers chip level processing and patterns are generated accordingly. If a JTAG controller was inserted in the chip design and used to access the memory built-in self-test, the BSDL file should be specified to `create_embedded_test` through the `bsdlinput=`*`BSDL_filename`* keyword.

The interface files are identified through use of the `interfacefiledir=`*`directory`* and `interfacefilelist=`*`comma_separated_filenames`* keywords. The interface files contain information describing the structures inserted into the netlist by `insert_dft mbist` and pattern generation control information. They are an abstract model of the memory built-in self-test and targeted memories for `create_embedded_test`. For bottom-up design flows inserting memory built-in self-test into multiple design blocks and/or creating multiple instances of blocks in which memory built-in self-test was inserted, it is possible to assign these blocks to different JTAG instruction sets requiring unique interface file sets. `create_embedded_test` can generate patterns for a single interface file set at each invocation.

If the specified pattern control file indicates that one of the multiple instruction sets is being used (for example, the file name *`design`*`_1_pattern_control.txt` indicates 1 as the instruction set index) all the files generated by `create_embedded_test` will be named with the instruction set index appended to the *`design`* name.

If ROMs exist in the design and are targeted for memory built-in self-test, the data load files must be identified through the `ROMPATH=`*`colon_separated_directories`* and `romcontentsfile=`*`comma_separated_filenames`* keywords, and optionally the `romdatamapfile=`*`fully_qualified_filename`* keyword. The `romdatamapfile` keyword allows you to specify unique ROM data load files for specific instances of ROMs in the design. This keyword is used in conjunction with the `ROMPATH` and `romcontentsfile` keywords and takes higher priority, therefore, when searching for ROM data load files, `romdatamapfile` is used first. If the memory instance is not present in `romdatamapfile`, then the data load file is searched using the `ROMPATH` and `romcontentsfile` keywords. Each line in `romdatamapfile` contains a memory instance, followed by one or more white space characters and then a fully qualified ROM load file. The memory instance name may start with the (`/`) character. If so, the string found between the first two (`/`) characters is assumed to be the top-level design, and removed for further processing by `create_embedded_test`. The ROM content files contain an image of the data present in the read-only memories. These data load files are typically created by the technology vendor ROM generators and used in logic simulation to load the memory contents. The file name must include the name of the memory module as a prefix within the filename: `ROM_module_name`. The files must be structured with one line per address, with the first line containing address zero contents and the last line containing the maximum address contents. The content of each line is a bit vector for that address, with the most significant bit on the left and bit zero on the right. The vector is either a binary string of ones and zeroes or a hexadecimal string with right-justified valid data.

Example 4-1 ROM file containing binary data

ROM cell name: `ROM128X17`. ROM contents file: `ROM128X17_verilog.rcf`.

ROM contains 128 addresses, and the data width is 17 bits.

Command line:

```
create_embedded_test ... ROMPATH=location_of_ROM_contents_file
romcontentsfile=ROM128X17_verilog.rcf
```

Contents of `ROM128X17_verilog.rcf`:

```
01000011101011011

10010110100011000

       ...

00001001010101000
```

There are 128 lines of data in the file, one for each address. There are 17 bits of data on each line as the data width of the memory is 17 bits.

Example 4-2 ROM file containing hexadecimal data

ROM cell name: `p10np_512x10cm16`. ROM contents file: `p10np_512x10cm16.hex`

ROM contains 512 addresses, and the data width is 10 bits.

Command line:

```
create_embedded_test ... ROMPATH=location_of_ROM_contents_file
romcontentsfile=p10np_512x10cm16.hex
```

Contents of `p10np_512x10cm16.hex`:

```
28A
1FE
 ...
12C
```

There are 512 lines of data in the file, one for each address. There are 10 bits of usable data on each line, as the data width of the memory is 10 bits. As the data is right justified, the 2 leftmost bits are ignored by `create_embedded_test` during processing.

Example 4-3 Using the `romdatamapfile` keyword

Command line:

```
create_embedded_test ... ROMPATH=location_of_default_ROM_load_files
romcontentsfile=ROM32X4.hex romdatamapfile=location_of_file/romdatamapfile
```

There are 2 instances of the ROM32X4 cell in this design:

```
/TopBlock/l1m1/rom
/TopBlock/l1m2/rom
```

Contents of the `romdatamapfile`:

```
/TopBlock/l1m1/rom location_of_instance_ROM_load_files/ROM32X4.hex.1
```

In this example, instance `/TopBlock/l1m1/rom` uses the `ROM32X4.hex.1` ROM data load file, and instance `/TopBlock/l2m2/rom` uses the `ROM32X4.hex` default ROM data load file.

Housekeeping Keywords

Specifying `cleanstart=yes` results in deletion of existing files generated by `create_embedded_test`, ensuring no conflicting or extraneous generated files exist from one pass of pattern generation to the next.

`genmodeinitonly=yes` is used to implement a sequence of events to set a state in the design prior to running memory test. This keyword causes `create_embedded_test` to execute only the code to generate the memory built-in self-test mode initialization sequence used for JTAG access patterns and the mode initialization sequence used for direct access patterns, if applicable. Comments are placed in this generated file to permit the designer to insert a sequence of events at the designated place. After the updated file is ready, specify the following keywords:

```
cleanstart=no genmodeinitonly=no
seqdef=jtag_access_mode_initialization_filename
```

These keywords enable use of this manually updated mode initialization sequence file in memory built-in self-test pattern generation for JTAG access patterns.

The following keywords enable use of a manually updated mode initialization sequence file in memory built-in self-test pattern generation for direct access patterns:

```
cleanstart=no genmodeinitonly=no
seqdefdirect=direct_access_mode_initialization_filename
```

The keyword `genscriptsonly=yes` enables the generation of the scripts executed within the `create_embedded_test` command but avoids actually executing them. This provides expert users more opportunity to vary the parameters of execution of the command.

The `BSDLPKGPATH=`*`colon_separated_directories`* keyword supports overriding the default BSDL package used within the Encounter Test installation.

Tester Description Rule Keywords

When the default memory built-in self-test tester description rule is not appropriate for pattern generation, three keywords exist which support varying degrees of control.

`maxsupportedclocks=`*`positive_integer`* is used to control the number of clocks that may be simultaneously controlled by the tester in a given tester cycle. `maxclockpulses=`*`positive_integer`* specifies the maximum number of clock pulses per tester cycle that may be applied to any of the active clocks in the pattern. These two keywords may reduce the current values of the active tester description rule, but not exceed them.

The keyword `testerdescriptionrule=`*`TDR_path/TDR_filename`* is used to replace the default tester description rule for the pattern generation. If this keyword is specified, all mode definition files used for test modes created by `create_embedded_test` are copied from the default location, *`installation_dir`*`/defaults/rules/modedef`, to the working directory. These copies of the mode definition files are then modified, so the `Tester_Description_Rule` entry points to the specified tester description rule file.

If the `testerdescriptionrule` keyword is specified, *`TDR_filename`* is added to the default test mode names created by `create_embedded_test`. Refer to the following sections for more information on file name changes:

- "Assign Files" on page 29
- "Mode Initialization Sequences" on page 29

- "File System Updates" on page 36

Pattern Generation Keywords

These keywords control the class of patterns and execution schedule followed during generation of the memory built-in self-test patterns during excecution of `create_embedded_test. createpatterns` selects one or more pattern classes for generation and the corresponding `*schedule` keyword selects the *desired* execution schedule. Each `*schedule` keyword allows for a fully `custom` schedule to be selected with the exception of the `directschedule` keyword or one of four broader engine/device schedule values.

Table 4-1 createpatterns Keyword Relationship to Scheduling Keywords

createpatterns value	corresponding `*schedule`
`bypass`	prodschedule
`production`	prodschedule
`diagnostic`	diagschedule
`bitmap`	bitmapschedule
`poweron`	directschedule
`burnin`	directschedule
`redundancy`	redundschedule

Table 4-2 Scheduling Keyword Value Descriptions

*schedule value	Memory BIST execution schedule
`custom`	fully customizable execution schedule
`parallel_parallel`	directive to schedule all engines in *parallel* and all devices in each engine in *parallel*
`parallel_serial`	directive to schedule all engines in *parallel* and all devices in each engine in *series*
`serial_parallel`	directive to schedule all engines in *series* and all devices in each engine in *parallel*

*schedule value	Memory BIST execution schedule
`serial_serial`	schedule all engines in *series* and all devices in each engine in *series*

The `failurelimit=`*`positive_integer`* keyword enables the test engineer to place a limit on the number of pause-and-resume iterations executed within the `diagnostic` and `redundancy` pattern classes.

The `buildtestmode` keyword allows you to select if the testmodes created by `create_embedded_test` for pattern generation are built or reused. You might want to execute `create_embedded_test` for a specific group of patterns, and then execute another invocation for different pattern types. By default, the testmodes used for pattern generation are built during each invocation of `create_embedded_test`. When existing testmodes are built again, any previously existing experiments are deleted automatically by the `build_testmode` command. To keep all experiments available for a testmode, use `buildtestmode=no` to force `create_embedded_test` to reuse the testmodes.

Design Verification Keywords

These keywords are only accessible from within an RTL Compiler session when the `write_mbist_testbench` command is invoked. They exist to permit memory built-in self-test generation of Verilog testbench patterns. `outputverilog=yes` enables Verilog patterns to be exported from Encounter Test and `outputverilogdir=`*`verilog_pattern_directory`* identifes the location where the files are written.

Manufacturing Test Keywords

`testblockname=`*`string`* enables the test engineer to indicate a string uniquely identifying a link between the `create_embedded_test` exported patterns and the tester data log created in manufacturing test. Refer to "Manufacturing Test Support" on page 65 for more information.

`splitretentionpatterns=yes` controls the manner in which patterns are generated that contain the checkerboard-retention algorithm. The original pattern is divided into multiple pattern subsets created to allow the test engineer to vary test environment variables between subsets. Refer to "Production Retention Pattern" on page 51 for more information.

Assign Files

`create_embedded_test` typically generates the necessary assign files for its own pattern generation based on interface file content. However, there may be situations where these need enhancement by the designer. In such situations they may be edited in place and used by specifying `cleanstart=no` on the `create_embedded_test` command line.

Table 4-3 Assign File Descriptions

File Name	Content	Usage
`assignfile.`*`design`*`.patt_gen`[1]	JTAG pins DFT configuration pins clock pins	JTAG access patterns
`assignfile.`*`design`*`.mda`[1]	clock pins	direct access patterns
`assignfile.`*`design`*`.1149`[1]	JTAG pins	Encounter Test boundary scan verification

[1]If the keyword `testerdescriptionrule=`*`TDR_path/TDR_filename`* is used, the file name is suffixed with `_`*`TDR_filename`*. If multiple instructions are detected based on the `interfacefiledir` and `interfacefilelist` keywords being specified, then the string *`design`* becomes *`design_instruction-set-index`*. The instruction set index is extracted from the pattern control file name. For example, if the pattern control file specified is *`design`*`_1_pattern_control.txt`, then the instruction set index would be `1`.

Mode Initialization Sequences

An Encounter Test test mode initialization sequence is used to establish a defined state of the design under test. `create_embedded_test` generates the necessary sequences based on its knowledge of the design and the pattern requirements. This includes

- establishing the state of test control pins defined in the RTL Compiler session,
- initializing the state of any JTAG controller and compliance enable pins,
- initializing the state of any memory built-in self-test direct access mechanism,
- and initializing the state of clock paths into the memory built-in self-test logic.

If the designer has additional requirements which must be satisfied, such as using the JTAG controller to establish a design for test control mode or locking a PLL for use as a clock source in memory built-in self-test, these can be added manually to the sequences in their predefined filesystem location and used by `create_embedded_test`.

Table 4-4 Mode Initialization Sequence Usage

File Name	Usage
`TBDseqPatt.`*`design`*`.patt_gen`[1]	JTAG access patterns initialization sequence
`TBDseqPatt.`*`design`*`.patt_gen_no_reset`[1]	intermediate JTAG access patterns; places the JTAG controller in Run-Test-Idle state
`TBDseqPatt.`*`design`*`.mda`[1]	direct access patterns initialization sequence
`TBDseqPatt.`*`design`*`.1149`[1,2]	Encounter Test boundary scan verification initialization sequence

[1]If the keyword `testerdescriptionrule=`*`TDR_path/TDR_filename`* is used, the file name is suffixed with `_`*`TDR_filename`*. If multiple instructions are detected based on the `interfacefiledir` and `interfacefilelist` keywords being specified, then the string *`design`* becomes *`design_instruction-set-index`*. The instruction set index is extracted from the pattern control file name. For example, if the pattern control file specified was *`design`*`_1_pattern_control.txt`, then the instruction set index would be `1`.

[2]In the mode initialization file created for boundary scan verification, all of the clocks used by MBIST insertion and unused clocks are set to their inactive value.

Pattern Classes

Patterns are classified in general terms by their execution time and failure analysis capability. In general terms, as the granularity of the failure analysis increases, the execution time increases. Memory built-in self-test patterns can be generated for a design, be it the block level or full chip, in most cases.

Table 4-5 Block and Chip Level Pattern Class Support

Pattern Class	Block-level Support	Chip-level Support
bypass (connectivity)	yes	yes

Pattern Class	Block-level Support	Chip-level Support
production	yes	yes
retention (production)	yes	yes
diagnostic	yes	yes
bitmap	no	yes
retention (bitmap)	no	yes
direct access	yes	yes
repair	no	yes

For the experiment names in the pattern classes which appear below, the *`clock-source`* variable has a value based on the source of the memory built-in self-test clock selected for pattern generation:

- mtclk

  `clock-source = design-port-name_engine-clock-frequency`
- stclk

  `clock-source = design-port-name`
- internal clock source

  `clock-source` `= ICS_`*`design-port-name`*

Bypass (connectivity)

The bypass pattern, selected with `createpatterns=bypass`, performs memory initialization in SRAMs, simply writing zeroes to all locations, while performing no memory accesses to any selected ROMs. This rudimentary pattern is intended to verify the controllability and observability of the memory built-in self-test logic.

Bypass pattern scheduling restrictions include the clock source and frequency.

Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_BYPASS_`*`positive-integer_clock-source`*

Production

The production pattern, selected with `createpatterns=production`, performs the execution of the specified algorithms in each memory BIST engine on its associated scheduled devices. Failure recording is limited to identifying failing memory devices as each selected algorithm executes once per scheduled device.

Production pattern scheduling restrictions include the clock source and frequency.

Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_PROD_`*`positive-integer_clock-source`*

Retention

The retention pattern is based on the selection of the checkerboard-retention algorithm for the scheduled devices when selected as part of the `createpatterns=production` execution of `create_embedded_test`. When selected as part of the production pattern it can be combined with other algorithms in the same generated pattern. Failure recording is limited to identifying failing memory devices as each selected algorithm executes once per scheduled device.

Retention production pattern scheduling restrictions include the clock source and frequency.

This pattern does not allow for user intervention at the points in the retention algorithm where the test pauses. Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*

If `createpatterns=production` is combined with `splitretentionpatterns=yes` the pattern allows for user intervention at the points in the retention algorithm where the test pauses. Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_RUN`

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_CONTINUE_1`

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_CONTINUE_2`

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_CONTINUE_3`[1]

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_CONTINUE_4`[1]

`MBIST_ATE_RETENTION_CONTROLLER_`*`positive-integer_clock-source`*`_DIAGNOSE`

[1]These are generated as required by the set of scheduled memory characteristics.

Refer to “Production Retention Pattern” on page 51 for more information.

The retention pattern is based on the selection of the checkerboard algorithm for the scheduled devices when selected as part of the `createpatterns=bitmap` execution of `create_embedded_test`. When selected as part of the bitmap pattern it cannot be combined with other algorithms in the same generated pattern. Failure recording identifies failing memory devices, addresses, bits/columns, and bit values.

Retention bitmap pattern scheduling restrictions include the clock source and frequency, and the separation of 2rw devices from {1rw, mrnw} devices due to algorithmic differences.

This pattern does not allow for user intervention at the points in the retention algorithm where the test pauses. Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source`*

If `createpatterns=bitmap` is combined with `splitretentionpatterns=yes` the pattern allows for user intervention after the first three sequence iterators in the algorithm for each diagnostic slice implemented in the memory built-in self-test logic. Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source_diagnostic-slice-iteration_1`*

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source_diagnostic-slice-iteration_2`*

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source_diagnostic-slice-iteration_3`*

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source_diagnostic-slice-iteration_4`*

Refer to “Bitmap Retention Pattern” on page 56 for more information.

Diagnostic

The diagnostic pattern, selected with `createpatterns=diagnostic`, performs the execution of the specified algorithms in each memory BIST engine on its associated scheduled devices. Failure recording identifies failing memory devices, addresses, and bits/columns based on the capabilities of the inserted memory built-in self-test structures for each of the scheduled devices. The diagnostic patterns rely upon a pause-and-resume failure detection mechanism in the memory built-in self-test hardware to gather the failing data. The `failurelimit=`*`positive_integer`* places an upper bound on the number of failures detected in a given memory device under test.

Diagnostic pattern scheduling restrictions include the clock source and frequency, the physical row and column structure, the read delay, and the number of diagnostic slices associated with the devices.

Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_DIAG_CONTROLLER_`*`clock-source_integer_integer`*

Bitmap

The bitmap pattern, selected with `createpatterns=bitmap`, performs the execution of the specified algorithm in each memory BIST engine on its associated scheduled devices. Failure recording identifies failing memory devices, addresses, bits/columns, and bit values based on the capabilities of the inserted memory built-in self-test structures for each of the scheduled devices. The bitmap patterns rely upon a stop-after-read failure detection mechanism in the memory built-in self-test hardware to gather the failing data. This resutls in a test that does not run at system speeds, but ensures the algorithm executes once per diagnostic slice for each memory device under test.

If bitmap patterns are selected and the clock frequencies at the port and the hookup pin are different, then a warning message is generated by `create_embedded_test`, and bitmap patterns are not generated. Only a 1:1 clock ratio between the port and hookup pin is supported for this pattern class.

Bitmap pattern scheduling restrictions include the clock source and frequency, each algorithm must be scheduled individually by the designer using `create_embedded_test`, and the constraints identified in the table below.

Table 4-6 Bitmap Pattern Algorithm Scheduling Constraints

Algorithms	Constraint	Comments
checkerboard march-C enhanced port interaction galloping ones	2rw devices split from other devices	`create_embedded_test` enforces this rule and checkerboard-retention is forced to checkerboard
pseudo-random galloping ones	engine address width[1]	designer is responsible for scheduling engines with different engine address widths in different groups[1]
wordline stripe march-LR’	none	none

[1]`engine_address_width` is found in the pattern control file in the design.

Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_BITMAP_`*`positive-integer_clock-source`*

Direct Access

The direct access patterns, selected with `createpatterns=poweron` or `createpatterns=burnin`, perform the execution of the specified algorithms in each memory BIST engine on its associated scheduled devices. Failure recording is limited to identifying passing or failing designs for poweron patterns only. Poweron patterns run through the applied test once only, reporting status at completion of the test. Burnin patterns run through the applied test repeatedly until the test engineer stops the test.

ROM data load files must embed the seed signature within the high order ROM addresses for direct access patterns. `create_embedded_test` generates replacement ROM data load files for use during pattern generation, which include this seed signature after verifying that all zeroes exist in the required locations. Each modified ROM data load file is written back to the original location with a `.seed` suffix appended to the filename for use in subsequent processing steps. The number of addresses required for the seed signature in each ROM is determined with the following formula using attributes for that ROM:

```
ceiling[(address_width+data_width+5)/data_width]
```

When both direct access patterns and JTAG patterns are generated for a design, the JTAG patterns utilize the same modified ROM data load files as the direct access patterns.

Direct access pattern scheduling restrictions include the limitations imposed by the tester description rule and the `create_embedded_test` tester description rule keyword overrides. The direct access hardware does not support the `checkerboard_retention` algorithm. If the `checkerboard_retention` algorithm is chosen, then `create_embedded_test` automatically forces the algorithm to checkerboard.

Experiments are created within Encounter Test `mda` test mode using the following naming convention:

`MBIST_ATE_POWERON_`*`positive-integer`*

`MBIST_ATE_BURNIN_`*`positive-integer`*

Repair

The repair pattern, selected with `createpatterns=redundancy`, performs the execution of the specified algorithms in each memory BIST engine on its associated scheduled devices. Failure recording identifies failing memory devices, addresses, and bits/columns based on the capabilities of the inserted memory built-in self-test structures for each of the scheduled devices. The diagnostic patterns rely upon a pause-and-resume failure detection mechanism in the memory built-in self-test hardware to gather the failing data. The `failurelimit=`*`positive_integer`* places an upper bound on the number of failures detected in a given memory device under test.

At present, the repair patterns are limited to built-in repair analysis, calculating a solution for the singular device targeted in this pattern.

Repair pattern scheduling restrictions include the clock source and frequency and a single memory device.

Experiments are created within Encounter Test `1149_patt` test mode using the following naming convention:

`MBIST_ATE_REPAIR_CONTROLLER_`*`clock-source_integer_integer`*

Outputs

In addition to the assign files, mode initialization sequences, test modes, and experiments `create_embedded_test` generates within Encounter Test, several scripts and support files are generated. The file system updates are noted below. Outputs are generated as requested by the user on the command line of `create_embedded_test`, but all possible files are show here. `create_embedded_test` may also modify in place the input pattern control file when memory built-in self-test changes are made under pattern generation scheduling constraints.

File System Updates

`WORKDIR` is an environment variable accessible within Encounter Test. This points to the branch of the directory tree where `create_embedded_test` reads input and writes user required output by default.

```
WORKDIR/testresults/
    logs/
    .   create_embedded_test logs and reports
    scripts/
        run.design.1149[2,3]
        run.design.instruction[3]
        run_sim.design.patt_gen[2,3]
        run_sim.design.mda_build_testmode[2,3]
        run_sim.design.mda_read_vectors[2,3]
    testmode_data/
```

```
        MBIST_ATE_PROD_integer_clock-source.map[1]
    .   sources/
            assignfile.design.1149[2,3]
            assignfile.design.mda[2,3]
            assignfile.design.patt_gen[2,3]
            TBDseqPatt.design.bsv[2,3]
            TBDseqPatt.design.mda[2,3]
            TBDseqPatt.design.patt_gen[2,3]
            TBDseqPatt.design.patt_gen_no_reset[2,3]
            MODE_JTAG.design.MBIST_instruction_name[3]
            TBDseqPatt.design.MBIST_instruction_name[3]
            TBDpatt.design.*[3]
```

[1] The map files are created in support of data-bit redundancy.

[2] If the keyword testerdescriptionrule=*TDR_path/TDR_filename* is used, the test mode string (located after the design name) in the file name is suffixed with _*TDR_filename*.

[3] If multiple instructions are detected based on the interfacefiledir and interfacefilelist keywords being specified, then the string *design* becomes *design_instruction-set-index*. The instruction set index is extracted from the pattern control file name. For example, if the pattern control file specified was *design*_1_pattern_control.txt, then the instruction set index would be 1.

The information available in the `*.`*`MBIST_instruction_name`* files support test data register analysis within Encounter Test.

Repeated invocations of create_embedded_test

Each invocation of `create_embedded_test` within a given Encounter Test model results in the recreation of the test modes by default, effectively deleting any generated patterns from a previous invocation. This behavior can be changed by specifying `buildtestmode=no`, causing `create_embedded_test` to reuse the existing testmodes. If multiple invocations are required within a given `WORKDIR`, then either execute `write_vectors` after each invocation of `create_embedded_test` to export the generated experiments for that session, or specify `buildtestmode=no` after the initial invocation of `create_embedded_test` so that the existing test modes and experiments are retained.

Invoking create_embedded_test from the GUI

The GUI enables access to all command line keyword settings in addition to being able to fully customize the memory built-in self-test execution schedule.

To invoke `create_embedded_test` from GUI, click *ATPG-Create Tests-Embedded Test* as shown in Figure 4-2 on page 38.

Figure 4-2 ATPG Menu for Create Embedded Test

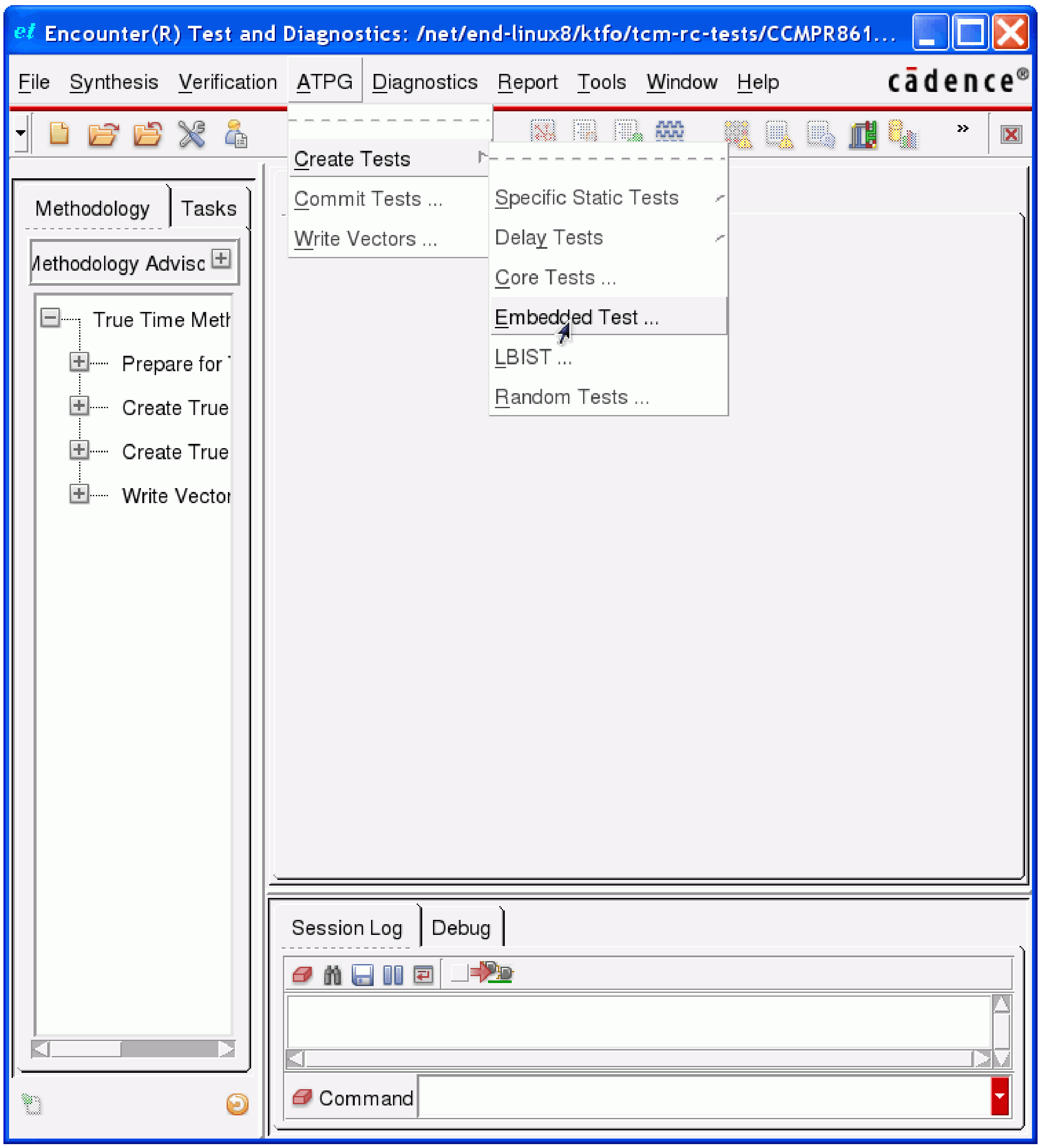
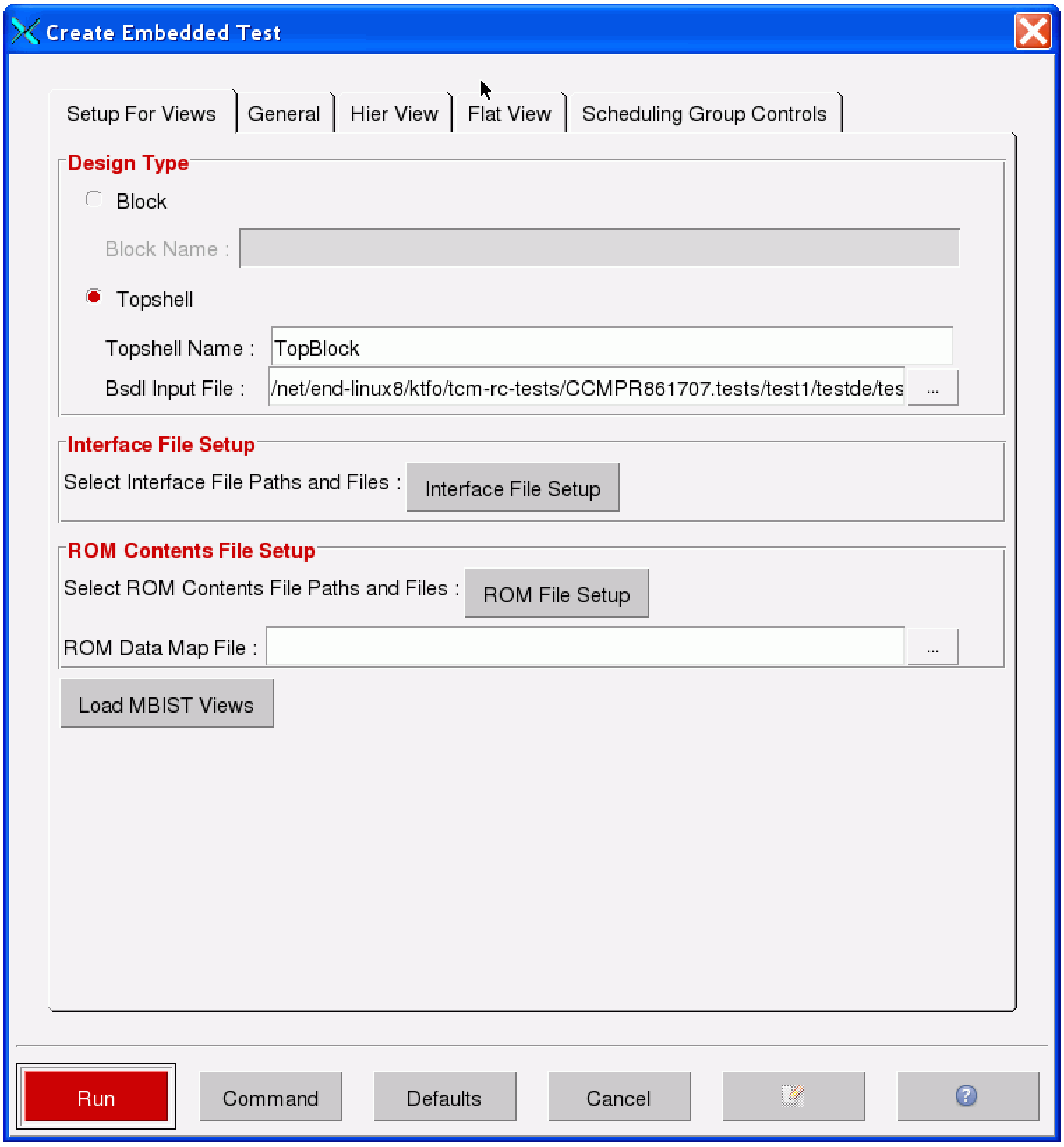


Figure 4-3 on page 39 displays the *Create Embedded Test* form.

Figure 4-3 Create Embedded Test Form



Design Type

Block

Block Name :

[✓ Topshell]

Topshell Name : TopBlock

Bsdl Input File : /net/end-linux8/ktfo/tcm-rc-tests/CCMPR861707.tests/test1/testde/tes

Interface File Setup

Select Interface File Paths and Files : Interface File Setup

ROM Contents File Setup

Select ROM Contents File Paths and Files : ROM File Setup

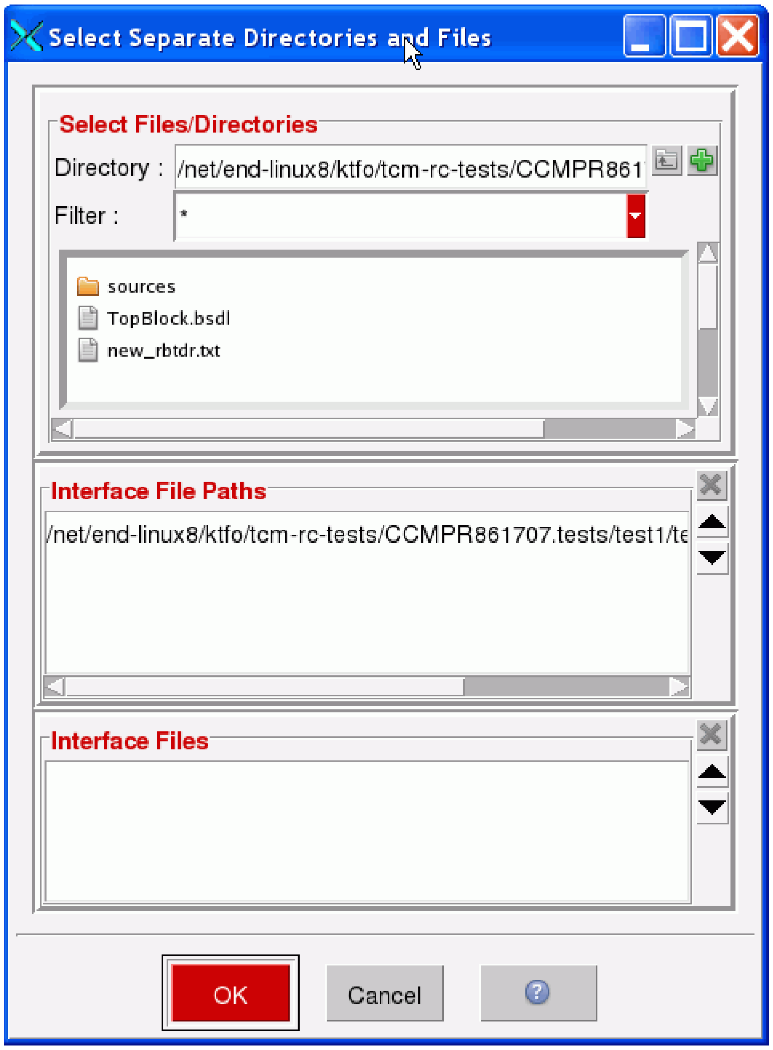
ROM Data Map File :

The *Setup For Views* tab is selected by default on the *Create Embedded Test* form.

- If working with a block level design (no JTAG controller), select the *Block* option and populate the *Block Name* field

- If working with a full-chip design:
  - ❑ Select the *Topshell* option
  - ❑ Populate the *Topshell Name* field
  - ❑ Either select a file or type the file name for the *Bsdl Input File* field.
- Clicking the *Interface File Setup* button opens the window to select the *Interface File Paths* and *Interface Files*. This window is displayed in Figure 4-4 on page 40.

Figure 4-4 Interface File Setup Window



Select Separate Directories and Files

Select Files/Directories

Directory : /net/end-linux8/ktfo/tcm-rc-tests/CCMPR861

Filter : *

sources
TopBlock.bsdl
new_rbtdr.txt

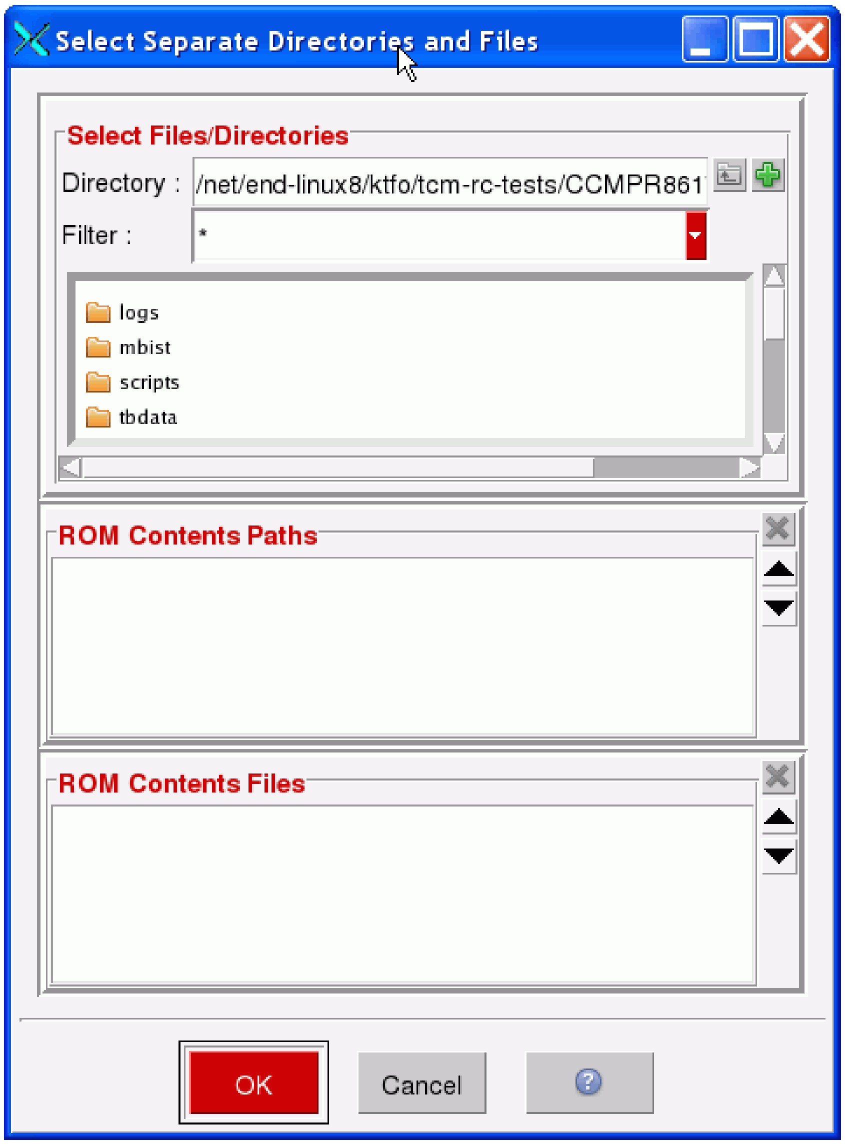
Interface File Paths

/net/end-linux8/ktfo/tcm-rc-tests/CCMPR861707.tests/test1/te

Interface Files

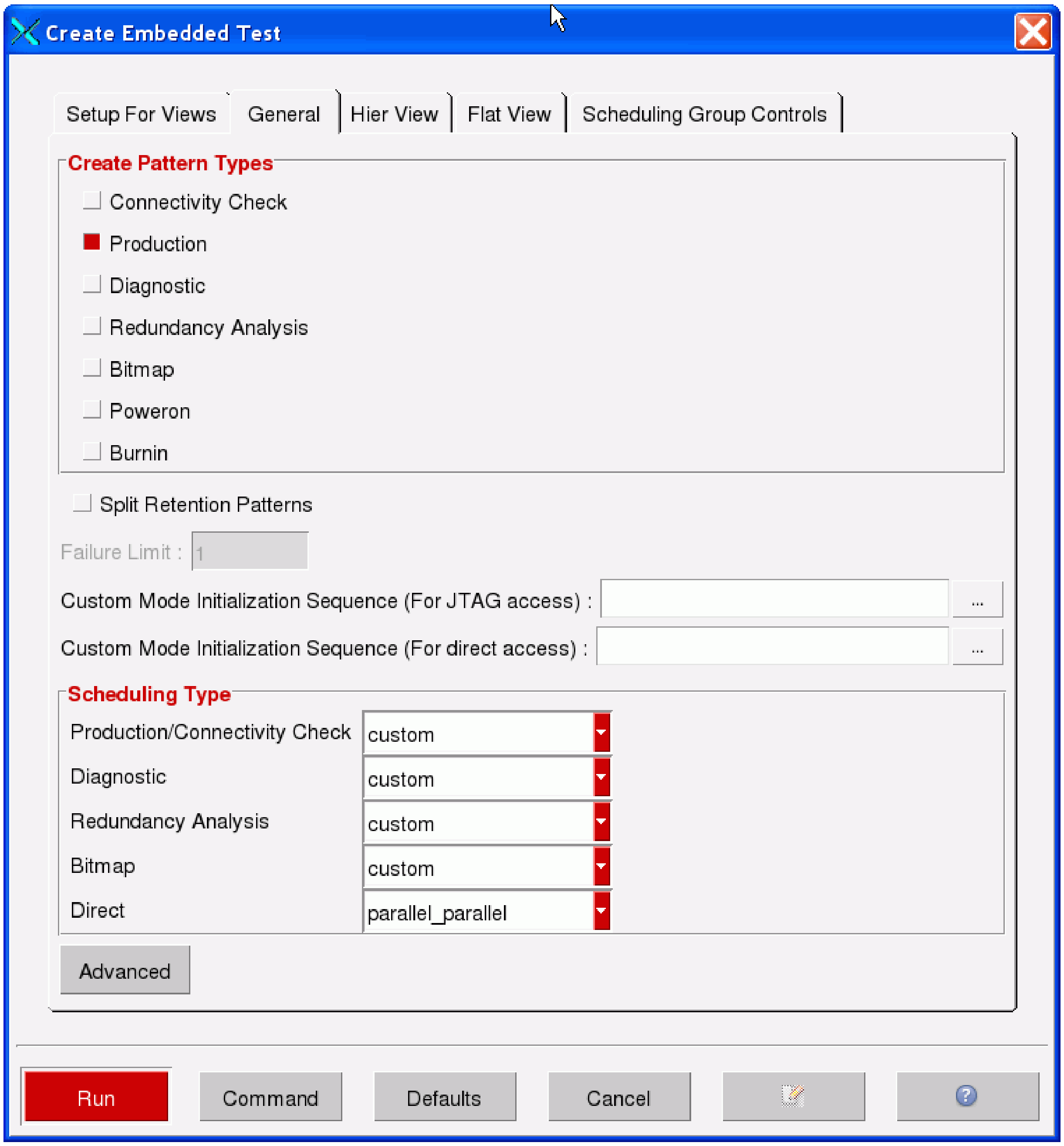
- If the design contains ROMs, click the *ROM File Setup* button to select the *ROM Contents Paths* and the *ROM Contents Files*. This window is displayed in Figure 4-5 on page 41.

Figure 4-5 ROM File Setup Window



Clicking the *General* tab on the *Create Embedded Test* form displays the window as shown in Figure 4-6 on page 42.

Figure 4-6 General Tab



Create Pattern Types

- ☐ Connectivity Check
- ☑ Production
- ☐ Diagnostic
- ☐ Redundancy Analysis
- ☐ Bitmap
- ☐ Poweron
- ☐ Burnin

☐ Split Retention Patterns

Failure Limit : 1

Custom Mode Initialization Sequence (For JTAG access) :

Custom Mode Initialization Sequence (For direct access) :

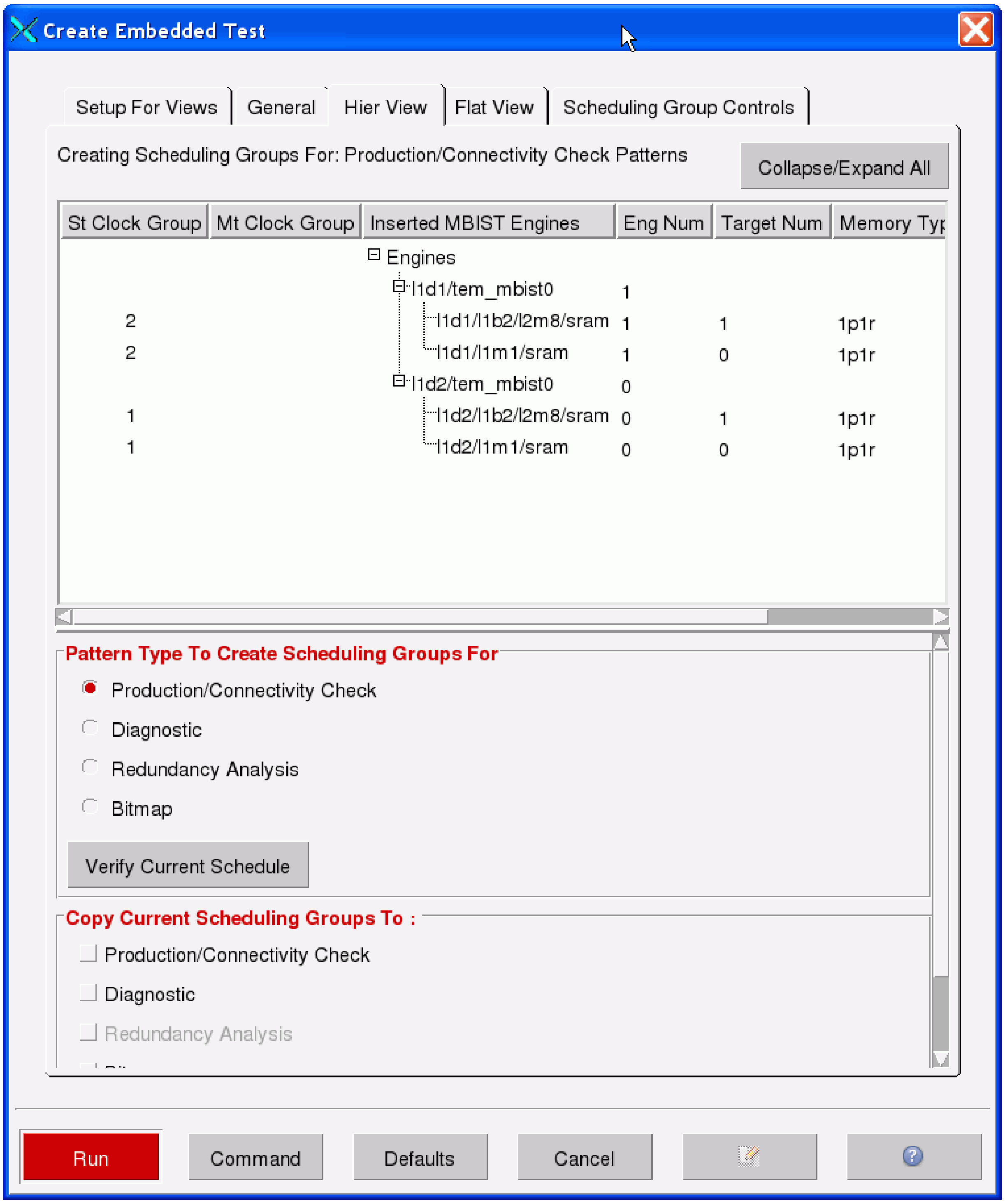
Scheduling Type

Production/Connectivity Check	custom
Diagnostic	custom
Redundancy Analysis	custom
Bitmap	custom
Direct	parallel_parallel

Select the pattern types to create from the *Create Pattern Types* group. Select the custom mode initialization sequences, if required for JTAG or direct access patterns. Finally, select a schedule from the *Scheduling Type* group for each of the pattern types for which patterns are to be generated.

If a custom scheduling type is selected for any of the patterns, the schedule can be modified using the *Hier View* and *Flat View* tabs. Both of these tabs allow you to view all the engines and devices in the design. Figure 4-7 on page 44 displays the *Hier View* window.

Figure 4-7 Hier View Tab



Creating Scheduling Groups For: Production/Connectivity Check Patterns

St Clock Group	Mt Clock Group	Inserted MBIST Engines	Eng Num	Target Num	Memory Typ
		Engines			
		l1d1/tem_mbist0	1		
2		l1d1/l1b2/l2m8/sram	1	1	1p1r
2		l1d1/l1m1/sram	1	0	1p1r
		l1d2/tem_mbist0	0		
1		l1d2/l1b2/l2m8/sram	0	1	1p1r
1		l1d2/l1m1/sram	0	0	1p1r

Pattern Type To Create Scheduling Groups For

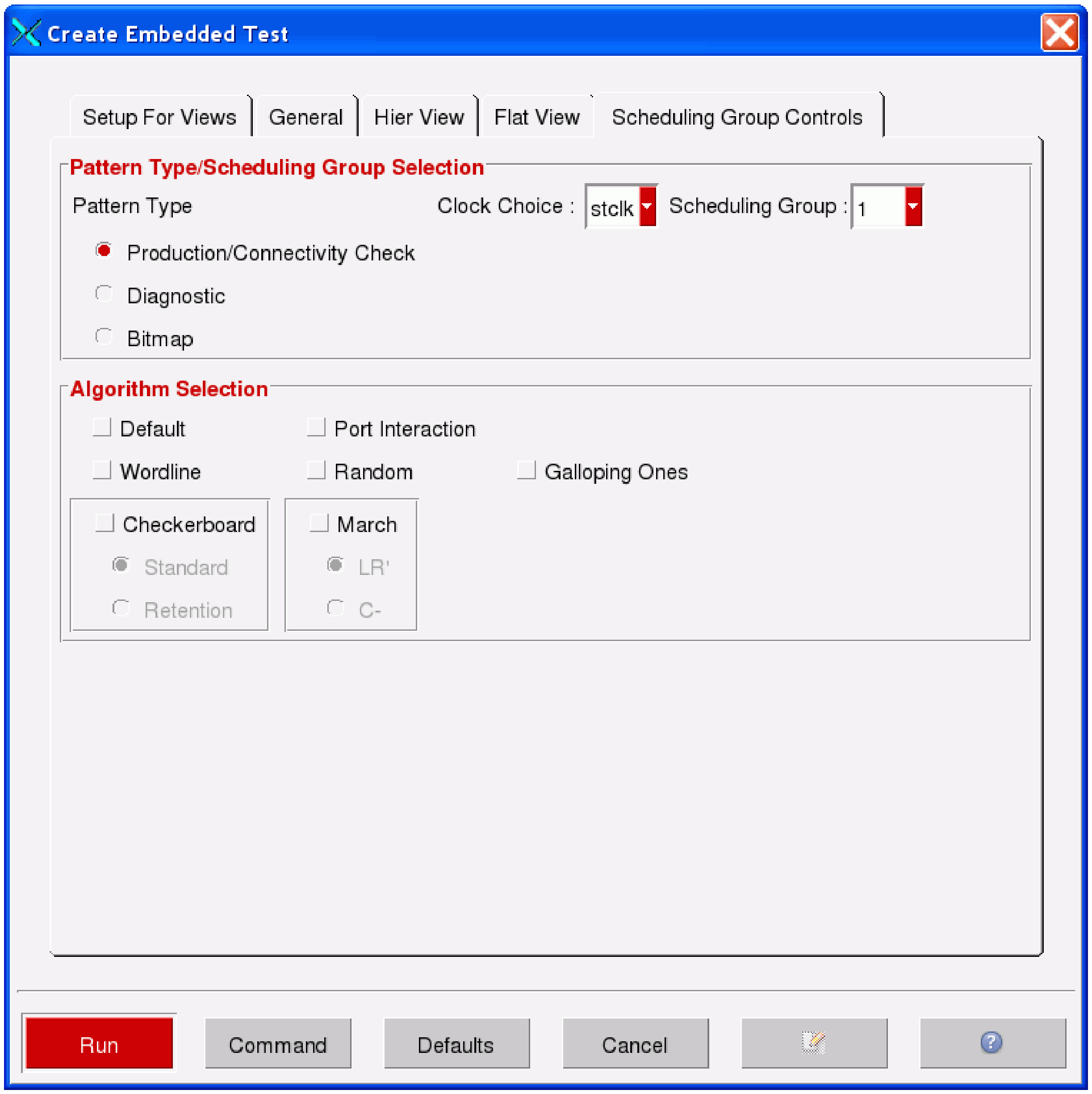
[✓ Production/Connectivity Check] Diagnostic Redundancy Analysis Bitmap

Copy Current Scheduling Groups To :

☐ Production/Connectivity Check
☐ Diagnostic
☐ Redundancy Analysis

The *Scheduling Group Controls* tab allows you to customize the algorithms for which the patterns are to be executed. This is based on the pattern type, clock source, and the scheduling group number. Figure 4-8 on page 45 displays the *Scheduling Group Controls* window.

Figure 4-8 Scheduling Group Controls Tab


Create Embedded Test

Setup For Views | General | Hier View | Flat View | Scheduling Group Controls

Pattern Type/Scheduling Group Selection

Pattern Type — Clock Choice : stclk — Scheduling Group : 1

- [✓ Production/Connectivity Check]
- Diagnostic
- Bitmap

Algorithm Selection

- Default
- Port Interaction
- Wordline
- Random
- Galloping Ones
- Checkerboard: [✓ Standard] Retention
- March: [✓ LR'] C-

After entering the relevant information using the tabs, click *Run* to execute `create_embedded_test` with the specified information.

5 Write Vectors

To export memory built-in self-test patterns from Encounter Test for use in design verification or manufacturing test, use the `write_vectors` command. Certain options should be applied for memory built-in self-test patterns.

- `compresspatterns=no` prevents the events from moving between test cycles, allowing analysis software to reliably predict actions within the memory test patterns.
- `cyclecount=yes` causes test cycle count informational comments to be added to the generated STIL and WGL output formats.
- `keyeddata=yes` causes information allowing consistency checks and failure data analysis to be added as comments to the STIL and WGL output formats.

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

The generally recommended values are listed below.

```
testpioffset=testbidioffet=0
teststrobeoffset=0.1*(TCK period)
testpioffset=TCK=0.25*(TCK period)
```

Typical values as they might be found on the write_vectors command line for a 20MHz JTAG TCK clock follow.

```
testtimeunits=ps \
testpioffset=0 testbidioffset=0 \
testperiod=50000 \
teststrobeoffset=5000 \
testpulsewidth=25000 \
testpioffsetlist=TCK=12500 \
```

6

Pattern Structure

Memory built-in self-test patterns are comprised of a mode initialization sequence as decribed in “Mode Initialization Sequences” on page 29 followed by a pattern generally comprising:

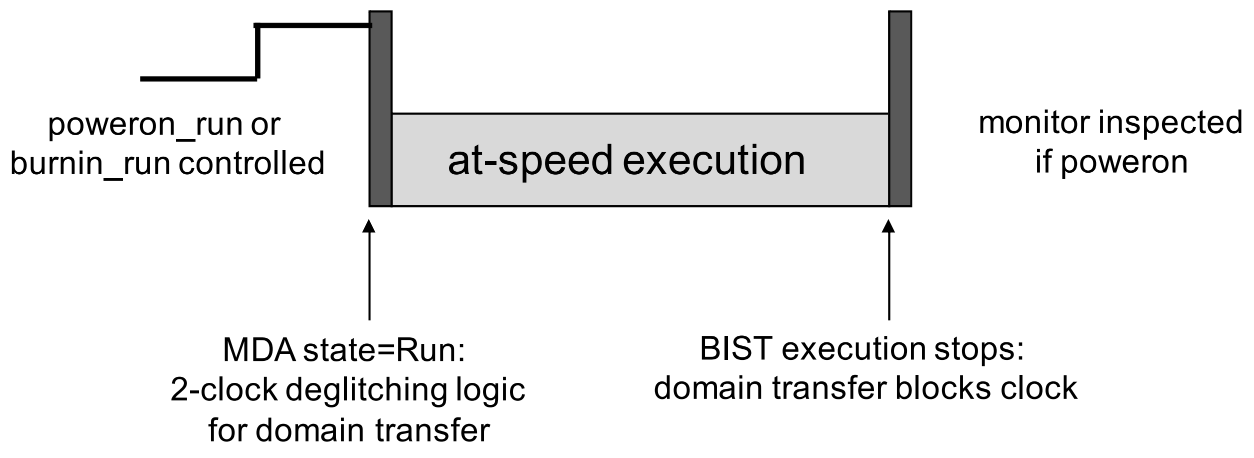
- setup of the targeted tests using the memory built-in self-test access method,
- execution of the selected memory built-in self-test algorithms under control of the system clocks
- examination of the results using the memory built-in self-test access method.

This chapter describes these sequences, distinguished by access method, direct or 1149 TAP, and pattern class.

Direct Access

Direct access is limited to executing production class patterns, as shown in the folloiwng figure, selected with `createpatterns=poweron` or `createpatterns=burnin`.

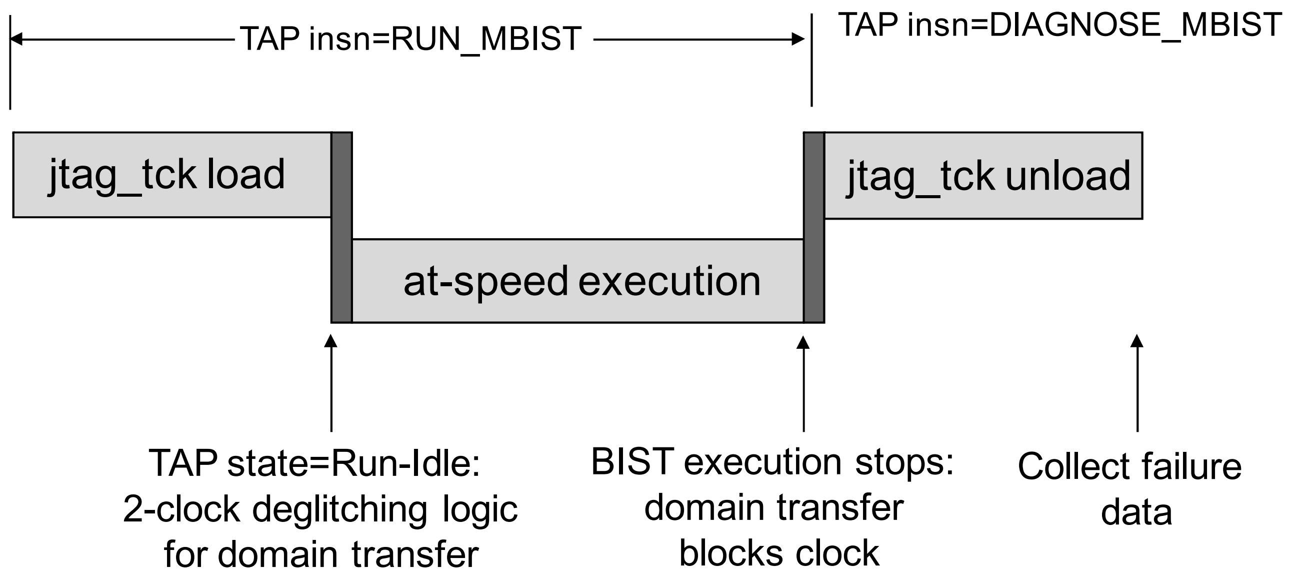
Figure 6-1 Direct Access Production Pattern Abstract View



1149 TAP Access

All pattern classes except direct access can be executed under control of the 1149 TAP controller.

Figure 6-2 1149 TAP Access Production Pattern Abstract View


TAP insn=RUN_MBIST
TAP insn=DIAGNOSE_MBIST
jtag_tck load
at-speed execution
jtag_tck unload
TAP state=Run-Idle: 2-clock deglitching logic for domain transfer
BIST execution stops: domain transfer blocks clock
Collect failure data

Production Retention Pattern

The `checkerboard_retention` algorithm must be selected within the `create_embedded_test` execution schedule to apply the memory retention test.

Checkerboard_retention Algorithm Descriptions

For 1rw memories, the sequence below is executed once in production patterns. For mrnw memories, the sequence below is executed once for each write port in production patterns. MBIST support for mrnw memories is currently limited to mr1w or mr2w.

```
{
W10, inc pause        # Write address space with checkerboard, wait
R10, R10, inc         # Do a double read of each address in the address space
W01, inc pause        # Write address space with inverted checkerboard, wait
R01, R01, inc       # Do a double read of each address in the address space
}
{
W01, inc            # this section of the algorithm tests any bit write enables
W10, inc bw disabled
R01, inc
W10, inc
W01, inc bw disabled
R10, inc
}
```

For 2rw memories, the sequence below is executed once in production patterns.

```
{
W10P0, inc pause       # Write address space with checkerboard via port0, wait
R10P0 & R10P1, R10P0 & R10P1, inc   # Double read of each address on each port
W01P0, inc pause       # Write address space with inverted checkerboard via port0, wait
R01P0 & R01P1, R01P0 & R01P1, inc   # Double read of each address on each port
}
{
W10P1, inc pause       # Write address space with checkerboard via port1, wait
R10P0 & R10P1, R10P0 & R10P1, inc   # Double read of each address on each port
W01P1, inc pause       # Write address space with inverted checkerboard via port1, wait
R01P0 & R01P1, R01P0 & R01P1, inc   # Double read of each address on each port
}
{
W01P0, inc             # this section of the algorithm tests any port0 bit write enables
W10P0, inc bw disabled
R01P0 & R01P1, inc
W10P0, inc
W01P0, inc bw disabled
R10P0 & R10P1, inc
}
{
W01P1, inc             # this section of the algorithm tests any port1 bit write enables
W10P1, inc bw disabled
R01P0 & R01P1, inc
W10P1, inc
W01P1, inc bw disabled
R10P0 & R10P1, inc
}
```

The `create_embedded_test` command generates one Encounter Test experiment for each generated production retention pattern scheduling group. Within this group, the set of devices are analyzed to determine the number of times the patterns must pause. For 1rw and mr1w devices this number is two. For mr2w and 2rw devices this number is four. Each of these pauses requires a CONTINUE operation. The runtime for each of these CONTINUE operations is minimized based on the set of active memory devices in this scheduling group at the time the algorithm pauses. Additionally, if other algorithms are scheduled for execution following checkerboard_retention, their runtimes are included in the last CONTINUE operation.

The experiment created by `create_embedded_test` for a production checkerboard_retention test is

MBIST_ATE_RETENTION_CONTROLLER_*integer_clock-source*

A fixed retention period is inserted by the software for this algorithm. This retention period is based on the following factors:

- The sharing of memories when processing this algorithm. For example, if two memories are sharing the same controller and they have different address widths, then the memory with the smaller address space will pause first, waiting for the second to complete its memory write operation.
- The number of TCK clock pulses needed to leave the Run-Test-Idle (RTI) state, load the CONTINUE_MBIST instruction, and return back to the RTI state, in which the MBIST controllers will start reading the memory. The variance from one design to another for this operation is the difference in lengths of the instruction register in the JTAG module.
- The frequency of TCK feeding the JTAG module defined when exporting the patterns from Encounter Test with the `write_vectors` command. Changing the frequency will directly affect the retention period.
- The minimum retention period required for memory testing is a function of the technology and memory provider.

Additional delay can be included with the following procedure:

1. Write the pattern in TBDpatt format.
2. Copy the RTI-> CONTINUE_MBIST->RTI method.
3. Place the first occurrence in a repeat statement.
4. Change the ending of the first occurrence to bypass the RTI state.
5. Read pattern back into Encounter Test.
6. Write the pattern back out in the vector format of choice.

The experiments created by `create_embedded_test splitretentionpatterns=yes` for a production checkerboard_retention test are

```
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_RUN
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_CONTINUE_1
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_CONTINUE_2
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_CONTINUE_3[1]
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_CONTINUE_4[1]
MBIST_ATE_RETENTION_CONTROLLER_integer_clock-source_DIAGNOSE
[1] These are generated when required by the memory device characteristics.
```

Though there are four to six experiments in this case, it is really one segmented pattern. All must be applied on the tester or within simulation for the MBIST algorithm to complete properly. Prior to merging the pattern segments, they must be modified using the Encounter Test contrib script, `fix_production_splitretentionpatterns`. The command help follows.

```
Description:
This script fixes MBIST production checkerboard_retention patterns that have
been created using the splitretentionpatterns=yes keyword. Before the
individual patterns can be merged together, several updates must be made to the
patterns. This script will make the necessary changes to allow for proper
merging.


After successful completion of the script, the patterns may then be merged
together.


Options:
-initial_verilog_data_file <file> -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence. (suffixed with _RUN by default)
-verilog_data_files <file>        -> Verilog data file(s) to modify
-initial_stil_vector_file <file>  -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence. (suffixed with _RUN by default)
-stil_vector_files <file>         -> STIL vector file(s) to modify
-initial_wgl_vector_file <file>   -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence. (suffixed with _RUN by default)
-wgl_vector_files <file>          -> WGL vector file(s) to modify


Example: fix_production_splitretentionpatterns -verilog_data_files
VER*RETENTION_CONTROLLER_1_ref_clka_CONTINUE_*.data*
         fix_production_splitretentionpatterns -verilog_data_files
VER*RETENTION_CONTROLLER_1_ref_clka_DIAGNOSE*.data*
         fix_production_splitretentionpatterns -initial_verilog_data_file
VER*RETENTION_CONTROLLER_1_ref_clka_RUN*.data*
```

For the set of devices active in the test the appropriate number of `*_CONTINUE_[1-4]` segments are generated, and all these pattern segments must be merged to complete the retention algorithm. As described earlier, one pause occurs when the checkerboard pattern is written and the other pause occurs when the inverted checkerboard pattern is written, and for each write port. Tester directives may be inserted between the pattern segments as they are merged, but they should not alter the number of test cycles executed.

Design Verification

Note that Verilog patterns have two files per segmented experiment, the testbench, typically including a `mainsim.v` suffix in the filename, and the data values, typically containting a `data.*.verilog` suffix in the filename. When merging Verilog patterns, the data values files must be merged into a single file to ensure proper operation. Attempts to reference multiple TESTFILE[1-99] in the testbench result in the introduction of X values on all design input pins between test file executions as the testbench assumes the test files are independent.

Manufacturing Test

Results generated using Automated Test Equipment must be converted to chip-pad-pattern (CPP) format prior to analysis with Encounter Test `analyze_embedded_test` command. The original STIL or WGL pattern created containing the checkerboard_retention test must be used in the conversion process from tester cycles to Encounter Test odometers to ensure accurate analysis.

Figure 6-3 1149 TAP Access Diagnostic Pattern Abstract View

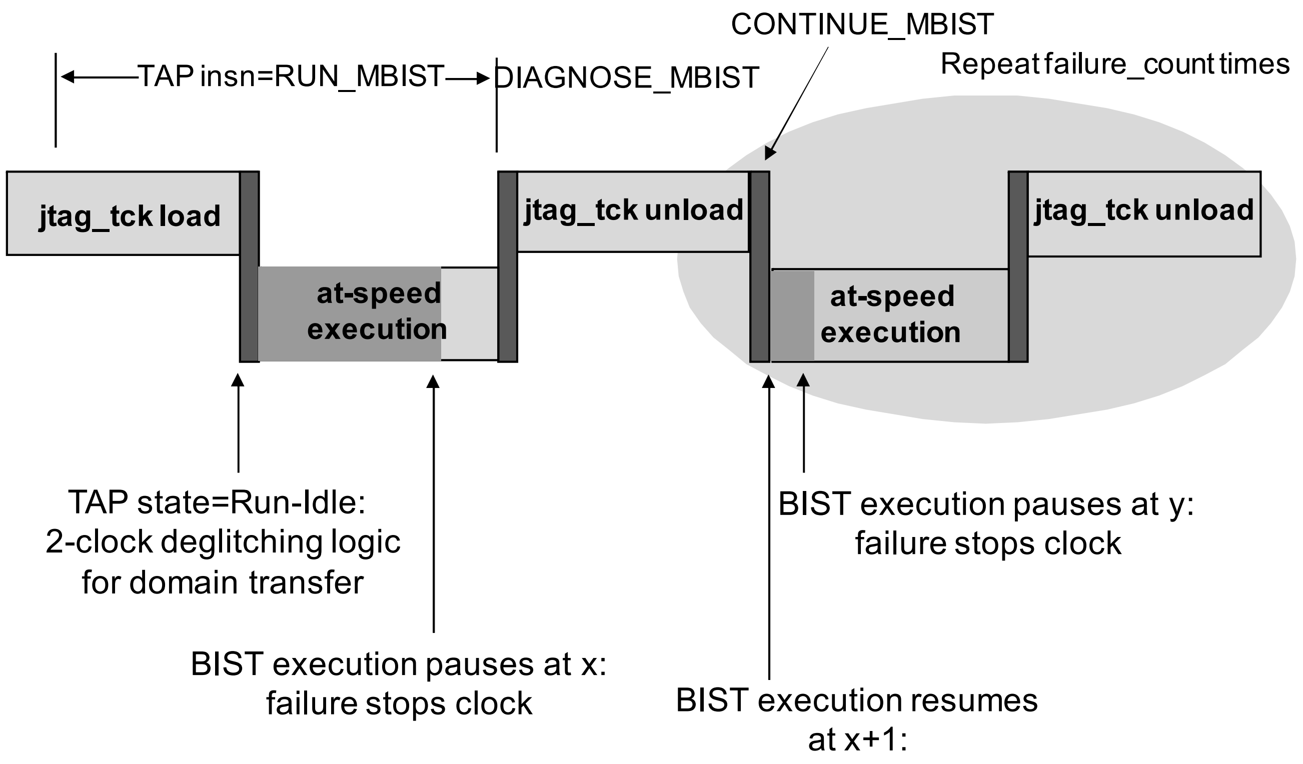
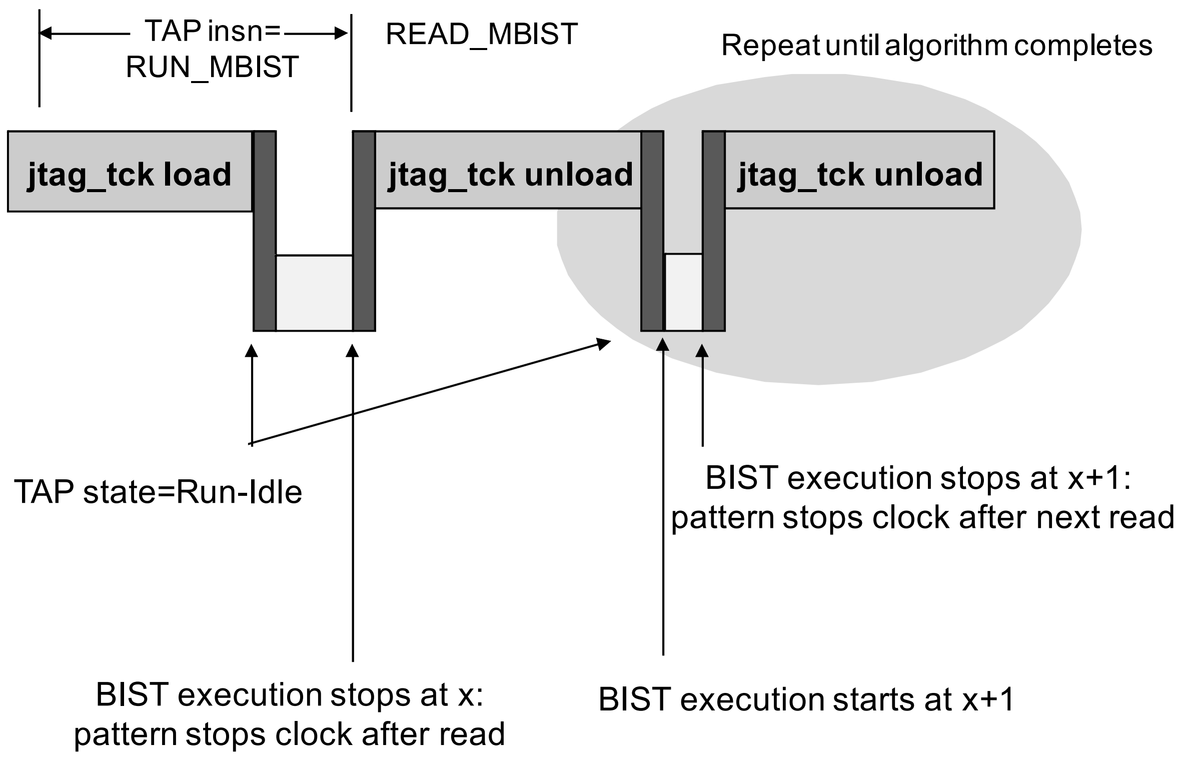

CONTINUE_MBIST
TAP insn=RUN_MBIST
DIAGNOSE_MBIST
Repeat failure_count times
jtag_tck load
jtag_tck unload
jtag_tck unload
at-speed execution
at-speed execution
TAP state=Run-Idle: 2-clock deglitching logic for domain transfer
BIST execution pauses at y: failure stops clock
BIST execution pauses at x: failure stops clock
BIST execution resumes at x+1:

Figure 6-4 1149 TAP Access Bitmap Pattern Abstract View


TAP insn=
RUN_MBIST
READ_MBIST
Repeat until algorithm completes
jtag_tck load
jtag_tck unload
jtag_tck unload
TAP state=Run-Idle
BIST execution stops at x+1:
pattern stops clock after next read
BIST execution stops at x:
pattern stops clock after read
BIST execution starts at x+1

Bitmap Retention Pattern

The checkerboard algorithm must be selected within the `create_embedded_test` execution schedule to apply the bitmap memory retention test. No fixed retention delay period is inserted by the software for this algorithm. Any retention period must be inserted where necessary by the test engineer using tester directives during pattern merging.

Checkerboard Algorithm Descriptions

For 1rw memories, the sequence below is executed once per diagnostic slice. For mrnw memories, the sequence below is executed once per diagnostic slice per write port, called an iteration.

```
{
W10, inc               # Write address space with checkerboard
R10, R10, inc          # Do a double read of each address in the address space
W01, inc               # Write address space with inverted checkerboard
R01, R01, inc          # Do a double read of each address in the address space
}
{
W01, inc               # this section of the algorithm tests any bit write
enables
W10, inc bw disabled
R01, inc
W10, inc
W01, inc bw disabled
R10, inc
}
```

For 2rw memories, the sequence below is executed once per diagnostic slice, called an iteration.

```
{
W10P0, inc                           # Write address space with checkerboard
via port0
R10P0 & R10P1, R10P0 & R10P1, inc    # Double read of each address on each port
W01P0, inc                           # Write address space with inverted
checkerboard via port0
R01P0 & R01P1, R01P0 & R01P1, inc   # Double read of each address on each port
}
{
W10P1, inc                          # Write address space with checkerboard via
port1
R10P0 & R10P1, R10P0 & R10P1, inc  # Double read of each address on each port
W01P1, inc                         # Write address space with inverted
checkerboard via port1
R01P0 & R01P1, R01P0 & R01P1, inc  # Double read of each address on each port
}
{
W01P0, inc                         # this section tests any port0 bit write
enables
W10P0, inc bw disabled
R01P0 & R01P1, inc
W10P0, inc
W01P0, inc bw disabled
R10P0 & R10P1, inc
}
{
W01P1, inc                         # this section tests any port1 bit write
enables
W10P1, inc bw disabled
R01P0 & R01P1, inc
W10P1, inc
W01P1, inc bw disabled
R10P0 & R10P1, inc
}
```

The `create_embedded_test` command generates one Encounter Test experiment for each generated bitmap checkerboard pattern scheduling group. `create_embedded_test splitretentionpatterns=yes` command generates four Encounter Test experiments for each iteration for each generated bitmap pattern scheduling group as delimited by the double underlines in the preceding checkerboard algorithm descriptions.

The experiment created by `create_embedded_test` for a bitmap checkerboard test is

```
MBIST_ATE_BITMAP_integer_clock-source
```

The experiments created by `create_embedded_test splitretentionpatterns=yes` for a bitmap checkerboard test are named as follows

```
MBIST_ATE_BITMAP_integer_clock-source_diagnostic-slice-iteration_segment
```

For example, assuming two iterations are required for the memories being tested, the experiments created by `create_embedded_test splitretentionpatterns=yes` for a bitmap checkerboard test are

```
MBIST_ATE_BITMAP_integer_clock-source_1_1
MBIST_ATE_BITMAP_integer_clock-source_1_2
MBIST_ATE_BITMAP_integer_clock-source_1_3
MBIST_ATE_BITMAP_integer_clock-source_1_4
MBIST_ATE_BITMAP_integer_clock-source_2_1
MBIST_ATE_BITMAP_integer_clock-source_2_2
MBIST_ATE_BITMAP_integer_clock-source_2_3
MBIST_ATE_BITMAP_integer_clock-source_2_4
```

Though there are eight experiments in this case, it is really one segmented pattern. All must be applied on the tester or within simulation for the MBIST algorithm to complete properly. Prior to merging the pattern segments, they must be modified using the Encounter Test contrib script, `fix_bitmap_splitretentionpatterns`. The command help follows.

```
Description:

This script fixes MBIST bitmap checkerboard patterns that have been created
using the splitretentionpatterns=yes keyword.  Before the individual patterns
can be merged together, several updates must be made to the patterns.  This
script will make the necessary changes to allow for proper merging.  Note that
the initial pattern file must be specified separately from the rest of the
patterns.


After successful completion of the script, the patterns may then be merged
together.


Options:

-initial_verilog_data_file <file> -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence.
-verilog_data_files <file>        -> Verilog data file(s) to modify
-initial_stil_vector_file <file>  -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence.
-stil_vector_files <file>         -> STIL vector file(s) to modify
-initial_wgl_vector_file <file>   -> When the patterns are merged together,
                                     this pattern contains the mode init
                                     sequence.
   -wgl_vector_files <file>          -> WGL vector file(s) to modify
```

```
Example: fix_bitmap_splitretentionpatterns
          -initial_verilog_data_file VER*BITMAP_1_ref_clka_1_1.data*
          -verilog_data_files VER*BITMAP_1_ref_clka_[2-9]_*.data*
```

Design Verification

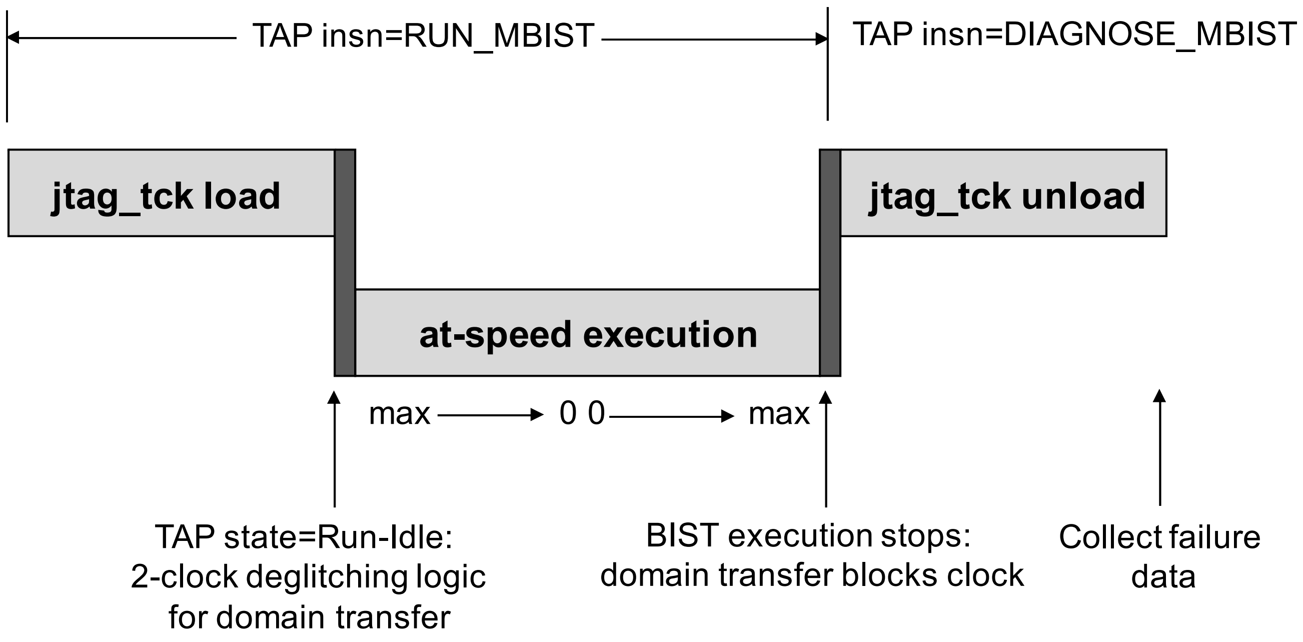
Note that Verilog patterns have two files per segmented experiment, the testbench, typically including a `mainsim.v` suffix in the filename, and the data values, typically containting a `data.*.verilog` suffix in the filename. When merging Verilog patterns, the data values files must be merged into a single file to ensure proper operation. Attempts to reference multiple TESTFILE[1-99] in the testbench result in the introduction of X values on all design input pins between test file executions as the testbench assumes the test files are independent.

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

Manufacturing Test

Results generated using Automated Test Equipment must be converted to chip-pad-pattern (CPP) format prior to analysis with Encounter Test `analyze_embedded_test` command. The original STIL or WGL pattern created containing the checkerboard test must be used in the conversion process from tester cycles to Encounter Test odometers to ensure accurate analysis.

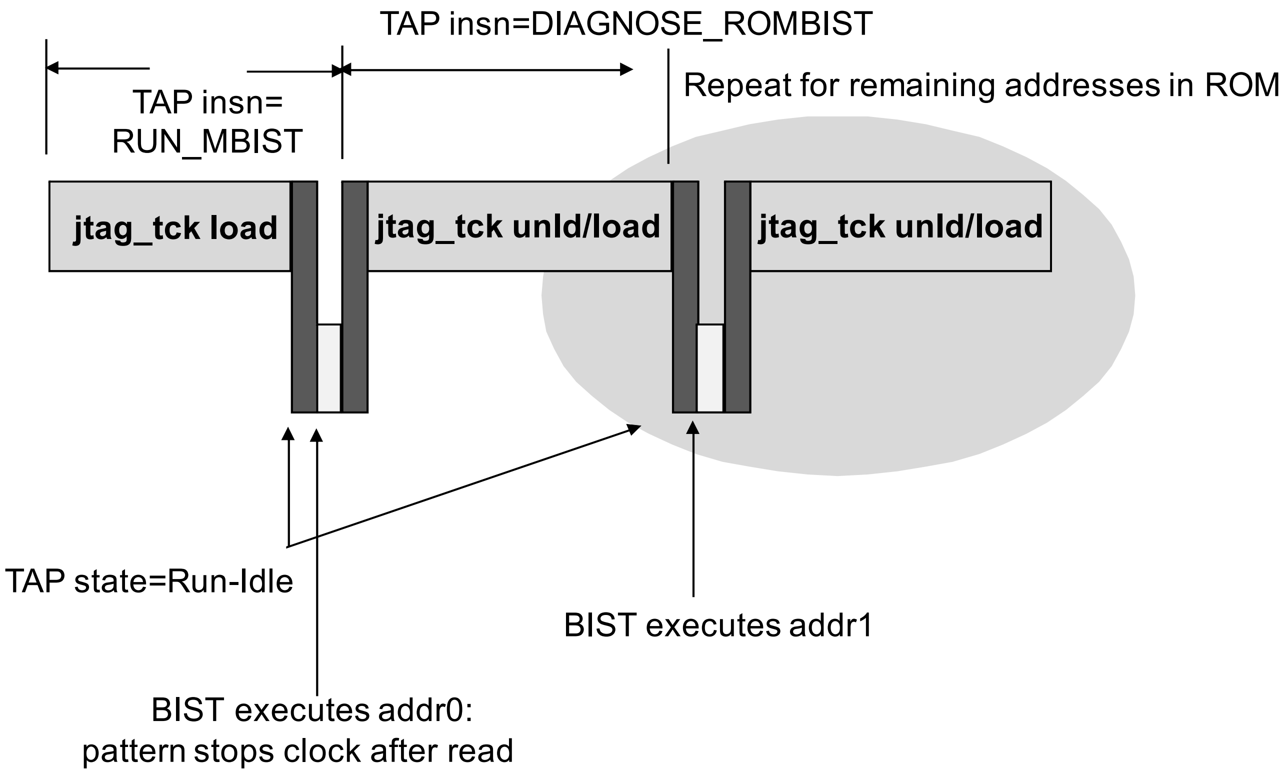
Figure 6-5 1149 TAP Access ROM Production Pattern Abstract View



Test algorithm:

- `create_embedded_test` uses ROM data load file to compute seed signature for MISR to achieve all zeroes result if good test
- Pattern loads seed signature into MISR via TAP access or designer stores in top ROM address locations when direct access is enabled
- Pattern reads ROM from maximum address down to 0, then 0 up to maximum less seed addresses, if present, compressing data into the MISR

Figure 6-6 1149 TAP Access ROM Diagnostic Pattern Abstract View



Test algorithm:

- `create_embedded_test` uses ROM data load file for each address from 0 to maximum
- Pattern loads MISR with addressed location contents, reads ROM, compares and examines result with all zeroes result yielding good test at each location

7

Design Verification Support

Pattern Simulation

Verilog testbenches suitable for simulation of the memory built-in self-test patterns for a design can be generated directly from an RTL Compiler session by using `write_mbist_testbench` or within an Encounter Test session after executing `create_embedded_test` by using `write_vectors language=verilog`. Using Incisive Enterprise simulation logs, it is also possible to generate chip-pad-pattern data and tester data logs. For certain analyses to proceed correctly, the exported Verilog testbenches require subsequent modification using contrib scripts supplied in the Encounter Test installation. This enables verification of pattern execution within the designer's simulation environment as well as verification of the *simulation-generated* manufacturing test output.

Stuck-at fault injection within memory models coupled with failure analysis is also possible with some user modification of the Verilog memory models and fully supported in the verification environment.

The Encounter Test `analyze_embedded_test` command is the results analysis tool.

1149 Verification

To execute `verify_11491_boundary` within Encounter Test to validate the memory built-in self-test JTAG instructions on a full chip design, the test mode must be created using the `TBDseqPatt.design.1149` file generated by `create_embedded_test` to properly initialize the state of the design.

Caution

If the IEEE1149.1 TAP clock is selected as a clock source for a target group, miscompares may arise in 1149 boundary scan verification of memory built-in self-test instructions due to a transition through Run-Test-Idle in the test sequence between the Update-IR and Select-DR-

Scan. This transition is not part of the generated memory built-in self-test execution sequence and, consequently, design verification does not encounter a similar problem.

8 Manufacturing Test Support

A mechanism is required to correlate the patterns generated by Encounter Test `create_embedded_test` with the result logs generated during memory test in manufacturing and subsequently analyzed by Encounter Test's `analyze_embedded_test`. The defined mechanism relies on a pointer, the `create_embedded_test testblockname`, being embedded within the STIL or WGL files exported from Encounter Test using `write_vectors keyeddata=yes`. The keyed data stored within the exported patterns is of the following format:

```
MBISTFailDataSync = "pattern-control-filename pattern-class clock-source
scheduling-index testblockname algorithm-list clkstep-vector odometer"
```

- *`pattern-control- filename`* is the pattern control file in the interface file set used in the invocation of the `create_embedded_test` command
- *`pattern-class`* is
  - `production` - covers production, production retention
  - `diagnostic`
  - `romdiagnostic` - covers ROM diagnostics
  - `bitmap` - covers bitmap, bitmap retention
  - `poweron`
  - `burnin`
- *`clock-source`* is
  - `stclk`
  - `mtclk`
  - `itclk`
- *`scheduling-index`* is the scheduling group for the appropriate pattern-class and clock-source in the pattern control file

- *`testblockname`* must be unique across the patterns generated and applied to a design and is supplied by the test engineer
- *`algorithm-list`* is the set of algorithms executed in the generated pattern as specified in the insert_dft mbist command input configuration file and *romtest* for ROMs
- *`clkstep-vector`* is a binary string indicating which devices in the pattern control file are active (1) and inactive (0) in this pattern;

  The string is ordered left to right to correlate with the devices top to bottom in the pattern control file.
- *`odometer`* contains the first measure statement in bitmap patterns; otherwise it has a value of zero.

The manufacturing memory test data log entries must identify the memory built-in self-test patterns by using `testblockname` specified when `create_embedded_test` generated the patterns. This data log must be converted to chip-pad-pattern data prior to Encounter Test `analyze_embedded_test` processing.

The STIL or WGL files exported from Encounter Test using `write_vectors` must use the keyword `cyclecount=yes`. This keyword causes comments to be printed in the patterns indicating the current cycle count. These comments are required when using the `prepare_memory_failset` command.

Pattern-related Data Flows

The data flow relevant to several pattern classes through manufacturing test and data log analysis is shown in the following sections. Cadence capabilities and customer responsibilities are identified in these flows. RC represents RTL Compiler and ET represents Encounter Test applications in the figures.

Figure 8-1 Production Pattern Data Flow

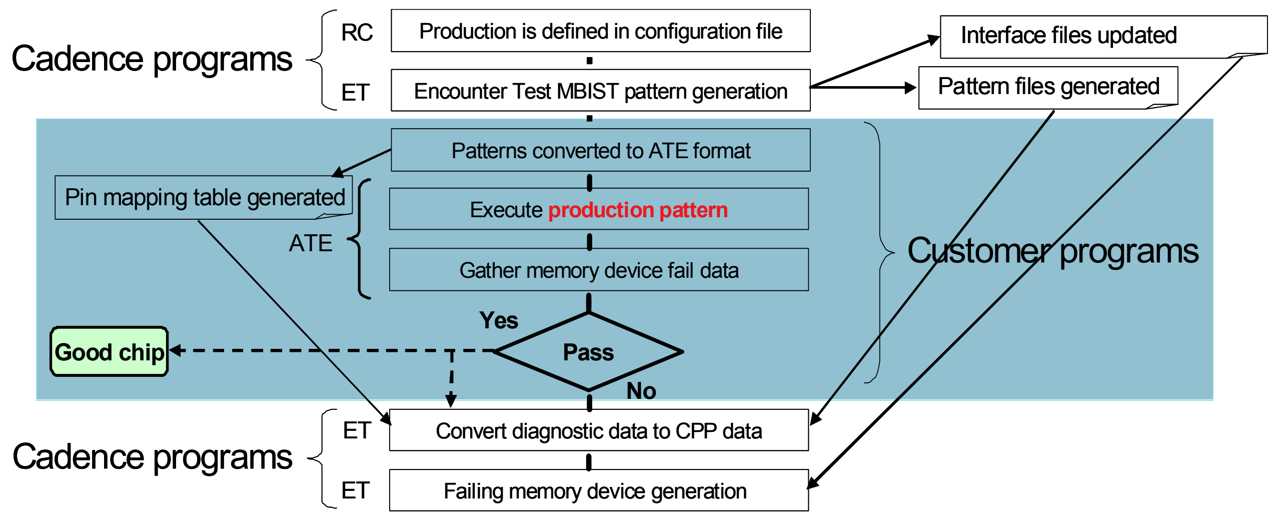

Cadence programs
RC
Production is defined in configuration file
ET
Encounter Test MBIST pattern generation
Interface files updated
Pattern files generated
Patterns converted to ATE format
Pin mapping table generated
Execute production pattern
ATE
Gather memory device fail data
Customer programs
Yes
Good chip
Pass
No
Cadence programs
ET
Convert diagnostic data to CPP data
ET
Failing memory device generation

Figure 8-2 Production Pattern with Data Bit Redundancy Data Flow

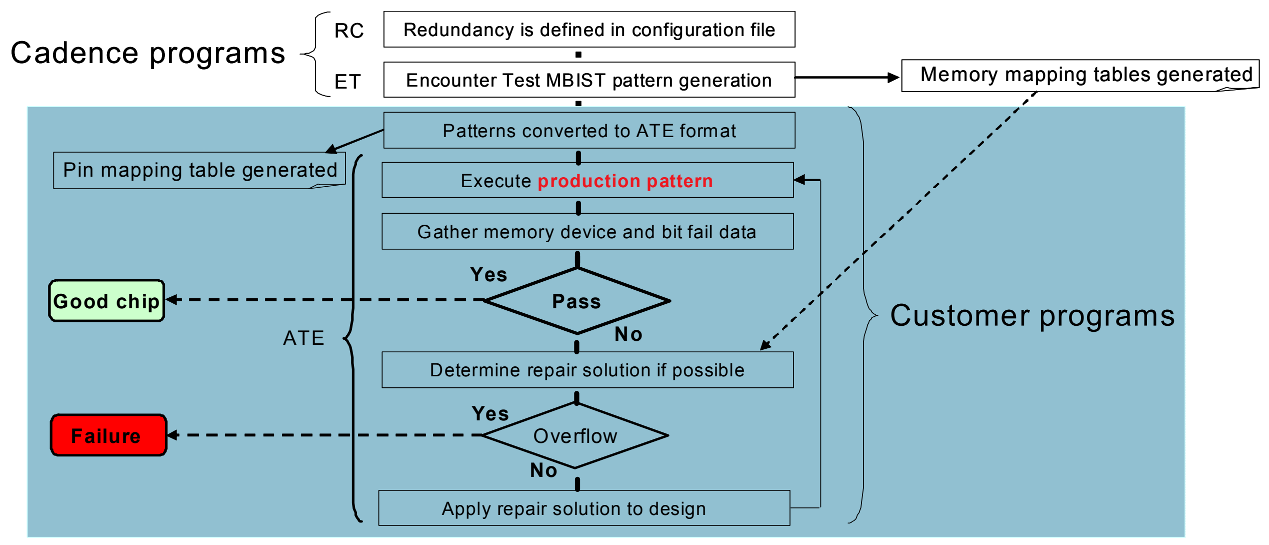

Cadence programs
RC
Redundancy is defined in configuration file
ET
Encounter Test MBIST pattern generation
Memory mapping tables generated
Patterns converted to ATE format
Pin mapping table generated
Execute production pattern
Gather memory device and bit fail data
Yes
Good chip
Pass
No
ATE
Customer programs
Determine repair solution if possible
Yes
Failure
Overflow
No
Apply repair solution to design

Figure 8-3 Diagnostic Pattern Data Flow

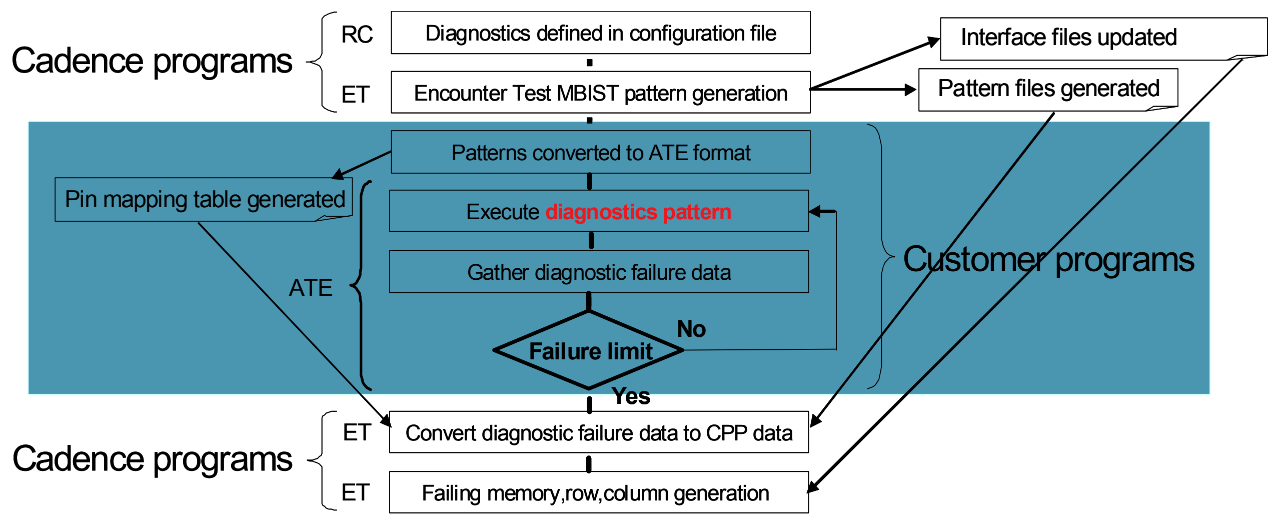
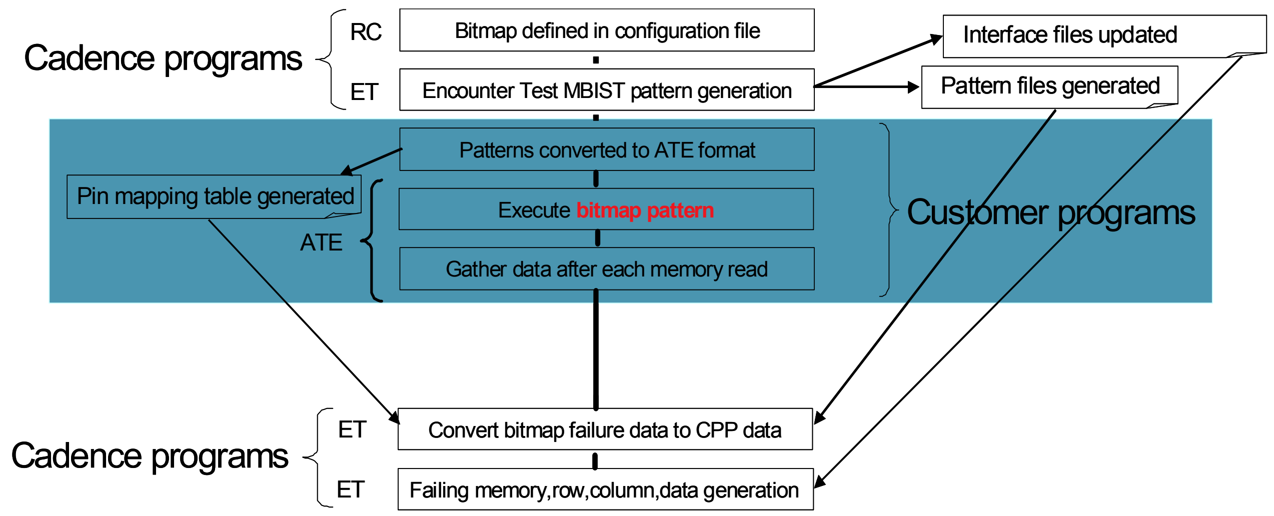

Cadence programs
RC
ET
Diagnostics defined in configuration file
Encounter Test MBIST pattern generation
Interface files updated
Pattern files generated
Patterns converted to ATE format
Pin mapping table generated
Execute diagnostics pattern
Gather diagnostic failure data
ATE
Failure limit
No
Yes
Customer programs
Cadence programs
ET
ET
Convert diagnostic failure data to CPP data
Failing memory,row,column generation

Figure 8-4 Bitmap Pattern Data Flow


Cadence programs
RC
ET
Bitmap defined in configuration file
Encounter Test MBIST pattern generation
Interface files updated
Pattern files generated
Patterns converted to ATE format
Pin mapping table generated
Execute bitmap pattern
ATE
Gather data after each memory read
Customer programs
Cadence programs
ET
ET
Convert bitmap failure data to CPP data
Failing memory,row,column,data generation

Data Bit Redundancy Support

If data bit redundancy is enabled for a device, then mapping files are created that indicate the tester cycle where a particular bit is measured. For each bit of the device, each read port is listed, along with the measure port and the tester cycle. These mapping files are located in the *workdir*/testresults/testmode_data directory. The files are named as *experiment-name.map*.

Example 8-1 Contents MBIST_ATE_PROD_1_REF_CLKA.map

```
RAMBST  : NAME  = /TopBlock/l1d2/l1m2/sram#0_32
        : BIT   = 4
        : WORD  = 128
        : OUT   = DR[0](jtag_mbist_tdo, 2213775),
                  DR[0](jtag_mbist_tdo, 2213783),
                  DR[1](jtag_mbist_tdo, 2213776),
                  DR[1](jtag_mbist_tdo, 2213784),
                  DR[2](jtag_mbist_tdo, 2213777),
                  DR[2](jtag_mbist_tdo, 2213785),
                  DR[3](jtag_mbist_tdo, 2213778),
                  DR[3](jtag_mbist_tdo, 2213786);
```

In this example, a failure of bit 0 of device /TopBlock/l1d2/l1m2/sram is measured on the port jtag_mbist_tdo will be detected on read port 0 at tester cycle 2213775 and on read port 1 at tester cycle 2213783. A failure of bit 1 will be detected on read port 0 at tester cycle 2213776 and on read port 1 at tester cycle 2213784.