SIEMENS EDA

Tessent™ Diagnosis User's Manual

Software Version 2022.4
Document Revision 27

SIEMENS

About Siemens Digital Industries Software

Revision History ISO-26262

Revision	Changes	Status/ Date
27	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Dec 2022
26	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Sep 2022
25	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Jun 2022
24	Modifications to improve the readability and comprehension of the content. Approved by Lucille Woo. All technical enhancements, changes, and fixes listed in the *Tessent Release Notes* for this product are reflected in this document. Approved by Ron Press.	Released Mar 2022

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

List of Figures

List of Tables

Chapter 1 The Diagnosis Process

Tessent Diagnosis uses failure data from manufacturing test, scan test patterns, and design information. With this data, Tessent Diagnosis identifies the location and classification of the defect causing the failure. Detailed analysis of devices that fail manufacturing test has been shown to greatly reduce the failure analysis effort and enables a diagnosis-driven yield analysis flow.

For the complete list of Tessent-specific terms, refer to the *Tessent Glossary*.

Tessent Diagnosis Features

Tessent Diagnosis operates within the Tessent Shell environment and provides advanced diagnostics for troubleshooting chips that fail on Automated Test Equipment (ATE). From the information you gather from the troubleshooting process, you can apply improvements for chip yield to your design and manufacturing processes.

Tessent Diagnosis provides the following features:

- Analyzes failures compiled by ATE
- Verifies test patterns

  As the first step of the diagnosis, Tessent Diagnosis runs a good machine simulation on the test patterns you specify and compares simulated values with expected values in the test patterns. The values must match before the diagnosis can continue.
- Verifies failure data

  As the second step of the diagnosis, Tessent Diagnosis checks the failure file for conversion errors and verifies consistency between the expected value of the failed test patterns in the failure file and the expected values of the failed patterns in the pattern file.
- Analyzes chain failures

  Tessent Diagnosis analyzes multiple faulty chains with single or multiple faults per chain (of the same type) with chain diagnostics.
- Analyzes logic failures
- Uses compressed or uncompressed patterns

  Tessent Diagnosis can use uncompressed or EDT-compressed patterns for diagnosis.

- Scores and classifies suspects

  Tessent Diagnosis analyzes the ATE failure file to identify the likely suspects for each failure, and ranks and classifies them into a configurable report. When Tessent Diagnosis completes diagnosis, the tool produces a report listing the fail subset (Symptom), possible suspect locations (Suspect), and the related pin, cell, and net paths.

- Displays suspects in physical layout view

  The diagnosis report provides links you can use with the Calibre® RVE™ tool for displaying suspect locations on the physical layout. You can also use the Calibre DESIGNrev™ tool in Calibre 2005.2 and later releases. This requires access to a Calibre software tree.

- Displays suspects in netlist logic view

  The diagnosis report provides links you can use with Tessent Visualizer for displaying suspect locations in a schematic representation of the netlist.

Limitations

Tessent Diagnosis currently has thefollowing limitations:

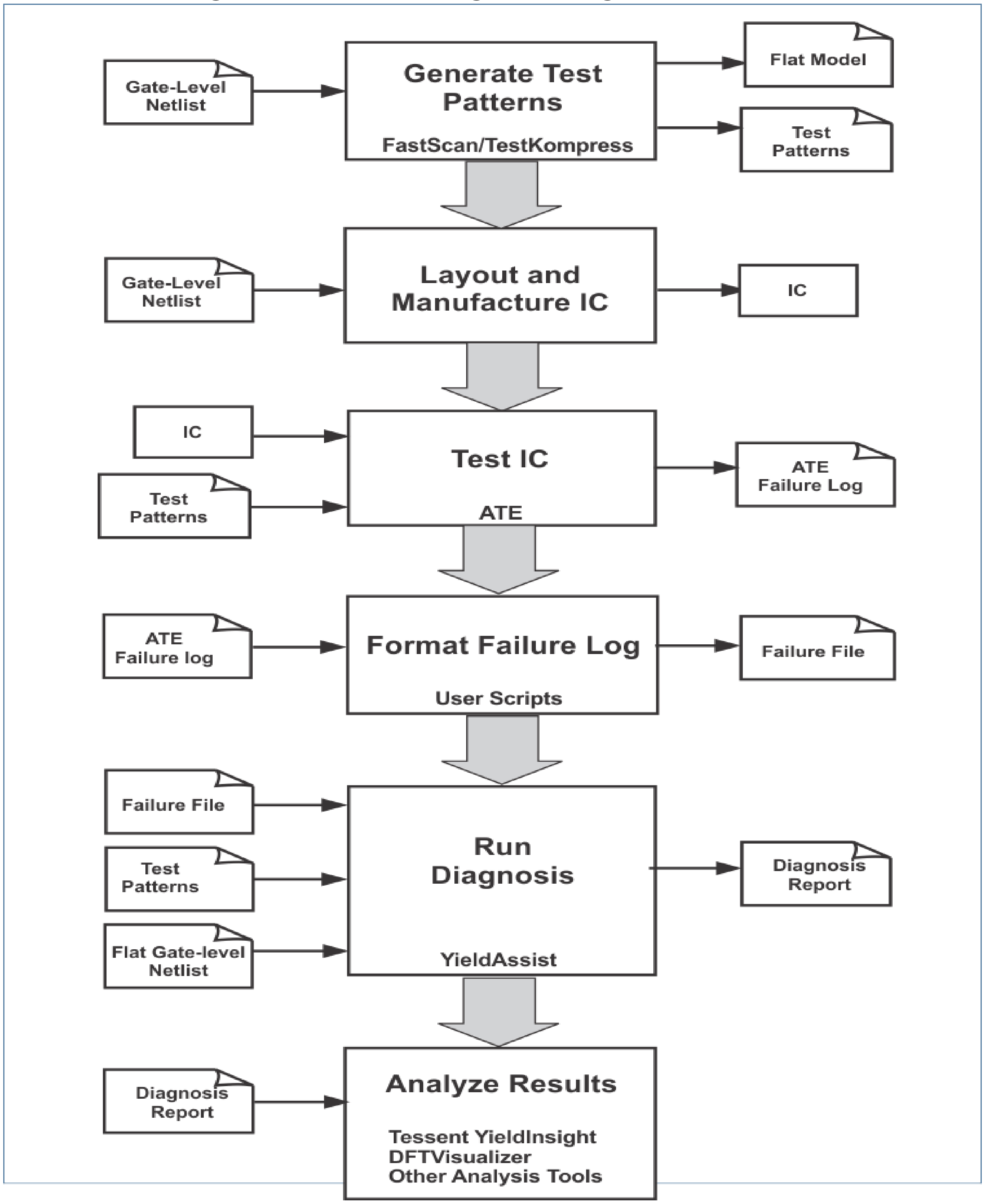
- Test pattern creation — You must create test patterns with Tessent FastScan™ or Tessent TestKompress™.
- No support for macro test patterns — You must separately save the macro test patterns separate from the test patterns you use for diagnosis. See "Preparing the Test Patterns."

Overview of the Diagnosis Process

Tessent Diagnosis is used to identify the defects on your chip that caused the scan test patterns to fail by reporting the location and classification of the faults.

Figure 1-1 shows an overview of the diagnosis process in Tessent Diagnosis and how this fits into the Automatic Test Pattern Generation (ATPG) process.

Figure 1-1. Tessent Diagnosis Diagnostic Process



Input File Requirements

Before running the diagnosis, you must prepare the following input files: test patterns, design netlist, and ATE failure file.

Note

To ensure the accuracy of the diagnosis, it is critical you synchronize the creation of the input files. Specifically, you should produce each of the input files from the identical version of the design, setups, and test pattern files.

You must prepare the following input files for an accurate diagnosis:

- Test patterns — see Preparing the Test Patterns.
- Design netlist — see Preparing the Design Netlist.
- ATE failure file— see Guidelines for Preparing the ATE Failure File.
- Verification data — see Reverse Mapping Top-Level Failures to the Core.

Preparing the Test Patterns

You must create the final version of the test patterns immediately before you save the design's flat netlist that you use for diagnosis. These test patterns must also be the same test patterns the ATE has applied when it produces the failure file for the diagnosis.

Table 1-1 lists the failure-file formats and Named Capture Procedure (NCP) modes that are compatible with supported test pattern formats for Tessent Diagnosis.

Table 1-1. Compatible Failure-File Formats and NCP Modes

Test Pattern Format	Compatible Failure-File Formats and NCP Modes
Parallel WGL	Format: cycle-based and pattern-based NCP mode: internal and external
ParallelSTIL	Format: cycle-based and pattern-based NCP mode: internal and external
ASCII	Format: pattern-based NCP mode: internal
Binary	Format: pattern-based NCP mode: internal and external

See "Saving Internal and External Patterns" in the *TessentScan and ATPG User's Manual* for complete information.

Prerequisites

- You can only use WGL and STIL patterns for diagnosis with cycle-based failure files.

  Note
  The cycle-based format is recommended for use with Tessent Diagnosis.

- Ensure that you are using ATPG cell libraries that model at the gate level. The tool does not diagnose down to gate-level defects if the ATPG model is at a higher-level block—for example, a module that includes RAM memories with scan cells and bypass logic—which can cause loss of diagnostic resolution and skewed RCD analysis results. In addition, ensure that the gate-level models are synchronized with their corresponding layout cells.

Procedure

1. Load the test patterns into Tessent Diagnosis using the read_patterns command as follows:

   read_patterns *pattern_file_name*

   This command syntax is for Tessent Diagnosis scan diagnosis mode. The syntax for Tessent Diagnosis server mode is different—see "Tessent Diagnosis Server Interface."

   By default, the read_patterns command assumes you are loading a single test pattern file, and replaces the existing test pattern file each time a new file is loaded. To load multiple test pattern files, you must merge them together with the -append switch.

2. Run the read_patterns command using the -append switch to load each additional test pattern file. For example:

   ```
   FAULT> read_patterns pat_file1
   FAULT> read_patterns pat_file2 -append
   FAULT> read_patterns pat_file3 -append
   FAULT> diagnose_failures scan_failure_file_mts_pat123
   ```

   Also see the multiple test suites example in "Cycle-Based Failure File Examples."

   Table 1-2 identifies the test pattern file compatibilities when using multiple test pattern files.

Table 1-2. Compatible Test Pattern Formats for Multiple Test Patterns

Test Pattern	Compatible With:
WGL	WGL, STIL
STIL	STIL, WGL
Binary	Binary, ASCII
ASCII	ASCII, Binary

The Design Netlist

After generating the final test patterns, you must immediately save your test pattern file and your flat model. Save the files right after issuing the create_patterns command to guarantee that the test patterns and netlist reflect the same data and setup parameters.

You must save different flat models if you have different settings for generating stuck-at and at-speed type of patterns. Refer to "Flat Models with Different Settings for Stuck-At and At-Speed Patterns" for more information.

Preparing the Design Netlist

When preparing your design netlist, avoid using HDL filename extensions such as .v, .vhd, or .vdhl. Instead, an extension such as .flat is a good mnemonic.

Do not use flat models saved in setup mode or before executing the create_patterns command.

Prerequisites

- You have generated your final test patterns.

Procedure

1. In ATPG analysis mode (scan -pattern context), run the following command:

   ```
   create_patterns
   ```

2. In ATPG analysis mode (scan -pattern context), run the following command to save the pattern file. For example:

   ```
   write_patterns pattern_file.stil.gz -stil
   ```

3. In ATPG analysis mode (scan -pattern context), run the following command to save the flat netlist. For example:

   ```
   write_flat_model flat_model_name.flat.gz
   ```

Results

By default, the tool saves and includes the necessary setup information in the flat model. The diagnosis requires all the same setup information, design rule checks, capture handling, sequential depth, and so on used to create the original test patterns.

Related Topics

Flat Models with Different Settings for Stuck-At and At-Speed Patterns

ATPG Change Impacts on Flat Netlists

Flat Models with Different Settings for Stuck-At and At-Speed Patterns

In ATPG, you must save different flat models if you have different settings for generating stuck-at and at-speed pattern types. Different settings can introduce different pin constrains and other settings.

For more information, see "ATPG Change Impacts on Flat Netlists."

The following example demonstrates saving two flat models when a pin constraint is set to a different value in order to generate stuck-at and at-speed patterns:

```
//set up pin constrain to generate stuck-at patterns
SETUP> add_input_constraints clk_sel -c0
...
//default to generate stuck-at faults
ATPG> create_patterns -auto
ATPG> write_patterns tst_sta.stil -stil
ATPG> write_flat_model design.v.sta.flat -all
//In the same atpg process, pin constrain is set to a different value for generating
//transtion patterns
SETUP> add_input_constraints clk_sel -c1
...
ATPG> set_fault_type transition
ATPG> create_patterns -auto
ATPG> write_patterns tst_tra.stil -stil
ATPG> write_flat_model design.v.tra.flat -all
```

ATPG Change Impacts on Flat Netlists

You must regenerate and save a new flat design model when certain changes occur in ATPG.

If any of the following changes occur in ATPG, regenerate and save a new flat design model:

- Capture handling and clock-off simulation — Changing the split capture or clock-off simulation using the set_split_capture_cycle or set_clock_off_simulation commands, respectively, requires a new flat model.
- ATPG constraints — Adding different ATPG constraints requires a new flat model.
- Cell constraints — Using different cell constraints requires a new flat model.

- Pin constraints — Specifying different pin constraints requires a new flat model.
- Force_pi — Changes to force_pi in test procedures even if there is no impact on the test mode require a new flat model when the force_pi sets different values.

You can use the original flat model without re-saving if you make any of the following changes in ATPG:

- Capture procedure variations — Using a different sequential depth, with or without named capture procedures, different pattern types, and so on. For example:
    - Using different clock sequential depth
    - Turning on or off named capture procedures
    - Using different pattern types, with the exception of RAM-sequential patterns, that cannot be loaded when the clock off simulation is ON

Pattern Verification and the Diagnosis Startup Cache

If the scan and chain patterns were not previously verified, Tessent Diagnosis automatically performs pattern verification when you issue the first diagnose_failures command.

During pattern verification, Tessent Diagnosis checks the following:

- If simulation values are the same as the expected values stored in pattern. If there is a binary mismatch (expected value is 1/0, but simulated value is 0/1), then pattern verification fails.
- If there is a X2B mismatch (expected value is X, simulated value is 0/1) and you run the following command:

  set_diagnosis_options –X2B_mismatch Warning

  Then, the Tessent Diagnosis pattern verification automatically flags the X2B mismatch such that in good machine simulation, the tool changes the simulation to X whenever the pattern has an expected X.
- If there is a B2X mismatch (expected value is 0/1, simulated value is X) and you run the following command:

  set_diagnosis_options –B2X_mismatch Warning // the default setting

  Then, the Tessent Diagnosis pattern verification automatically flags the B2X mismatch such that if any B2X mismatch shows up in a fail file, the tool removes the mismatch.
- For chain patterns, Tessent Diagnosis reports the verification results as follows:
  - Chain pattern verification passes: no message.
  - Chain pattern verification fails: warning message and continue.

    During verification, the tool can produce the following warnings for failing chain patterns:

    - For EDT, if an observed internal scan chain has no corresponding chain masking pattern, then the tool issues a warning similar to the following:

      ```
      Warning: Chain chain_1 has no corresponding masking chain
      pattern.
      ```

    - For non-EDT designs, if all chain patterns for a chain do not cover “00”, “01”, “10”, and “11” sequences, or for EDT designs, if all masking chain patterns for a chain do not cover the “00”, “01”, “10”, and “11” sequences, then the tool issues a warning similar to the following:

      ```
      Warning: Chain chain_1 does not cover sequence "01”
      ```

By default, Tessent Diagnosis performs scan and chain pattern verification any time you set or change the pattern source. You specify the pattern source using the read_patterns command.

Alternatively, you can force pattern verification before you perform diagnosis by using the Diagnosis Startup Cache.

Diagnosis Startup Cache

You can create and update a Diagnosis Startup Cache that stores the pattern verification results, including a masking file, and X2B or B2X mismatches. Using the Diagnosis Startup Cache can greatly reduce the diagnosis runtime by re-using the pattern verification information without re-running the verification, assuming you have made no changes to the patterns.

You only need one file as the startup cache. You create or update the Diagnosis Startup Cache using the verify_patterns command.

Note

Currently, creating and updating a Diagnosis Startup Cache with the verify_patterns command is limited to the Tessent Diagnosis scan diagnosis point tool. When using the Tessent Diagnosis Server, you can only load an existing Diagnosis Startup Cache—see the add_startup_cache server command.

You can update or use an existing cache for as many combinations of tool version, flat model, pattern, and pattern mask file as you want. In essence, the Diagnosis Startup Cache is a database. For example, the following command sequence demonstrates loading two different patterns sets (*pattern_set_01.stil* and *pattern_set_02.stil*) into the same Diagnosis Startup Cache (*my_database.db*).

```
read_patterns pattern_set_01.stil

verify_patterns -create_startup_cache my_database.db

diagnose_failures fail1

read_patterns pattern_set_02.stil              //assuming pattern_set_02.stil and
                                              //pattern_set_02.stil use the same flatmodel

verify_patterns -load_startup_cache my_database.db

diagnose_failures fail2
```

For each pattern verification run, Tessent Diagnosis stores an associated timestamp for the run. During subsequent Diagnosis Startup Cache updates, any previous data more than 300 days old is deleted automatically from the Diagnosis Startup Cache.

Diagnosis Startup Cache Loading Errors

Tessent Diagnosis may not be able to load the Diagnosis Startup Cache due to matching errors with an existing startup cache.

The tool provides matching errors for the following circumstances:

- The current flat model MD5 does not match what is stored in the cache. In this case, the tool produces the following error:

  ```
  // Error: The specified diagnosis startup cache does not include the
  current flat model. Please use "verify_pattern -update
  <startup_cache>" command to update the startup cache.
  ```

- The current pattern set MD5 does not match what is stored in the cache. In this case, the tool produces the following error:

  ```
  // Error: The specified diagnosis startup cache does not include the
  current pattern set(s). Please use "verify_pattern -update
  <startup_cache>" command to update the startup cache.
  ```

- The tool version does not match what was used to create the cache. In this case, the tool produces the following error:

  ```
  // Error: The specified diagnosis startup cache does not include the
  current version of tool. Please use "verify_pattern -update
  <startup_cache>" command to update the startup cache.
  ```

- X2B_is_on does not match what is stored in the cache. In this case, the tool produces the following error:

  ```
  // Error: The specified diagnosis startup cache does not include the
  current X2B flag. Please use "verify_pattern -update
  <startup_cache>" command to update the startup cache.
  ```

- The specified mask file's (if any) MD5 does not match what is stored in the cache. In this case, the tool produces the following error:

  ```
  // Error: The specified diagnosis startup cache does not include the
  current mask file(s). Please use "verify_pattern -update
  <startup_cache>" command to update the startup cache.
  ```

Diagnosis Startup Cache Usage Examples

In addition to creating a Diagnosis Startup Cache, you can update existing caches, use existing caches with Tessent Diagnosis server, and run pattern verification without creating a Diagnosis Startup Cache.

Example: Creating a Diagnosis Startup Cache

In the Tessent Diagnosis scan diagnosis point tool, the following command sequence loads the test patterns, performs the pattern verification, and creates a Diagnosis Startup Cache named *diagnosis_startup_cache*:

```
read_patterns pat1
read_patterns pat2 -append
verify_patterns –create_startup_cache ./diagnosis_startup_cache
exit
```

Subsequently, you re-invoke Tessent Diagnosis, load the patterns and newly-created Diagnosis Startup Cache (*diagnosis_startup_cache*), and perform diagnosis using the following command sequence:

```
read_patterns pat1
read_patterns pat2 -append
verify_patterns –load_startup_cache ./diagnosis_startup_cache
diagnose_failures log1
diagnose_failures log2
……
exit
```

Example: Updating an Existing Diagnosis Startup Cache

This example command sequence illustrates updating an existing Diagnosis Startup Cache (*diagnosis_startup_cache*) for future use by Tessent Diagnosis:

```
read_patterns pat1
read_patterns pat2 –append
read_patterns pat3 -append
verify_patterns –update_startup_cache ./diagnosis_startup_cache
exit
```

Subsequently, you re-invoke Tessent Diagnosis, load the patterns and updated Diagnosis Startup Cache (*diagnosis_startup_cache*), and perform diagnosis using the following command sequence:

```
read_patterns pat1
read_patterns pat2 –append
read_patterns pat3 -append
verify_patterns –load_startup_cache ./diagnosis_startup_cache
diagnose_failures log1
diagnose_failures log2
……
exit
```

Example: Using an Existing Diagnosis Startup Cache with the Tessent Diagnosis Server

The following example provides the command sequence you would perform to use an existing Diagnosis Startup Cache with the Tessent Diagnosis Server:

1. Create the Diagnosis Startup Cache (*diagnosis_startup_cache*) with the following commands using the Tessent Diagnosis scan diagnosis point tool:

   ```
   read_patterns pat1
   read_patterns pat2 -append
   verify_patterns –create_startup_cache ./diagnosis_startup_cache
   exit
   ```

2. In the Tessent Diagnosis Server, add the patterns and load the Diagnosis Startup Cache (*diagnosis_startup_cache*) using the following command sequence:

   ```
   add_pattern monitor1 pat1
   add_pattern monitor1 pat2 -append
   add_startup_cache monitor1 ./diagnosis_startup_cache
   ...
   ```

Example: Running Pattern Verification Without Creating a Diagnosis Startup Cache

In this example, Tessent Diagnosis does not create a Diagnosis Startup Cache or re-run pattern verification during diagnosis:

```
read_patterns pat1
read_patterns pat2 -append
```

```
verify_patterns –diagnosis          // assume pattern verfication passed
diagnose_failures log1              // does not re-run pattern verification
```

Turning Off Test Pattern Verification

By default, each time a read_patterns command is run, the test patterns are reverified before diagnosis. The verification consists of simulating the test patterns and comparing the captured values with the values expected by the test patterns.

Prerequisites

- Patterns that have been verified at least once.
- A startup cache created with the "verify_patterns -create_startup_cache" command.

Note

Test pattern verification is required for accurate diagnosis results. Use a startup cache to reduce run time. See verify_patterns in the *Tessent Shell Reference Manual* for details and examples of creating, updating, and loading a startup cache.

Siemens strongly recommends that you do not turn off pattern verification.

Procedure

Use the "verify_patterns -load_startup_cache" command to load a startup cache. This is part of the recommended method of using a startup cache to achieve run-time savings.

verify_patterns -load_startup_cache <*startup_cache_name*>

The following deprecated method is strongly discouraged:

- set_diagnosis_options -verify_patterns off

Note

You must not use the "set_diagnosis_options -verify_patterns off" command without verifying the patterns at least once. You should only use this command if you are confident that you fully understand its impact.

Displaying Test Pattern Mismatches

In the first step of the diagnosis, the Tessent Diagnosis tool simulates the design and verifies the test pattern accuracy. If the expected values do not match the simulated values, an error displays and the diagnosis aborts.

Prerequisites

- You have loaded the design netlist(s) and test patterns.

- You have performed the pattern verification.

Procedure

Use the report_failures command to display the test pattern mismatches. For example, the following command reports all pattern mismatches

```
report_failures -exact
```

Results

A display of the test pattern mismatches.

Guidelines for Preparing the ATE Failure File

Typical industry practice is to create scripts that convert the ATE failure files into the necessary format. Depending on the ATE, the specific format of the failure file varies, but there are usually four key pieces of information contained in the failure files.

ATE failure files normally contain the following information:

- Cycle each failure occurred
- Signal on which each failure occurred
- Expected value
- Actual value

Note

In the Tessent Shell environment, if you have used the patterns -scan_retargeting context to retarget core-level test patterns at the top level, refer to "Reverse Mapping Top-Level Failures to the Core." For information about scan pattern retargeting, refer to "Scan Pattern Retargeting."

ATE Failure File Format Requirements

You must convert the ATE failure file into one of two special ASCII formats compatible with Tessent Diagnosis.

Tessent Diagnosis supports the following ASCII formats:

- Cycle-based format — The cycle-based format is recommended. The cycle-based format is similar to the format of the ATE failure file, so it is the most accurate and easiest of the two formats to produce. The cycle-based format is only compatible with test pattern files saved in parallel WGL and STIL. See "The Cycle-Based Failure File" for more information.
- Pattern-based format — The pattern-based format is supported, but creating this type of failure file can be quite difficult and error-prone because failing cycles must be mapped to the test patterns. See "The Pattern-Based Failure File" for more information.

Chain Diagnosis Requirements

Chain diagnosis is a two-step process: identify the chain that fails and then identify which scan cells fail on that chain.

The following requirements apply to diagnosing chain failures:

- Include the fail data from the chain tests in the failure file. Tessent Diagnosis uses the fail data to automatically determine the faulty chains and their fault types. This is the preferred method.
    - The preferred method requires fail data from the chain tests and scan tests.
    - There is an alternative method if you only have fail data from the scan tests. You can specify the failing chains and fault types. For this case, your failure file must include the fail data from the scan tests only. It may not include any fail data from chain tests.
- Include the fail data from at least 100 failing scan test patterns to achieve good chain diagnosis resolution that locates the failing scan cell in the chain.

Tip

We recommend that you use 100 failing scan test patterns for optimal results.

You can estimate the required number of ATE test cycles to collect the scan test pattern failures by using the formula:

cycles = *patterns* * *length_longest_chain* * (*percentage_of_failing_cycles* / 100)

where:

- *patterns* — The number of failing scan chain patterns to use for data collection. We recommend 100 failing scan test patterns.
- *length_longest_chain* — The length of the longest scan chain in your design.
- *percentage_of_failing_cycles* — Percentage of cycles that cause a failure. For the stuck-at fault, the percentage is 50%.

For example, a chain has a stuck-at-zero fault. The length of the chain is 913, but the longest reported chain is 1195. Using the recommended 100 patterns:

cycles = 100 * 1195 * (50/100)

You should collect 59,750 test cycles on your ATE for the optimal result.

Logic Diagnosis Requirements

To run an accurate diagnosis on logic failures, chains must be functioning correctly. If chains are not functioning properly, it is unclear whether failures are due to defective chains or defective logic whose effect is captured in the chains.

Include failures from at least 30 failing scan test patterns to achieve good logic diagnosis resolution. It is recommended that you use 100 failing scan test patterns for optimal results.

The Cycle-Based Failure File

The cycle-based failure file format can only be used with test patterns saved in parallel WGL or STIL format.

Note
If you plan to use Tessent YieldInsight, your ATE failure files must contain certain keywords in the tracking_info section. See "Requirements for ADB Generation" in the *Tessent YieldInsight User's Manual* for details.

Use the keywords as described in Table 1-3 to create a cycle-based failure file. Precede comment text with a pair of forward slashes (//).

Table 1-3. Cycle-Based Formatting Keywords

Keyword (s)	**Usage Rules**
format {cycle | pattern}	Optional. Use this keyword on the first non-comment line of the failure file to identify the file format. Depending on the application, use one of the following arguments: • cycle — Indicates the failure file uses a cycle-based format. • pattern — Indicates the failure file uses a pattern-based format. By default, pattern-based is assumed. Use only one format keyword per failure file.
tracking_info_begin <user_defined_text> tracking_info_end	Optional. Use these keywords to place user-defined tracking information in a failure file. If you plan to use Tessent YieldInsight, you must add specific keywords in this section that propagate to the ADB. See "Requirements for ADB Generation" in the *Tessent YieldInsight User's Manual* for a complete list of supported keywords. Use the following rules when entering tracking information: • Use the keywords only once to create a single tracking information section for the entire file. • Place the keywords anywhere in the file except within the failing cycle (failures_begin and failures_end keywords) information. This information is not used for diagnosis; it is placed in the diagnosis report verbatim. **Note:** For hierarchical designs, after remapping, the tracking information contains fields with "core_", such as "core_name core1." The tool uses these fields to detect the remapping, and they must not be modified or removed. The "core_instance" pathname is a reflection of the logical hierarchy in the TCD.

Table 1-3. Cycle-Based Formatting Keywords (cont.)

Keyword (s)	**Usage Rules**
set_expected_z_handling <value>	Optional. Use this keyword to identify how the ATE handles the high impedance (Z) values on tri-state nets. Specify one of the following values: • X (unknown) • H (logic 1) • L (logic 0) • Z (high impedance) Use only one set_expected_z_handling keyword per failure file. By default, Z values are handled as an unknown (X) state. See "High Impedance (Z) Handling."
test_suite_begin <test_suite_id> test_suite_end	Optional. Use these keywords to list the fail.log data produced when multiple test sets are used for testing, and the ATE produces a separate *failure.log* for each test set. See "Multiple Test Suite Failure Data." By default, a single test pattern set is assumed.

last_cycle_applied <cycle_number>	Optional. Use this keyword to specify the last cycle applied to the Design-Under-Test (DUT). By default, the last cycle in the test set is assumed to be the last cycle applied to the DUT. If a last cycle is specified, all cycles past the specified cycle are masked out.
	If used, you must put this keyword outside the failures_begin and failures_end keywords. For example:
	`failures_begin`
	`.....`
	`failures_end`
	`last_cycle_applied 200`

Table 1-3. Cycle-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules
failures_begin <failing_cycle_number> <failing_pin_name> <expected_value> <faulty_value> failures_end	Required. Use one set of these keywords to enter all the failing cycle data from a test set. Place the failing data between these keywords on separate lines. Enter the cycle data for all failing patterns up to and including the last failing pattern. Place all data for a single failing cycle on one line, in the following order, from left to right: • failing_cycle_number — This is the number that identifies the cycle. Cycle zero must correspond to cycle zero in the scan test. If the tester applies the scan test set after other tests, then you must deduct the offset (corresponding to the total cycles of the test sets preceding the scan test set) from the cycle number in the ATE failure file to obtain the correct cycle number. Cycles can be listed in any order. • failing_pin_name — This is either a primary output pin or a scan chain/channel output pin associated with the specified cycle. The ATE failure file typically lists the load board pin names. You must convert the load board pin names into primary output or scan chain/channel output pin names. • expected_value — This is the expected test result for the cycle. Use one of the following characters to indicate the expected test result value: --H (logic 1) --L (logic 0) --Z (high impedance) The expected values entered here are compared with the expected values in the test patterns to check the accuracy of the converted failure file. Use Z if the ATE differentiates a Z value from an H or L value. • faulty_value — This is the actual test result for the cycle. Use one of the following characters to indicate the actual test result value: --H (logic 1) --L (logic 0) --Z (high impedance) Use Z if the ATE differentiates a Z value from an H or L value.

Table 1-3. Cycle-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules
failure_buffer_limit_reached {none | last_cycle_logged | <pin_name> | all | unknown}	Optional. Use this keyword to indicate the pins that reach their failure buffer limit during testing. If the current test suite (or the fail file if there are no test suites) contains fail bits and no failure buffer limit keyword applies to the test suite, then the "unknown" switch is assumed. If no fail bits are present, the "none" switch is assumed. See "Failure Truncation Handling" on page 65. • none — Indicates all pins are within buffer limits and no cycle adjustments are necessary. • last_cycle_logged — Indicates the buffer limit was reached on the last/highest failing cycle in the test set. • pin_name — Specifies the name of an individual pin that reaches the failure buffer limit. If necessary, use the failure_buffer_limit_reached keyword multiple times to specify additional pins. • all — Specifies that buffer limits were reached by all failing pins. This option only applies to per-pin ATE fail buffers. • unknown — Specifies using the safest way to handle ATE log truncations. Equivalent to specifying the following: failure_buffer_limit_reached all last_cycle_applied <last_failed_cycle_number> Except when specifying multiple single pin names, use the failure_buffer_limit_reached keyword only once per test suite. An FBLR on a different test suite does not change the handling for the current test suite.
total_cycles <number_of_cycles>	Optional. Use this keyword to specify the total number of cycles in each test suite. When provided, the number is used to perform a data consistency check before diagnosis. If the total number of cycles in the test pattern file supplied for diagnosis does not match this value, an error message displays and diagnosis aborts.
failure_file_end	Optional. Use this keyword to indicate the end of the failure file.

Cycle-Based Failure File Examples

A cycle-based failure file contains one fail per row and the information about each fail follows a four-column format. Optional keywords can provide additional information to generate the diagnosis results and enable yield learning.

Basic Cycle-Based Failure File

The following example is a basic cycle-based failure file. The columns of data that describe each fail are summarized below.

- **Cycle** — the failing cycle number
- **Pin_Name** — the failing pin name
- **Exp** — the expected value for that cycle and pin
- **Act** — the actual value from the ATE

```
// Cycle Pin_Name Exp Act
   13450  pin100   H   L
   15345  pin140   L   H
   15900  pin130   H   L
   15900  pin140   L   H
   17120  pin130   H   L
   19201  pin130   L   H
   19320  pin130   H   L
```

Keywords in a Cycle-Based Failure File

Tessent Diagnosis can process optional information about the ATE to improve diagnosis results. Use optional keywords to provide this information. Tessent Diagnosis also uses keywords to pass tracking information to yield learning tools such as Tessent YieldInsight. Cycle-based failure files can contain different keywords depending on the use case.

The following example comes from a yield learning exercise that includes three split lots from a foundry and two wafers from each lot to analyze. Tessent YieldInsight is a tool in the analysis flow. The example correlates information from the datalog of the ATE test program to the cycle-based failure file. The following information is pertinent to the example:

- The design name is SL9010A.
- The initial failing die is on wafer number 3 from lot number RA9226.00.
- The ATE handles the high impedence state as a logic 1, or H.
- The pattern set has 60150 total cycles.
- The ATE uses a centralized memory architecture that has a fail memory buffer to collect the failures of all pins.
- The ATE fail memory buffer is large enough to capture all the failures.

- The following lines from the test program datalog contain all the scan fails to diagnose.

```
Failing cycle 13450 pin pin100 was L expected H
Failing cycle 15345 pin pin140 was H expected L
Failing cycle 15900 pin pin130 was L expected H
Failing cycle 15900 pin pin140 was H expected L
Failing cycle 17120 pin pin130 was L expected H
Failing cycle 19201 pin pin130 was H expected L
Failing cycle 19320 pin pin130 was L expected H
```

The following cycle-based failure file conveys all this information to the tool using some optional keywords, shown in red. It presents the scan fails in the four-column format.

```
// cycle-based failure file
format cycle
set_expected_z_handling H
tracking_info_begin
lot_id RA9226.00
wafer_id RA9226-03
design_id SL9010A
tracking_info_end
failures_begin
13450 pin100 H L
15345 pin140 L H
15900 pin130 H L
15900 pin140 L H
17120 pin130 H L
19201 pin130 L H
19320 pin130 H L
failures_end
total_cycles 60150
failure_file_end
```

Last Cycle Applied

This example shows the use of the last_cycle_applied keyword in red. All the information of the previous example remains in effect except the last cycle applied by the ATE is cycle 20000. This communicates to the tool that pass and fail information to consider for diagnosis ends at cycle 20000 (that is, the tool does not consider cycle 20001 through the final cycle 60150 as passing cycles for diagnosis).

```
// cycle-based failure file
format cycle
set_expected_z_handling H
tracking_info_begin
lot_id RA9226.00
wafer_id RA9226-03
design_id SL9010A
tracking_info_end
failures_begin
13450 pin100 H L
15345 pin140 L H
15900 pin130 H L
15900 pin140 L H
17120 pin130 H L
19201 pin130 L H
19320 pin130 H L
failures_end
last_cycle_applied 20000
total_cycles 60150
failure_file_end
```

Full Failure Buffer Limit

This example shows the use of the failure_buffer_limit_reached keyword in red. All the information of the previous examples remain in effect except the ATE applies all 60150 cycles. However, the fail memory buffer of the ATE fills up during cycle 19320 because there are more scan fails in the test program's datalog. For brevity, this example uses "…" to represent those additional failing cycles and pins.

```
…
Failing cycle 13450 pin pin100 was L expected H
…
Failing cycle 15345 pin pin140 was H expected L
…
Failing cycle 15900 pin pin130 was L expected H
…
Failing cycle 15900 pin pin140 was H expected L
…
Failing cycle 17120 pin pin130 was L expected H
…
Failing cycle 19201 pin pin130 was H expected L
…
Failing cycle 19320 pin pin130 was L expected H
```

The following cycle-based failure file conveys this information to the tool.

```
// cycle-based failure file
format cycle
set_expected_z_handling H
tracking_info_begin
lot_id RA9226.00
wafer_id RA9226-03
design_id SL9010A
tracking_info_end
failures_begin
…
13450 pin100 H L
…
15345 pin140 L H
…
15900 pin130 H L
…
15900 pin140 L H
…
17120 pin130 H L
…
19201 pin130 L H
…
19320 pin130 H L
failures_end
failure_buffer_limit_reached last_cycle_logged
total_cycles 60150
failure_file_end
```

Multiple Test Suites

The following example shows how to format the failure data of multiple test suites in a failure file for diagnosis.

```
test_suite_begin suite_1    // suite_1 contains all failures in fail_log_1
failures_begin
......
failure_buffer_limit_reached none
failures_end
test_suite_end

test_suite_begin suite_2   // suite_2 not tested on ATE, no failure log
failures_begin
......
failures_end
last_cycle_applied -1       // -1 indicates suite not tested
test_suite_end

test_suite_begin suite_3    // suite_3 contains all failures in fail_log_3
failures_begin
......
failure_buffer_limit_reached all
failures_end
test_suite_end

test_suite_begin suite_4    // fail_log_4 has no failures so suite_4 is
                            // empty
failures_begin
failure_buffer_limit_reached none
failures_end

test_suite_end
```

Here is another example:

```
format cycle

test_suite_begin scanpat_suite1
failures_begin
100 pin1 H L
150 pin1 Z L
100 pin2 L H
failures_end
last_cycle_applied 499
total_cycles 500
test_suite_end

test_suite_begin scanpat_suite2
failures_begin
10 pin1 H L
15 pin1 Z L
8 pin2 L H
failures_end
last_cycle_applied 299
total_cycles 300
test_suite_end
failure_file_end
```

Multiple Test Suites with Full Fail Buffer Limit

The following example shows how the failure_buffer_limit_reached keyword displays when you have multiple suites.

```
format cycle

test_suite_begin scanpat_suite1
failures_begin
100 pin1 H L
100 pin2 L H
failure_buffer_limit_reached last_cycle_logged
failures_end
last_cycle_applied 499
total_cycles 500
test_suite_end

test_suite_begin scanpat_suite2
failures_begin
10 pin1 H L
8 pin2 L H
failure_buffer_limit_reached last_cycle_logged
failures_end
last_cycle_applied 299
total_cycles 300
test_suite_end
```

High Impedance (Z) Handling

Some ATEs measure high impedance (Z) and some do not. Even if a tester can measure the Z value, it may not be enabled. When the Z measuring capability is off or unavailable, all expected Z values in the test pattern file are converted to unknown (X) values during the test program creation. The ATE may perform this conversion dynamically without changing the test pattern file.

For an effective diagnosis, you must have accurate information on how the ATE handles Z values. For example, if the ATE enables the Z measuring capability and no fail cycles/bits appear with expected Z values, then all cycles expecting Z values pass. If the ATE turns off the Z measuring capability, then it masks the cycles containing Z values and they never fail.

You must determine from the ATE equipment or operator how the Z values are handled during testing and specify this behavior for diagnosis. Use the set_expected_z_handling keyword in the failure file to specify the Z handling for diagnosis.

By default, Z values are handled as an unknown (X) state. For information on changing the default Z handling behavior for Tessent Diagnosis, see the set_z_handling command.

Multiple Test Suite Failure Data

In ATPG, you can break test patterns into multiple pattern files for use on the ATE.

The following example splits the test patterns into two patterns files. The first file contains patterns 0 to 999, and the second file contains patterns 1000 to 1999:

create_patterns

write_patterns tst_sta_0_999.stil -stil -begin 0 -end 999

write_patterns tst_sta_1000_1999.stil -stil -begin 1000 -end 1999

If you load the two pattern files on the ATE, the ATE can generate two failure files that correspond to the two test patterns. If you want to run diagnosis with these two test pattern files in a single diagnosis run, then you must merge the failure files using test_suite_begin keyword in the failure file.

When a test pattern set is broken into multiple test suites for testing, the ATE produces a separate *fail.log* for each test suite. Each *fail.log* should be represented separately in the failure file loaded into Tessent Diagnosis—see also "Setting Up the Tessent Diagnosis Server."

Use the following rules to format the multiple test suite data in a cycle-based or pattern-based failure file:

- List the failing data between the failures_begin/failures_end keywords.

  List the failing data for a test suite as follows:

  ```
  test_suite_begin suite_1
  failures_begin
  ......
  failures_end
  test_suite_end
  ```

- Document each fail.log.

  Specify the multiple test suites as follows:

  ```
  test_suite_begin suite_1
  failures_begin
  ......
  failure_buffer_limit_reached none
  failures_end
  test_suite_end
  ```

  - Even if there are no failures in a test suite or a test suite is omitted from testing, the test suite must still be represented in the failure file as follows:
  - Use the following placeholder for each test suite that is omitted from testing:

    ```
    test_suite_begin <suite_name>
    failures_begin
    ......
    failures_end
    last_cycle_applied -1
    test_suite_end
    ```

Use the following placeholder for each test suite that contains no failures:

```
test_suite_begin <suite_name>
failure_buffer_limit_reached none
test_suite_end
```

- Create one failure file for the entire test set.

  Document each *fail.log* produced for each test suite inside one failure file.

- You must list test suites within the failure file in the same order that you load the test pattern sets on the ATE.

  Each test suite is independent from other test suites, in that the cycle count or pattern count for each test suite corresponds to the pattern set order you loaded on the ATE. For example, the first pattern set read in corresponds to the first test suite, the second pattern set read in corresponds to the second test suite, and so on. You must load the corresponding test pattern sets into Tessent Diagnosis with the read_patterns -append command.

  Additionally, the failure_buffer_limit_reached settings do not carry from one test suite to another.

Cycle Offset Adjustment for Failure Files

When you apply a pattern set on the ATE, extra cycles may be introduced, causing cycle offset issues in the failure file. These issues can cause verification to fail and the diagnosis to abort.

To adjust the cycle offset for a given failure file, enter the following command:

set_diagnosis_options -cycle_offset *offset*

When you specify this option, Tessent Diagnosis adjusts all failing cycles by the integer value offset and uses the adjusted cycles for failure verification and diagnosis.

For hierarchical ATPG flows, this option is available in the patterns -failure_mapping context through the set_failure_mapping_options command. In this context, the option ensures that the tool correctly translates the failure cycles to the core level during reverse mapping of the top-level failures to the core. Refer to "Reverse Mapping Top-Level Failures to the Core" for details.

As shown in the following example, you can create a TCL procedure to automatically identify the cycle offset by sweeping a range of user-defined cycle offset values.

```
// flog: failure filename
// cycle_offset_lb: cycle offset lower bound
// cycle_offset_ub: cycle offset upper bound
// return: 0, if one valid cycle offset is found; 1, if no valid cycle
// offset is found; 2, if more than one valid cycle offset is found

proc findValidCycleOffset { flog cycle_offset_lb cycle_offset_ub } {
   set result 0;  // return value
   set cycle_offset_list { } ;
   set verbose 0 ;     // print out detailed results
   set cycle_offset_identified 0;      // true if a single valid offset is found

 // Scan the offset range to find valid offset value(s)
 if { $verbose == 0 }  { set_screen_display off }
 for { set offset $cycle_offset_lb } { $offset <= $cycle_offset_ub } { incr offset } {
      puts "*** Try cycle offset ( $offset ) with failure file ( $flog ) ***"
         set_diagnosis_options -cycle_offset $offset

         if { [ catch { read_failure $flog } res ]  } {
            puts "Found one invalid cycle offset ( $offset ), result ( $res )" ;
         } else {
              puts "Found one possible valid cycle offset ( $offset )" ;
              lappend cycle_offset_list $offset;
         }
   }

   // Report the valid cycle offset identified
   set_screen_display on
      set num_valid_offset [ llength $cycle_offset_list ];
   if { $num_valid_offset == 0 } {
      puts "//  Error: Failed to identify any valid cycle offset within \[$cycle_offset_lb,
$cycle_offset_ub\]"
         set result 1;
   } else {
      if { $num_valid_offset == 1 } {
         puts "// Found $num_valid_offset valid cycle offset within \[$cycle_offset_lb,
$cycle_offset_ub\] : $cycle_offset_list";

         set cycle_offset_identified 1;
         set final_offset [ lindex $cycle_offset_list 0 ];
         set_diagnosis_options -cycle_offset $final_offset;
         puts "// Set diagnosis option '-cycle_offset' to value ( $final_offset )"
            //read_failure $flog
      } else {
         puts "// Found $num_valid_offset valid cycle offsets within \[$cycle_offset_lb,
$cycle_offset_ub\] : $cycle_offset_list";
         puts "// Further investigation is needed to determine which offset value should be
used." ;
         puts "// Hint: Another failure file with more failure data may help identify the
right offset value." ;
         set result 2;
      }
   }

   return $result;
}
```

The following example illustrates how to use the TCL procedure.

```
// For a given failure file, identify its cycle offset first,
// then run diagnosis
set min_offset -1;
set max_offset 1;
set flog_list { ./flog1 ./flog2 ./flog3 }

foreach f $flog_list {
   if { [ findValidCycleOffset $f $min_offset $max_offset ] == 0 } {
      diagnose_failure $f ;
   } else {
      puts "//  Error: Can't diagnose '$f' without a valid cycle offset.";
   }
}
```

The Pattern-Based Failure File

By default, Tessent Diagnosis generates pattern-based failure files. As with cycle-based failure files, you specify formatting keywords.

Note

If you plan to use Tessent YieldInsight, your ATE failure files must contain certain keywords in the tracking_info section. See "Requirements for ADB Generation" in the *Tessent YieldInsight User's Manual* for details.

Use the keywords as described in Table 1-4 to create a pattern-based failure file. Precede comment text with a pair of forward slashes (//).

Table 1-4. Pattern-Based Formatting Keywords

Keyword (s)	**Usage Rules**
format {cycle | pattern}	Optional. Use this keyword on the first non-comment line of the failure file to identify the file format. Depending on the application, use one of the following arguments: • cycle — Indicates the failure file uses cycle-based format. • pattern — Indicates the failure file uses pattern-based format. By default, pattern-based is assumed. Use only one format keyword per failure file.
tracking_info_begin <user_defined_text> tracking_info_end	Optional. Use these keywords to place user-defined tracking information in a failure file. If you plan to use Tessent YieldInsight, you must use specific keywords in this section that propagate to the ADB. See "Requirements for ADB Generation" in the *Tessent YieldInsight User's Manual* for a complete list of supported keywords. Use the following rules when entering tracking information: • Use the keywords only once to create a single tracking information section for the entire file. • Place the keywords anywhere in the file except within the failing bit (scan test and chain test information) information. This information is not used for diagnosis; it is placed in the diagnosis report verbatim. **Note:** For hierarchical designs, after remapping, the tracking information contains fields with "core_", such as "core_name core1." The tool uses these fields to detect the remapping, and they must not be modified or removed.

Table 1-4. Pattern-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules
failures_begin <failure_data> failures_end	Required for pattern-based multiple test suites, otherwise omit. Use one set of these keywords to enter all the failing data from a single test set. Place the failing data between these keywords on separate lines. See "Multiple Test Suite Failure Data."
set_expected_z_handling <value>	Optional. Use this keyword to identify how the ATE handles the impedance (Z) values on tri-state nets. Specify one of the following values: • X (unknown) • H (logic 1) • L (logic 0) • Z (high impedance) Use only one set_expected_z_handling keyword per failure file. By default, Z values are handled as an unknown (X) state. See "High Impedance (Z) Handling."
test_suite_begin <test_suite_id> test_suite_end	Optional. Use these keywords to list the fail.log data produced when multiple test sets are used for testing, and the ATE produces a separate fail.log for each test set. See "Multiple Test Suite Failure Data." By default, a single test set is assumed.

Table 1-4. Pattern-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules
chain test scan test	Required when there is both scan test data and chain test data in the failure file. Use these keywords to enter the failing pattern data. If the failure file contains only scan test data, you do not need to use these keywords. Omit if using an empty test suite. Use the keywords as follows: • Place all chain test data before scan test data. • Place keyword within the failures_begin/failures_end block. • Place the keyword on the line immediately before the failure data it identifies. • Enter the patterns in ascending order, by number. • Enter the failing cycle information for all patterns up to and including the last failing pattern. • Enter all data for a single failing cycle on one line, in the following order, from left to right: • Pattern number — Number of the failing pattern. • Primary output — Scan channel name or primary output. • Cycle — Number that identifies the failing cell within a scan chain. The cycle number is required when the primary output is a scan chain name. The cycle number always starts from 0, which is closest to the scan out pin. • Expected_value — Character that represents the expected test result for the cycle as follows: --H (logic 1) --L (logic 0) --Z (high impedance) The expected values listed here are compared with the expected values in the test patterns to check the accuracy of the converted failure file. Use Z if the ATE differentiates a Z value from an H or L value. • Faulty_value — Character that represents the actual/ faulty value as follows: --H (logic 1) --L (logic 0) --Z (high impedance) Use Z if the ATE differentiates a Z value from an H or L value.

Table 1-4. Pattern-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules
failure_buffer_limit_reached {none | last_pattern_logged | <pin_name> | all | unknown}	Optional. Use this keyword to indicate the pins that reach their failure buffer limit during testing. If the current test suite (or the fail file if there are no test suites) contains fail bits and no failure buffer limit keyword applies to the test suite, then the "unknown" switch is assumed. If no fail bits are present, the "none" switch is assumed. See "Failure Truncation Handling" on page 65. • none — Indicates all pins are within buffer limits and no pattern adjustments are necessary. • last_pattern_logged — Indicates the buffer limit was reached on the last/highest failing pattern in the test set. • pin_name — Specifies the name of an individual pin that reaches the failure buffer limit. If necessary, use the failure_buffer_limit_reached keyword multiple times to specify additional pins. • all — Specifies that buffer limits were reached by all failing pins. This option only applies to per-pin ATE fail buffers. • unknown — Specifies using the safest way to handle ATE log truncations. Equivalent to specifying the following: last_pattern_applied <last_failed_pattern_number> failure_buffer_limit_reached all Except when specifying multiple single pin names, use the failure_buffer_limit_reached keyword only once per test suite. An FBLR on a different test suite does not change the handling for the current test suite.

Table 1-4. Pattern-Based Formatting Keywords (cont.)

Keyword (s)	Usage Rules

last_pattern_applied <test_pattern_number>	Optional. Use this keyword to identify the last pattern applied to the Design-Under-Test (DUT). By default, the last pattern in the test set is assumed to be the last pattern applied to the DUT. If a last pattern is specified, all patterns past the specified pattern are masked out.
	
	You can specify one of these keywords for both, the scan test section and chain test section.
	
	The specified number must be a positive integer, equal to or less than the total number of test patterns, and not less than any failing pattern number. For example, if the ATE applied 100 patterns, the last pattern applied would be pattern 99 (patterns are numbered starting at 0 for the first pattern) and the failure file entry would be:
	
	`last_pattern_applied 99`
	
	Due to a full failure buffer or a truncated test, the tester may stop before applying all test patterns. Specifying the last pattern applied prevents unapplied test patterns from being interpreted as passing.
	
	If used, you must enclose this keyword with the failures_begin and failures_end keywords. For example:
	
	`failures_begin`
	`last_pattern_applied -1`
	`failures_end`

failure_file_end	Optional. Use this keyword to indicate the end of the failure file.
allow_missing_expected_actual_values	Optional. This is a reverse mapping process generated keyword for SSN designs that could appear in the tracking_info section for reverse-mapped failure files. This keyword instructs the tool to inhibit the failure file verification for patterns/cycles that are missing expected and actual values in the failure file.

Pattern-Based Failure File Examples

The information contained in the pattern-based failure file depends on the formatting keywords you choose.

Suppose you have the following snippet from an ATE failure log:

```
 9500 pin99L L H
10000 pin100 L H
13450 pin100 H L
15345 pin140 L H
15900 pin130 H L
15900 pin140 L H
17120 pin130 H L
19201 pin130 L H
19320 pin130 H L
```

The pattern-based failure file format is:

```
// pattern-based failure file
format pattern
tracking_info_begin
lot_id 12345
wafer_id 98765
chip_id 54820
x_coord 72
y_coord 57
tracking_info_end
set_expected_z_handling H
failures_begin
chain test
0 chain2 0 L H
0 chain3 1 L H
failures_end
last_pattern_applied 1
failures_begin
scan test
75 chain3 10 H L
90 chain20 30 L H
90 chain19 100 H L
90 chain20 100 L H
100 chain19 50 H L
125 chain19 55 L H
125 chain19 200 H L
failures_end
last_pattern_applied 266
failure_file_end
```

The following example shows how multiple test suite failure data is formatted in the chain failure file for diagnosis.

```
tracking_info_begin

 defect_info_begin
   injected_fault     STUCK_AT_0    pm7202_core_inst/reg_9_inst/Q
   faulty_chain       chain23
   triggering_prob    100
   cell_id            218
 defect_info_end

tracking_info_end


// pattern_id  chain/PO_name  cell_number  expected_value  simulated_value
// cell_path_name

test_suite_begin  SUITE_1
failures_begin
chain test
0 chain23 2 H L // pm7202_core_inst/reg_93_inst
0 chain23 3 H L // pm7202_core_inst/reg_92_inst
...
last_pattern_tested 13
failures_end
test_suite_end
test_suite_begin SUITE_2
failures_begin
scan test
0 chain22 1197 H L // pm7202_core_inst/reg_11_inst
0 chain22 1198 H L // pm7202_core_inst/reg_10_inst
...last_pattern_tested 5
failures_end
test_suite_end
```

Guidelines for Mapping ATE Failure Logs to Pattern-Based Failure Files

There are several challenges in accurately mapping the ATE failure log data to the pattern-based failure file format. To ensure accurate diagnostics, you need to take special care to map the data correctly. The following sections describe situations that need special consideration when mapping the ATE failure log to a pattern-based failure file:

Failures During Scan Chain Unloading

If scan cell failures are detected during unloading of the scan chains, the failure file requires the names of the chains involved. For example, if a chain is named “chain1” and its primary input and output pins are named “si_1” and “so_1”, the failure file requires the “chain1” name, not the pin names.

For STIL test patterns, use the scan chain name in the ScanStructures block.

For example, in the following STIL excerpt, the chain name corresponding to the “si_1” and “so_1” pins is “chain1” (the names are highlighted in bold for clarity):

```
ScanStructures {
      ScanChain chain1 {
      ScanLength 26;
      ScanInversion 1;
      ScanCells "\portb_reg[7] " ! "\portb_reg[6] " ! "\portb_reg[5]
" ! "\portb_reg[4 ] " ! "\portb_reg[3] " ! "\portb_reg[2] " !
"\portb_reg[1] " ! "\portb_reg[0] " ! "\portc_reg[7] " !
"\portc_reg[6]" ! "\portc_reg[5] " ! "\portc_reg[4] " !
"\portc_reg[3] " ! "\portc_reg[2] " ! "\portc_reg[1] " !
"\portc_reg[0] " ! "\phase_reg[3] " ! "\phase_reg[0]
" ! "regs.†out_reg[7] " ! "regs.†out_reg[6] " ! "regs.†out_reg[5]
" !  "regs.†out_reg[4] " ! "regs.†out_reg[3] " ! "regs.†out_reg[2]
" ! "regs.†out_reg[1] " ! "reg s.†out_reg[0] " ! ;
      ScanIn "si_1";
      ScanOut "so_1";
      ScanMasterClock "clk";
   }
}
```

Failures During Scan Chain Unloading (Compressed Patterns)

Tessent TestKompress (EDT On) outputs a different format when a test fails during scan chain unloading. In addition to the cycle the failure was detected on during the unload process, you must provide the name of the external scan channel where the failure occurred as follows:

- Use “edt_channel” followed by the index number of the external scan channel. For example, if the channel pins are named “my_edt_input7” and “my_last_edt_output”, the correct name for the external scan channel is “edt_channel7”, assuming the index number is accurate.
- Verify the index number of the external scan channel reflected in the pin names with the report_edt_pins command. The pin description column of the report lists the index number for each channel.

In STIL patterns, you can obtain the channel name from the ScanStructures block. For example, in the following STIL excerpt, the channel name corresponding to the “my_edt_input7” and “my_last_edt_output” pins is “edt_channel7” (the names are highlighted in bold for clarity):

```
ScanStructures {
   ScanChain edt_channel7 {
      ScanLength 36;
      ScanInversion 0;
      ScanCells "edt_channel1/cell_35" "edt_channel1/cell_34"
"edt_channel1/cell_33" "edt_channel1/cell_32"
"edt_channel1/cell_31" "edt_channel1/cell_30"
"edt_channel1/cell_29" "edt_channel1/cell_28"
...
"edt_channel1/cell_1" "edt_channel1/cell_0" ;
   ScanIn "my_edt_input7";
   ScanOut "my_last_edt_output";
   ScanMasterClock "clk" "edt_clock";
   }
}
```

Multiple Test Pattern Types in a Set

Test pattern sets typically contain several types of test patterns arranged in a particular order. The number of cycles in a given test pattern varies depending on the pattern type. Use the information in the following sections of the *Tessent Scan and ATPG User’s Manual* to determine the correct cycle for a given test pattern:

- Basic
- Clock Sequential
- Multiple Load

You should work with the pattern designer in order to understand the types and arrangement of the patterns in a given pattern set. For an introduction to the different Tessent FastScan/Tessent TestKompress pattern types, refer to “Tessent FastScan Pattern Types” in the *TessentScan and ATPG User’s Manual*.

Reordered Patterns

Patterns are often saved in multiple pattern sets. For example, pattern set #1 might contain patterns 0 to 999, pattern set #2 might contain patterns 1000 to 1999, and so on. When applied on the tester, the pattern sets might be reordered (pattern set #2 applied first followed by pattern set #1 for example). If patterns are reordered, you need to verify and compensate, if necessary, in the following situations:

- Test_setup procedure

  When the patterns are saved in different sets, be aware the tool may or may not include the test_setup procedure in each individual pattern set, depending on the settings used when the patterns were saved.

- Pattern order

  Although the pattern sets are applied in a different order on the tester, the patterns should be numbered starting from 0 in the pattern-based failure file. In the preceding example, pattern 0 in the pattern-based failure file should correspond to pattern 1000 of pattern set #2.

  Tip

  The recommended practice is to load patterns into Tessent Diagnosis in the same order the patterns were applied on the tester. This helps prevent conversion errors due to incorrect determination of pattern boundaries.

- Overlap cycles

  In pattern set #1, which is applied later on the tester, be aware that the initial load_unload cycles in the first pattern load the scan chains but do not result in actual comparison for the values that are simultaneously shifted out. Comparison only happens in the load_unload cycles that occur after capture.

Last Pattern Tested not at Pattern Boundary

In some cases, the ATE may only capture a certain number of failing cycles, and the end point may not be at the boundary of the pattern. For example, in a STIL pattern, if the failure happened to be a bit in the pattern 2 macro, the actual last pattern tested should be pattern 1 because the scan unloading of pattern 1 happens at the scan chain loading of pattern 2.

Failure Truncation Handling

For cycle-based (and pattern-based) failure logs, the Tessent Diagnosis failure file format enables you to specify failures captured in a variety of ways that reflect various tester configurations, including support for different ways in which failure data may be truncated. In the failure logs, the failure_buffer_limit syntax reflects any truncations in the logging of failing cycles.

Note

The tool can detect invalid fail file annotations that may lead to lower diagnosis resolution or inaccurate diagnosis. See "diagnose_failures" in the *Tessent Shell Reference Manual* for more details and limitations.

Two methods of failure logging are typically used by ATE, pin-based and central buffer.

- Pin-based — In pin-based failure logging, each pin has its own failure buffer. This means each pin can reach the buffer limit independent of other pins.
- Central-buffer — In central-buffer failure logging, there is a common failure buffer memory where every cycle/pattern stores at least one failing cell. Typically, when failures are reported externally, only the failing cell information is reported.

By default, Tessent Diagnosis stops observing a failure buffer when it becomes full. Anything after the last failing bit is truncated (therefore, unknown). Consequently, Tessent Diagnosis assumes:

- All pass/fail information on failing pins beyond their last failing cycle is not available.
- All pass/fail information on passing pins beyond the last cycle where any pins failed is not available.

For example:

- Pin A:failing pin, last failing cycle is 200 — pass/fail information beyond cycle 200 is unavailable for pin A.
- Pin B:failing pin, last failing cycle is 400 — pass/fail information beyond cycle 400 is unavailable for pin B.
- Pin C:passing pin — pass/fail information beyond cycle 400 is unavailable, but cycles 0 - 400 on pin C passed.
- Pin D:passing pin — pass/fail information beyond cycle 400 is unavailable, but cycles 0 - 400 on pin D passed.

To increase the accuracy of the diagnosis, specify the failure buffers that become full for either central-buffer or per-pin failure logging architectures.

Note

If you do not specify a failure truncation annotation, the tool defaults to "failure_buffer_limit_reached unknown" and prints a warning similar to the following:

```
Warning: Failure truncation was not specified in failure file
'fail_log_name'. Using default failure truncation handling
'failure_buffer_limit_reached unknown' which could impact diagnosis
resolution. Please use appropriate truncation annotation in the
failure file.
```

Specify last_cycle_applied, which indicates a global buffer filling in addition to any local buffers. Use one of the following arguments with the failure_buffer_limit_reached (FBLR) keyword to refine the failure full buffer status:

- none — Indicates all pins are within buffer limits and no cycle/pattern adjustments are necessary. When the FBLR keyword is not specified, the tool assumes the None condition.
- pin_name — For per-pin failure logging, specifies the names of individual pins that reach their failure buffer limit on the last failing cycle/pattern of the pin. All pass/fail information beyond the last failing cycle/pattern is unavailable.
- all — For per-pin failure logging, specifies that all failing pins reached their failure buffer limit on the last failing cycle/pattern. If there are no failures in a test suite, then the tool assumes truncation on all pins and the status is unknown for all pins. If there are failures in the test suite, the pins that are failing are considered truncated after the last failure, and the status is unknown after the last failure. The non-failing pins are considered passing (no truncation).
- last_cycle_logged — For central-buffer failure logging and cycle-based failure file format, specifies that the central buffer reached its limit at the last failing cycle. All pass/fail information beyond the last failing cycle on all pins is unavailable.
- last_pattern_logged — For central-buffer failure logging and pattern-based failure file format, specifies that the central buffer reached its limit at the last failing pattern. All pass/fail information beyond the last failing pattern on all pins is unavailable.
- unknown — For central-buffer failure logging and per-pin failure logging, specifies the safest most conservative way to handle truncations. Status is unknown unless a test suite has no failing bits, in which case it is pass.

For multi-suite scenarios, each suite is treated independently. The tool does not carry over any truncation information from one test suite to another. Therefore, if the failure buffer limit is reached in a test suite and the buffer remains full for the next test suite, the failure buffer limit reached keyword(s) must be specified again for the next suite. If this is not done, the tool assumes that the failure buffer was reset between test suites.

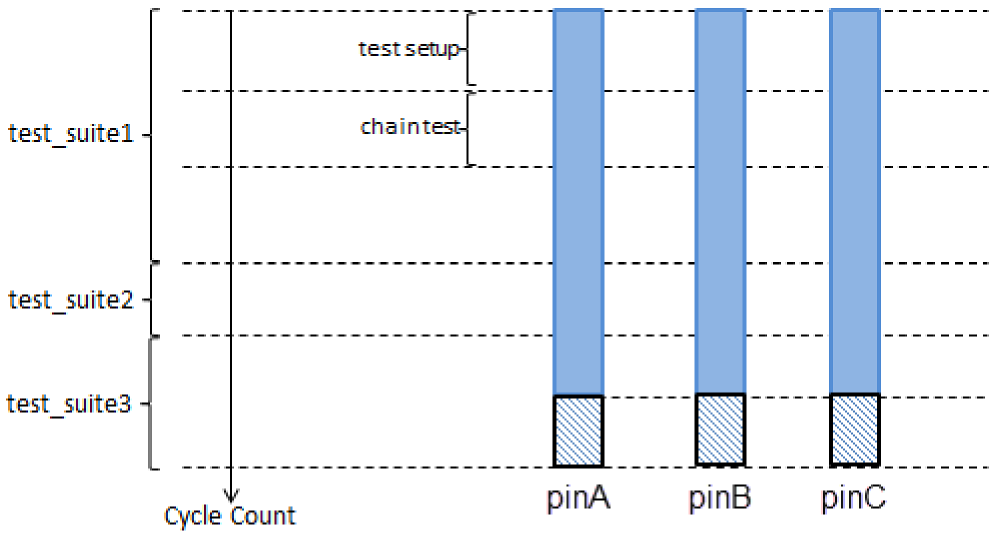
Truncated Failure File Examples

The examples illustrate supported failure truncation scenarios.

Example 1: Truncation of Failing Cycles for All Pins

There are three pins in the design: PinA, PinB, and PinC. Assume that the pin truncations occur for all of the pins in the same cycle, as shown in the figure below, where the blue areas depict the cycles for which failing cycles were logged. The cross-hatched areas depict the cycles where no failures were collected and are, therefore, truncated or ignored for diagnosis. The tool makes no assumptions regarding whether the patterns related to those cycles are passing or failing for diagnosis.

Figure 1-2. Truncation of Failing Cycles for all Pins



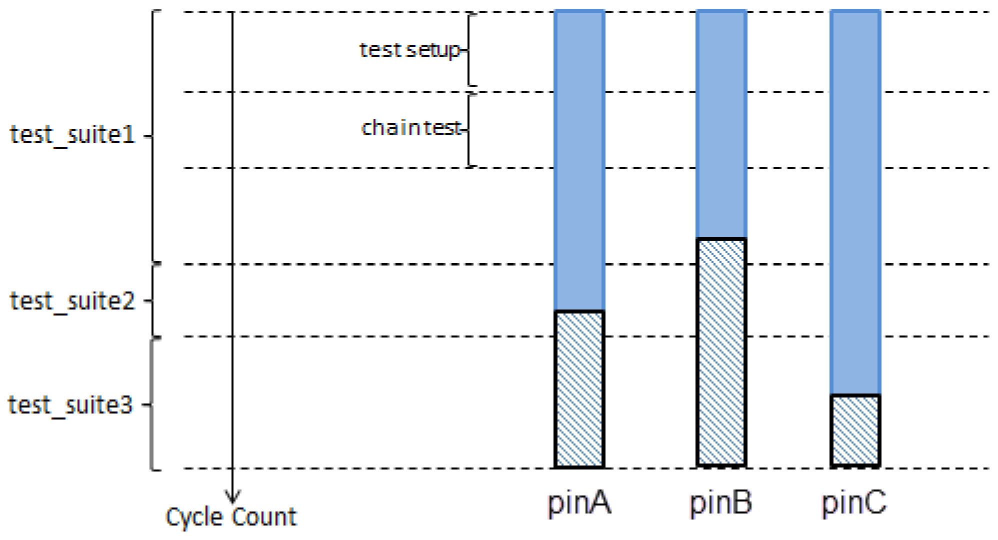
The failure_buffer_limit_reached occurs only in the last test suite, test_suite3, as shown below:

```
format cycle
test_suite_begin test_suite1
failures_begin
100 pinA H L
100 pinB L H
failures_end
last_cycle_applied 499
total_cycles 500
test_suite_end
test_suite_begin test_suite2
failures_begin
101 pinA H L
202 pinB L H
failures_end
last_cycle_applied 299
total_cycles 300
test_suite_end
test_suite_begin test_suite3
failures_begin
320 pinA H L
462 pinB L H
failures_end
failure_buffer_limit_reached last_cycle_logged
last_cycle_applied 599
total_cycles 600
test_suite_end
```

Example 2: Contiguous Truncation of Failing Cycles Per Pin in Test Suites

There are three pins in the design: PinA, PinB, and PinC. Assume that the pin truncations occur for all of the pins in different cycles, as shown in the figure below. As in the previous example, the blue areas depict the cycles for which failing cycles were logged. The cross-hatched areas depict the cycles where no failures were collected and are, therefore, truncated or ignored for diagnosis. The tool makes no assumptions regarding whether the patterns related to those cycles are passing or failing for diagnosis.

Figure 1-3. Contiguous Truncation of Failing Cycles Per Pin in Test Suites



In this scenario, the reverse mapping flow ignores the failures for the pin(s) where the failure buffer limit was reached for the rest of that suite and for all the following test suites, as shown below:

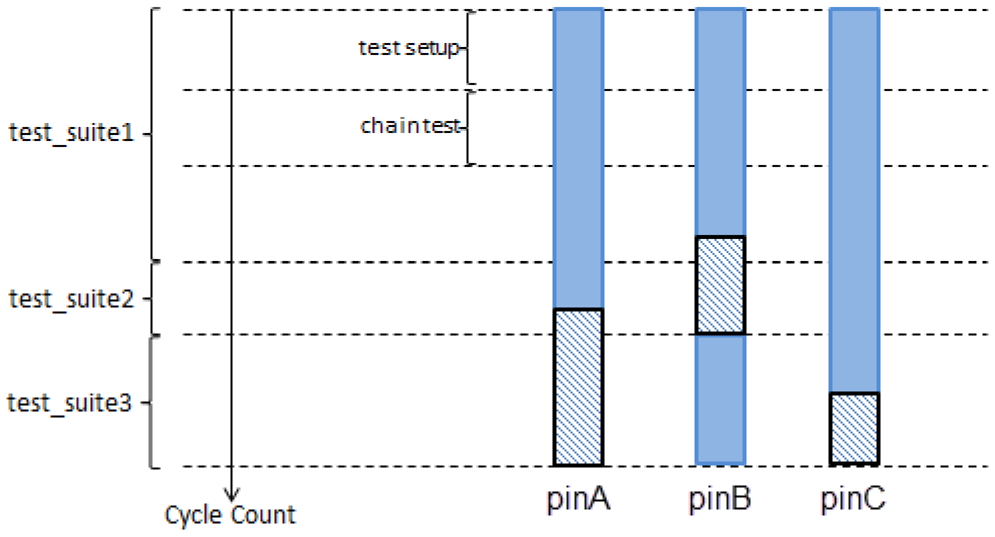
```
format cycle
test_suite_begin test_suite1
failures_begin
100 pinA H L
100 pinB L H
150 pinC H L
failures_end
failure_buffer_limit_reached pinB
last_cycle_applied 499
total_cycles 500
test_suite_end
test_suite_begin test_suite2
failures_begin
101 pinA H L
202 pinC L H
failures_end
failure_buffer_limit_reached pinA
failure_buffer_limit_reached pinB
last_cycle_applied 299
total_cycles 300
test_suite_end
test_suite_begin test_suite3
failures_begin
120 pinC H L
422 pinC L H
failures_end
failure_buffer_limit_reached pinC
failure_buffer_limit_reached pinA
failure_buffer_limit_reached pinB
last_cycle_applied 599
total_cycles 600
test_suite_end
```

Example 3: Noncontiguous Truncation of Failing Cycles Per Pin in Test Suites (Holes)

Given PinA, PinB, and PinC, assume that the pin truncations occur for all of the pins in different cycles but the fail buffer for pinB is flushed between test_suite2 and test_suite3 that creates a noncontiguous "hole" in the test data, as shown in the following figure.

Tessent automatically handles noncontiguous test data by adding an internal record to the failure file during remapping. See "Internal Records for Failure Buffer Limits" on page 71 for more information.

Figure 1-4. Noncontiguous Truncation of Failing Cycles per Pin in Test Suites



```
format cycle
test_suite_begin test_suite1
failures_begin
100 pinA H L
100 pinB L H
150 pinC H L
failures_end
failure_buffer_limit_reached pinB
last_cycle_applied 499
total_cycles 500
test_suite_end
test_suite_begin test_suite2
failures_begin
101 pinA H L
202 pinC L H
failures_end
failure_buffer_limit_reached pinA
failure_buffer_limit_reached pinB
last_cycle_applied 299
total_cycles 300
test_suite_end
test_suite_begin test_suite3
failures_begin
120 pinC H L
422 pinB L H
failures_end
failure_buffer_limit_reached pinA
failure_buffer_limit_reached pinC
last_cycle_applied 599
total_cycles 600
test_suite_end
```

Internal Records for Failure Buffer Limits

When you apply reverse mapping to a top-level failure file for SSN or retargeted patterns in a hierarchical DFT design, Tessent automatically handles any noncontiguous truncation by adding an internal record to the failure file to indicate the “hole” in the test data.

Note

The internal records are for internal use by Tessent when remapping. They are not intended for general use and they are not documented as failure file keywords.

The failure_buffer_limit_reached keyword in failure files indicates that the testing buffer filled with failures before the end of the patterns. The keyword indicates that failure data may be truncated after the last failing bit in the failure file. Bits after the buffer limit are considered “unknown” rather than “passing”. When testing a top-level design with multiple test suites, this kind of truncation may occur in any of the suites.

If the truncation is in the last suite, the existing failure_buffer_limit_reached keyword can indicate the truncation. However, if the truncation is in a suite before the last, it results in a “hole” in the passing bits between the last failing bit of the truncated suite and the beginning of the next suite. For hierarchical test using retargeted patterns, Tessent preserves this information by automatically adding records to the remapped failure file to indicate the truncated data.

The following example shows a simple multi-suite failure file where failure buffer limits were reached in the TS1 test_suite:

```
format cycle

test_suite_begin TS1
failures_begin
        688  bc_2_core_2_blk1_edt_channels_out1             H L
        720  bc_2_core_2_blk1_edt_channels_out1             H L
        752  bc_2_core_2_blk1_edt_channels_out1             H L
failures_end
failure_buffer_limit_reached bc_2_core_2_blk1_edt_channels_out1
test_suite_end

test_suite_begin TS2
failures_begin
        720  bc_2_core_2_blk1_edt_channels_out1             L H
        752  bc_2_core_2_blk1_edt_channels_out1             H L
        816  bc_2_core_2_blk1_edt_channels_out1             L H
failures_end
failure_buffer_limit_reached none
test_suite_end
```

When you reverse map this failure file to the core level, Tessent combines the test suites into a single test suite to map the failing bits. To preserve the truncation information at the end of the TS1 test suite, Tessent automatically adds an internal record that indicates the missing range as in the remapped failure file:

```
format pattern

scan_test

failures_begin
  0 blk1_edt_channels_out1 2 H L
  1 blk1_edt_channels_out1 2 H L
  2 blk1_edt_channels_out1 2 H L
  32 blk1_edt_channels_out1 2 L H
  33 blk1_edt_channels_out1 2 H L
  35 blk1_edt_channels_out1 2 L H
failures_end

fail_pattern_unknown_range blk1_edt_channels_out1 2 3 30
failure_buffer_limit_reached none
failure_file_end
```

The fail_pattern_unknown_range record indicates that there was a channel truncation in pattern 2 starting at cycle 3 and continuing to the end of pattern 30, which is the last pattern in the test suite.

Internal Records and Their Usage

Tessent has internal records for scan and chain tests that enable you to specify individual channels or global for all channels.

Note

The internal records are for internal use by Tessent when remapping. They are not for general use. Tessent automatically inserts them. They are documented here because you may see them in the remapped failure files and want to understand their meaning.

Scan Unknown Records

If the top level test suite is truncated in the scan test section with

```
failure_buffer_limit_reached <channel>
```

the remapped failure log has the record:

```
fail_pattern_unknown_range <channel> <startPat> <startCycle> <endPat>
```

The fail_pattern_unknown_range record indicates the remapped range of unknown bits. The range is for the specified <*channel*>. The range starts with the first cycle after the last failing bit that is specified by <*startPat*> and <*startCycle*>. The range ends with the last cycle of <*endPat*>.

If the top level fail log is truncated with

```
failure_buffer_limit_reached all
```

the failure log is remapped to a fail_pattern_unknown_range for each channel with a failing bit.

If the top level fail log is truncated with

```
last_cycle_applied
```

the remapped failure log has the record

```
fail_pattern_unknown_range_global <startPat> <startCycle> <endPat>
```

The fail_pattern_unknown_range_global record indicates a global range for all channels. The range starts with the first cycle after the last failing bit that is specified by <*startPat*> and <*startCycle*>. The range ends with the last cycle of <*endPat*>.

Chain Unknown Records

Holes in the chain test are represented by

```
fail_pattern_unknown_range_chain <channel> <startPat> <startCycle> \
   <endPat>

fail_pattern_unknown_range_chain_global <startPat> <startCycle> <endPat>
```

These are identical to the scan unknown records except that the range is part of the chain test. If the buffer fills in the chain test and extends over the scan test, both chain and scan unknown records may be output to the remapped failure log.

Modulo Unknown Records

SSN can lead to another form of unknown holes that repeat. These SSN holes are represented with the following records:

```
fail_pattern_unknown_modulo_ranges <channel> <startPat> <startOffset> \
   <length> <repeatOffset> <endPattern>

fail_pattern_unknown_modulo_ranges_global <startPat> <startOffset> \
   <length> <repeatOffset> <endPattern>

fail_pattern_unknown_modulo_ranges <channel> <startPat> <startOffset> \
   <length> <repeatOffset> <endPattern>

fail_pattern_unknown_modulo_ranges_global <startPat> <startOffset> \
   <length> <repeatOffset> <endPattern>
```

These specify ranges that start at <*startPat*> and <*startOffset*>. The range extends for <*length*> bits and repeats at an interval of <*repeatOffset*> bits from the <*startOffset*>. The range ends at the end of <*endPattern*>. There are channel/global and scan/chain variants.

Substituting Instance Text for Diagnosis Reporting

If you are running diagnosis on multi-core designs, Tessent Diagnosis can perform instance text substitution for the core names. In the diagnosis report, Tessent Diagnosis substitutes the instance text, specifically the pin_pathname and net_pathname report fields. You specify this instance text substitution in the ATE failure file. You can perform the substitution for both scan and chain diagnosis.

Use the following items as a guide for using instance text substitution:

- The instance_modeled core name must be an instance that exists in the flat model.
- There is no requirement that the core name correspond to a name that exists in the flat model.
- You cannot use instance text substitution in conjunction with the write_diagnosis command’s -encoded switch.
- You can specify multiple cores as in the following example:

  ```
  tracking_info_begin
  INSTANCE_SUBSTITUTE core_flat1 core_ATE_fail1
  INSTANCE_SUBSTITUTE core_flat2 core_ATE_fail2
  tracking_info_end
  ```

  In the event there are multiple substitutions or substitutions overlap, then Tessent Diagnosis processes these using an order-dependent method.

Prerequisites

- An ATE failure file. See “Guidelines for Preparing the ATE Failure File” for more information.

Procedure

Insert the following keyword and arguments between the tracking_info_begin and tracking_info_end failure file keywords:

INSTANCE_SUBSTITUTE ***instance_modeled instance_tested***

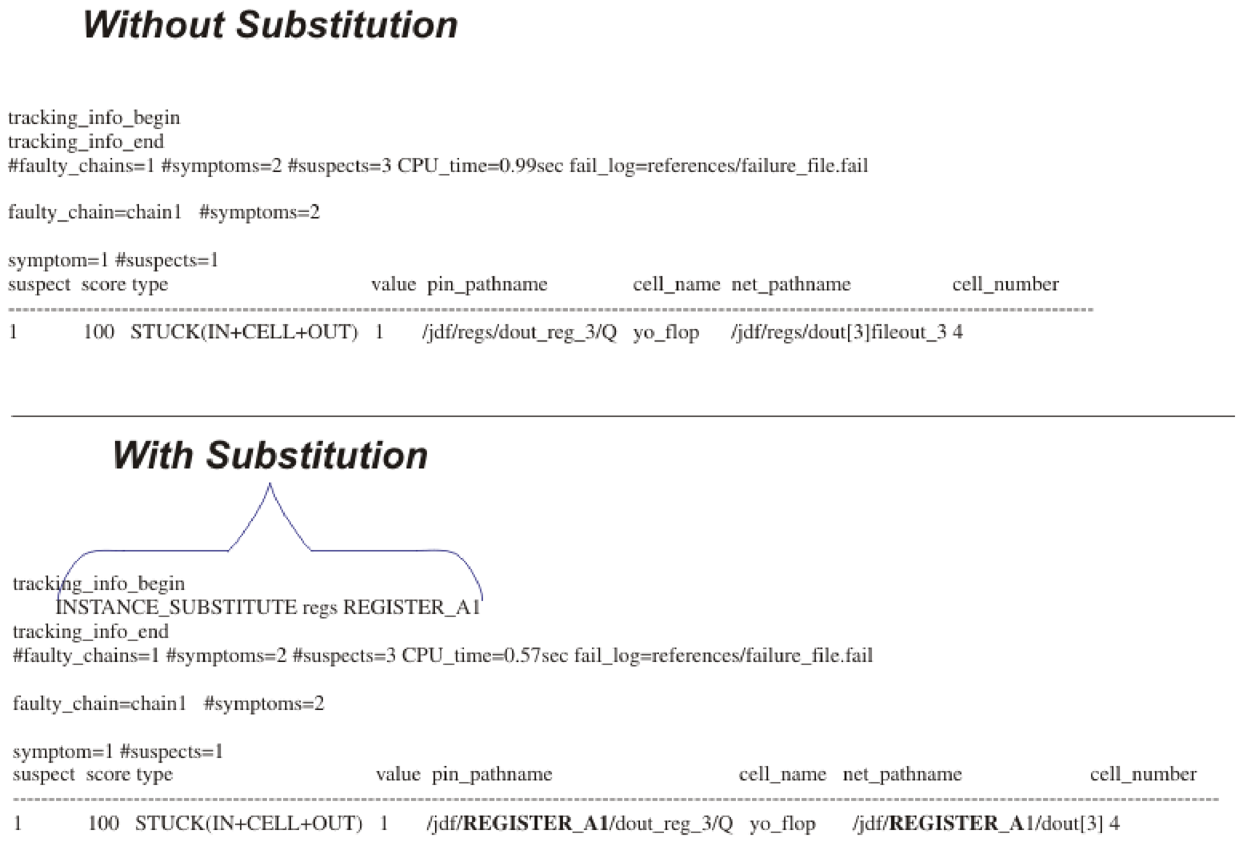
where:

- *instance_modeled* — Specifies the core name in the flat model.
- *instance_tested* — Specifies the core name that failed on the ATE.

Results

Figure 1-5 shows an example chain diagnosis report and the identical report with the instance text substitution. It shows that Tessent Diagnosis substitutes the name regs with the name REGISTER_A in the resulting diagnosis report:

Figure 1-5. Diagnostic Report Example Original and With Instance Substitution



Without Substitution

```
tracking_info_begin
tracking_info_end
#faulty_chains=1 #symptoms=2 #suspects=3 CPU_time=0.99sec fail_log=references/failure_file.fail

faulty_chain=chain1   #symptoms=2

symptom=1 #suspects=1
suspect  score type                       value  pin_pathname          cell_name  net_pathname              cell_number
-------------------------------------------------------------------------------------------------------------------------
1        100   STUCK(IN+CELL+OUT)  1      /jdf/regs/dout_reg_3/Q   yo_flop     /jdf/regs/dout[3]fileout_3 4
```

With Substitution

```
tracking_info_begin
     INSTANCE_SUBSTITUTE regs REGISTER_A1
tracking_info_end
#faulty_chains=1 #symptoms=2 #suspects=3 CPU_time=0.57sec fail_log=references/failure_file.fail

faulty_chain=chain1   #symptoms=2

symptom=1 #suspects=1
suspect  score type                       value  pin_pathname                  cell_name   net_pathname                cell_number
------------------------------------------------------------------------------------------------------------------------------------
1        100   STUCK(IN+CELL+OUT)  1      /jdf/REGISTER_A1/dout_reg_3/Q   yo_flop     /jdf/REGISTER_A1/dout[3] 4
```

Archiving Data for Re-Running Diagnosis

To re-run diagnosis, you should archive a complete set of ATPG data.

Prerequisites

- Ensure you have a complete set of ATPG data.
- Verified test patterns and other data.

Procedure

1. Before archiving, perform a Verilog simulation to verify the test patterns by saving the test patterns in a testbench format. For example:

   write_patterns pattern_file -verilog

2. Simulate the design to verify ASCII vector file format test patterns.
3. Archive enough information to allow the recreation of the original ATPG scenario as follows:
   - Netlist model
   - Dofiles and test procedure files for each test mode and pattern type
   - Tool version used to create the patterns
   - Parallel binary, STIL, and ASCII (if you want to read and edit them) test patterns
   - Log file for each test mode and test pattern type

Results

If the simulation is successful, your binary and ASCII patterns are valid. If the simulation fails, you must fix the design configuration. The ASCII patterns can be archived with the required binary patterns, or they can be regenerated from the binary patterns at diagnosis time.

STDF-V4 2007-Formatted File Support

Tessent Diagnosis provides the dlogutil utility that you can use to extract scan failures from STDF-V4 2007 and STDF-V4 2007.1-formatted files and output failure files that you can use for diagnosis with Tessent Diagnosis.

For more information about dlogutil, see "dlogutil Utility."

The dlogutil utility supports the following functionality to support scan failures in STDF format:

- Loading an ATE-generated STDF-V4 2007(.1)-formatted file containing scan test failure information.
- Reporting information for pattern and device.
- Reporting and extracting scan failures for a given device.
- Reporting and extracting scan failures for devices tested in a multi-site environment.
- Writing an ASCII-formatted ATDF file.

STDF-V4 2007 Records and Tessent Diagnosis

You can verify that your STDF file is version STDF-V4 2007 or later after you write the STDF file to an ASCII-based ATDF file with the write_atdf_file dlogutil command. If you see an STR failure record, then your STDF file is STDF-V4 2007 or later.

As described in "Multi-Site Support," STR records should be located within their associated PIR/PRR pairs.

The following table shows the STDF-V4 2007 records and their use with Tessent Diagnosis. See also "STDF-V4 2007 ATDF Record Examples."

Table 1-5. STDF-V4 2007 Records

Record Name	Required or Optional	Usage Comments
Version Update Record (VUR)	Required	Record header.

Table 1-5. STDF-V4 2007 Records (cont.)

Record Name	Required or Optional	Usage Comments
Name Map Record (NMR)	Required[1]	Recommended for Tessent Diagnosis, ATPG signal names may not be defined in Pin Map Records (PMR) based on the STDF V4 standard.
Cell Name Record (CNR)	Optional	Can produce large data volume. Not recommended for use with Tessent Diagnosis.
Scan Structure Record (SSR)	Optional	Not used by Tessent Diagnosis.
Chain Description Record (CDR)	Optional	Not used by Tessent Diagnosis.
Pattern Sequence Record (PSR)	Required	Main record for test pattern file information.
Scan Test Record (STR)	Required	Main record for logging scan failures.

1. Required only if the PMR does not contain ATPG signal names.

STDF-V4 2007 Tracking Information

By default, the dlogutil utility extracts tracking information from the STDF-V4 2007(.1) files during the conversion process. In the diagnosis report, this information is displayed in the tracking_info section.

Table 1-6 shows the STDF-V4 2007(.1) source information that dlogutil extracts for Tessent Diagnosis failure files.

Table 1-6. Extracted STDF-V4 2007 Tracking Information

Tessent Diagnosis Keyword	Source	STDF-V4 2007 Record
lot_id	LOT_ID	Master Information Record (MIR)
wafer_id	WAFER_ID	Master Results Record (MRR) or Master Information Record (MIR)
x_coord	X_COORD	Part Results Record (PRR)
y_coord	Y_COORD	Part Results Record (PRR)
part_id	PART_ID	Part Results Record (PRR)
dlog_temp	TST_TEMP	MIR

Table 1-6. Extracted STDF-V4 2007 Tracking Information (cont.)

Tessent Diagnosis Keyword	Source	STDF-V4 2007 Record
dlog_volt	string(s) in COND_LST of first STR record, starting with "VCC=" or "VDD="	STR
dlog_freq	string(s) in COND_LST of first STR record, starting with "SHIFT_FREQ=" or "CAPTURE_FREQ="	STR
unknown keyword	string(s) in COND_LST of first STR record, not covered by dlog_volt or dlog_freq	STR

Test Suite Naming

By default, the dlogutil utility names the test suite based on PSR_INDX for a given STR.

The utility uses the following naming convention:

```
test_suite_PSR_INDX
```

where *PSR_INDX* is the unique PSR_INDX field referenced by the STR. For example:

```
test_suite_1
```

You can use the dlogutil report_stdf_pattern_sequences command to determine the corresponding pattern file for a specific pattern sequence.

For Verigy testers, if the pattern file is stored in the PSR_NAM field of the PSR record, you can use the stdf_test_name_source dlogutil variable to append the pattern file name to the test suite name:

set stdf_test_name_source PSR_NAM

Failure Truncation Handling

By default, the dlogutil uses the failure_buffer_limit_reached none keyword for an extracted failure if all of its failing cycles are logged. Otherwise, the dlogutil utility uses the failure_buffer_limit_reached unknown keyword.

You can optionally change the handling of truncated failures by setting the following dlogutil variable:

stdf_fail_trunc_handling unknown | all | last_cycle_logged

There is no support in the dlogutil utility for per-pin-based failure truncation.

Support for Unknown Captured Values

The Tessent Diagnosis failure file only expects the H, L, and Z values for the expected values and actual values for failing bits. However, the STDF file may contain other values that have special meanings for testers. For example, Teradyne ultraFlex testers use G to indicate glitches and M to indicate mid-band voltage. These values are unknown to Tessent Diagnosis.

By default, dlogutil issues an error message for failing bits with unknown values and does not create failure files for parts with unknown values. However, you can map these values to valid values to generate valid failure files. To do this, specify the stdf_cap_data_mapping variable.

For more information about the stdf_cap_data_mapping variable, see "stdf_cap_data_mapping."

Multi-Site Support

For multi-site testing, the dlogutil utility supports a flexible file structure to accommodate both synchronous and asynchronous testing environments. The dlogutil utility can extract the failure information as long as each PIR record comes before its PRR counterpart, and the STR failure records are located within their PIR/PRR pairs. Each failure record contains a HEAD_NUM and SITE_NUM reference that dlogutil uses to match each record to its PIR/PRR pair.

Extracting Scan Failures from STDF-V4 2007 Files and Creating Failure Files

You can use the dlogutil utility either interactively on the command line or in a dofile.

See "dlogutil Utility" for complete dlogutil invocation syntax and arguments.

Prerequisites

- An STDF-V4 2007(.1)-formatted file.

Procedure

1. Load the STDF file using the load_stdf_file command. For example:

   load_stdf_file my_test.stdf

2. Report the part or test information from the STDF file executing any of the following commands:

   - report_stdf_conditions
   - report_stdf_parts

- report_stdf_pattern_sequences

These report commands are useful for debugging.

3. Extract the information from the STDF file you have loaded using the extract_stdf_failures command.

Results

A failure file that you can use with Tessent Diagnosis.

Examples

The following dlogutil dofile example provides basic usage instruction.

```
//loading the STDF file
dlogutil> load_stdf_file stdf/my_test.stdf
// Note: STDF ( my_test.stdf ) is successfully loaded using schema
(/home/tessent/installDir-rls/public/share/SiliconInsight/stdf/schema/
stdf_V4_2007.schema).
// Warning: No WIR/WRR record found.

//reporting STDF information
dlogutil> report_stdf_conditions

Field         Value

___________   ______
Temperature     27C

dlogutil> report_stdf_parts

No. PIR PRR HEAD SITE FLAG #TEST SOFT_BIN TIME PART_ID LOT_ID   Results

___ ___ ___ ____ ____ ____ _____ ________ ____ _______ ______ ___________
  0 141 146    1    1    8     4        3 1046       1      1 test failed

dlogutil> report_stdf_pattern_sequences

 ID CONT_FLG PSR_INDX                  PSR_NAM OPT_FLG TOTP_CNT LOCP_CNT

___ ________ ________ ___________________________ _______ ________ ______
119        0        1                chain_test1      14       10       10
120        0        2                 scan_test1      14      100      100
121        0        3                 scan_test2      14      100      100
```

```
//writing an STDF file into an ATDF file
dlogutil> load_stdf_file stdf/my_test.stdf
dlogutil> write_atdf_file results/my_test.atdf -rep

//extracting scan failures form an STDF file
dlogutil> extract_stdf_failures stdf/my_test.stdf results/flog_base

// Note: STDF ( my_test.stdf ) is
successfully loaded using schema
(/home/tessent/installDir-rls/public/share/SiliconInsight/stdf/schema/
stdf_V4_2007.schema).
//  Warning: No WIR/WRR record found.
//  Note: extract 8 failing bits from STR records [145,145], format=CYCLE.
//  Note: write failure information for part ( 0 ) into file
// (results/flog_base_0 ).
//  Note: faillog ( results/flog_base_0 ) was generated for failure ( 0 ).

//extracting failures for a subset of failing parts for selected tests
dlogutil> load_stdf_file stdf/my_test.stdf
dlogutil> report_stdf_pattern_sequences
dlogutil> stdf_selected_psr_ids {1 3}
dlogutil> report_stdf_parts
dlogutil> stdf_selected_parts { 0 1 3 5 }
dlogutil> extract_stdf_failures stdf/my_test.stdf results/flog_base
```

Diagnosis

Running scan diagnosis requires you to set the proper context and specify the pathname to the flat design netlist. You use Tessent Shell to start scan diagnosis.

You can perform scan diagnosis in batch mode in addition to performing other types of diagnosis such as gross delay defect, slow-clock compound hold-time, IDDQ diagnosis, and at-speed diagnosis.

Performing Scan Diagnosis

Before Tessent Diagnosis runs scan diagnosis, it simulates the design, verifies test pattern accuracy, and verifies failure file accuracy.

Prerequisites

- Design netlist — See Preparing the Design Netlist.
- Test patterns — See Preparing the Test Patterns.
- ATE failure file — See Guidelines for Preparing the ATE Failure File.
- Verification data — See Reverse Mapping Top-Level Failures to the Core.

Procedure

1. From a Linux/UNIX shell, enter:

   ```
   Tessent_Tree_Path/bin/tessent -shell
   ```

   where *Tessent_Tree_Path* is the path to where the Tessent Shell application tree is installed.

2. After Tessent Shell has started, enter:

   ```
   set_context patterns -scan_diagnosis
   ```

3. Enter:

   ```
   read_flat_model flat_model
   ```

   where *flat_model* is the pathname of the flat design netlist.

For more information on invoking Tessent Diagnosis, see the tessent shell command in the *Tessent Shell ReferenceManual*.

4. Load the test pattern file. Enter the read_patterns command and specify the pathname of the final test patterns similar to:

    read_patterns my_patterns.bin

    Where:

    - *my_patterns.bin* — The pathname of the test pattern file to load.

    If masking was used for the patterns run on the ATE, you must use the read_patterns command's -mask_file switch and provide a masking file that masks out the appropriate patterns. Check with the engineer that created the final test patterns to see if masking was used.

    See also "Preparing the Test Patterns" for additional information.

5. Run diagnosis on the failure file. Enter the diagnose_failures command and specify the pathname of the failure file similar to:

    diagnose_failures *failure_file* -output *ascii_report*

    Where:

    - ***failure_file*** — The pathname of the properly formatted failure file to diagnose.
    - **-output** — A switch that directs the tool to write the diagnosis results report to a file.
    - ***ascii_report*** — A user-specified pathname for the diagnosis results report. By default the report displays on screen. This report should be saved to a file for subsequent analysis.

    The diagnosis begins and Tessent Diagnosis performs four tasks in the following order:

    a. Simulates the design and verifies the test pattern accuracy. If expected values do not match the simulated values, an error displays and the diagnosis aborts. Use the report_failures command to display the test pattern mismatches. For more information, see the "Displayed Failure File Errors/Mismatches" section in this chapter.

    b. Verifies the failure file accuracy. If the data in the specified failure file contains errors or does not match the test patterns in the external database, an error displays and the diagnosis aborts. Use the set_diagnosis_options command to specify how many errors and pattern mismatches display before diagnosis aborts. For more information, see the "Displayed Failure File Errors/Mismatches" section in this chapter.

    c. Runs the diagnosis.

Results

Tessent Diagnosis outputs the results to an ASCII file. For details, refer to "Diagnosis Reporting."

Related Topics

Batch Mode in Tessent Diagnosis

Gross Delay Defect Diagnosis

Slow Clock Compound Hold-Time Diagnosis

IDDQ Diagnosis

Batch Mode in Tessent Diagnosis

You can run diagnosis in batch mode by using a dofile to pipe commands into the application. Dofiles let you automatically control the operations of the tool. The dofile is a text file you create that contains a list of application commands that you want to run.

Note
Do not include DRC handlings or ATPG settings in the dofile. All the DRCs and ATPG settings required by Tessent Diagnosis should already be saved in the flat design netlist written out from Tessent FastScan or Tessent TestKompress.

If you place all commands, including the exit command, in a dofile, you can run the entire diagnostic session as a batch process.

Create the dofile, and enter the following command to start scan diagnosis in the Tessent Shell environment and run the dofile:

Tessent_Tree_Path*/bin/tessent -shell -dofile *my_dofile.do

where:

- ***Tessent_Tree_Path*** the path to where the Tessent Shell application tree is installed.
- -dofile is the invocation switch required to run a dofile.
- ***my_dofile.do*** is the pathname of the dofile to run.

Tessent Shell invokes and runs the commands listed in the dofile.

The following dofile runs scan diagnosis in batch mode:

```
// Get the date for the session.
system date

// set_tcl_shell_options -abort_dofile_on_error off
// (useful if there are erroneous failure files and
// you do not want the tool to exit on an error during the batch run).

set_tcl_shell_options -abort_dofile_on_error off

// Set the context.
set_context patterns -scan_diagnosis

//Read in the flat netlist.
read_flat_model my_model

//Load the patterns.
read_patterns my_patterns/pat.bin

// Perform scan diagnosis only.
set_diagnosis_options -mode scan

// run the diagnosis. The following diagnose_failures command:
//   1) Verifies the test patterns.
//   2) Verifies the failure file.
//   3) Runs the diagnosis and saves the results.
diagnose_failures failure_file1 -output diagnosis_file1.txt -replace

// The next (and all subsequent) diagnose_failures commands do not
// reverify patterns until preceded by another read_patterns
// command.

diagnose_failures failure_file2 -output diagnosis_file2.txt -replace

// exit the session.
exit -force
```

By default, if the tool encounters an error when running one of the commands in the dofile, it stops dofile execution. You can, however, use the "set_tcl_shell_options -abort_dofile_on_error" command to turn this setting off.

Gross Delay Defect Diagnosis

You can configure Tessent Diagnosis to perform net open gross delay defect diagnosis. A gross delay defect has delay that is large enough to slow down a transition at the defect site and cause failure at all the observation points where the transition propagates to, regardless of the propagation path delay. A gross delay defect causes no failures if there is no transition at the defect site; that is, it passes a conventional static scan test. This capability helps separate at-speed and static defects in cases where it is unknown whether the failures are timing sensitive.

Enable gross delay defect diagnosis using the following command, depending on whether you are using scan diagnosis or Tessent Diagnosis Server:

- Tessent Diagnosis — Set the set_diagnosis_options command's optional -GRoss_delay argument to ON. For example:

```
set_diagnosis_options -gross_delay on
```

- Tessent Diagnosis Server — Set the set_diagnosis_options command's optional -GRoss_delay argument to ON. For example:

```
set_diagnosis_options -gross_delay on
```

You can perform the gross delay diagnosis with any of the following:

- Stand-alone Tessent Diagnosis
- Layout-aware diagnosis
- Server-based Tessent diagnosis

Tessent Diagnosis performs gross delay diagnosis for net open defects. Currently, the tool does not support:

- Chain failures
- Gross delay bridges

Gross Delay Diagnosis Reporting

The following table lists the gross delay defect diagnosis reporting, depending on the mode, done by the tool.

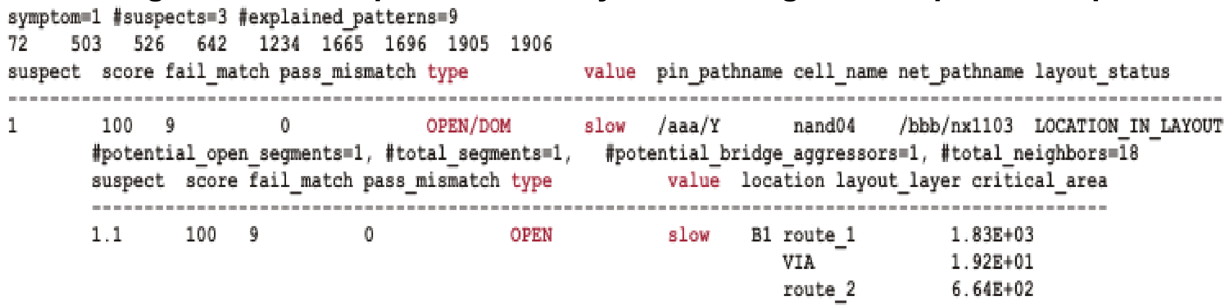
Table 1-7. Gross Delay Defect Diagnosis Reporting

Mode	Diagnosis Report type	Diagnosis Report value
Logic diagnosis	OPEN/DOM	slow/STR/STF
Logic diagnosis using a layout-aware diagnosis Layout Database (LDB)	OPEN	slow/STR/STF

Gross Delay Diagnosis Example

The following table provides an example Tessent Diagnosis tool diagnosis report showing a gross delay defect.

Figure 1-6. Example Gross Delay Defect Diagnosis Report Excerpt



```
symptom=1 #suspects=3 #explained_patterns=9
72    503   526   642   1234  1665  1696  1905  1906
suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name net_pathname layout_status
----------------------------------------------------------------------------------------------------------------
1        100   9          0             OPEN/DOM       slow   /aaa/Y       nand04    /bbb/nx1103  LOCATION_IN_LAYOUT
         #potential_open_segments=1, #total_segments=1,   #potential_bridge_aggressors=1, #total_neighbors=18
         suspect  score fail_match pass_mismatch type           value  location layout_layer critical_area
         ----------------------------------------------------------------------------------------------
         1.1      100   9          0             OPEN           slow   B1 route_1       1.83E+03
                                                                          VIA           1.92E+01
                                                                          route_2       6.64E+02
```

Slow Clock Compound Hold-Time Diagnosis

Tessent Diagnosis can perform diagnosis of scan chain compound hold-time faults. A compound hold-time fault is caused by a spot delay defect in the clock structure resulting in delayed shift and capture clocks to some cells. This defect can result in a hold-time failure during shift mode and capture mode.

Enable compound hold-time diagnosis using the following command in either Tessent Diagnosis or the Tessent Diagnosis server. The option only applies to hold times on shift paths, not capture paths.

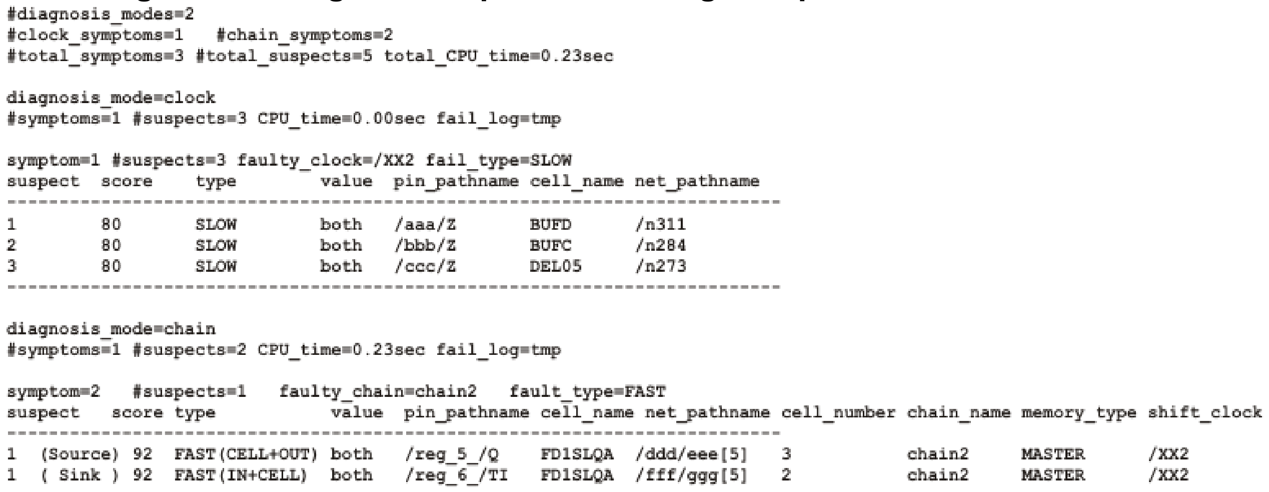
set_diagnosis_options -compound_hold_time_fault_diagnosis on

Compound Hold-Time Diagnosis Reporting

For each chain hold-time suspect, Tessent Diagnosis reports both the source cell and sink cell. The tool also reports the possible slow clock signals that are causing hold time errors on scan chains.

The following table shows a Tessent Diagnosis reporting that contains compound hold-time information.

Figure 1-7. Diagnosis Report Containing Compound Hold-Time Results



```
#diagnosis_modes=2
#clock_symptoms=1   #chain_symptoms=2
#total_symptoms=3 #total_suspects=5 total_CPU_time=0.23sec

diagnosis_mode=clock
#symptoms=1 #suspects=3 CPU_time=0.00sec fail_log=tmp

symptom=1 #suspects=3 faulty_clock=/XX2 fail_type=SLOW
suspect  score    type        value  pin_pathname cell_name net_pathname
-------------------------------------------------------------------------
1        80       SLOW        both   /aaa/Z       BUFD      /n311
2        80       SLOW        both   /bbb/Z       BUFC      /n284
3        80       SLOW        both   /ccc/Z       DEL05     /n273
-------------------------------------------------------------------------

diagnosis_mode=chain
#symptoms=1 #suspects=2 CPU_time=0.23sec fail_log=tmp

symptom=2   #suspects=1   faulty_chain=chain2   fault_type=FAST
suspect   score type           value  pin_pathname cell_name net_pathname cell_number chain_name memory_type shift_clock
-------------------------------------------------------------------------
1  (Source) 92  FAST(CELL+OUT) both   /reg_5_/Q    FD1SLQA  /ddd/eee[5]   3           chain2     MASTER      /XX2
1  ( Sink ) 92  FAST(IN+CELL)  both   /reg_6_/TI   FD1SLQA  /fff/ggg[5]   2           chain2     MASTER      /XX2
```

For symptom 2, suspect 1, the source cell has an appendix of (CELL+OUT) for the fault type, while the sink cell has an appendix of (IN+CELL) for the fault type.

When a hold-time error occurs at sink cell *N*, the error could be due to one of the following:

- The cell *N*'s clock comes late.
- The cell (*N*+1)'s clock comes early.
- The delay is smaller than expected between these two cells.

To determine the cause, you should check from the source cell and its scan output all the way to the sink cell and its scan input.

IDDQ Diagnosis

Tessent Diagnosis enables you to perform IDDQ diagnosis on patterns that have passed stuck-at testing but failed IDDQ testing. IDDQ diagnosis can help you pinpoint the locations on defective devices that are causing the failures. It supports single and multiple defects. The failure files you use as input to Tessent Diagnosis should include IDDQ measurements for passing and failing IDDQ patterns.

When you set the set_diagnosis_options -mode auto command, the tool automatically detects IDDQ failure data from the fail log and runs IDDQ diagnosis. The first line of the fail log must specify the keyword "iddq_test" as shown below in Figure 1-8.

Tessent Diagnosis Server and layout-aware diagnosis do not support IDDQ mode.

IDDQ Failure File Format

The following example shows the supported failure file format for the IDDQ failure data. For IDDQ diagnosis, you must convert your failure data to this format.

Figure 1-8. Supported IDDQ Failure File Format

```
iddq_test
format pattern
tracking_info_begin
   lot_id lot_id
   part_id part_id
   wafer_id wafer_id
   x_coord x_coord
   y_coord y_coord
tracking_info_end
iddq_jump_size jump_size_value
failures_begin                  //list of actual current measurements for
                                //each scan test pattern
   pattern_0 iddq_current_0
   ...
   pattern_n iddq_current_n
failures_end
```

The iddq_jump_size keyword is required to define the current threshold that differentiates the measured IDDQ current clusters. The threshold determines the status of each pattern. After sorting the failure patterns in ascending order by IDDQ current, the tool considers a jump to be any two adjacent patterns with a current differential larger than the specified jump size.

The tool clusters the sorted patterns based on iddq_jump_size. Patterns in the first cluster are considered passing patterns and other patterns are treated as failing patterns. If the tool only detects one cluster, it assumes all patterns are passing and identifies no suspects.

Tip

For the IDDQ jump size value, 10% of the average IDDQ current from a good die could be a good starting point. Ideally, the value should be greater than the hardware current measurement variation or noise.

Tessent Diagnosis does not support multiple test suites for the IDDQ failure file.

The following figure shows an IDDQ failure file:

Figure 1-9. IDDQ Failure File

```
iddq_test
format pattern
tracking_info_begin
   lot_id ABC
   part_id 1
   wafer_id 1
   x_coord 2
   y_coord 3
tracking_info_end
iddq_jump_size 2
failures_begin
   0 18.5626
   1 18.3624
   2 18.5376
   3 43.8
   …
   97 18.5
   98 18.5
   99 18.5251
failures_end
```

IDDQ Diagnosis Report

The following figure shows a sample IDDQ diagnosis report.

Figure 1-10. IDDQ Diagnosis Report

```
tracking_info_begin
   lot_id YA_lot
   part_id 1
tracking_info_end

#diagnosis_modes=1
#logic_symptoms=1
#total_symptoms=1 #total_suspects=10
total_CPU_time=71.25sec

diagnosis_mode=iddq
#symptoms=1 #suspects=10
CPU_time=71.25sec fail_log=flogs.ya/flog_iddq_ya_1
#failing_patterns=23,
#passing_patterns=77
#unexplained_failing_patterns=0

symptom=1 #suspects=10
#explained_patterns=23
70    34    86    85
   53    54    23    82    68    76
59    25    93    31
   90    87    43    42    65    15
11    3     12
suspect  score fail_match
pass_mismatch type    value  pin_pathname cell_name net_pathname
-------------------------------------------------------------------------------------------
1        100   23          0             STUCK
  0      …/C1/Z       H250      …/foo_5_1
2        100   23          0             EQ1
    0      …/C1/A       H250      …/reg_1
3        100   23          0             EQ1
    0      …/C1/B       H250      …/data_s[5]
4        100   23          0             STUCK
  0      …/outbuf/Z   SBUF10    …/epwr_70[5]
5        100   23          0             EQ4      0      …/outbuf/A   SBUF10    …/foo_5_1
…
-------------------------------------------------------------------------------------------
```

At-Speed Failure Diagnosis

You can perform at-speed failure diagnosis with Tessent Diagnosis and subsequently visually examine, with Tessent Visualizer, any failing paths written to the failure diagnosis report.

See "Viewing Failing Paths for a Pattern" for more information.

Specify performing at-speed failure diagnosis by adding the following switch to the set_diagnosis_options command:

set_diagnosis_options ... -at_speed on

The following example shows an at-speed failure diagnosis run.

```
...
// Finished reading "design.flat". sim_gates=989 PIs=19 POs=23.
FAULT> set_diagnosis_options -mode scan -at_speed on    // perform at-speed diagnosis
FAULT> set_pattern_source external pat_good.stil -stil
// Reading STIL input file "pat_good.stil"
FAULT> diagnosis_failures 1.fail -ouput x1.diag -replace -csv x1.csv // create fail report
// Verifying 42 external patterns.
// incorrect_patterns=0, simulated_patterns=42, simulation_time=0.01sec.
FAULT> diagnose_failures 2.fail -output x2.diag -replace -csv x2.csv
... converting 190 cycles to patterns (5038 total cycles loaded)
FAULT> display_diagnosis_report x1.diag       // view failure report in Tessent Visualizer
// command: open_visualizer -display debug
FAULT> display_diagnosis_report x1.diag
FAULT> write_failing_paths x1.slow_paths.txt
```

You cannot use at-speed diagnosis with the layout-aware diagnosis flow. Even though an LDB may have been opened prior to executing diagnose_failures, at-speed diagnosis results does not include layout information.

Tessent Diagnosis calculates failing capture cycles for at-speed diagnosis using the following criteria:

- Launch-off-capture — The failing capture cycles occur during the second capture clock. The first capture clock is slow speed.
- Launch-off-shift — The failing capture cycles occur on the first and all subsequent capture cycles.

When performing at-speed diagnosis, for each failing bit, the tool attempts to identify the paths that can cause a particular failure and converge the paths to a single location. If there are too many failing bits and too many paths, it cannot isolate a defect location and reports zero symptoms. When this occurs, adjust the test conditions so that there are fewer failing bits to diagnose. Do this by adjusting the test conditions closer to passing conditions, such as by reducing the test frequency or adjusting the voltage on the power supply. Fewer paths should fail, and Tessent Diagnosis should be able to identify some symptomatic paths.

Note

This is the opposite approach than you would use when running gross delay or static diagnosis, and should only be applied during at-speed diagnosis.

Tessent Diagnosis expects that there exists a timing and voltage configuration under which the part passes the test. Additionally, Tessent Diagnosis expects that the test can be made to fail by only shortening the delay (compared to the passing condition) between clock edge 1 and clock edge 2.

When You Should Perform At-Speed Diagnosis

At-speed diagnosis provides meaningful results when the failures logged are characterized by small delays, not gross delays. Both gross delay faults and small delay faults require a transition to control them; the difference is in how the tool observes the fault. A gross delay fault causes a delay large enough that it is observed at all sensitized observe points (scan flops and primary outputs). A small delay fault, on the other hand, causes a delay large enough to be observed at one or more sensitized observe points, but not the full set of sensitized observe points. A small delay fault therefore corresponds to two main kinds of silicon issues:

- A defect introducing a very small delay; that is, a via whose resistance has increased moderately as a result of electrical stress.
- A path in a defect-free die with insufficient timing slack; that is, a frequency-limiting path in first silicon that needs to be identified and corrected in a subsequent tape out.

Use at-speed diagnosis to diagnose only small delay failures, whether they be defects or slow paths.

To differentiate small delay failures from gross delay failures, you must consider the test conditions required to produce the failures.

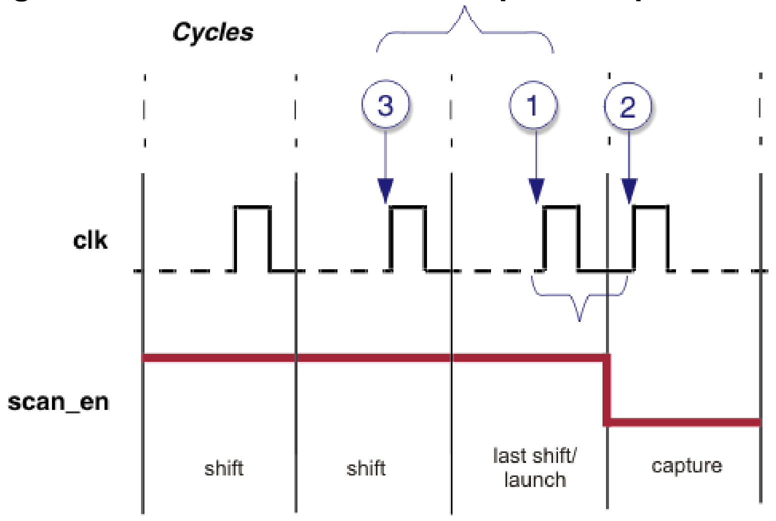
- Small delay failures fail only when applying path delay fault patterns or transition fault patterns (any patterns including multiple, fast clock pulses) at or near the spec operating frequency defined for the part.
- Gross delay failures are those that persist well below the spec operating frequency defined for the part.

When you diagnose a small delay failure using at-speed diagnosis, the result is a list of suspects (nodes and cell pins) that are of type SLOW. If the failure is a defect, this is the preferred report format. However, if the failure is known to be a slow path in defect-free silicon, you can use the write_failing_paths command immediately after the diagnose_failures command to generate report data that can help identify and fix any slow paths.

Perform at-speed diagnosis when the following conditions exist:

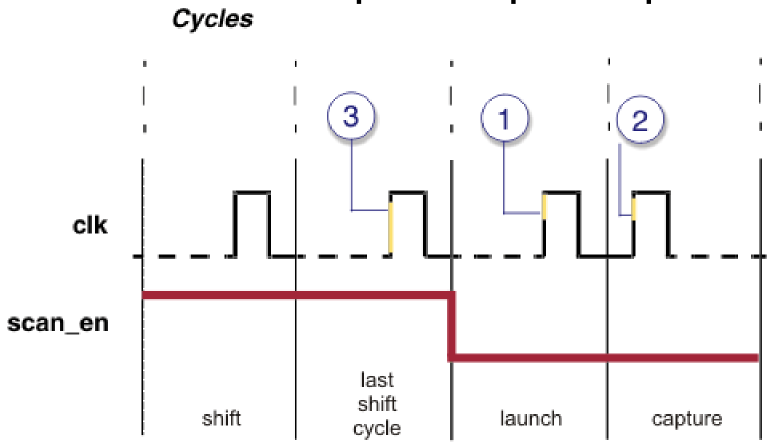
- The failures are actual at-speed failures, specifically the following:
    - The test pattern should pass at a low speed but fail at a high speed.
    - For launch-off-shift, the following figure illustrates how the clk-to-clk edge between 1 and 2 should be smaller than 3 to 1.

Figure 1-11. Launch-Off-Shift At-Speed Requirements



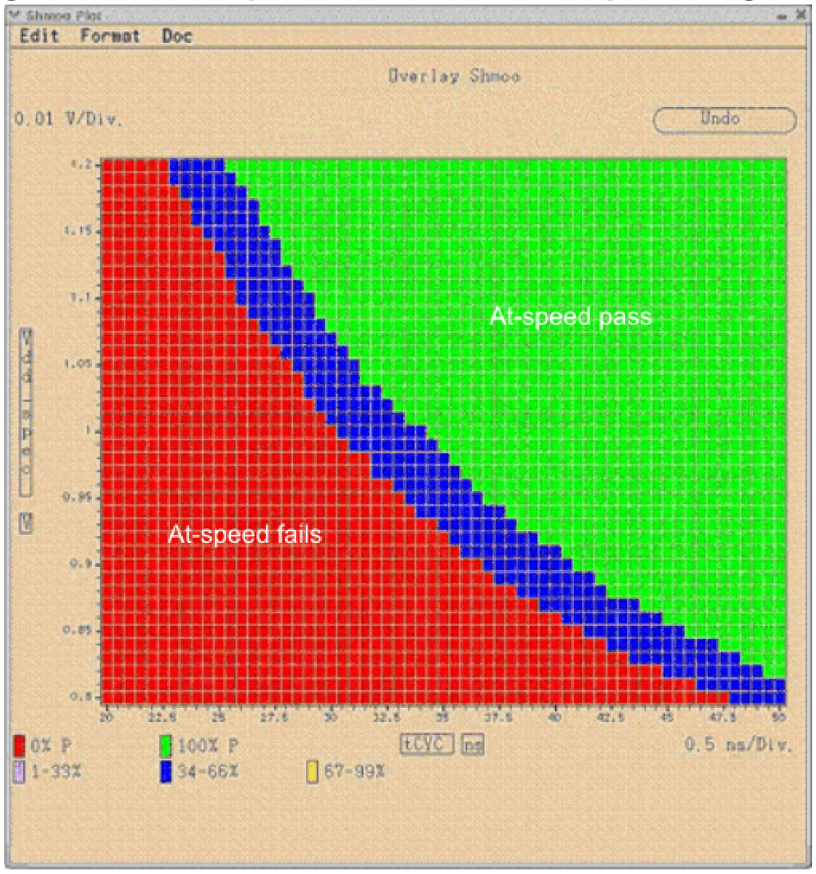
- o For launch-off-capture, the following figure illustrates how the timing between edge 1 and 2 should be smaller than 3 and 1.

Figure 1-12. Launch-Off-Capture At-Speed Requirements



- o Consider experimenting on a couple of sampling devices. For example, shmooing is a good technique. Additionally, sweep the frequency from low to high so you can determine if the device fails consistently at higher frequencies.

Figure 1-13. Example Shmoo Plot for At-Speed Diagnosis



- Voltage levels: If you lower the voltage, ensure the device passes at low frequency and fails at high frequency.

When You Should Not Perform At-Speed Diagnosis

Use gross delay diagnosis mode to diagnose gross delay failures. The vast majority of process-induced manufacturing defects typically introduce gross delays, not small delays. For diagnosing manufacturing defects in bulk, therefore, gross delay mode is virtually always the best choice.

In rares cases, you might need to use at-speed diagnosis to diagnose manufacturing defects in volume when the manufacturing process generates large quantities of delay failures at operating spec frequency and those failures always pass after reducing the operating frequency.

Small delay failures are most often encountered in early silicon (slow paths) or as defects in small quantities. Large-volume, frequency-sensitive failures are predominantly gross delay failures.

Guidelines for Customizing the Diagnostic Session

Tessent Diagnosis provides many options for customizing the diagnostic session. Additional diagnosis capabilities such as chain diagnosis, cell internal analysis, and bridge and open analysis are available using the set_diagnosis_options command arguments.

You can modify how the diagnosis runs as described in the following topics:

Log File Generation

Log files provide a useful way to examine the operation of the tool, especially when you run the tool in batch mode using a dofile. If errors occur, you can examine the log file to see exactly what happened. The log file contains all software application operations and any notes, warnings, or error messages that occur during the session.

To generate a log file, use the -Logfile switch when you invoke Tessent Shell as the following example demonstrates:

```
shell> Tessent_Tree_Path/bin/tessent -shell -logfile my_logfile.log
```

You can also create a log file from a Tessent Diagnosis session with the set_logfile_handling command.

By default, Tessent Diagnosis creates a new log file. If you want to overwrite a previously generated log file, you must use the -Replace switch.

Note
If you create a log file during a tool session, the log file only contain notes, warnings, or error messages that occur after you enter the set_logfile_handling command. Therefore, you should enter it as one of the first commands in the session.

System Mode Toggles

Tessent Diagnosis invokes in either Fault or Good system mode depending on the system mode of the tool that was used to save the design netlist.

Table 1-8 shows the Tessent Diagnosis invocation modes.

Table 1-8. Tessent Diagnosis Invocation Modes

If netlist is saved from mode...	Tessent Diagnosis invokes in this mode...
Atpg or Fault	Fault
Good	Good
Setup mode	Does not invoke

To toggle between system modes, use the set_system_mode command. For example, to toggle between Fault mode and Good mode:

```
FAULT> set_system_mode good
GOOD> set_system_mode fault
FAULT>
```

Reported Suspects

By default, for each symptom, Tessent Diagnosis reports the top three suspects and all suspects with a score greater than 80. When you start the diagnosis, you can set Tessent Diagnosis to report a percentage of the suspects with the highest scores for each symptom.

Enter the set_diagnosis_options command with the -Report switch and an integer between 1 and 100 to indicate the percentage. For example, the following command directs Tessent Diagnosis to report the top 50% of all suspects for each symptom:

```
set_diagnosis_options -report 50
```

Saved Diagnosis Reports

You can either view the diagnostic report on screen or save it to a file. By default, the diagnosis report displays on screen. You can save the diagnosis report to a file by using the write_diagnosis command or the diagnose_failures command.

- write_diagnosis command — Writes diagnosis reports to ASCII text, layout marker, and Comma Separated Value (CSV) formats. For example:

  ```
  write_diagnosis -format text csv -file report_filename
  ```

- diagnose_failures command — Writes diagnosis reports to ASCII text and CSV formats. Enter the diagnose_failures command with the -Output option to direct the tool to save the diagnostic report in ASCII format. For example:

  ```
  diagnose_failures -output report_filename
  ```

Diagnosis Time Limit

By default, the diagnosis continues until all the failures in the specified failure file are analyzed. You can use the set_diagnosis_command command with the -Time_limit switch to specify the number of seconds the diagnosis should run before it aborts with an error message.

For example, the following command limits subsequent diagnoses to 60 seconds:

```
set_diagnosis_options -time_limit 60
```

Displayed Failure File Errors/Mismatches

The diagnose_failures command verifies the data in the specified failure file is accurately translated and matches the test patterns in the external database before running diagnosis. By default, an error displays and the diagnosis aborts when an error or mismatch is encountered. You can specify the number of error/pattern mismatches that display before diagnosis aborts with the set_diagnosis_options command.

Enter the set_diagnosis_options command as follows to display all errors and mismatches before the diagnosis aborts:

```
set_diagnosis_options -failurefile_mismatch_verbosity all
```

Enter the set_diagnosis_options command with the -Failurefile_mismatch_verbosity switch and an integer that specifies the number of errors and mismatches to display before the diagnosis aborts. The following example displays 50 errors/mismatches before the diagnosis aborts:

```
set_diagnosis_options -failurefile_mismatch_verbosity 50
```

For information on troubleshooting failure file errors/mismatches, see the "Pattern and Failure File Mismatches" section in this manual.

Chapter 2 Diagnosis Reporting and Troubleshooting

Depending on the contents of the failure file, Tessent Diagnosis produces a chain diagnosis report or a logic diagnosis report.

Note

You should write the diagnosis to a file so you can analyze and manipulate the results at the conclusion of the diagnosis.

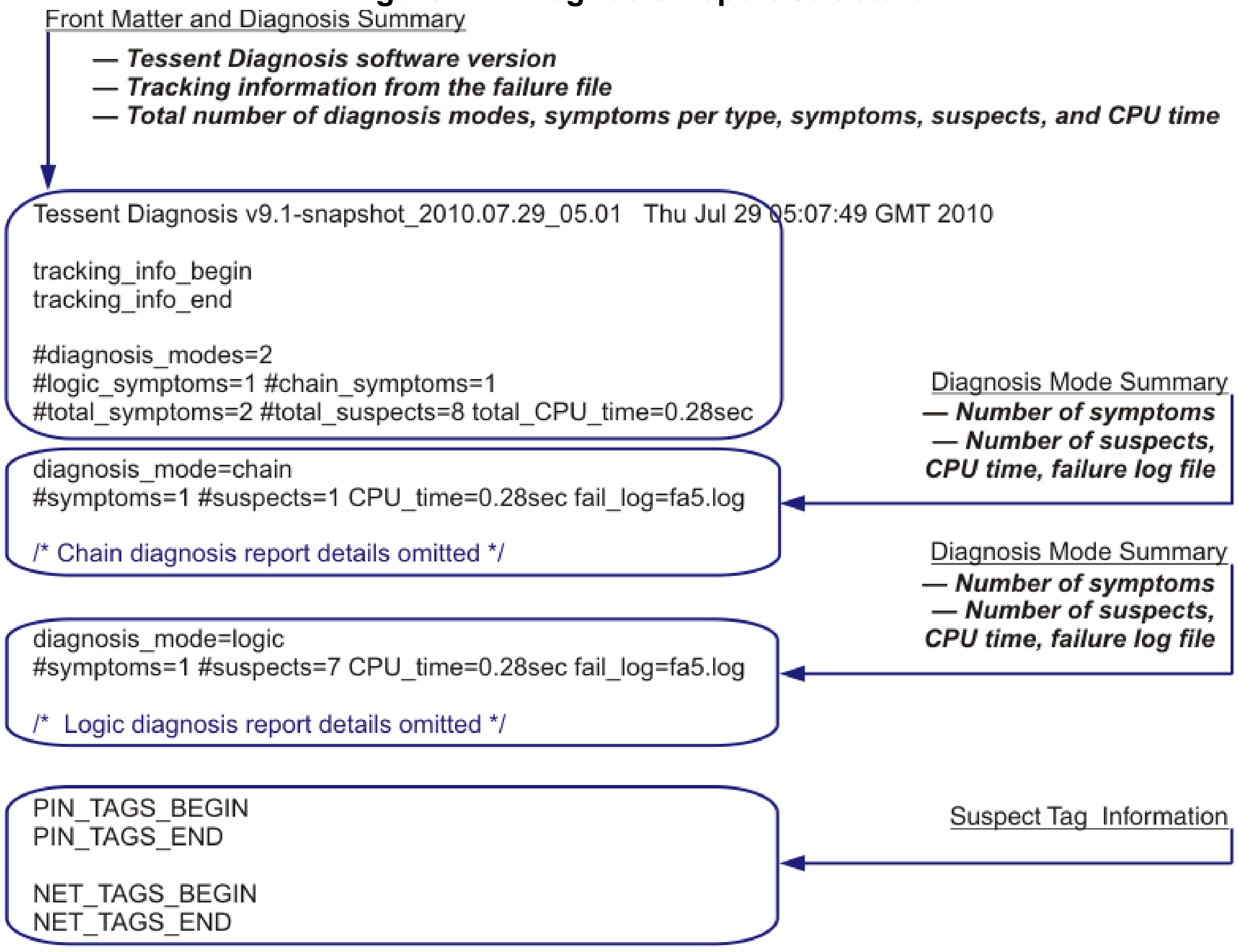
In addition to the chain diagnosis and logic diagnosis reports, this chapter discusses tips for troubleshooting common Tessent Diagnosis issues.

Diagnosis Reporting

All Tessent Diagnosis reports share a common report header followed by one or more diagnostic mode report sections.

The common report header information is explained in "Front Matter and Diagnosis Summary," the chain diagnosis report is detailed in "Chain Diagnosis Section," and the logic diagnosis report is detailed in "Logic Diagnosis Section."

Figure 2-1. Diagnosis Report Structure


Front Matter and Diagnosis Summary
— Tessent Diagnosis software version
— Tracking information from the failure file
— Total number of diagnosis modes, symptoms per type, symptoms, suspects, and CPU time
Diagnosis Mode Summary
— Number of symptoms
— Number of suspects, CPU time, failure log file
Diagnosis Mode Summary
— Number of symptoms
— Number of suspects, CPU time, failure log file
Suspect Tag Information

```
Tessent Diagnosis v9.1-snapshot_2010.07.29_05.01   Thu Jul 29 05:07:49 GMT 2010

tracking_info_begin
tracking_info_end

#diagnosis_modes=2
#logic_symptoms=1 #chain_symptoms=1
#total_symptoms=2 #total_suspects=8 total_CPU_time=0.28sec

diagnosis_mode=chain
#symptoms=1 #suspects=1 CPU_time=0.28sec fail_log=fa5.log

/* Chain diagnosis report details omitted */

diagnosis_mode=logic
#symptoms=1 #suspects=7 CPU_time=0.28sec fail_log=fa5.log

/*  Logic diagnosis report details omitted */

PIN_TAGS_BEGIN
PIN_TAGS_END

NET_TAGS_BEGIN
NET_TAGS_END
```

Front Matter and Diagnosis Summary

The front matter and diagnosis summary reports the Tessent Diagnosis software version, including the release number, date, and time in addition to tracking information and a diagnosis summary.

Tracking Information

The tracking information section replicates the tracking information from the failure file verbatim and is enclosed by the keywords tracking_info_begin and tracking_info_end. The keywords are included even if the failure file does not contain any tracking information.

Diagnosis Summary

The summary information immediately follows the tracking information. The diagnosis summary section contains two sets of information: a summary containing the total number of diagnosis modes, symptoms, suspects, and CPU time; and a similar set of information for each reported diagnosis mode.

The following table lists the diagnosis report summary elements.

Table 2-1. Diagnosis Report Summary Elements

Report Field	Description
#diagnosis_modes	Total number of diagnosis mode sections appearing in the diagnosis report.
#logic_symptoms[1]	Total number of logic diagnosis symptoms.
#scan_enable_symptoms[1]	Total number of scan_enable symptoms.
#clock_symptoms[1]	Total number of clock diagnosis symptoms.
#chain_symptoms[1]	Total number of chain diagnosis symptoms.
#total_symptoms	Total number of symptoms reported in all of the diagnosis_mode sections.
#total_suspects	Total number of suspects reported in all of the diagnosis_mode sections.
Per diagnosis_mode Summary Elements (repeatable)	
diagnosis_mode=	Diagnosis mode header followed by one of the following keywords: logic | scan_enable | clock | chain
#symptoms	Number of symptoms diagnosed. Each symptom represents a group of failure information (fail subsets) that is linked to a single probable source defect. Specific failure information may be linked to multiple symptoms.

Table 2-1. Diagnosis Report Summary Elements (cont.)

Report Field	Description
#suspects	Number of suspects diagnosed. A suspect is a circuit location determined to be a possible cause of a symptom.
CPU_time	CPU time used for the diagnosis.
fail_log	Name of the failure log associated with the diagnosis.

1. This report element is only reported when a corresponding diagnosis mode appears in the report; otherwise this entry is suppressed. A diagnosis mode appears only when the number of symptoms for this mode is greater than 0.

Chain Diagnosis Section

The chain diagnosis report lists the results of the chain diagnosis in the diagnosis summary and symptom descriptions.

Diagnosis Summary

The diagnosis summary of the chain diagnosis report summarizes the diagnosis including the total number of symptoms, and suspects; CPU time; and the failure file path.

Table 2-2 describes the chain diagnosis report summary elements.

Table 2-2. Chain Diagnosis Report Summary Elements

Report Field	Description
diagnosis_mode=chain	Identifies the Chain Diagnosis section of the report.
#symptoms	Number of logic symptoms diagnosed. Each symptom represents a group of failure information (fail subsets) that is linked to a single probable source defect. Specific failure information may be linked to multiple symptoms. One faulty chain is one symptom.
#suspects	Number of suspects diagnosed. A suspect is a circuit location determined to be a possible cause of a symptom.
CPU_time	CPU time used for the diagnosis.
fail_log	Name of the failure log associated with the diagnosis.
Per symptom Summary Elements (repeatable)	
symptom=*n*	Symptom ID number assigned by the tool.
#suspects	Number of suspects diagnosed. A suspect is a circuit location determined to be a possible cause of the symptom.
faulty_chain=*name*	The chain name associated with the reported symptom.

Table 2-2. Chain Diagnosis Report Summary Elements (cont.)

Report Field	Description
fault_type=*type*	When you are diagnosing down to the scan cell, this is the fault type diagnosed for the faulty scan call. The value may be one of: • STUCK • INDETERMINATE • FAST • SLOW When you are diagnosing down to the scan chain because there are no scan pattern failures, this is the fault type for the diagnosed faulty scan chain. The value may be one of: • STUCK_AT_1 • STUCK_AT_0 • FAST_TO_RISE • FAST_TO_FALL • FAST • SLOW_TO_RISE • SLOW_TO_FALL • SLOW • INDETERMINATE **Note:** The fault type is with regard to the faulty scan chain output.

The following example shows a report when no scan pattern failures are present.

```
tracking_info_begin

defect_info_begin
   injected_fault     STUCK    upintercomp/compval_reg3/QN
   faulty_chain       chain1
   triggering_prob    100
   cell_id            10
defect_info_end

tracking_info_end

#diagnosis_modes=1
#chain_symptoms=1
#total_symptoms=1 #total_suspects=0 total_CPU_time=0.00sec

#symptoms=1 #suspects=0 CPU_time=0.00sec fail_log=fa_stuck_at_1_chain
symptom=1 #suspects=0 faulty_chain=chain1 fault_type=STUCK
//  It might have 10 hold time (fast) faults on chain 1.
//  9 hold time (fast) faults could be intermittent fault.
...
```

Symptom Descriptions

For each symptom associated with a particular faulty chain, the tool lists the symptom ID, number of corresponding suspects, and an information table.

The information table describes the following information for each suspect:

- Suspect — Lists the tool-assigned identification number (ID) of the suspect. The suspect ID resets to 1 for each symptom.
- Score — Indicates the ranking of a suspect. See "Suspect Scores in Chain Diagnosis."
- Type — Indicates the fault type for the faulty chain as follows:

  STUCK. It has the following three scenarios:

  - STUCK(CELL+OUT): Stuck fault if the tool can distinguish a suspect is at scan_in or scan_out. This type indicates the defect could be one of the following:
    - Inside the cell.
    - At the scan_out pin of the cell.
    - On the scan path from the scan_out pin of the cell to the scan_in of the next cell.
  - STUCK(IN+CELL): Stuck fault if the tool can distinguish a suspect is at scan_in or scan_out. This type indicates the defect could be one of the following:
    - Inside the cell.
    - At the scan_in pin of the cell.
    - On the scan path from the scan_in pin of the cell to the scan_out of the previous cell.
  - STUCK(IN+CELL+OUT): Stuck fault if the tool cannot distinguish a suspect is at scan_in or scan_out. This type indicates the defect could be one of the following:
    - At the scan_in pin of the cell or on the scan path from the scan_in pin of the cell to the scan_out pin of the previous cell.
    - Inside the cell.
    - At the scan_out pin of the cell or on the scan path from the scan_out pin of the cell to the scan_in of the next cell.

  FAST_TO_RISE: The fault causes a 0-to-1 transition to occur faster than expected. It has one scenario:

  - (Source Cell:FAST_TO_RISE(CELL+OUT))/(Sink Cell:FAST_TO_RISE(IN+CELL)) pair: The fault causes a 0-to-1 transition to occur faster than expected and indicates the defect could be one of the following:
    - Inside the cell (for example, clock-data race).

- On the scan path from the scan_out pin of the source cell to the scan_in pin of the sink cell (for example, the shift path delay is smaller than expected).

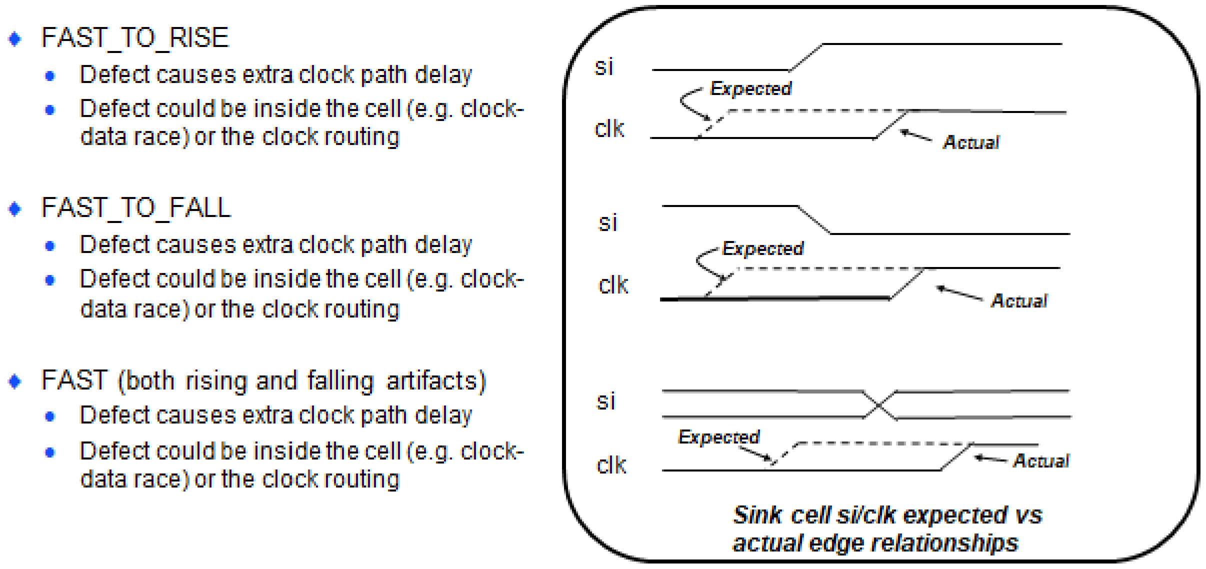
FAST_TO_FALL: The fault causes a 1-to-0 transition to occur faster than expected. It has one scenario:

- o (Source Cell:FAST_TO_FALL(CELL+OUT))/(Sink Cell:FAST_TO_FALL(IN+CELL)) pair: The fault causes a 1-to-0 transition to occur faster than expected and indicates the defect could be one of the following:
  - Inside the cell (for example, clock-data race).
  - On the scan path from the scan_out pin of the source cell to the scan_in pin of the sink cell (for example, the shift path delay is smaller than expected).

FAST: The fault causes both 0-to-1 and 1-to-0 transitions to occur faster than expected. It has one scenario:

- o (Source Cell:FAST(CELL+OUT))/(Sink Cell:FAST(IN+CELL)) pair: The fault causes 0-to-1 and 1-to-0 transitions to occur faster than expected and indicates the defect could be one of the following:
  - Inside the cell (for example, clock-data race).
  - On the scan path from the scan_out pin of the source cell to the scan_in pin of the sink cell (for example, the shift path delay is smaller than expected).

Figure 2-2. Chain Fault Types FAST_TO...



- FAST_TO_RISE
  - Defect causes extra clock path delay
  - Defect could be inside the cell (e.g. clock-data race) or the clock routing
- FAST_TO_FALL
  - Defect causes extra clock path delay
  - Defect could be inside the cell (e.g. clock-data race) or the clock routing
- FAST (both rising and falling artifacts)
  - Defect causes extra clock path delay
  - Defect could be inside the cell (e.g. clock-data race) or the clock routing

SLOW_TO_RISE: The fault causes a 0-to-1 transition to occur slower than expected. It has one scenario:

- SLOW_TO_RISE(IN+CELL): The fault causes a 0-to-1 transition to occur slower than expected and indicates the defect could be one of the following:
  - Inside the cell (for example, clock-data race).
  - On the scan path from the scan_in pin of the cell to the scan_out of the previous cell (for example, the shift path delay is larger than expected).

SLOW_TO_FALL: The fault causes a 1-to-0 transition to occur slower than expected. It has one scenario:

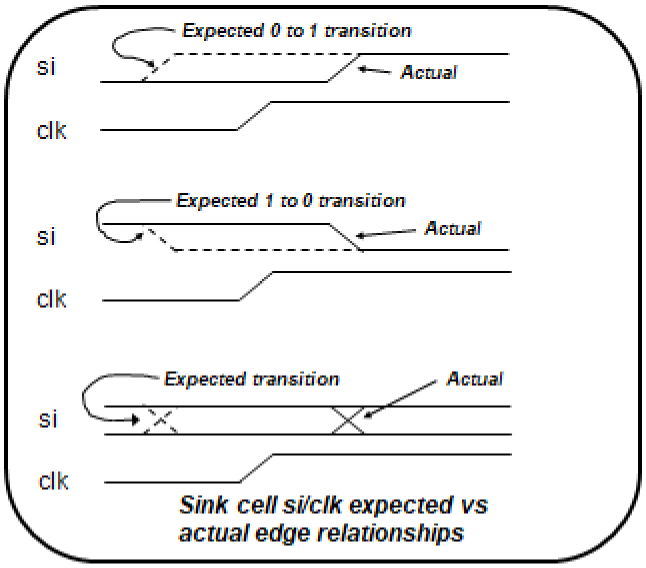
- SLOW_TO_FALL(IN+CELL): The fault causes a 1-to-0 transition to occur slower than expected and indicates the defect could be one of the following:
  - Inside the cell (for example, clock-data race).
  - On the scan path from the scan_in pin of the cell to the scan_out of the previous cell (for example, the shift path delay is larger than expected).

SLOW: The fault causes both 0-to-1 and 1-to-0 transitions to occur slower than expected. It has one scenario:

- SLOW(IN+CELL): The fault causes a 0-to-1 and 1-to-0 transition to occur slower than expected and indicates the defect could be one of the following:
  - Inside the cell.
  - On the scan path from the scan_in pin of the cell to the scan_out of the cell (for example, the shift path delay is larger than expected).

Figure 2-3. Chain Fault Types SLOW_TO...

- SLOW_TO_RISE
  - Defect causes extra rising scan path delay
  - Could be inside the cell (e.g. clock-data race) or on the path between the scan_out pin of one cell to scan_in of the following cell
- SLOW_TO_FALL
  - Defect causes extra rising scan path delay
  - Could be inside the cell (e.g. clock-data race) or on the path between the scan_out pin of one cell to scan_in of the following cell
- SLOW (*both rising and falling artifacts*)
  - Defect causes extra rising and falling scan path delay
  - Could be inside the cell (e.g. clock-data race) or on the path between the scan_out pin of one cell to scan_in of the following cell


Expected 0 to 1 transition
si
Actual
clk
Expected 1 to 0 transition
si
Actual
clk
Expected transition
Actual
si
clk
Sink cell si/clk expected vs actual edge relationships

INDETERMINATE: Diagnosis cannot classify the fault into any of the preceding types. It has three scenarios:

- INDETERMINATE(CELL+OUT): This type indicates the defect could be one of the following:
    - Inside the cell.
    - At the scan_out pin of the cell.
    - On the scan path from the scan_out pin of the cell to the scan_in of the next cell.
- INDETERMINATE(IN+CELL): This type indicates the defect could be one of the following:
    - Inside the cell.
    - At the scan_in pin of the cell.
    - On the scan path from the scan_in pin of the cell to the scan_out of the previous cell.
- INDETERMINATE(IN+CELL+OUT): Indeterminate fault if the tool cannot distinguish a suspect is at scan_in or scan_out. This type indicates the defect could be one of the following:
    - At the scan_in pin of the cell or on the scan path from the scan_in pin of the cell to the scan_out pin of the previous cell.
    - Inside the cell.
    - At the scan_out pin of the cell or on the scan path from the scan_out pin of the cell to the scan_in of next cell.

- Value — Indicates the faulty value, 0, 1 or both.
- Pin_pathname — Pin pathname to the location of the suspect. Pin pathname to the location of the suspect. If the tool cannot distinguish a suspect is at scan_in or scan_out, for example STUCK(IN+CELL_OUT), the scan_out pins are reported in the pin_pathname column. Keep in mind that PFA should also check scan-in and cell internal signals.
- Cell_name — Name of the cell associated with the suspect. The cell name is the top-level library cell name rather than the model name of scan cell inside the library cell.
- Net_pathname — Net pathname to the location of the suspect. If the tool cannot distinguish a suspect is at scan_in or scan_out, for example STUCK(IN+CELL_OUT), the scan_out net is reported in the net_pathname column. Keep in mind that PFA should also check scan-in and cell internal signals.
- Cell_number — Number of the cell associated with a suspect. For example, a cell_number of 0 would be the cell closest to the scan output pin.

- Chain_name — Name of the faulty chain.
- Memory_type — Type name of the reported scan latch (for example, MASTER, REMOTE, COPY). The tool does not report SHADOW. If any logic gates are between the scan cells or between latches within one scan cell, then the tool does not report these.
- Shift_clock — Clock name used for shifting each scan latch.
- Lib_internal_pathname — Scan latch instance pathname inside a library.
- Lib_internal_pathname reporting is off by default: you activate the reporting by specifying the -CHAIN_LIB_internal_pathname ON switch and literal with the set_diagnosis_options or set_diagnosis_options command.

Suspect Scores in Chain Diagnosis

Suspects found during chain diagnosis are scored based on many factors. To obtain a score of 100, the tool needs to have enough data and no mismatches.

Suspect scores are based on:

- The statistical probability that a suspect cell is an actual defect location. The tool uses a proprietary statistical formula to calculate this probability. The higher the probability, the higher the score.
- Population of data that can be used in calculating the statistics. The greater the population, the higher the score.
- Simulation match and mismatch data. The more simulation matches, the higher the score.

Layout Information in the Chain Diagnosis Report

The chain diagnosis report includes chain failure layout information that you can find in two XMAP tables.

The two XMAP tables display the following information:

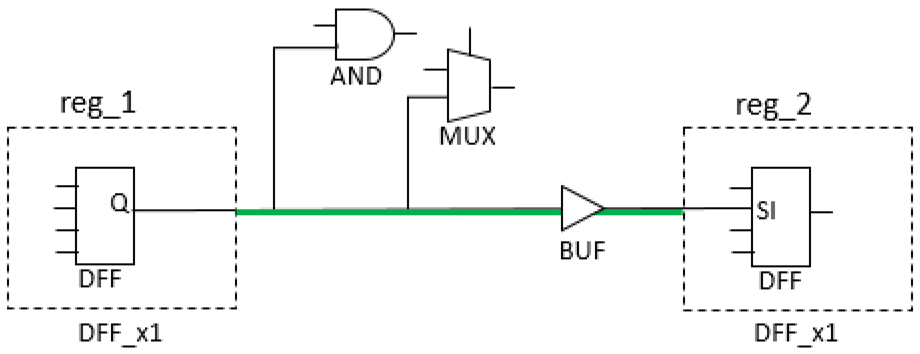
- CELL_LOCATION: displays the cell polygons for the associated suspect.
- OPEN_LOCATION: displays the net polygons for the associated suspect.

You can import the chain failure results into Tessent YieldInsight and view them in the layout viewer.

Note
This layout information is only available for scan chain suspects, not for scan_enable and clock fault suspects.

The tool reports layout polygons of nets' branches that belong directly to the suspect's scan path, as highlighted in green in the following figure.

Figure 2-4. Layout Polygon Reporting



The following example shows a chain diagnosis report.

Example 2-1. Chain Diagnosis Report Example

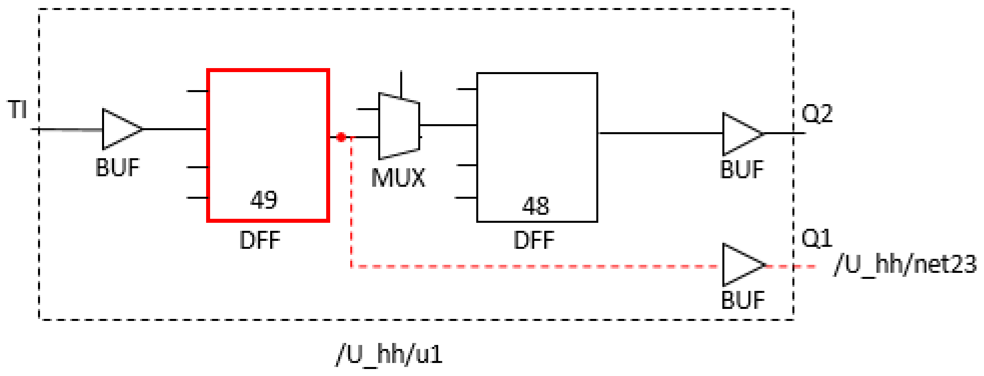
```
suspect  score type             value  pin_pathname cell_name net_pathname cell_number   \
chain_name memory_type shift_clock
-------------------------------------------------------------------------------------------
1        95    STUCK(CELL+OUT)     0     /cpu_i/uCNTR/ix1103/Q sff /cpu_i/uCNTR/$foo[4] 10 \
chain1 MASTER /CLK3        -----------------------------------------------------------------
-------------------------
XMAP_TABLE_BEGIN
1.0
UNITS_DISTANCE_MICRONS 1000
DIEAREA -4 -4 8881.69 9328
 CELL_LOCATION_BEGIN
  symptom suspect layout_layer category critical_area x_coord1  y_coord1 x_coord2  y_coord2
  1       1       CELL         NA       NA            4410.5000 304.5000 4611.0000 424.5000
 CELL_LOCATION_END
 OPEN_LOCATION_BEGIN

symptom suspect layout_layer category critical_area x_coord1  y_coord1 x_coord2  y_coord2
  1       1       route_1      OP       3.26E+02      4431.0000 512.0000 4588.0000 515.0000
  1       1       route_1      OP       1.34E+01      4430.5000 511.5000 4434.5000 515.5000
  1       1       route_1      OP       1.34E+01      4584.5000 511.5000 4588.5000 515.5000
... OPEN_LOCATION_END
```

Multi-Bit Flip-Flop Handling

Chain diagnosis reports pin names of suspect scan cells in scan chains. However, for MBFFs, the suspect scan cells may reside within the MBFF, which is a library-level instance. Pins of scan cells inside MBFFs are unnamed because they are under the library level, so they cannot be reported. In addition, connections inside MBFFs do not have names so the tool cannot report net pathnames either.

When there are suspect scan cells within MBFFs, the tool attempts to report the name of the pin at the MBFF library instance level that connects to the pin of the internal suspect scan cell. In the following example, suppose internal MBFF element 49 is a STUCK(CELL+OUT) suspect.

Figure 2-5. MBFF Internal Scan Chain Suspect Chain Diagnosis Reporting



In the resulting chain diagnosis report, the report traces the path inside MBFF to detect the connection between the pin of the internal suspect and the functional pin of the MBFF instance: pin pathname /U_hh/u1/Q1. In addition, the tool reports the net pathname /U_hh/net23. This is not a direct connection to the suspect scan cell, but it supplies information about which accessible pin propagates the faulty signal.

```
diagnosis_mode=chain
#symptoms=1 #suspects=1 CPU_time=2.87sec fail_log=fal

symptoms=1 #suspects=1 faulty_chain=100 fault_type=STUCK
suspect score type          value pin_        cell_   net_      cell_  chain_  memory_  shift_
                                  pathname    name    pathname number name     type     clock
------------------------------------------------------------------------------------------
1       100   STUCK(CELL+OUT) 1 /U_hh/u1/Q1 cmos281 /U_hh/net23 49   100     MASTER   /clk_st
------------------------------------------------------------------------------------------
```

Pin Tagging to Report MBFF Internal Suspect Scan Cell Layout Data

For layout data, the tool cannot report the physical layout data for internal suspect scan cells unless you enhance the LEF files and then (re-)generate the LDB. To enhance the LEF file, add virtual ports to the MBFF instance definition. The following example adds virtual port out_0 to internal element 49.

```
END
PIN out_0
  DIRECTION OUTPUT ;
  PORT
    LAYER metal1 ;
    RECT 3.07 0.9 3.082 0.905 ;
  END
END out_0
```

Do not connect the virtual pins to nets in the DEF file; they only function to provide additional layout data. The polygon of each pin should correspond to the actual layout coordinates of pins inside the MBFF instance.

Note

Boundary pins inside MBFFs do not have virtual names assigned to them because they are assigned names at the library level. In the figure, the input pin for element 49 is TI and the output pin for element 48 is Q2.

The tool ignores additional layout data associated with internal scan cells unless you turn on MBFF pin tagging with the set_diagnosis_options command. When you turn on pin tagging, the tool searches for the additional data and includes it in the chain diagnosis report.

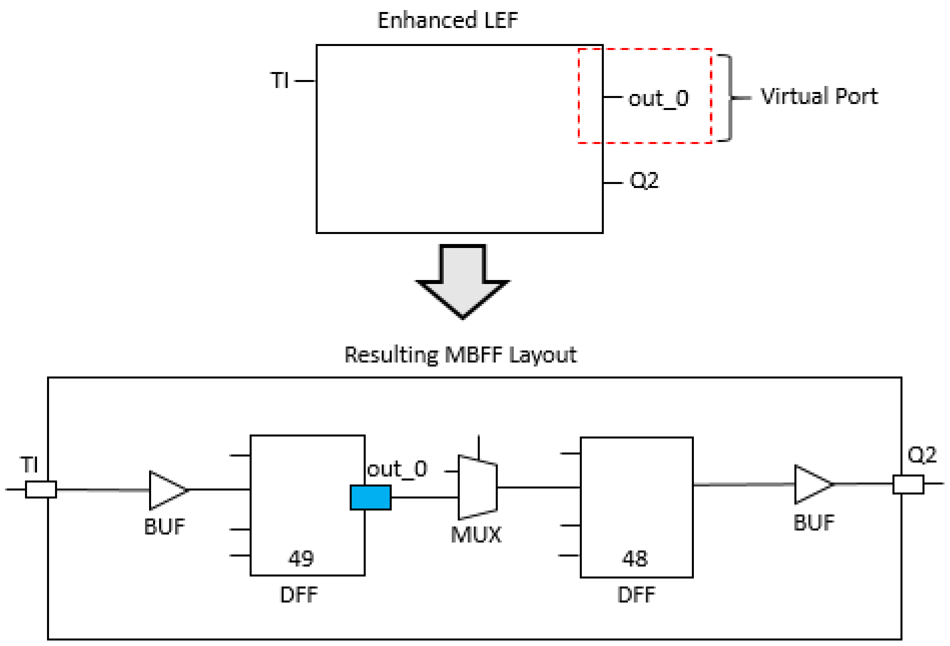
Suppose the chain diagnosis report for the STUCK(CELL+OUT) suspect corresponding to internal element 49 lists the following results in the OPEN_LOCATION section before you turn on pin tagging:

```
OPEN_LOCATION_BEGIN
 symptom suspect layout_layer category critical_area x_coord1 y_coord1 x_coord2 y_coord2
 1       3       metal5       OP       9.62E-03      83.4350  45.7100  83.5750  45.8500
 1       3       metal5       OP       9.62E-03      126.2925 45.7100  126.4325 45.8500
OPEN_LOCATION_END
```

Suppose you turn on pin tagging as follows:

```
set_diagnosis_options -mbff_tag_sci_template in_
set_diagnosis_options -mbff_tag_sco_template out_
```

Figure 2-6. Pin Tagging to Report MBFF Internal Suspect Scan Cell Layout Data



You can turn on tagging for either scan inputs or scan outputs of internal MBFF elements independently. When you specify a template for the scan input, then all internal input pins of scan elements inside the MBFF (excluding the boundary input pin) are assigned virtual names in accordance with sci template. When you specify template for the scan output then all internal output pins of scan elements inside the MBFF (excluding the boundary output pin) are assigned virtual names in accordance with sco template.

The switches set the internal pin tag naming convention for the SCI and SCO pins for elements inside the MBFF in the format "<template><N>", where *N* is the index for an internal element in the MBFF. Indexing starts from 0. During diagnosis, the tool checks whether pins of internal MBFF elements were assigned virtual names. If so, the tool checks the LDB for layout polygon data associated with the virtual names.

If the template name you specify does not match the virtual SCI and SCO names in the enhanced LEF file, the tool returns no additional data for pins of internal scan elements inside the MBFF.

The following example shows the OPEN_LOCATION section with the additional pin polygon information for the STUCK(CELL+OUT) suspect. The line in bold font corresponds to the coordinates for out_1.

```
OPEN_LOCATION_BEGIN
 symptom suspect layout_layer category critical_area x_coord1 y_coord1 x_coord2 y_coord2
 1       3       metal5       OP       9.62E-03      83.4350  45.7100  83.5750  45.8500
 1       3       metal5       OP       9.62E-03      126.2925 45.7100  126.4325 45.8500
 1       10      metal1       OP       5.18E-02      128.1010 73.0385  128.1130 73.0435
OPEN_LOCATION_END
```

You can also turn on MBFF pin tagging when generating reports with the report_scan_path command as described in the *Tessent Shell Reference Manual*.

Global Signal Suspect Reporting

Tessent Diagnosis detects and reports potential suspects for global signals (for example, scan_enable or a clock tree).

Tessent Diagnosis reports the following depending on the global signal:

- scan_enable — STUCK 0
- clock fault — INDETERMINATE both

Figure 2-7 shows the chain diagnosis report syntax Tessent Diagnosis uses when reporting these suspects.

Figure 2-7. Global Signal Suspect Diagnosis Report Format

```
suspect  score  type   value  pin_pathname     cell_name net_pathname
1        60     STUCK  0      /.../U350_112/O  BUF       /.../U350_112/nx456
```

The diagnosis of faults on the scan clock tree or scan enable tree is limited to one fault. If the scan enable signal has a defect, it must make the scan cells affected by the defect always take the value from the system data input, so chain patterns fail.

The following example shows how a scan enable defect is reported:

```
#diagnosis_modes=2
#scan_enable_symptoms=1 #chain_symptoms=1
#total_symptoms=2 #total_suspects=3 total_CPU_time= 667.28sec


diagnosis_mode=scan_enable
#symptoms=1 #suspects=1 CPU_time=50sec fail_log=fa5.cyc.ya

symptom=1 #suspects=1 faulty_scan_enable=scan_en fault_type=STUCK
suspect  score  type           value  pin_pathname cell_name net_pathname
-------------------------------------------------------------------------
1        100    STUCK           0      /U102/Z      DEL05     /n70
-------------------------------------------------------------------------
diagnosis_mode=chain
#symptoms=1 #suspects=2 CPU_time=617.28sec fail_log=fa5.cyc.ya


symptom=2 #suspects=2 faulty_chain=chain59  fault_type=STUCK

suspect score type           value pin_pathname   cell_name net_pathname
cell_number chain_name memory_type shift_clock
-------------------------------------------------------------------------
1     89   STUCK(IN+CELL+OUT) 0 /...regx10x/SO FD2TQHV  /.../n31    255
chain59    MASTER      /TCK
2     89   STUCK(IN+CELL)    0 /...regx17x/TI FD2TQHV  /.../n31    254
chain59    MASTER      /TCK
-------------------------------------------------------------------------
PIN_TAGS_BEGIN
1.0
......
PIN_TAGS_END


NET_TAGS_BEGIN
1.0
......
NET_TAGS_END
```

Masking Scan Patterns for Chain Diagnosis Failures

When diagnosing chain failures, you can improve the diagnosis resolution of the failure by using masking patterns. A masking pattern masks other internal chains and enables only a small number of internal chains to be observed. If no masking patterns or only a few masking patterns exist in the pattern set, it may not be possible to isolate the failing scan chain associated with a specific EDT channel. The ability to isolate the failing scan chain can be improved by including additional masking patterns.

The additional masking patterns are created by using the iterative diagnosis create_diagnosis_patterns command. In general, the more diagnosis patterns you create, the better the resolution. See "Diagnosis Improvements and Retrieving Internal Scan Cell Information" for complete information.

Chain Diagnosis Reporting for Failure Files Missing Failing Scan Test Information

For chain diagnosis, both failing chain test patterns and failing scan test patterns are required. In some cases, the failure file is either missing or contains insufficient failing scan pattern information to perform diagnosis. In those cases, chain diagnosis analysis is limited and affects the diagnosis report produced.

Three such failure file scenarios and the resulting chain diagnosis reporting are described below:

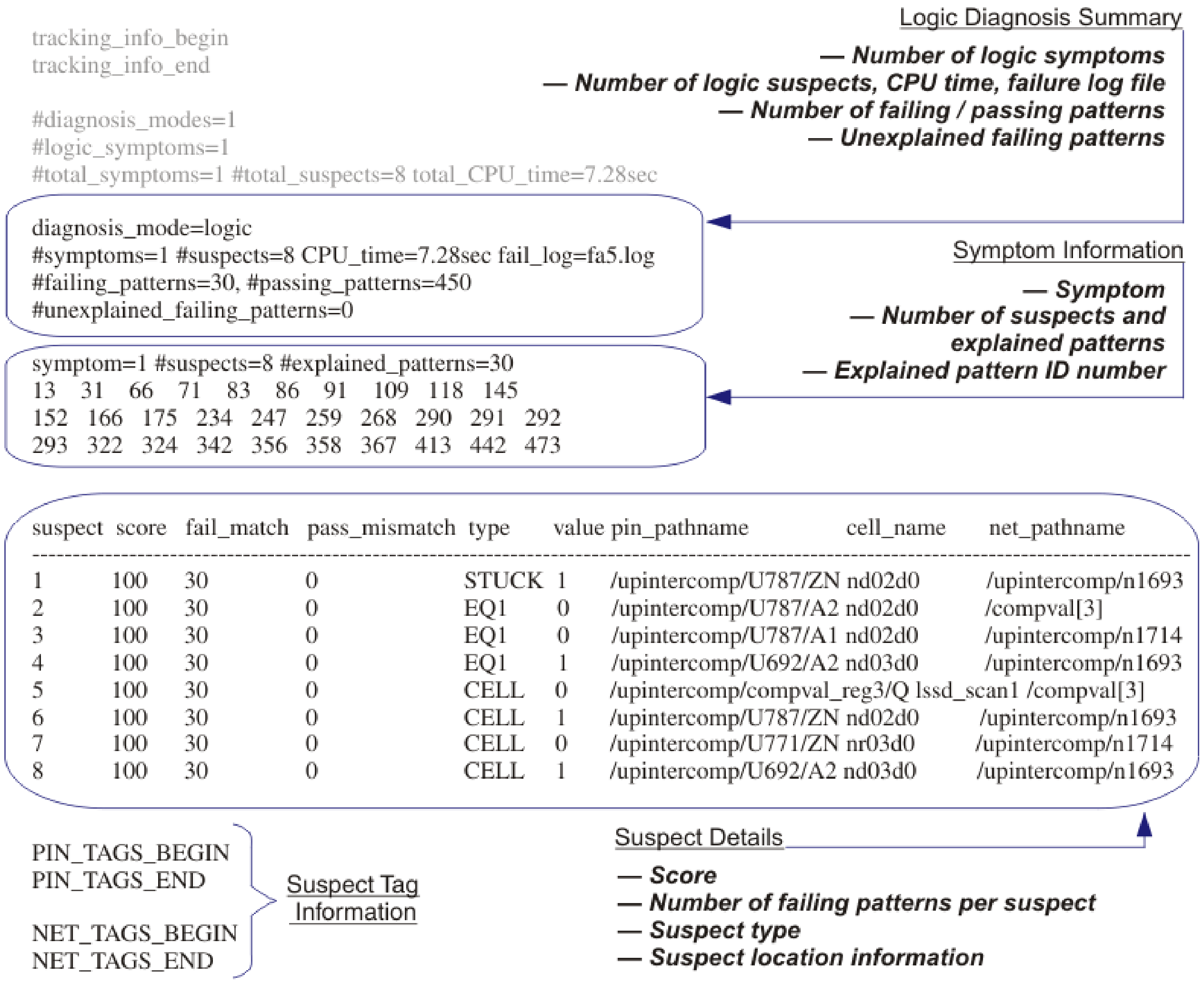
1. Scenario 1: Failing scan patterns are not logged (only failing chain test patterns are logged).
    a. Point tool diagnosis: Failing chain is reported (current behavior). Additional message regarding missing failing scan patterns in failure file is reported.
    b. Dynamic partitioning flow: Behavior is the same as point tool.
2. Scenario 2: Failing scan patterns do not contain a capture cycle.
    a. Point tool diagnosis: Failing chain is reported. Additional message regarding shortcoming of failing scan patterns in failure file is reported.
    b. Dynamic partitioning flow: Behavior is the same as point tool.
3. Failing scan patterns are logged (with capture patterns) but are unrelated to failing chain or compactor output.
    a. Point tool diagnosis: Chain diagnosis is performed but a warning message about impact to diagnosis resolution due to failure file shortcomings is reported.
    b. Dynamic partitioning flow: Chain diagnosis is aborted for the failure file with a message about the insufficiency of the failure file data.

Logic Diagnosis Section

The logic diagnosis section of the diagnosis report contains a logic diagnosis summary, symptom information, and suspect details.

The following example shows a logic diagnosis section.

Figure 2-8. Example Logic Diagnosis Section



```
tracking_info_begin
tracking_info_end

#diagnosis_modes=1
#logic_symptoms=1
#total_symptoms=1 #total_suspects=8 total_CPU_time=7.28sec

diagnosis_mode=logic
#symptoms=1 #suspects=8 CPU_time=7.28sec fail_log=fa5.log
#failing_patterns=30, #passing_patterns=450
#unexplained_failing_patterns=0

symptom=1 #suspects=8 #explained_patterns=30
13   31   66   71   83   86   91   109  118  145
152  166  175  234  247  259  268  290  291  292
293  322  324  342  356  358  367  413  442  473

suspect  score  fail_match  pass_mismatch  type    value pin_pathname              cell_name   net_pathname
--------------------------------------------------------------------------------------------------------------
1        100    30          0              STUCK  1     /upintercomp/U787/ZN nd02d0        /upintercomp/n1693
2        100    30          0              EQ1     0     /upintercomp/U787/A2 nd02d0        /compval[3]
3        100    30          0              EQ1     0     /upintercomp/U787/A1 nd02d0        /upintercomp/n1714
4        100    30          0              EQ1     1     /upintercomp/U692/A2 nd03d0        /upintercomp/n1693
5        100    30          0              CELL    0     /upintercomp/compval_reg3/Q lssd_scan1 /compval[3]
6        100    30          0              CELL    1     /upintercomp/U787/ZN nd02d0        /upintercomp/n1693
7        100    30          0              CELL    0     /upintercomp/U771/ZN nr03d0        /upintercomp/n1714
8        100    30          0              CELL    1     /upintercomp/U692/A2 nd03d0        /upintercomp/n1693

PIN_TAGS_BEGIN
PIN_TAGS_END

NET_TAGS_BEGIN
NET_TAGS_END
```

Logic Diagnosis Summary
— Number of logic symptoms
— Number of logic suspects, CPU time, failure log file
— Number of failing / passing patterns
— Unexplained failing patterns
Symptom Information
— Symptom
— Number of suspects and explained patterns
— Explained pattern ID number
Suspect Details
— Score
— Number of failing patterns per suspect
— Suspect type
— Suspect location information
Suspect Tag Information

Logic Diagnosis Summary

The logic diagnosis header summarizes the diagnosis details, including the total number of symptoms, and suspects; CPU time; the failure file path; number of failing patterns; number of passing patterns; and the number of unexplained patterns.

Table 2-3 describes the logic diagnosis section summary elements.

Table 2-3. Logic Diagnosis Section Summary Elements

Report Field	Description
diagnosis_mode=logic	Identifies the Logic Diagnosis section of the report.
#symptoms	Number of logic symptoms diagnosed. Each symptom represents a group of failure information (fail subsets) that is linked to a single probable source defect. Specific failure information may be linked to multiple symptoms.
#suspects	Number of suspects diagnosed. A suspect is a circuit location determined to be a possible cause of a symptom.
CPU_time	CPU time used for the diagnosis.
fail_log	Name of the failure log associated with the diagnosis.
#failing_patterns	Number of failing test patterns in the failure log.
#passing_patterns	Number of passing test patterns in the failure log.
#unexplained_failing_patterns	Number of failing patterns that cannot be explained. The number for each unexplained failing pattern is also printed.

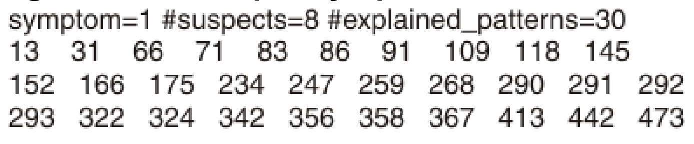
Symptom Information Section

The symptom information section immediately follows the diagnostic summary, and it lists symptom, suspect, and failing pattern information.

- symptom — Unique number that identifies the symptom.
- #suspects — Number of suspects identified for the symptom.
- #explained_patterns — Number of failing patterns explained by the symptom along with their identification (ID) number.

Figure 2-9 shows a representative symptom information section of the report.

Figure 2-9. Example Symptom Information Section



```
symptom=1 #suspects=8 #explained_patterns=30
13   31   66   71   83   86   91   109  118  145
152  166  175  234  247  259  268  290  291  292
293  322  324  342  356  358  367  413  442  473
```

Suspect Details

The suspect details section immediately follows the symptom information section, and it lists symptom, score, fail and pass match, and type information.

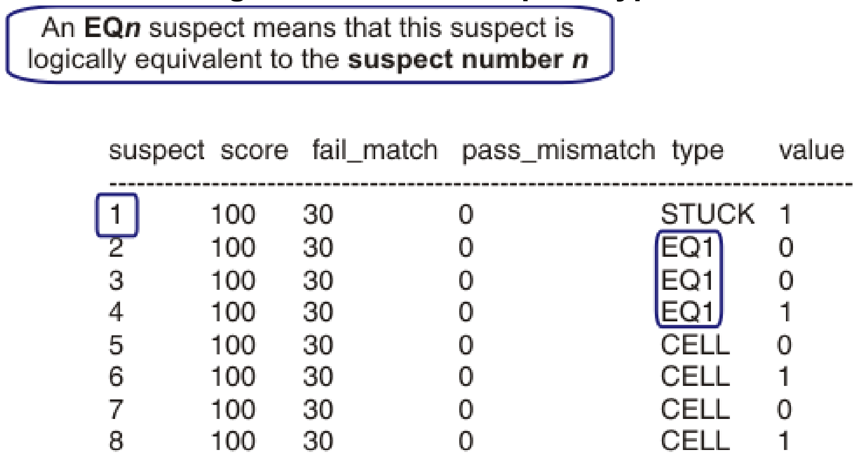
- suspect — ID number of the suspect. The suspect ID numbers restart at 1 for each symptom.
- score — Computed score for each suspect. The higher the score, the more likely the suspect is the cause of the symptom—see "Suspect Scores in Logic Diagnosis."
- fail_match — Number of failing patterns that can be explained by the suspect.
- pass_mismatch — Number of passing patterns that cannot be explained by the suspect.
- type — Specifies a suspect type. Table 2-4 lists each type and the corresponding fault type.

Table 2-4. Logic Diagnosis Suspect Types

Type	Fault Type
STUCK	Stuck-at fault.
CELL	Cell internal fault. The pin_pathname is one of the pins of the cell with an internal defect.
SLOW	Transition fault when running at-speed.
BRIDGE_2WAY	Two-way bridge fault. Each suspect of this type has two components with the same suspect ID.
BRIDGE_3WAY	Three-way bridge fault. Each suspect of this type has three components with the same suspect ID.
OPEN/DOM	Either an open or a dominant bridge victim with unknown aggressor.
RES_OPEN	Indicates the open suspect behaves as a resistive open rather than a complete open.
INDETERMINATE	Diagnosis cannot classify the fault into any of the preceding types.
EQ	Indicates the suspect type is equivalent to the suspect that precedes it in the report. See Figure 2-10; applies exclusively to the STUCK, INDETERMINATE, and OPEN/DOM suspect types.

The following figure shows what the output looks like when you have an EQ suspect type.

Figure 2-10. EQn Suspect Type



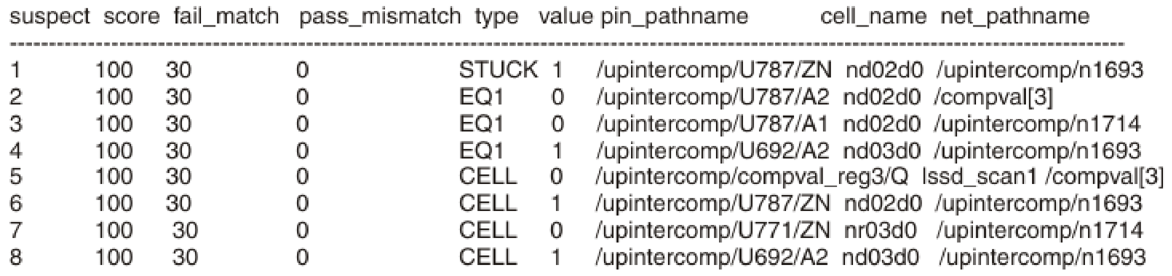
An **EQ*n*** suspect means that this suspect is logically equivalent to the **suspect number *n***

```
suspect  score  fail_match  pass_mismatch  type      value
-------------------------------------------------------------------------
1        100    30          0              STUCK     1
2        100    30          0              EQ1       0
3        100    30          0              EQ1       0
4        100    30          0              EQ1       1
5        100    30          0              CELL      0
6        100    30          0              CELL      1
7        100    30          0              CELL      0
8        100    30          0              CELL      1
```

- value — Indicates a fault value associated with the suspect, which can be 0, 1, or both. For transition faults (SLOW type), the value can be rise for slow-to-rise, fall for slow-to-fall, or both for slow-at-both transitions.
- pin_pathname — Pin pathname to the location of the suspect.
- cell_name — Name of the cell associated with the suspect.
- net_pathname — Net pathname to the location of the suspect.
- layout_status — Layout aware diagnosis reports the marker location in the LDB in this column. This column does not appear for logic-only diagnosis.

Figure 2-11 shows an example suspect details section of a logic diagnosis report.

Figure 2-11. Example Suspect Details Diagnosis Report Information



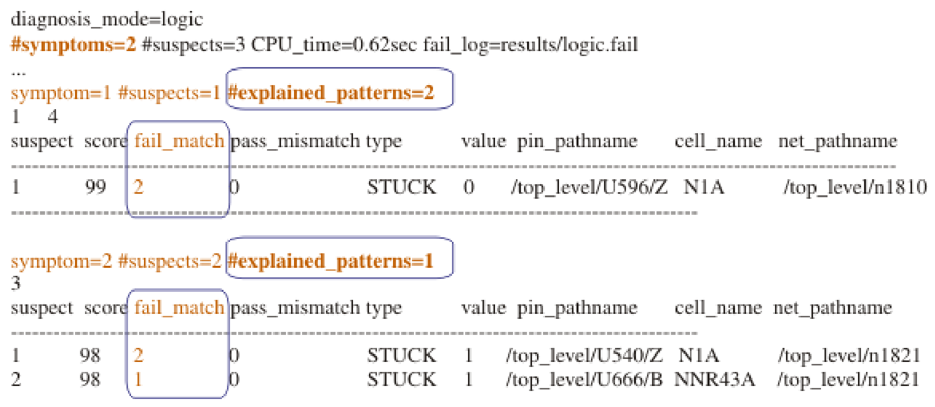
```
suspect  score  fail_match  pass_mismatch  type   value pin_pathname           cell_name  net_pathname
------------------------------------------------------------------------------------------------------------
1        100    30          0              STUCK  1     /upintercomp/U787/ZN   nd02d0  /upintercomp/n1693
2        100    30          0              EQ1    0     /upintercomp/U787/A2   nd02d0  /compval[3]
3        100    30          0              EQ1    0     /upintercomp/U787/A1   nd02d0  /upintercomp/n1714
4        100    30          0              EQ1    1     /upintercomp/U692/A2   nd03d0  /upintercomp/n1693
5        100    30          0              CELL   0     /upintercomp/compval_reg3/Q  lssd_scan1 /compval[3]
6        100    30          0              CELL   1     /upintercomp/U787/ZN   nd02d0  /upintercomp/n1693
7        100     30         0              CELL   0     /upintercomp/U771/ZN   nr03d0  /upintercomp/n1714
8        100     30         0              CELL   1     /upintercomp/U692/A2   nd03d0  /upintercomp/n1693
```

Explained Patterns and Fail Match

For single defect failing cases, to explain a failing pattern, a suspect needs to be excited and propagate to all observe points with the exact failing behavior as observed on the ATE.

For multiple defect failing cases, a suspect can team up with other suspects to jointly explain a failing pattern if their combined failing behavior matches what is observed on the ATE. In this case, a symptom's #explained_patterns entry may not match its suspect's fail_match entry. The following figure illustrates a multiple defect failing case and shows the diagnosis report for a die with two defect locations and three failing patterns, one of which activates both defects. In the example, Tessent Diagnosis produces two symptoms and only reports this failing pattern in one of the symptoms.

Figure 2-12. Explained Patterns Different Than Fail_Match



```
diagnosis_mode=logic
#symptoms=2 #suspects=3 CPU_time=0.62sec fail_log=results/logic.fail
...
symptom=1 #suspects=1 #explained_patterns=2
1   4
suspect  score fail_match pass_mismatch type       value  pin_pathname      cell_name  net_pathname
----------------------------------------------------------------------------------------------------
1         99   2          0            STUCK     0     /top_level/U596/Z  N1A        /top_level/n1810
------------------------------------------------------------------------------

symptom=2 #suspects=2 #explained_patterns=1
3
suspect  score fail_match pass_mismatch type       value  pin_pathname      cell_name  net_pathname
------------------------------------------------------------------------------
1        98    2          0            STUCK     1     /top_level/U540/Z  N1A        /top_level/n1821
2        98    1          0            STUCK     1     /top_level/U666/B  NNR43A     /top_level/n1821
```

Consequently, the diagnosis report lists these two symptoms and failing patterns as follows:

- symptom 1 #explained_patterns=2
- symptom 2 #explained_patterns=1

In the diagnosis report example, symptom 1 accounts for the failing pattern that activated both defects, and its suspect has a fail_match=2.

For symptom 2, suspect 1 has a fail_match=2, which accounts for the failing pattern that activated both defects. Symptom 2's suspect 2 has a fail_match=1.

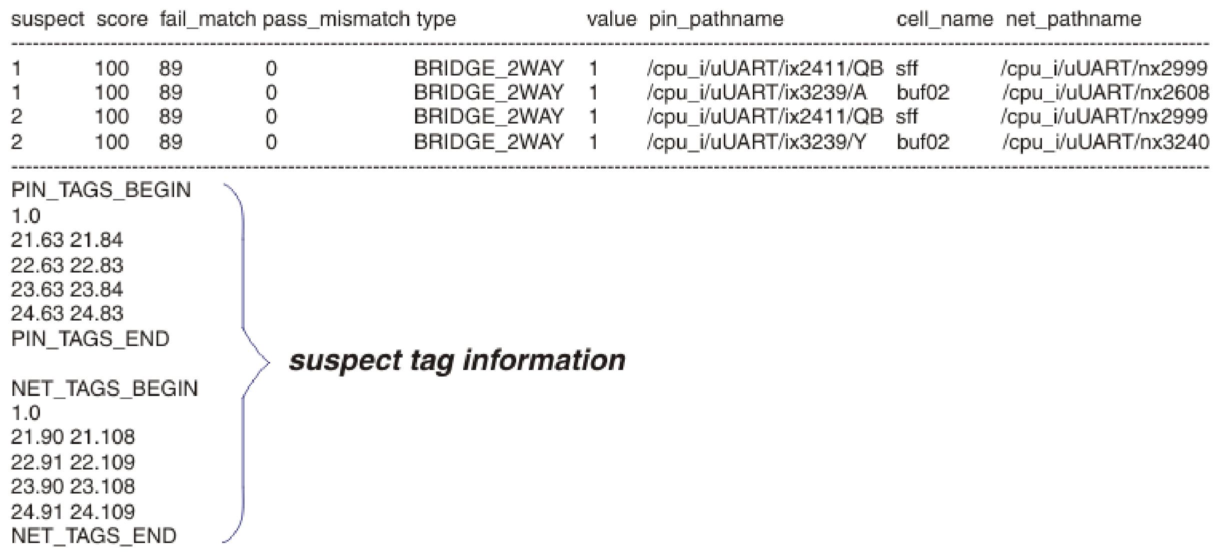
In other words, all suspects in one symptom may not have the same fail_match, but all of the suspects must have a fail_match greater than or equal to the number of patterns explained by the symptom.

Suspect Tags

The suspect tag information is used by Calibre to display suspect locations in the physical layout.

Figure 2-13 shows a typical suspect tags portion of the diagnostic report.

Figure 2-13. Suspect Tags Portion of Diagnostic Report



```
suspect  score  fail_match pass_mismatch type               value  pin_pathname               cell_name  net_pathname
--------------------------------------------------------------------------------------------------------------------------
1        100    89          0            BRIDGE_2WAY   1     /cpu_i/uUART/ix2411/QB  sff         /cpu_i/uUART/nx2999
1        100    89          0            BRIDGE_2WAY   1     /cpu_i/uUART/ix3239/A   buf02       /cpu_i/uUART/nx2608
2        100    89          0            BRIDGE_2WAY   1     /cpu_i/uUART/ix2411/QB  sff         /cpu_i/uUART/nx2999
2        100    89          0            BRIDGE_2WAY   1     /cpu_i/uUART/ix3239/Y   buf02       /cpu_i/uUART/nx3240
--------------------------------------------------------------------------------------------------------------------------
PIN_TAGS_BEGIN
1.0
21.63 21.84
22.63 22.83
23.63 23.84
24.63 24.83
PIN_TAGS_END

NET_TAGS_BEGIN
1.0
21.90 21.108
22.91 22.109
23.90 23.108
24.91 24.109
NET_TAGS_END
```

The information in this section is segmented into the following areas:

- Pin pathnames — The tags in this section are enclosed by the keywords: PIN_TAGS_BEGIN and PIN_TAGS_END. The pin pathname tags for each suspect are listed between these keywords. Each line respectively provides the beginning and ending coordinates within the fail log file for each pin pathname. Each coordinate consists of two integers separated by a period where the first integer is the line number and the second integer is the column number.
- Net pathnames — The tags in this section are enclosed by the keywords: NET_TAGS_BEGIN and NET_TAGS_END. The net pathname tags for each suspect is listed between the keywords. Each line respectively provides the beginning and ending coordinates within the fail log file for each net pathname. Each coordinate consists of two integers separated by a period where the first integer is the line number and the second integer is the column number.

The first line in each section specifies a version number to facilitate tag compatibility—see "Troubleshooting."

Suspect Scores in Logic Diagnosis

Suspects found during logic diagnosis are scored using the following formula. The highest possible score is 100. If the score is less than 1, it is reported as 1. The higher the score, the more likely the suspect is the cause of the symptom.

The final suspect score is calculated as follows:

final_score = 70* (F / (F+P)) + 10* (F / (F+100P)) + 10* (F / (F+1000P)) + 10* F

where:

F = (fail_match/max_fail_match) * (total_fails_explained / total_tester_fails)

P = pass_mismatch / total_tester_pass

fail_match = number of failing patterns explained by a specific suspect.

max_fail_match = maximum number of failing patterns explained by any suspect.

total_fails_explained = total number of failing patterns explained by all suspects.

total_tester_fails = number of failing patterns in the fail file used for diagnosis.

pass_mismatch = number of passing patterns that cannot be explained by one specific suspect.

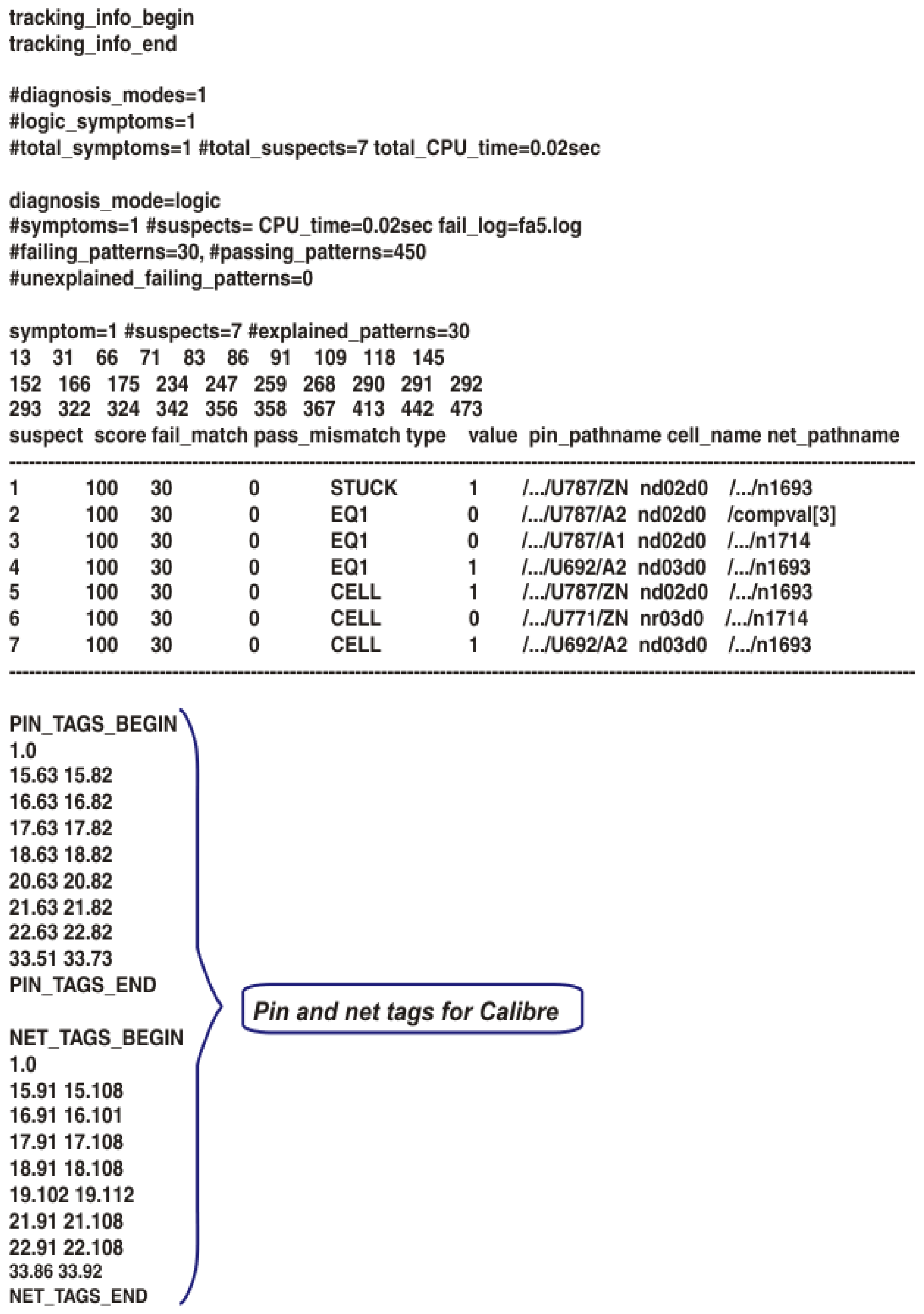
total_tester_pass = number of passing patterns in the fail file used for diagnosis.

Logic Diagnosis Report Examples

The typical logic diagnosis report begins with tracking information and ends with the pin and net tags.

Example 2-14 shows a typical logic diagnosis report.

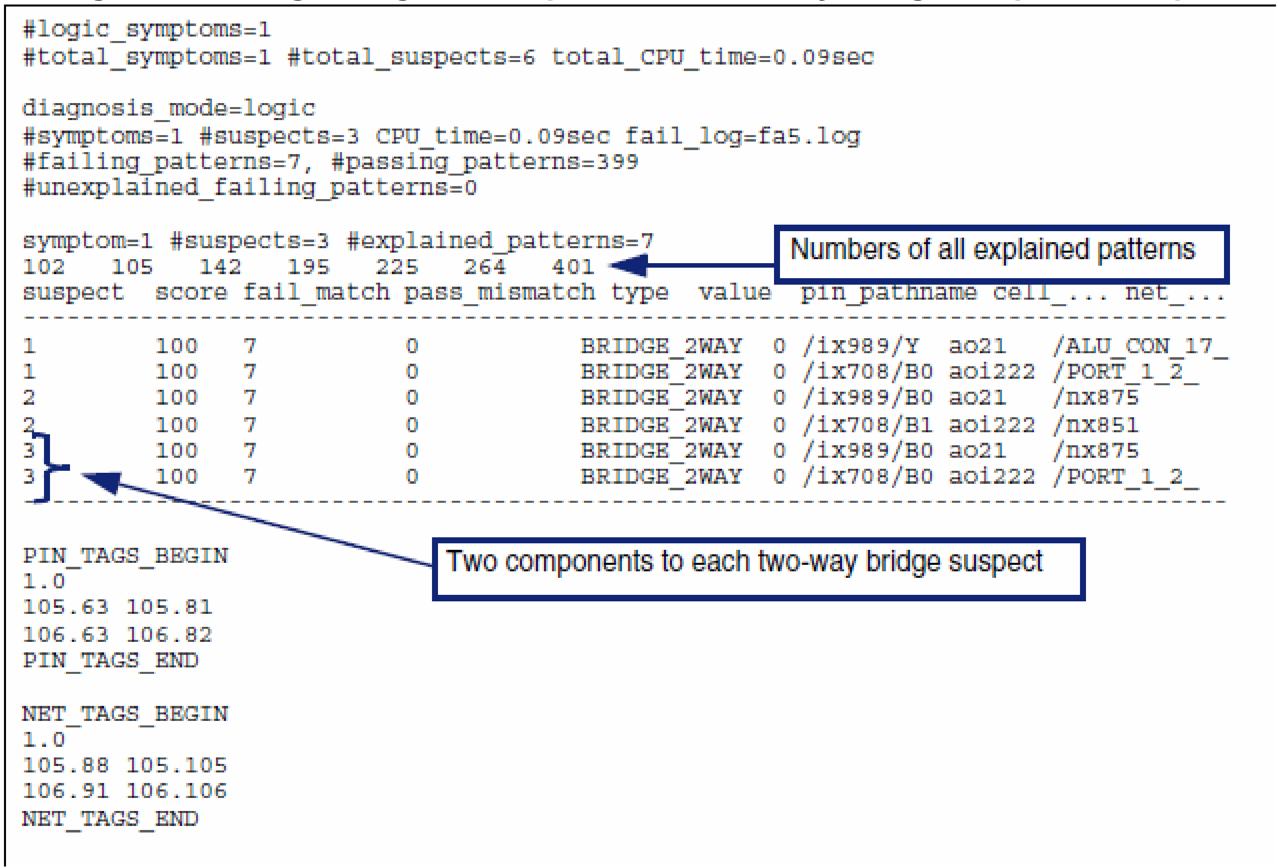
Figure 2-14. Typical Logic Diagnosis Report Example



```
tracking_info_begin
tracking_info_end

#diagnosis_modes=1
#logic_symptoms=1
#total_symptoms=1 #total_suspects=7 total_CPU_time=0.02sec

diagnosis_mode=logic
#symptoms=1 #suspects= CPU_time=0.02sec fail_log=fa5.log
#failing_patterns=30, #passing_patterns=450
#unexplained_failing_patterns=0

symptom=1 #suspects=7 #explained_patterns=30
13   31   66   71   83   86   91   109  118  145
152  166  175  234  247  259  268  290  291  292
293  322  324  342  356  358  367  413  442  473
suspect  score fail_match pass_mismatch type    value  pin_pathname cell_name net_pathname
-----------------------------------------------------------------------------------------------------
1        100    30          0          STUCK        1      /.../U787/ZN  nd02d0   /.../n1693
2        100    30          0          EQ1          0      /.../U787/A2  nd02d0   /compval[3]
3        100    30          0          EQ1          0      /.../U787/A1  nd02d0   /.../n1714
4        100    30          0          EQ1          1      /.../U692/A2  nd03d0   /.../n1693
5        100    30          0          CELL         1      /.../U787/ZN  nd02d0   /.../n1693
6        100    30          0          CELL         0      /.../U771/ZN  nr03d0   /.../n1714
7        100    30          0          CELL         1      /.../U692/A2  nd03d0   /.../n1693
-----------------------------------------------------------------------------------------------------

PIN_TAGS_BEGIN
1.0
15.63 15.82
16.63 16.82
17.63 17.82
18.63 18.82
20.63 20.82
21.63 21.82
22.63 22.82
33.51 33.73
PIN_TAGS_END

NET_TAGS_BEGIN
1.0
15.91 15.108
16.91 16.101
17.91 17.108
18.91 18.108
19.102 19.112
21.91 21.108
22.91 22.108
33.86 33.92
NET_TAGS_END
```

The following example shows a typical logic diagnosis report for a two-way bridge suspect.

Figure 2-15. Logic Diagnosis Report for a Two-Way Bridge Suspect Example



```
#logic_symptoms=1
#total_symptoms=1 #total_suspects=6 total_CPU_time=0.09sec

diagnosis_mode=logic
#symptoms=1 #suspects=3 CPU_time=0.09sec fail_log=fa5.log
#failing_patterns=7, #passing_patterns=399
#unexplained_failing_patterns=0

symptom=1 #suspects=3 #explained_patterns=7
102    105    142    195    225    264    401
suspect  score fail_match pass_mismatch type   value  pin_pathname cell_... net_...
------------------------------------------------------------------------------------
1        100   7           0          BRIDGE_2WAY  0 /ix989/Y  ao21   /ALU_CON_17_
1        100   7           0          BRIDGE_2WAY  0 /ix708/B0 aoi222 /PORT_1_2_
2        100   7           0          BRIDGE_2WAY  0 /ix989/B0 ao21   /nx875
2        100   7           0          BRIDGE_2WAY  0 /ix708/B1 aoi222 /nx851
3        100   7           0          BRIDGE_2WAY  0 /ix989/B0 ao21   /nx875
3        100   7           0          BRIDGE_2WAY  0 /ix708/B0 aoi222 /PORT_1_2_
------------------------------------------------------------------------------------

PIN_TAGS_BEGIN
1.0
105.63 105.81
106.63 106.82
PIN_TAGS_END

NET_TAGS_BEGIN
1.0
105.88 105.105
106.91 106.106
NET_TAGS_END
```

Numbers of all explained patterns
Two components to each two-way bridge suspect

In this example, the value column states 0 for all of the two-way bridge suspects. This means that when these nets fail, they fail to 0. The possible results for this column are:

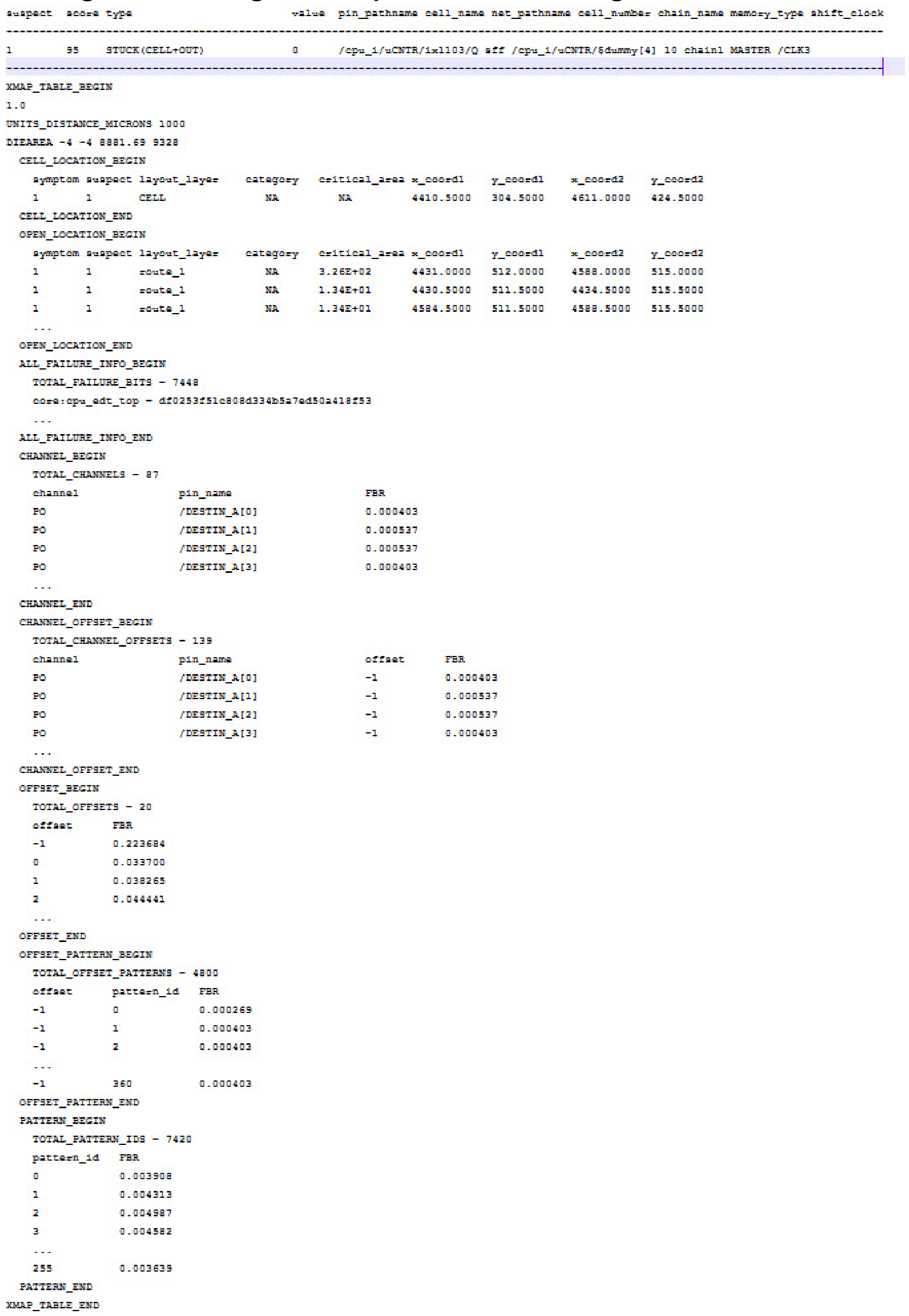
- 0. For all failure patterns, when the net fails, it fails to 0.
- 1. For all failure patterns, when the net fails, it fails to 1.
- Both. For some failure patterns, the net fails to 0, and for other failure patterns, the net fails to 1.

Failure Signature Information in the Diagnosis Report

In Tessent Diagnosis releases v9.2 or later, each diagnosis report includes, by default, the failure signature information enclosed with XMAP_TABLE_BEGIN and XMAP_TABLE_END.

Figure 2-16 shows a diagnosis report with failure signature information.

Figure 2-16. Diagnosis Report With Failure Signature Information



```
suspect  score type                          value  pin_pathname cell_name net_pathname cell_number chain_name memory_type shift_clock
------------------------------------------------------------------------------------------------------------------------------------
1        95    STUCK(CELL+OUT)               0      /cpu_i/uCNTR/ix1103/Q aff /cpu_i/uCNTR/$dummy[4] 10 chain1 MASTER /CLK3
------------------------------------------------------------------------------------------------------------------------------------
XMAP_TABLE_BEGIN
1.0
UNITS_DISTANCE_MICRONS 1000
DIEAREA -4 -4 8881.69 9328
  CELL_LOCATION_BEGIN
    symptom suspect layout_layer    category    critical_area x_coord1     y_coord1     x_coord2     y_coord2
    1       1       CELL              NA           NA         4410.5000    304.5000     4611.0000    424.5000
  CELL_LOCATION_END
  OPEN_LOCATION_BEGIN
    symptom suspect layout_layer    category    critical_area x_coord1     y_coord1     x_coord2     y_coord2
    1       1       route_1           NA        3.26E+02      4431.0000    512.0000     4588.0000    515.0000
    1       1       route_1           NA        1.34E+01      4430.5000    511.5000     4434.5000    515.5000
    1       1       route_1           NA        1.34E+01      4584.5000    511.5000     4588.5000    515.5000
    ...
  OPEN_LOCATION_END
  ALL_FAILURE_INFO_BEGIN
    TOTAL_FAILURE_BITS - 7448
    core:cpu_edt_top - df0253f51c808d334b5a7ed50a418f53
    ...
  ALL_FAILURE_INFO_END
  CHANNEL_BEGIN
    TOTAL_CHANNELS - 87
    channel               pin_name                     FBR
    PO                    /DESTIN_A[0]                 0.000403
    PO                    /DESTIN_A[1]                 0.000537
    PO                    /DESTIN_A[2]                 0.000537
    PO                    /DESTIN_A[3]                 0.000403
    ...
  CHANNEL_END
  CHANNEL_OFFSET_BEGIN
    TOTAL_CHANNEL_OFFSETS - 139
    channel               pin_name                     offset      FBR
    PO                    /DESTIN_A[0]                 -1          0.000403
    PO                    /DESTIN_A[1]                 -1          0.000537
    PO                    /DESTIN_A[2]                 -1          0.000537
    PO                    /DESTIN_A[3]                 -1          0.000403
    ...
  CHANNEL_OFFSET_END
  OFFSET_BEGIN
    TOTAL_OFFSETS - 20
    offset      FBR
    -1          0.223684
    0           0.033700
    1           0.038265
    2           0.044441
    ...
  OFFSET_END
  OFFSET_PATTERN_BEGIN
    TOTAL_OFFSET_PATTERNS - 4800
    offset      pattern_id   FBR
    -1          0            0.000269
    -1          1            0.000403
    -1          2            0.000403
    ...
    -1          360          0.000403
  OFFSET_PATTERN_END
  PATTERN_BEGIN
    TOTAL_PATTERN_IDS - 7420
    pattern_id    FBR
    0             0.003908
    1             0.004313
    2             0.004987
    3             0.004582
    ...
    255           0.003639
  PATTERN_END
XMAP_TABLE_END
```

The failure signature information is intended for use by Tessent YieldInsight to create an Analysis database (ADB) from the diagnosis reports—see "Creating the ADB" in the *Tessent YieldInsight User's Manual*.

By default, each diagnosis report includes the failure signature information with maximum 256 rows per table—see "Failure Signature Format." You can modify the number of rows per table by using either of the following methods depending on the used tool before performing the diagnosis:

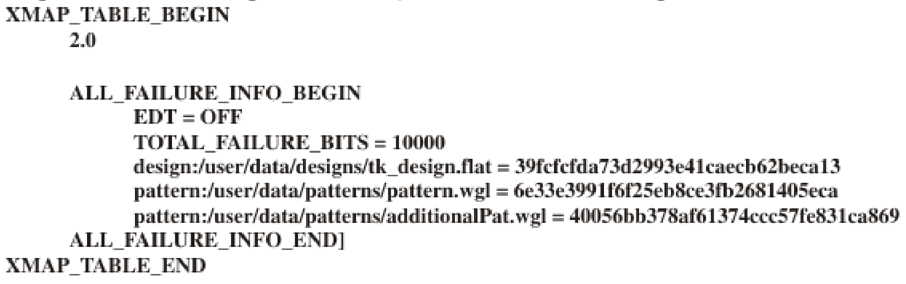
- Tessent Diagnosis — The set_diagnosis_options command's -INCLude_fail_signatures_size switch to specify the number of rows per table.
- Tessent Diagnosis Server — The set_diagnosis_options command's -INCLude_fail_signatures_size switch to specify the number of rows per table.

MD5 Signature Information in the Diagnosis Report

In Tessent Diagnosis releases v9.3 or later, each diagnosis report includes, by default, the MD5 signature information for the pattern or patterns, and flat model Tessent Diagnosis used to generate the report.

The MD5 signature information is enclosed with XMAP_TABLE_BEGIN and XMAP_TABLE_END as shown in Figure 2-17.

Figure 2-17. Diagnosis Report With MD5 Signature Information



```
XMAP_TABLE_BEGIN
        2.0

        ALL_FAILURE_INFO_BEGIN
                EDT = OFF
                TOTAL_FAILURE_BITS = 10000
                design:/user/data/designs/tk_design.flat = 39fcfcfda73d2993e41caecb62beca13
                pattern:/user/data/patterns/pattern.wgl = 6e33e3991f6f25eb8ce3fb2681405eca
                pattern:/user/data/patterns/additionalPat.wgl = 40056bb378af61374ccc57fe831ca869
        ALL_FAILURE_INFO_END]
XMAP_TABLE_END
```

See also "MD5 Signature Format."

The MD5 signature information is intended for use by Tessent YieldInsight when creating an Analysis database (ADB)—see "MD5 Signature Mismatches When Importing Diagnosis Reports" in the *Tessent YieldInsight User's Manual*.

CSV Diagnosis Report Format

When using the Tessent Diagnosis point tool, you can use the diagnose_failures -csv switch to specify the generation of a CSV file. If the same file is used on multiple diagnose_failures commands, Tessent Diagnosis performs concatenation of the CSV information.

When using the Tessent Diagnosis server, set the set diagnostic_CSV variable to true to generate the CSV information. Alternatively, use the add_reporting_format command to generate CSV-formatted reports. These files accumulate in the results directory.

The following table lists the CSV report variables and respective data types.

Table 2-5. CSV Variables and Data Types

Column Title	Data Type	Description
fail_log	string	Name and location of the failure file.
tracking	string	Tracking information from the failure file.
faulty_chains	integer	Number of faulty chains.
symptoms	integer	Total symptom count in the diagnosis result.
suspects	integer	Total suspect count in the diagnosis result.
enclosing_circle_diameter	floating point	Defect enclosing circle—see "Defect Location Information." Corresponds to the diameter around all the bounding boxes for layout layers for the symptom corresponding to the suspected listed in the current row.
symptom	integer	Tool-assigned symptom ID number.
explained_patterns	integer	Number of explained patterns by the symptom.
suspect	integer	Tool-assigned ID number of the suspect.
score	integer	Computed score for each suspect.
fail_match	integer	Number of failing patterns that can be explained by the suspect.
pass_mismatch	integer	Number of passing patterns that cannot be explained by the suspect.
type	string	Suspect type.
value	string	Fault value associated with the suspect.
pin_pathname	string	Name of the pin associated with the suspect.
cell_name	string	Name of the cell associated with the suspect.
cell_number	integer	Number of the cell associated with a suspect.
net_pathname	string	Name of the net associated with the suspect.
chain_name	string	Name of the faulty chain.
memory_type	string	Type name of a reported scan latch.
shift_clock	string	Clock name used for shifting each scan latch.
lib_internal_pathname	string	Name of the scan latch instance pathname inside a library.

Table 2-5. CSV Variables and Data Types (cont.)

Column Title	Data Type	Description
layout_status	string	Layout status.
layout_layer	string	Layer of the bounding box in the current row.
category	string	Defect category—see "The XMAP Table."
critical_area	floating point	Critical areas for a defect bounding box.
x_coord1	floating point	Bounding box coordinates.
y_coord1	floating point	Bounding box coordinates.
x_coord2	floating point	Bounding box coordinates.
y_coord2	floating point	Bounding box coordinates.
cpu_time	string	CPU time used for the diagnosis.
fail_patterns	integer	Number of failing test patterns in the failure file.
passing_patterns	integer	Number of passing test patterns simulated for the diagnosis.
unexplained_fail_patterns	integer	Number of failing patterns that cannot be explained.
tool	string	Tool name used for diagnosis.
version	string	Tessent Diagnosis version number used for the diagnosis.
date	string	Date of the diagnosis run.

Depending on the failure type under diagnosis, the details of the data appearing in the report vary. The following table details the data items per diagnosis type, an "X" indicates the data item is reported; non reported data items are empty fields in the CSV report.

Table 2-6. CSV Report Data Items by Diagnosis Type

Data Item	Layout-Aware Diagnosis	Chain Diagnosis Down to Cell	Chain Diagnosis Down to Chain	Scan_ Enable Diagnosis	Clock Diagnosis
fail_log	X	X	X	X	X
tracking	X	X	X	X	X
faulty_chains	X	X	X	X	X
symptoms	X	X	X	X	X
suspects	X	X	X	X	X
symptom	X	X	X	X	X

Table 2-6. CSV Report Data Items by Diagnosis Type (cont.)

Data Item	Layout-Aware Diagnosis	Chain Diagnosis Down to Cell	Chain Diagnosis Down to Chain	Scan_ Enable Diagnosis	Clock Diagnosis
explained_pattern	X				
suspect	X	X		X	X
score	X	X		X	X
fail_match	X				
pass_mismatch	X				
type	X	X	X	X	X
value	X	X		X	X
pin_pathname	X	X		X	X
cell_name	X	X		X	X
cell_number	X	X			
net_pathname	X	X		X	X
chain_name		X	X		
memory_type		X			
shift_clock		X			
lib_internal_ pathname		X			
layout_status	X	X			
layout_layer	X	X			
category	X	X			
critical_area	X	X			
x_coord1	X	X			
y_coord1	X	X			
x_coord2	X	X			
y_coord2	X	X			
cpu_time	X	X	X	X	X
fail_patterns	X				
passing_patterns	X				

Table 2-6. CSV Report Data Items by Diagnosis Type (cont.)

Data Item	Layout-Aware Diagnosis	Chain Diagnosis Down to Cell	Chain Diagnosis Down to Chain	Scan_ Enable Diagnosis	Clock Diagnosis
unexplained_fail_ patterns	X				
tool	X	X	X	X	X
version	X	X	X	X	X
date	X	X	X	X	X

Graphical Results in Tessent Visualizer

Tessent Visualizer is a graphical user interface available from within Tessent Diagnosis. After you have run diagnosis and saved the diagnosis report, you can view that report and any failing paths using the Tessent Visualizer Diagnosis Report Viewer and schematic tabs.

Displaying Suspects in the Schematic View

You can display any suspect listed in the diagnosis report in Tessent Visualizer.

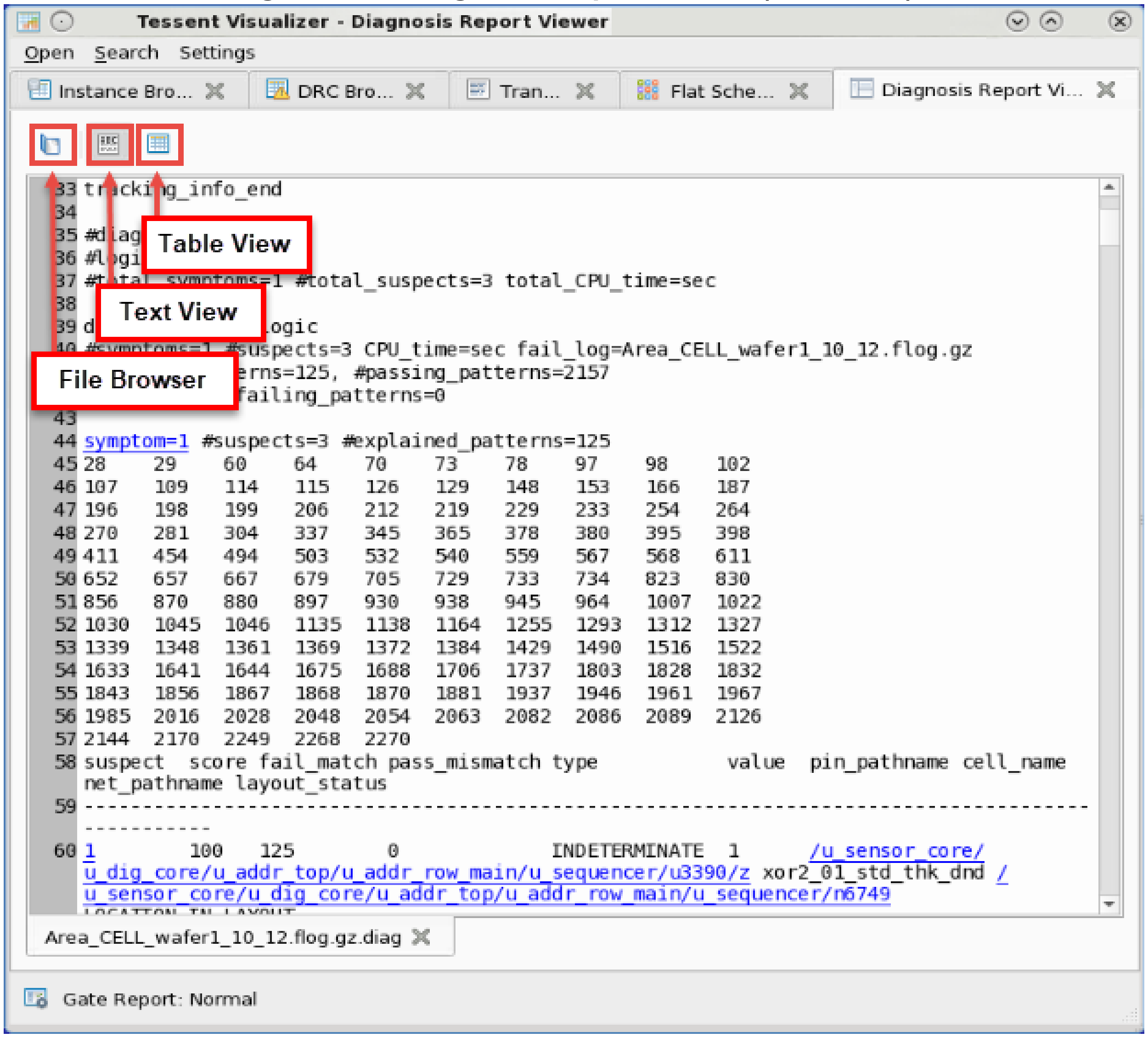
Prerequisites

- Tessent Diagnosis must be invoked on the flat design netlist associated with the diagnosis report you want to view.
- Diagnosis must have run successfully, and the results must be saved to a file.

Procedure

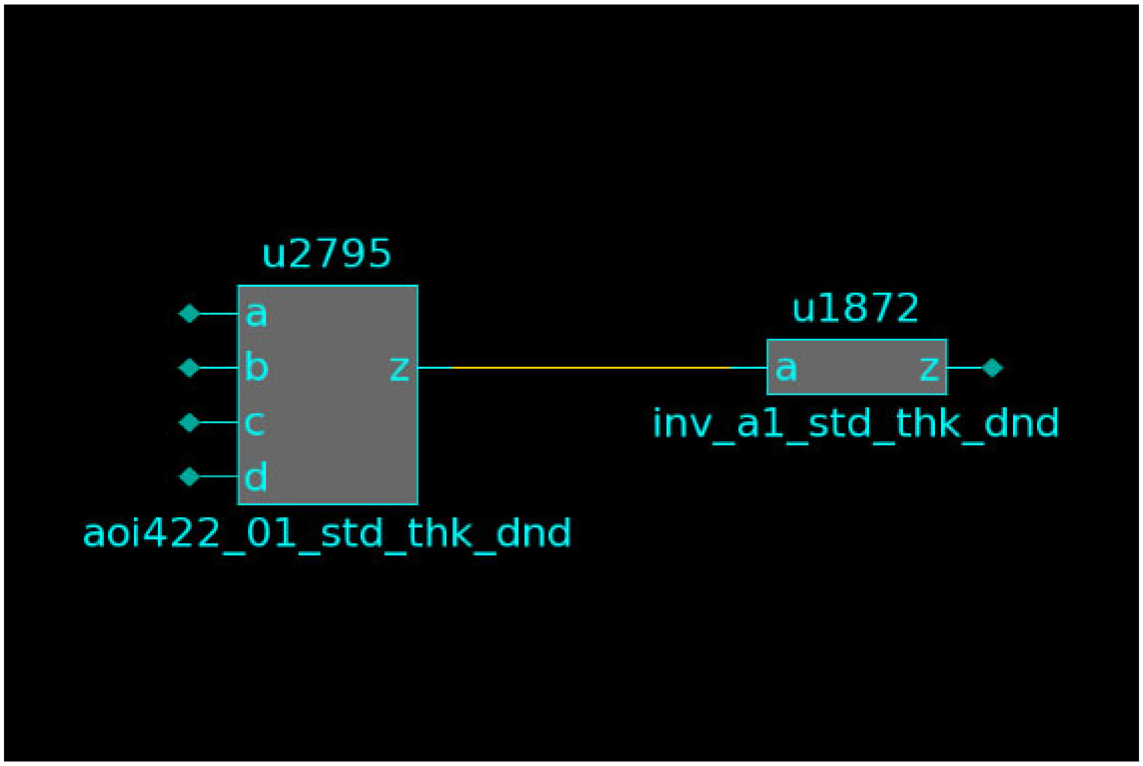
1. At the command line, enter open_visualizer. Tessent Visualizer opens.
2. Select **Open** > **Diagnosis Report Viewer.** Open the file browser from the icon in the toolbar.
3. Browse to and select the diagnosis report to load, and click **Open**. The Diagnosis Report Viewer displays all symptom (symptom=1, for example) and suspect locations (pin and net pathnames) as active links in the diagnosis report. You can choose a textual or a tabular view for the report.

Figure 2-18. Diagnosis Report Viewer (Text View)

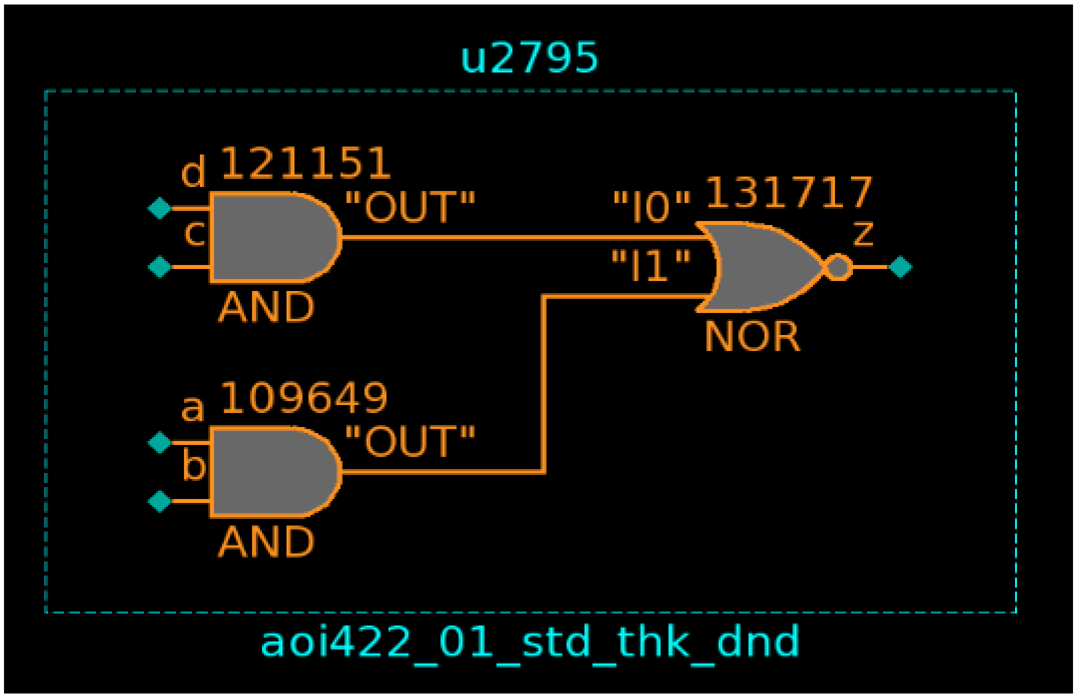


```
tracking_info_end

#diag
#logi
#total_symptoms=1 #total_suspects=3 total_CPU_time=sec

d                    logic
#symptoms=1 #suspects=3 CPU_time=sec fail_log=Area_CELL_wafer1_10_12.flog.gz
                    erns=125, #passing_patterns=2157
                    failing_patterns=0

symptom=1 #suspects=3 #explained_patterns=125
28     29     60     64     70     73     78     97     98     102
107    109    114    115    126    129    148    153    166    187
196    198    199    206    212    219    229    233    254    264
270    281    304    337    345    365    378    380    395    398
411    454    494    503    532    540    559    567    568    611
652    657    667    679    705    729    733    734    823    830
856    870    880    897    930    938    945    964    1007   1022
1030   1045   1046   1135   1138   1164   1255   1293   1312   1327
1339   1348   1361   1369   1372   1384   1429   1490   1516   1522
1633   1641   1644   1675   1688   1706   1737   1803   1828   1832
1843   1856   1867   1868   1870   1881   1937   1946   1961   1967
1985   2016   2028   2048   2054   2063   2082   2086   2089   2126
2144   2170   2249   2268   2270
suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
net_pathname layout_status
-------------------------------------------------------------------------------------
-----------
1        100   125        0             INDETERMINATE  1      /u_sensor_core/
u_dig_core/u_addr_top/u_addr_row_main/u_sequencer/u3390/z xor2_01_std_thk_dnd /
u_sensor_core/u_dig_core/u_addr_top/u_addr_row_main/u_sequencer/n6749
```

4. Click a link in the diagnosis report to display a representation of the associated circuitry in the Tessent Visualizer hierarchical schematic tab.



Any object thus displayed can also be selected for display in the flat schematic tab:



Continue to click links to display the desired circuitry as follows:

- Click a symptom to display all the gates/pins listed as suspects for the symptom.
- Click a pin pathname to display the associated gate.
- Click a net pathname to display all the associated gates.

5. Use Tessent Visualizer features as necessary to display the desired suspect information. For more information, see Tessent Visualizer in the *Tessent Shell User's Manual*.

   In addition to displaying suspects via the open_visualizer command, Tessent Diagnosis is capable of opening a diagnosis report and Tessent Visualizer when you use the display_diagnosis_report command, as shown in "Viewing Failing Paths for a Pattern" and "Viewing Failing Paths for a Suspect."

Viewing Failing Paths for a Pattern

A failing path has PI or sequential elements starting points. For a failing path of one failing pattern, the starting points must have value transition and its end points must have failures. Besides showing the failing paths, Tessent Visualizer displays all pin values through failing paths of the failing pattern. The failing paths display in the logic schematic view.

Prerequisites

- Tessent Diagnosis must be invoked on the flat design netlist associated with the diagnosis report you want to view.
- Diagnosis must have run successfully, and the results must be saved to a file.

Procedure

1. At the command line, enter display_diagnosis_report *diagnosis_report_name*.

   Example:

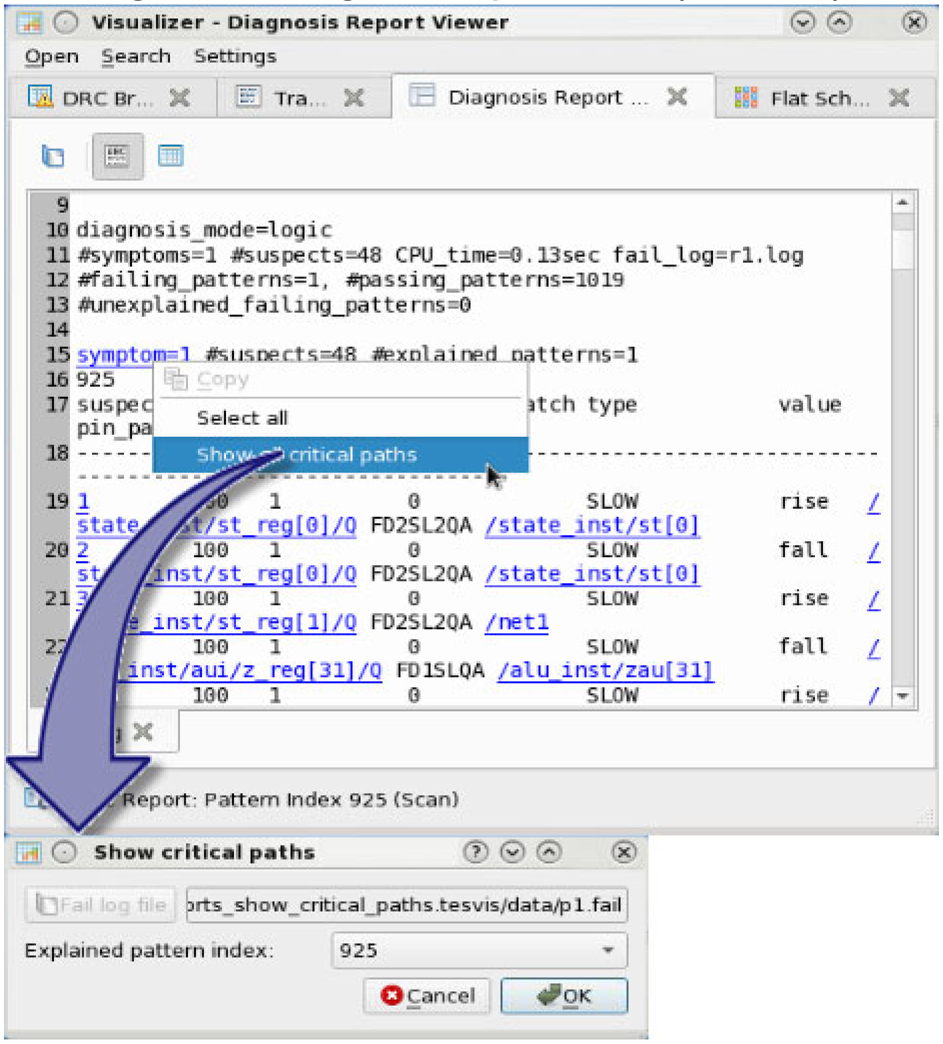
   ```
   FAULT> display_diagnosis_report p1.diag
   ```

   Tessent Visualizer opens and displays the report.

   Tip
   If you have Tessent Visualizer up and running, you can also view the report by choosing **Open > Diagnosis Report Viewer** and browsing to the report.

2. Place the mouse pointer over a symptom, right-click, and select **Show all critical paths** as shown in Figure 2-19.

Figure 2-19. Diagnosis Report Viewer (Text View)



```
diagnosis_mode=logic
#symptoms=1 #suspects=48 CPU_time=0.13sec fail_log=r1.log
#failing_patterns=1, #passing_patterns=1019
#unexplained_failing_patterns=0

symptom=1 #suspects=48 #explained_patterns=1
925
suspec                               atch type          value
pin_pa
------                               ----------------------------
----------------------------------
1         00    1          0          SLOW          rise    /
state  st/st_reg[0]/Q FD2SL2QA /state_inst/st[0]
2         100    1          0          SLOW          fall    /
st   inst/st_reg[0]/Q FD2SL2QA /state_inst/st[0]
3         100    1          0          SLOW          rise    /
    e_inst/st_reg[1]/Q FD2SL2QA /net1
          100    1          0          SLOW          fall    /
   inst/aui/z_reg[31]/Q FD1SLQA /alu_inst/zau[31]
          100    1          0          SLOW          rise    /
```

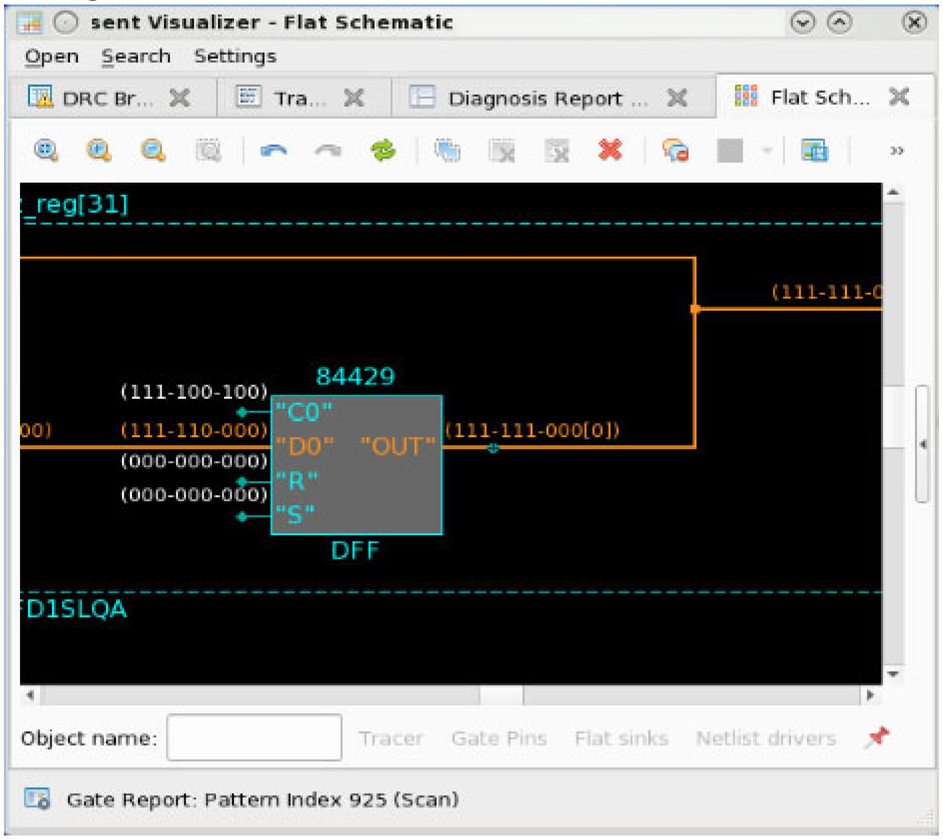
orts_show_critical_paths.tesvis/data/p1.fail

Explained pattern index: 925

3. Enter the failure log file or browse to the file.

4. Analyze the results in the Tessent Visualizer Flat Schematic Window as shown in Figure 2-20.

Figure 2-20. Tessent Visualizer Flat Schematic Window



Note

This procedure shows how to view the critical paths using the text view of the Diagnosis Report Viewer. This capability is also available from the table view. See Diagnosis Report Viewer in the Tessent Shell User's Manual for more information on the text view and table view.

Viewing Failing Paths for a Suspect

The failing paths for suspects display in the Tessent Visualizer hierarchical schematic view.

Prerequisites

- Tessent Diagnosis must be invoked on the flat design netlist associated with the diagnosis report you want to view.
- Diagnosis must have run successfully, and the results must be saved to a file.

Procedure

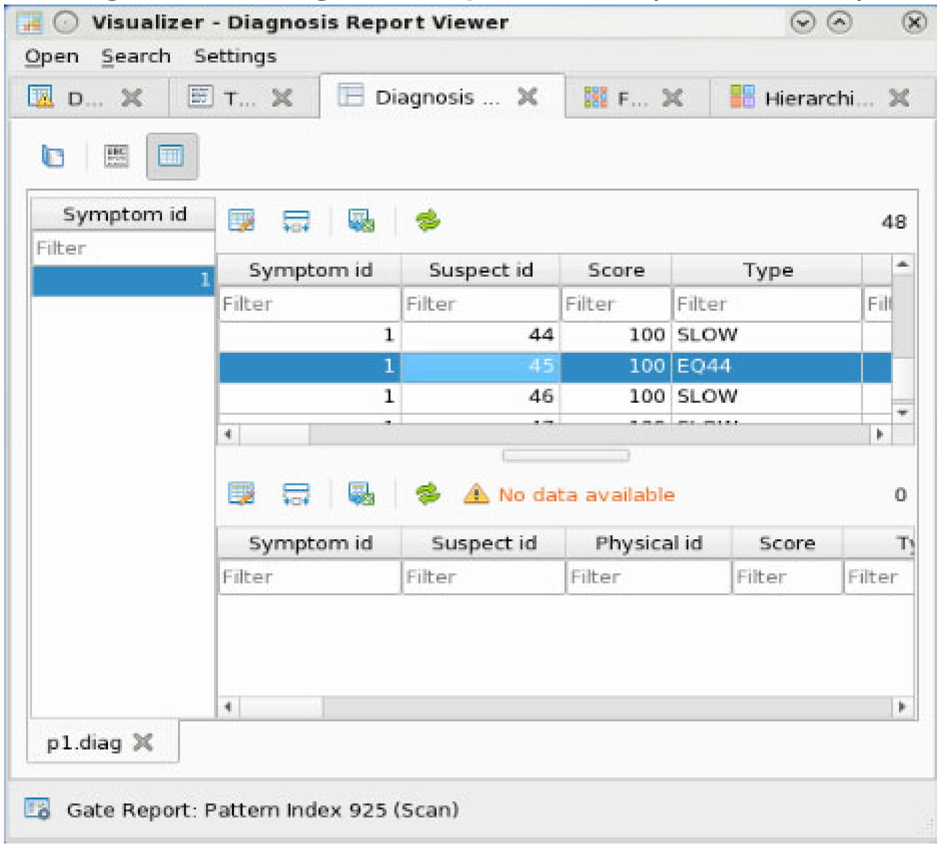
1. At the command line, enter display_diagnosis_report *diagnosis_report_name*.

Example:

```
FAULT> display_diagnosis_report p1.diag
```

Tessent Visualizer opens and displays the report.

Figure 2-21. Diagnosis Report Viewer (Tabular View)



Symptom id
1

48

Symptom id	Suspect id	Score	Type
1	44	100	SLOW
1	45	100	EQ44
1	46	100	SLOW

No data available 0

Symptom id	Suspect id	Physical id	Score	Ty

Gate Report: Pattern Index 925 (Scan)

Tip

If you have Tessent Visualizer up and running, you can also view the report by choosing **Open > Diagnosis Report Viewer**, and select the report.

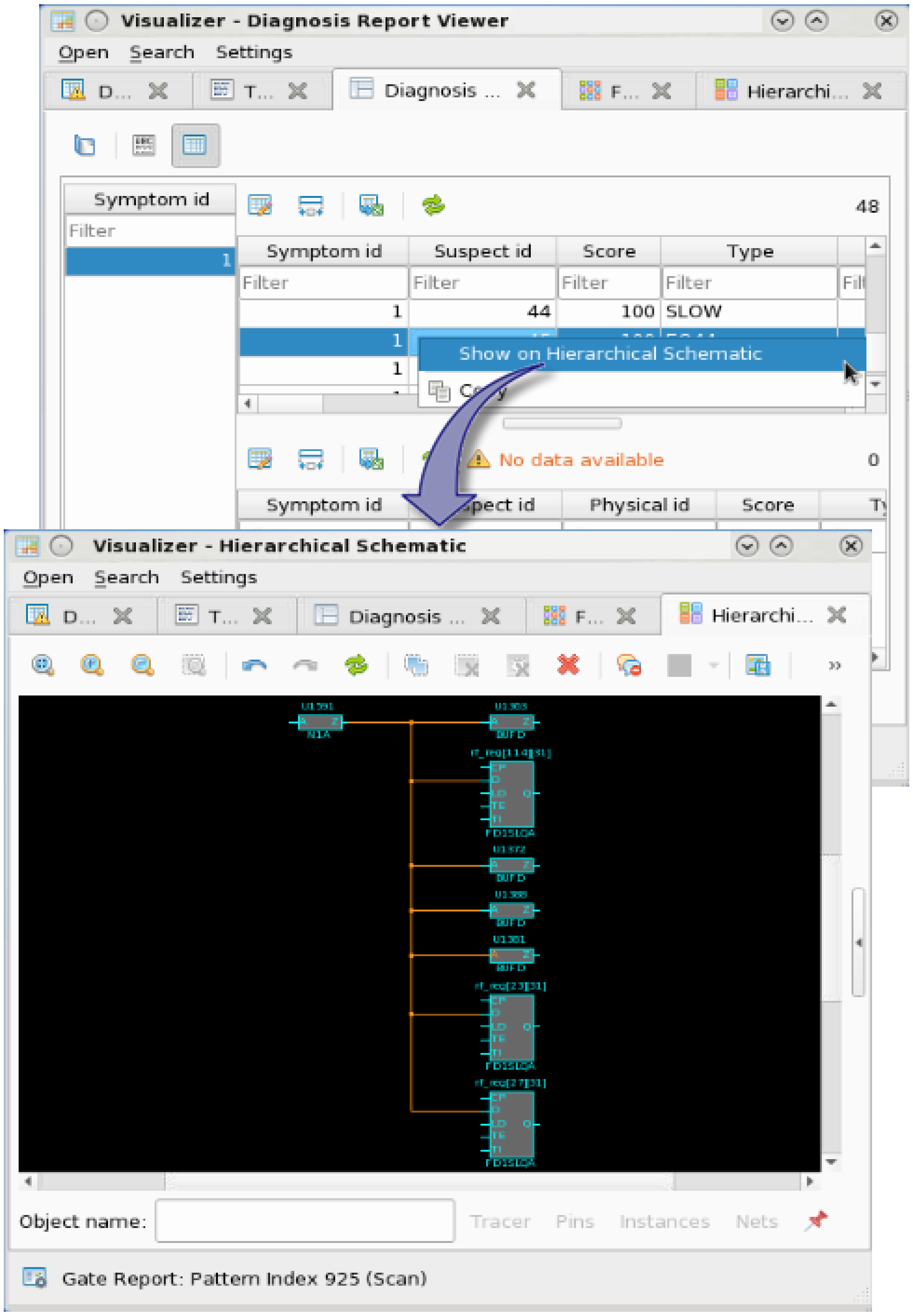
2. Place the mouse pointer over a suspect, right-click, and select **Show on Hierarchical Schematic**.

3. Analyze the results.

The following figure shows a pictorial view of steps 2 and 3.

2 Place pointer over a suspect, right click, and select **Show on Hierarchical Schematic.**

3 Analyze the results.


Visualizer - Diagnosis Report Viewer
Open Search Settings
D... T... Diagnosis ... F... Hierarchi...
Symptom id
Filter
48
Symptom id Suspect id Score Type
Filter Filter Filter Filter
1 44 100 SLOW
1
1
Show on Hierarchical Schematic
Copy
No data available
0
Symptom id Suspect id Physical id Score
Visualizer - Hierarchical Schematic
Open Search Settings
D... T... Diagnosis ... F... Hierarchi...
Object name:
Tracer Pins Instances Nets
Gate Report: Pattern Index 925 (Scan)

Diagnosis Improvements and Retrieving Internal Scan Cell Information

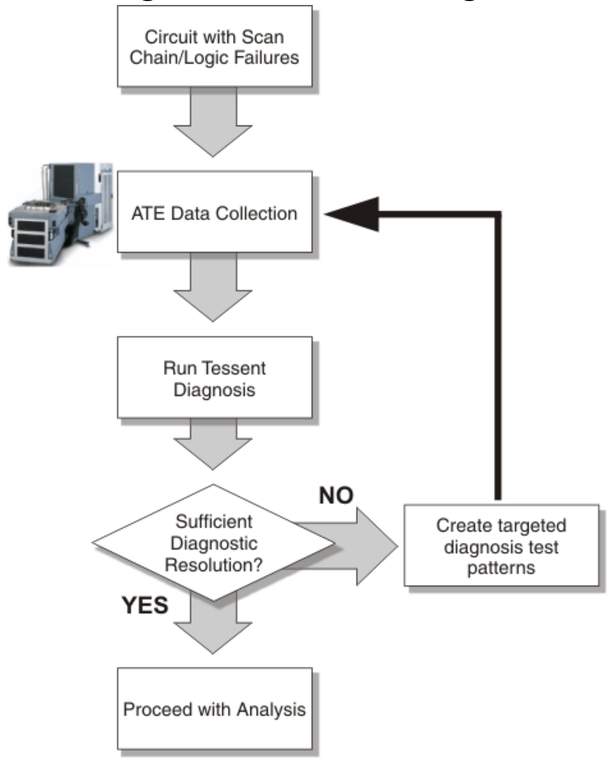
You can improve your diagnosis results by generating additional test patterns with iterative diagnosis, which can improve diagnostic resolution. In addition, for compressed patterns, you can retrieve internal scan cell information through various techniques, most notably, internal scan cell profiling.

Iterative Diagnosis

Iterative diagnosis is a method of generating additional test patterns in order to improve diagnostic resolution with Tessent Diagnosis. This technique is available for both scan chain and logic diagnosis. Better resolution for diagnosis is often a requirement for a Tessent Diagnosis user that works in a failure analysis lab.

Figure 2-22 shows the high-level iterative diagnosis process.

Figure 2-22. High-Level Iterative Diagnosis Process


Circuit with Scan Chain/Logic Failures
ATE Data Collection
Run Tessent Diagnosis
Sufficient Diagnostic Resolution?
NO
Create targeted diagnosis test patterns
YES
Proceed with Analysis

When To Use Iterative Diagnosis

Iterative diagnosis is the process of creating additional patterns for use on the ATE in order to improve diagnostic resolution. Normally, you use the iterative diagnosis flow with Tessent Diagnosis when a chip fails on the ATE and the subsequent failure files diagnosis does not yield the desired results.

Possible criteria that would merit further refinement of diagnosis results with iterative diagnosis include the following:

- Too many suspects reported
- No highly-distinguished suspects
- Too few failing patterns
- Too few parts available for Failure Analysis

Performing Iterative Diagnosis

Tessent Diagnosis provides a simple direct test generation flow. You must decide when to use this functionality: specifically, you must determine on what kind of failure mode or on what kind of diagnosis situation to run iterative diagnosis.

The following process is exemplified in Iterative Logic Diagnosis Examples 3 and 4.

Prerequisites

- Iterative diagnosis only works if there is a list of suspects reported from a previous diagnosis report. In other words, if the diagnosis run fails to converge, iterative diagnosis cannot generate a pattern.

Procedure

1. Collect the failure logs from the ATE on the device of interest using the normal production pattern.
2. Run diagnosis on the failure logs using Tessent Diagnosis and save the diagnosis report.
3. Run iterative diagnosis with Tessent Diagnosis using the create_diagnosis_patterns command. Create a targeted pattern set for use on the ATE. Save the patterns using the write_patterns command. Ensure the new test patterns pass timing simulations.

   Note
   When you save the new patterns with the write_patterns command, ensure you check with your designer to find out what switches you should use with this command.

4. If required, convert the diagnostic pattern into the ATE-required format and modify the test program to apply the pattern.
5. Collect the failure logs from the ATE on the device of interest using the new iterative diagnosis pattern.
6. Re-run Tessent Diagnosis with the new pattern and the new failure log.

Note

It is good practice to include the original failure logs and pattern file during the iterative diagnosis step, because this data supplies additional information to Tessent Diagnosis, resulting in a single diagnosis based on the two sets of data. The goal of iterative diagnosis is to provide as much fail data as possible to Tessent Diagnosis.

Related Topics

Iterative Logic Diagnosis Examples

Iterative Scan Chain Diagnosis Examples

Iterative Logic Diagnosis Examples

Run iterative diagnosis with Tessent Diagnosis using the create_diagnosis_patterns command, and save the patterns using the write_patterns command.

Example 1

The following example shows the command for specifying a Tessent Diagnosis ASCII diagnosis report, *Prod_A_w200x5y8.diag*, for diagnostic ATPG:

```
create_diagnosis_patterns –diagnosis_report Prod_A_w200_x5y8.diag

write_patterns Prod_A_w200_x5y8.diag.pattern
```

Upon issuing this command, Tessent Diagnosis reads the contents of this diagnosis report file. If there are any system logic defects reported in this diagnosis report file Tessent Diagnosis creates new test patterns to target the stuck-at-0 and stuck-at-1 faults on these candidate signals.

If the diagnosis report contains both logic defects and scan chain defects, then Tessent Diagnosis generates diagnostic test patterns targeting both the scan chain defect and the system logic defects. For diagnostic test generation for logic failures, Tessent Diagnosis skips any command line switches for chain failure diagnostic test generation. For diagnostic test generation for chain failures, Tessent Diagnosis skips any command line switches used by logic failure diagnostic test generation. Tessent Diagnosis prints a warning message for each skipped switch.

Example 2

The following example shows how to run diagnostic ATPG without providing a diagnosis report file:

```
read_patterns design_A.pattern

diagnose_failures chip1.fail

create_diagnosis_patterns

write_patterns chip1.diagnosis.pat
```

In this example, the “diagnose_failures chip1.fail” command diagnoses failure file *chip1.fail* and creates the diagnosis results. While running the “create_diagnosis_patterns” command, the internal diagnosis results are used as input to the diagnostic ATPG process.

Example 3

The following example shows the usage scenario that might be encountered by a failure analyst intending to use iterative diagnosis:

```
//collect the datalog on the ATE; assume you name the datalog “fail1.log”

//traditional diagnosis
read_patterns src/bridge_patterns.ascii

//diagnose the failure
diagnose_failures results/fail1.flog

//view the results, and if diagnosis is needed, write the diagnosis
//result to a file
write_diagnosis -file results/fail1 -rep

//create_diagnosis_patterns based on the internally stored diagnosis
// result
create_diagnosis_patterns

//save the targeted diagnosis patterns
write_patterns results/fail1.bin.gz

//save Verilog patterns and run simulation to check for mismatches

//go back onto the ATE and collect the datalog with the new pattern;
//assume you name the datalog “fail1_iter_diag.log”

//ITERATIVE DIAGNOSIS

//load the pattern the first datalog was collected with
read_patterns src/bridge_patterns.ascii

//append the diagnostic pattern
read_patterns results/fail1.bin.gz -append

//read in the original fail log
read_failures results/fail1.flog

//read in the diagnostic pattern generated fail log
read_failures results/fail1_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new combined diagnosis result
write_diagnosis -file results/fail1_iter_diag -rep
```

Example 4

In the following usage example, a failure analyst has been given five units of the same product in a failure analysis job. The analyst is to pick one and perform failure analysis on it. The analyst has already run diagnosis on all five parts. In order to achieve better diagnostic resolution, the analyst creates a single pattern, collects additional fail data on all five units, and decides which part would be best for failure analysis.

```
//diagnosis results already collected and diagnosed
YAlogs = part1.diag, part2.diag, part3.diag, part4.diag, part5.diag

//create_diagnosis_patterns based on the internally stored
//diagnosis result
create_diagnosis_patterns -diagnosis_report part1.diag part2.diag \
part3.diag part4.diag part5.diag

//save the targeted diagnosis patterns
write_patterns results/fail1-2-3-4-5.bin.gz

//take pattern to ATE and collect five more datalogs with the one \
//compound pattern
diagnostic_datalog = part1_iter_diag.log,\
part2_iter_diag.log,part3_iter_diag.log,\
part4_iter_diag.log, part5_iter_diag.log

//load the pattern the first datalog was collected with
read_patterns src/bridge_patterns.ascii

//append the diagnostic pattern
read_patterns results/fail1-2-3-4-5.bin.gz  -append

//begin diagnosis - note that the pattern source doesn't need \
//to be reloaded
read_failures results/part1.flog  //read in the original fail log

//read in the diagnostic pattern generated fail log
read_failures results/part1_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new diagnosis result
write_diagnosis -file results/part1_iter_diag -rep

//read in the original fail log
read_failures results/part2.flog

//read in the diagnostic pattern generated fail log
read_failures results/part2_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new diagnosis result
write_diagnosis -file results/part2_iter_diag -rep

//read in the original fail log
read_failures results/part3.flog
```

```
//read in the diagnostic pattern generated fail log
read_failures results/part3_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new diagnosis result
write_diagnosis -file results/part3_iter_diag -rep

//read in the original fail log
read_failures results/part4.flog

//read in the diagnostic pattern generated fail log
read_failures results/part4_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new diagnosis result
write_diagnosis -file results/part4_iter_diag -rep

//read in the original fail log
read_failures results/part5.flog

//read in the diagnostic pattern generated fail log
read_failures results/part5_iter_diag.flog -append

//diagnose both of the fail logs that are stored in internal memory
diagnose_failures -internal

//save the new diagnosis result
write_diagnosis -file results/part5_iter_diag -rep
```

Iterative Scan Chain Diagnosis Examples

Perform iterative scan chain diagnosis the same way you would iterative logic diagnosis.

Example 1

The following example shows how to specify a diagnosis report for diagnostic ATPG for chain failures:

create_diagnosis_patterns –diagnosis_report w200x5y8.diag

write_patterns w200x5y8.diag.ascii

In this example the file *w200x5y8.diag* is a diagnosis report file created by Tessent Diagnosis in ASCII format. Iterative diagnosis reads the content of this diagnosis report file. If there are any scan chain failures reported as failing scan chain, new test patterns are created to target the candidate scan cells of the faulty scan chain.

Example 2

This example runs diagnostic ATPG with a chain name, cell range, and fault type you specify.

create_diagnosis_patterns –chain chain1 –cell_range 10 20 –fault_type stuck_at_0

write_patterns chain1_10_20_sa0.diag.ascii

The above command creates diagnostic scan ATPG patterns that target fault isolation for scan cells 10 through 20 of scan chain chain1, the target fault type is stuck-at 0 fault.

Example 3

In this example, iterative chain diagnosis is performed by producing test patterns and saving the chain-test and the scan-test into separate pattern files. Thus, a multiple test suite fail log must be employed. An example of the production fail log is below:

```
// Sample multiple test suite fail log
format cycle
tracking_info_begin
tracking_info_end
test_suite_begin PRODUCTION_CHAIN_TEST
failures_begin
150 PADX L H
...
1704 PADX H L
failure_buffer_limit_reached none
failures_end
test_suite_end
test_suite_begin PRODUCTION_SCAN_TEST
failures_begin
150 PADX L H
...
7801 PADX L H
failure_buffer_limit_reached last_cycle_logged
failures_end
test_suite_end
failure_file_end
system date
```

The dofile for performing iterative diagnosis is below:

```
// Iterative diagnosis dofile
// Read in multiple test suite production patterns
read_patterns PRODUCTION_CHAIN_TEST_PATTERN.stil.gz
read_patterns PRODUCTION_SCAN_TEST_PATTERN.stil.gz -append
diagnose_failures PRODUCTION_MULTI_SUITE_FAILING_CHAIN_TEST_FLOG.flog

// create_diagnosis_patterns based on the internally stored diagnosis
// result for a chain failure
create_diagnosis_patterns
// Save the targeted diagnosis patterns
write_patterns ITERATIVE_CHAIN_TEST_PATTERN.stil.gz -chain_test -stil
write_patterns ITERATIVE_SCAN_TEST_PATTERN.stil.gz -scan_test -stil

// Go collect the new failure logs and assemble the multiple test suite
// fail data into a new flog called
// ITERATIVE_MULTI_SUITE_FAILING_CHAIN_TEST_FLOG.flog

// Read in all four of the patterns
read_patterns PRODUCTION_CHAIN_TEST_PATTERN.stil.gz
read_patterns PRODUCTION_SCAN_TEST_PATTERN.stil.gz -append
read_patterns ITERATIVE_CHAIN_TEST_PATTERN.stil.gz -append
read_patterns ITERATIVE_SCAN_TEST_PATTERN.stil.gz -append

// Diagnosis execution on two datalogs
read_failures PRODUCTION_MULTI_SUITE_FAILING_CHAIN_TEST_FLOG.flog
read_failures ITERATIVE_MULTI_SUITE_FAILING_CHAIN_TEST_FLOG.flog -append

diagnose_failures -internal
write_diagnosis
```

Techniques for Finding Internal Scan Cells in a Compressed Pattern

There are many techniques that you can use to retrieve the internal scan cells for diagnosis suspects.

Table 2-7 describes various techniques for finding internal scan cells.

Table 2-7. Techniques for Finding Internal Scan Cells

Technique	Pros	Cons
Internal scan cell profiling	Ease of use with set_diagnosis_options -internal_failing_cells command	n/a
1hot pattern expansion	If failure is consistent, should tell you exactly where the failure is located	You must first know which compressed patterns failed, and then create new patterns and test them

Table 2-7. Techniques for Finding Internal Scan Cells (cont.)

Technique	Pros	Cons
Based on the suspect, use the report_gate command, which enables you to trace the suspects' output logic cones to find the potential capture scan cells	Flexible and enables you to use your own judgement	Labor-intensive and may lack accuracy
Use the read_failure command to report all corresponding scan cells for a failure bit	Simple technique	Results are not specific
Use the report path to report the control and observe scan cells	Tool automatically provides this information	This only applies to at-speed patterns, and you could have a significant number of paths
Use a bypass pattern	You can precisely locate the failed scan cell	You must first create the bypass pattern and then test it

Internal Scan Cell Profiling for Compressed Patterns

When debugging failures, it is sometimes necessary to understand which scan chains and which group of cells on those chains observe the scan failure. When using compressed patterns, this is difficult to extrapolate from the failing patterns alone because multiple chains are compressed into a single compactor output. Internal scan cell profiling enables you to report the internal scan cells and scan chains that have the highest confidence of observing the defect when using compressed scan patterns. This method is only available after you have performed scan diagnosis and have reported suspects.

During scan cell profiling, the tool traces the failures back to the scan cells that have the highest possibility of observing the defect location (suspect). The tool reports the failing pattern and the chain and cell combination that observe the failing bits. It also includes a confidence number for the scan chain/cell combination for each of the failing patterns; some failing patterns might be better for observing a defect.

To enable internal scan cell profiling, specify the following command:

set_diagnosis_options -internal_failing_cells on

When enabled, Tessent Diagnosis appends an "Internal failing cells profiling" section to the diagnosis report.

The following example shows that there are four failing patterns. However, only patterns 112 and 376 are observed on a compactor output. (The other two failing patterns, 98 and 378, are observed on a functional output.) The tool reports the failing patterns that failed for a compactor output.

```
//  command: diagnose_failures injected_1.flog
//  Verifying 455 external patterns.
//  incorrect_patterns=0, simulated_patterns=455, simulation_time=0.87sec.
Tessent Diagnosis <. . .>
tracking_info_begin

tracking_info_end                         #diagnosis_modes=1
#logic_symptoms=1
#total_symptoms=1 #total_suspects=10 total_CPU_time=0.14sec

diagnosis_mode=logic
#symptoms=1 #suspects=10 CPU_time=0.14sec fail_log=injected_1.flog
#failing_patterns=4, #passing_patterns=451
#unexplained_failing_patterns=0
symptom=1 #suspects=10 #explained_patterns=4
98    112   376   378
suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
net_pathname
-------------------------------------------------------------------------------------------
1        100   4          0             STUCK          0      /cpu_i/uUART/ix1003/Y nor02 /
cpu_i/uUART/nx1002
2        100   4          0             EQ1            1      /cpu_i/uUART/ix1003/A1 nor02 /
cpu_i/uUART/nx3064
3        100   4          0             EQ1            1      /cpu_i/uUART/ix1003/A0 nor02 /
cpu_i/uUART/nx3346
4        100   4          0             EQ1            0      /cpu_i/uUART/ix1013/B0 ao21 /
cpu_i/uUART/nx1002
5        100   4          0             CELL           1      /cpu_i/uUART/ix2571/QB sff /
cpu_i/uUART/nx3064
6        85    4          6             INDETERMINATE  0      /cpu_i/uSDM/ix475/A0 ao22 /
cpu_i/SSFRDI_2_
7        85    4          6             EQ6            0      /cpu_i/uSDM/ix475/A1 ao22 /
cpu_i/uSDM/nx617
8        85    4          6             EQ6            0      /cpu_i/uUART/ix1013/Y ao21 /
cpu_i/SSFRDI_2_
9        81    4          15           INDETERMINATE  0      /cpu_i/uSDM/ix475/Y ao22 /cpu_i/
uSDM/nx474
10       81    4          15            EQ9            0      /cpu_i/uSDM/ix702/C0 aoi221 /
cpu_i/uSDM/nx474
-------------------------------------------------------------------------------------------

Internal failing cells profiling
  Total failing patterns=4, Total external compressed bits=2
  Failing Pattern 112 has 1 compressed failing bits
    chain=chain23, cell=16, confidence=100.00%
  Failing Pattern 376 has 1 compressed failing bits
    chain=chain9, cell=2, confidence=19.36%
    chain=chain12, cell=2, confidence=80.64%
```

In this case, the tool reports that for failing pattern 112, one compressed failing bit (failing cycle) was found in the fail log. The analysis shows that scan chain chain23 and scan cell 16 are likely to observe the suspects reported in the diagnosis report.

The tool also reports that for failing pattern 376, one compressed failing bit (failing cycle) was found in the fail log. The analysis shows that scan chain chain9 and scan cell 2 or scan chain chain12 and scan cell 2 may observe the suspects reported in the diagnosis report but with lower confidence.

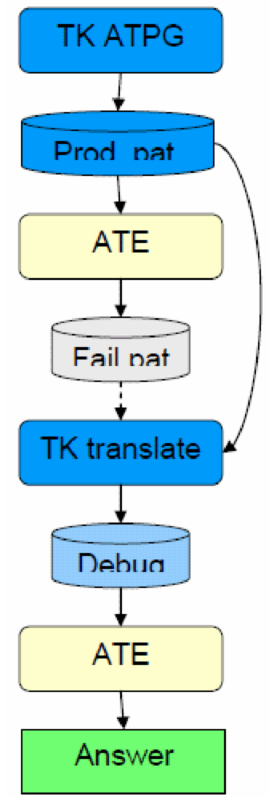
Internal scan cell profiling is not supported in server mode.

1hot Compressed Pattern Expansion

1hot compressed pattern expansion, like internal scan cell profiling, enables you to report the internal scan cells for compressed failing patterns. When EDT patterns fail on the ATE, some users may choose to discover the internal failing flops, mask them, move on and run other tests. 1hot pattern expansion enables you to identify the internal failing flops.

Compressed patterns usually do not observe per internal scan chain. If the compressed patterns are expanded to be 1hot patterns, then each 1hot pattern can observe the individual internal chain for an EDT channel. The newly created 1hot patterns are expected to have the same scan load values into the internal scan chains (except the EDT-related scan cells) as the original compressed patterns. This ensures that retesting the 1hot patterns maintains the same responses at the internal chains as their original compressed patterns. The only difference would be that the individual chain is observed.

Figure 2-23. Compressed Pattern Expansion Flow



Note
If you have a pattern observing an unbalanced number of chains on channels, the tool only observes a single chain exactly once across all patterns generated to expand the original compressed pattern. In the case of unbalanced chains for multiple channels, the tool may set some channels (connected to fewer chains than other channels) to <no_chains_observed> because you already have patterns observing chains in this channel.

Performing 1hot Compressed Pattern Expansion

When you perform 1hot compress pattern expansion, you can specify a set of selected test patterns or a set of selected scan chains for expansion. After applying the 1hot patterns on the ATE, you can retrieve the failing flop IDs based on the ATE failure log.

Compressed pattern expansion applies to either non-masking patterns or partial-masking patterns. These patterns can be transformed into multiple 1hot patterns, where only a single internal chain is observed at each channel. You can only use this capability on external pattern sets. If Tessent Diagnosis detects any internal patterns, then command execution quits and an error message is generated.

Prerequisites

- A core-level failure file.

Procedure

1. Run production compressed pattern on the ATE. Identify failing patterns on the ATE, or optionally use the read_failures command to get the failing file, if the failure file is cycle-based format.
2. Optionally, use set_pattern_filtering to select which patterns to expand.
3. Use expand_compressed_patterns to expand selected patterns to 1hot patterns.
4. Save expanded patterns using the write_patterns command.
5. Run the new patterns on ATE and generate a new failure file.
6. Identify failing scan cells in the Tessent Diagnosis tool using the read_failures command.

Related Topics

One-Hot Compressed Pattern Expansion Examples

One-Hot Compressed Pattern Expansion Examples

To expand one-hot compressed patterns, use the expand_compressed_patterns command.

Example 1

A design has five edt_channels with chain/channel ratio of 100. A set of 1000 Tessent TestKompress patterns were tested on the ATE. Some devices passed chain test but failed scan patterns 57, 58, and 65. Assume that patterns 57, 58, and 65 are compressed patterns, pattern 57 and 58 observe all 100 internal scan chains, and pattern 65 observes some of the scan chains, for example 50 scan chains. To identify the internal scan chain cell that is failing on these three failing patterns, use the following sequence of commands:

read_patterns production_edt_pat.stil

set_pattern_filtering –external -list 57 58 65

expand_compressed_patterns

```
// Pattern 57 observes 100 internal chains and is expanded to 100 1hot patterns
// Pattern 58 observes 100 internal chains and is expanded to 100 1hot patterns
// Pattern 65 2 observes 50 internal chains and is expanded to 50 1hot patterns
```

write_patterns debug_pat_57_58_65_.stil –stil2005

The debug_pat_57_58_65_.stil file contains 250 patterns that were expanded from patterns 57, 58 and 65. After the set_pattern_filtering command, only the original patterns 57, 58, and 65 are expanded to 1hot test patterns.

The messages from the expand_compressed_patterns command shows how many 1hot test patterns were created based on patterns 57, 58, and 65. A default map file is created, this map file maps the new 1hot pattern indices to the original test pattern indices. The default map file name is *<external_pattern_file_name>.map*. For this example, the default map file name is *production_edt_pat_.stil.map*.

The contents of the map file are similar to the following:

```
1hot_pat_id       original_pat_id  observedChainName  EdtChannelName
-----------------------------------------------------------------
    0              57              block1_chain0      edt_block1_channel1
                                   block1_chain100    edt_block1_channel2
                                   block1_chain200    edt_block1_channel3
                                    ...
                                   block1_chain500    edt_block1_channel5
    1              57              block1_chain1      edt_block1_channel1
                                   block1_chain101    edt_block1_channel2
                                   block1_chain201    edt_block1_channel3
                                    ...                ...
                                   block1_chain501    edt_block1_channel5
    2              57               ...                ...
    ...            ...              ...                ...
    99             57              block1_chain99     edt_block1_channel1
                                   block1_chain199    edt_block1_channel2
                                   block1_chain299    edt_block1_channel3
                                    ...
                                   block1_chain599    edt_block1_channel5
    100            58
    101            58
... <patterns 102-198 omitted from this example>
    199            58
    200            65
... <patterns 210-248 omitted from this example>
    249            65
```

Note

The expanded 1 hot pattern set can be very large unless the set_pattern_filtering command is used.

Example 2

Using alternative pattern filtering options can produce very large expanded pattern sets, and yields a warning from the tool, as shown:

read_patterns production_edt_pat.wgl

set_pattern_filtering –external –clock clkA

expand_compressed_patterns

```
//Warning: There are 500 TestKompress patterns to expand, it ends up
//with 50,000 patterns. By default, the maximum number of 1hot patterns to
//be created is 1000. You can use the "-max__patterns N" switch for
//"expand_compressed_patterns" command to increase this limit, where N is
//the maximum number of 1hot patterns to create.
```

Example 3

Expand selected patterns for patterns 37, 38, and 45, and scan chains chain1, chain2, and chain3. In this example pattern 45 does not observe any of the three scan chains, so pattern 45 is not expanded as a result of this command sequence.

```
read_patterns production_edt_pat.stil

set_pattern_filtering –external 37 38 45        // Assuming the users are interested
                                                // in chain1 to chain5

expand_compressed_patterns
–chain chain1 chain2 chain3    // Pattern 37 observes X internal chains and is expanded to Z
        // 1hot patterns
        // Pattern 38 observes Y internal chains and is expanded to W 1hot patterns
        //… similar message for all patterns

write_patterns debug_pat.stil –stil2005
```

Example 4

If you have a pattern observing an unbalanced number of chains on channels:

```
Channel 0 – observing chain0, chain1
Channel 1 – observing chain2
```

This generates the following expanded patterns:

```
Pattern 0 – observing chain0 (channel 0) and chain2 (channel 1)
Pattern 1 – observing chain 1 (channel 0) and none-observed (channel 1)
```

For failure on chain2, exactly 1 expanded pattern is expected to fail.

Troubleshooting

At times, issues can arise that impact the Tessent Diagnosis results and performance. These issues include pattern and failure file mismatches, unexpected diagnosis results, fault injection issues, abort condition for chain diagnosis, and long

Pattern and Failure File Mismatches

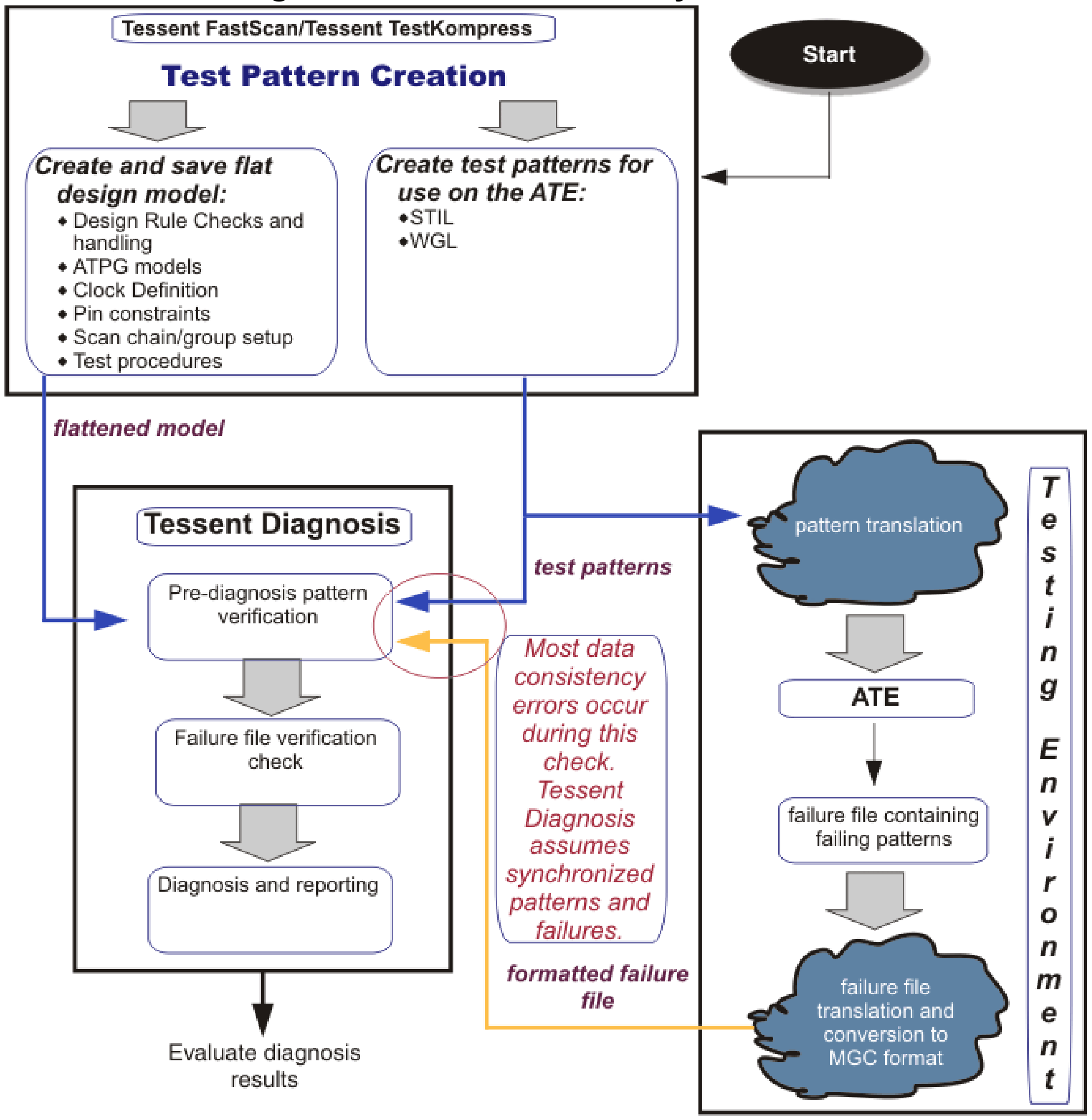
Pattern and failure file mismatch errors are usually caused by failure of the input data files to pass the data consistency and accuracy tests performed before the actual diagnosis begins. If Tessent Diagnosis encounters problems during the run, then the tool aborts the diagnosis and sends an error message to standard out.

Data Consistency Checks

In the data flow from the test-pattern creation stage to the testing environment, the ATE runs the patterns with the actual DUT and, subsequently, creates the failure file for subsequent consumption by Tessent Diagnosis.

Figure 2-24 shows a high-level view of the data flow between the ATPG tools (Tessent FastScan and Tessent TestKompress), Tessent Diagnosis, and the actual testing environment.

Figure 2-24. Data Consistency Data Flow


Tessent FastScan/Tessent TestKompress
Test Pattern Creation
Start
Create and save flat design model:
• Design Rule Checks and handling
• ATPG models
• Clock Definition
• Pin constraints
• Scan chain/group setup
• Test procedures
Create test patterns for use on the ATE:
•STIL
•WGL
flattened model
Tessent Diagnosis
Pre-diagnosis pattern verification
Failure file verification check
Diagnosis and reporting
Evaluate diagnosis results
test patterns
Most data consistency errors occur during this check. Tessent Diagnosis assumes synchronized patterns and failures.
formatted failure file
pattern translation
ATE
failure file containing failing patterns
failure file translation and conversion to MGC format
Testing Environment

Note

The recommended practice is to load patterns into Tessent Diagnosis in the same order the patterns were applied on the tester. This helps prevent conversion errors due to incorrect determination of pattern boundaries.

In general, the process of preparing the test patterns for use in the testing environment consists of the following data translation stages:

- Pattern translation — Post-ATPG pattern processing, specifically creating the program for use on the ATE. Data corruption can occur during this stage if you modify the patterns (for example, adding extra cycles).

- Failure file translation — Converting the raw ATE failure file into the format Tessent Diagnosis reads.

As Figure 2-24 shows, Tessent Diagnosis uses the unmodified patterns from the test-pattern creation stage independent of the testing environment and compares these unmodified patterns to the failure file from the ATE. Before diagnosis, Tessent Diagnosis performs the following data consistency checks before proceeding to diagnosis:

- Pattern consistency — Compares the simulated capture values to the expected values in the test patterns—for an example, see “Pattern Verification Unsuccessful.” Identifies inconsistencies and inaccuracies involving the design netlist, test patterns, or settings—see “Preparing the Test Patterns.”
- Failure file consistency — Compares pattern expected values with the corresponding value in the failure file. Identifies erroneous failure file conversions—see “Guidelines for Preparing the ATE Failure File.”

Consequently, if you do not account for any pattern modifications in the failure file you input to Tessent Diagnosis, you can encounter data inconsistency warnings and errors. If Tessent Diagnosis identifies data consistency errors during these checks, then the tool halts diagnosis. You must resolve any data consistency check errors.

Pattern Verification Unsuccessful

Unsuccessful pattern verification consists of three types of mismatch errors: B2X, X2B, and binary.

The following list describes the three types of unsuccessful pattern verification:

- B2X mismatch — B2X mismatches occur during pattern verification when the simulator returns X values instead of the expected binary values. These errors are usually due to software enhancements to the simulator or using a mask file when reading patterns.

  By default, Tessent Diagnosis reports a B2X mismatch as a warning.
- X2B mismatch — X2B mismatches occur during pattern verification when the simulator returns binary values instead of the expected X values.

  The tool most likely introduces X2B mismatches because it removes cell constraints when performing pattern verification and diagnosis. This enables the tool to perform diagnosis using one flat model for potentially multiple modes, each of which have different constraints, or to use flat models that were written at different times.

  By default, Tessent Diagnosis ignores X2B mismatches.
- Binary mismatch — Binary mismatches occurring during pattern verification when the simulator returns incorrect binary values (for example, a 1 instead of an expected 0). In the event that a binary mismatch occurs, you should investigate and determine the root cause, otherwise the mismatch could lead to improper diagnosis results.

By default, Tessent Diagnosis reports a binary mismatch as an error.

The following example depicts a typical Tessent Diagnosis pattern verification mismatch error message:

read_patterns tst_fail.ascii

diagnose_failures tst_failure.log -output tst_diag.log -replace

```
// Verifying patterns.
0 /paddr[9]  Z  L
0 /paddr[8]  Z  L
0 /paddr[6]  Z  L
0 /paddr[4]  Z  L
0 /debugpc[9]  Z  L
0 /debugpc[8]  Z  L
0 /debugpc[6]  Z  L
0 /debugpc[4]  Z  L
0 /piccpu_i/edt_so4  Z  L
// incorrect_patterns=1, simulated_patterns=32, simulation_time=0.00sec.
// To verify all external patterns, use command 'report_failures -exact'.
// Error: Pattern verification unsuccessful. Diagnosis cannot be performed
// without successful pattern verification.
```

This error message indicates the test pattern verification was unsuccessful and there are mismatches in the data. Possible causes include:

- Improper setting of the write_flat_model command when the flat version of the netlist was saved.
- Not all relevant commands used to set up for ATPG were run prior to saving the flat model.
- An improper or incomplete ATPG library was used in the pattern creation session in which the flat model was saved.

To report all the mismatches, including expected binary versus X in external patterns, use the report_failures -Exact command.

Failure File Errors

Failure files can have issues that cause Tessent Diagnosis to issue errors.

Example 1

The tool issues an error when the tester's expected value does not match the Tessent Diagnosis good simulation value.

read_patterns tst_fail.bin

diagnose_failures tst_failure.log -output tst_diag.log -rep

```
// Error: File "tst_failure.log", line number 10: " 80 chain2 19 L H"
fail_log_expected_value=L, pattern_expected_value=H
// Error: Incorrect Failure File Format.
```

To correct this issue, ensure that the:

- Test patterns loaded in are the same patterns used on the ATE when generating the failure logs.
- Expected binary values on the ATE are not changed to the opposite state.
- Conversion of the ATE failure log is correct. Verify that you account for any test cycles applied before the test patterns.

See also "Input File Requirements."

Example 2

This following error message indicates duplicate lines in the failure file. Delete one to correct the problem.

diagnose_failures ../Failog/fs_fail_st0.log

```
// Error: Duplicated fail data at chain chain1 cell 10.
// Error: File "../Failog/fs_fail_st0.log", line number 2: " 69 chain1...
```

Example 3

The following error message indicates the failure file is empty or contains syntax errors in keywords. Verify the conversion of the ATE failure log is correct, and that the failure file contains failing cycles.

diagnose_failures /Failog/fs_fail_st0.log output fs_diag.log replace

```
//Error: No valid failing pattern(s) in failure file
// …/Failog/fs_fail_st0.log with the specified options.
```

See "Input File Requirements."

Unexpected Diagnosis Results

Unexpected diagnosis results can include having too few failing patterns, many suspects for one symptom, lower than expected suspect scores, too many reported faulty cell suspects, unusable clock domain patterns, and too many suspects, symptoms, or unexplained patterns in logic diagnosis.

Very Few Total Failing Patterns

Very few total failing patterns are reported in the diagnosis report. If there are not many failing patterns, then the number of failing suspects may become quite large as the tool is unable to differentiate between a large set of faults.

The most common reasons for a low number of failing patterns are:

- Low excitation or observation of the defect site.
- High number of failing cycles with truncation.

Low Excitation or Observation of the Defect Site

When this happens, Tessent Diagnosis has relatively few patterns able to excite and observe the defect, which can lead to a very low number of failing patterns: this can translate to low resolution. In this case, consider performing the following activities:

- Collect as much failing data as possible.
- If multiple patterns induce failures, then you should use multiple test suites to accumulate the failing patterns.

Make sure that pattern sampling of the failing patterns has been turned off.

High Number of Failing Cycles with Truncation

In this case, a defect that has very high observability causes many cycles to fail. If the datalog was truncated, then there may only be a small number of failing patterns that were exhaustively collected. To obtain more failing patterns, you need to collect a much larger number of failing cycles on the ATE.

Many Suspects for One Symptom

In some cases, Tessent Diagnosis may report many suspects for one symptom. Use other physical failure analysis methodologies, such as viewing in Calibre, to determine which net or even which section of the net is causing the problem.

In the following example, there are 12 suspects total and 11 of them are equivalent to the first suspect.

```
suspect score fail_match pass_mismatch type value pin_pathname cell_ ...
-----------------------------------------------------------------------
1        100   70         0             STUCK 0    /U1680/Y     inv02 ...
2        100   70         0             EQ1   1    /U1680/A     inv02 ...
3        100   70         0             EQ1   1    /U1544/Y     nand03...
4        100   70         0             EQ1   0    /U1544/A0    nand03...
5        100   70         0             EQ1   0    /U1544/A2    nand03...
6        100   70         0             EQ1   0    /U1544/A1    nand03...
7        100   70         0             EQ1   0    /U1584/Y     and03 ...
8        100   70         0             EQ1   0    /U1584/A2    and03 ...
9        100   70         0             EQ1   0    /U1584/A0    and03 ...
10       100   70         0             EQ1   0    /U1584/A1    and03 ...
11       100   70         0             EQ1   0    /U480/Y      inv01 ...
12       100   70         0             EQ1   1    /U480/A      inv01 ...
```

Low Score for Suspects

Tessent Diagnosis may assign suspects lower-than-expected scores. If this is the case, verify the accuracy of the failure file contents. If the ATE is a per-channel type, adjust the failure file contents as described.

Tessent Diagnosis assigns a suspect a lower score for any of the following cases:

- There are mismatches between simulated faulty behavior and the behavior that was observed on the ATE (pass_mismatch).
- There are failing patterns on the ATE that cannot be explained by any suspect (unexplained_failing_patterns).
- There are multiple symptoms in the report.

In the following example, Tessent Diagnosis assigned a low score when a pattern passed on the tester, but based on simulation of the suspect, it should have failed.

```
suspect score fail_match pass_mismatch type value pin_pathname cell_...
------------------------------------------------------------------------
1       1     10         20            OPEN/DOM 0 /U1680/Y     inv02 ...
```

Large Faulty Scan Cell Range in Chain Diagnosis

Too many faulty cell suspects may be reported for chain diagnosis.

For example:

```
faulty_chain=chain3 #symptoms=1
symptom=1 #suspects=50
```

Check to see how many failing scan test patterns are listed in the failure file. Increasing the number of failing scan patterns should reduce the number of suspects reported. You should use at least 32 failing scan patterns.

Too Many Suspects, Symptoms, or Unexplained Patterns in Logic Diagnosis

In general, you should make sure the chain tests are passing. Logic diagnosis assumes that chains are functioning correctly. Without properly operating chains, it is unclear if the failures are due to defective chains or defective logic whose effect is captured in the chains and leads to inaccurate results.

If for a particular die both chain and scan tests fail, but you do not provide information about the failing chain(s) either by specifying the faulty chain(s) and fault type(s) or by including the chain test failures in the failure file, then logic diagnosis is performed. This could lead to inaccurate and poor results.

Lots of Unexplained Patterns with Poor Diagnosis Resolution

In some rare cases, the chain test passes, but the scan test fails, which produces too many unexplained patterns with poor diagnosis resolution. This can be due to actual defects on the scan chains.

In general, these type of scan chain defects are power-related scan chain hold-time issues that exhibit the following behavior:

- Large number of failing bits on one chain/channel compared to others.
- The failures vary with vdd.

For non-EDT designs, you can debug these by determining the fault model (while considering the chain inversion) and the chain that is failing more frequently. Once you have determined the fault model, then you can specify the model during diagnosis with the diagnose_failures command’s option -faulty_chain switch and arguments.

For EDT designs, debugging the issue is more complex. Assume you have *N* chains associated with one EDT channel that had most of the failures. In this case, you could diagnose one chain at a time and compare the scores and #suspects in all diagnosis results, and choose the best candidates with largest score and minimum #suspects. When using this method, consider the following issues:

- If *N* is a large number, then the tool runtime could be impacted.

- If the design has multiple faulty chains in one EDT channel, the tool cannot diagnose these.
- You should try at least three fault models (fast/fast_to_rise/fast_to_fall) because you may not know the faulty value is one-sided or double-sided for EDT.

Fault Injection Issues

When you inject faults using the write_failures and report_failures commands, the tool check the fault types, and issues warnings certain types of faults.

Tessent Diagnosis checks for the following fault types:

- Tied gate — If the gate is a tied gate (Tie0 / Tie1/ TieX / TieZ), Tessent Diagnosis creates no fail log, skips the fault injection, and issues a warning similar to the following:

  ```
  Pin ... is tied to .... Faul injection skipped.
  ```

- Gate driving a clock — If the gate is driving a clock, Tessent Diagnosis issues a warning similar to the following:

  ```
  Pin ... drives a clock port of a sequential element.
  ```

  Injecting a clock fault can cause chain test failures.

- Gate driving a set/reset — If the gate is driving a set/reset, Tessent Diagnosis issues a warning similar to the following:

  ```
  Pin ... drives set/reset of a sequential element.
  ```

  Injecting a fault that causes a scan cell always in set/reset active mode can cause chain test failures.

- Gate driving a scan_enable — If the gate is driving a scan_enable, Tessent Diagnosis issues a warning similar to the following:

  ```
  Pin ... drives scan enable of a sequential element.
  ```

  Injecting a fault that causes chain always in non-scan mode can cause chain test failures.

For the gate driving a clock, gate driving set/reset, and gate driving a scan_enable, Tessent Diagnosis simulates the injected fault assuming the scan chain works correctly. You should ensure the scan chain works even with defective clock, set/reset, or scan_en.

Abort Conditions for Chain Diagnosis

During chain diagnosis, the Tessent Diagnosis tool can abort chain diagnosis because the tool's requirements for the diagnosis are not met.

Cannot Identify Faulty Chain

Tessent Diagnosis may not be able to identify faulty chains because the failure log does not contain a chain failure or because, in EDT mode, there are no masking chain patterns or failed masking chain patterns.

First Case

There is no chain failure in the failure log, but you have specified the following command:

```
set_diagnosis_options -mode chain
```

Under this circumstance, the tool aborts diagnosing and does not produce a diagnosis report written in the transcript, report text file, or CSV file.

The tool issues the following messages and recommendations:

- Tessent Scan Diagnosis

```
// Note: Cannot identify faulty chains in failure log XXX because
// there is no chain test pattern failures.
// Recommendation: Please use `set_diagnosis_options -mode auto` and
// then run logic diagnosis on this case
```

- Tessent Diagnosis Server

```
// Note: Cannot identify faulty chains in failure log XXX because
// there is no chain test pattern failures.
// Recommendation: Please use `set_diagnosis_options <MONITOR>
// -mode auto`
// and then run logic diagnosis on this case"
```

Second Case

In EDT mode, the tool uses the masking chain patterns to determine the faulty chain. If there are no masking chain patterns or there are no failed masking chain patterns, then the tool cannot identify the faulty chain. Under this circumstance, the tool aborts diagnosing and does not produce a diagnosis report written in the transcript, report text file, or CSV file.

The tool issues the following messages and recommendations:

```
// Note: Cannot identify faulty chains in failure log XXX because there is
// no failed EDT masking chain patterns.
// Recommendation: Please check:
// (1) if there are EDT masking chain patterns in the pattern file
// (2) if these EDT masking failures are truncated
```

No “Usable” Scan Patterns for Chain Diagnosis

Chain diagnosis requires “usable” failed scan patterns. To be usable, the scan pattern must pulse the capture clock for at least one scan cell on at least one faulty chain.

IDDQ patterns or other patterns that do not measure scan cells are always not usable for chain diagnosis. In addition, if a design has multiple clock domains, and if a scan pattern did not pulse the capture clock of any scan cells on a faulty chain, it is equivalent to a chain pattern for this specific chain. If a pattern is equivalent to a chain pattern on all failed scan chains, it is not usable for chain diagnosis.

If there are no “usable” failed scan patterns at all for all failed scan chains, we only report the faulty chains and their fault type after a message.

The tool issues the following messages and recommendations:

```
// Note: Cannot find any scan patterns that pulse the capture clock of
// scan cells on faulty chains in failure log XXX.
// Recommendation: Please use diagnostic ATPG to regenerate scan patterns
// for chain diagnosis
```

Too Many Failing Scan Chains

In some cases, the chain test results show that multiple scan chains had failures in one chip.

There are two possible reasons for this:

- There are indeed multiple defects on multiple scan chains.
- Some global control signals, such as clock tree / scan_enable tree / edt logic have defects.

The more scan chains that failed, the higher the possibility of the second reason. Currently, you can specify when to diagnose global control signals using the following command and switch:

set_diagnosis_options -max_faulty_chains N

Where *N* specifies the maximum number of failing scan chains within a single datalog that are diagnosed using a chain fault model. By default, this is *N*=2.

You can bypass diagnosis when the number of faulty chains (#faulty_chains) is larger than *N* specified with the -max_faulty_chains switch by using the set_diagnosis_options switch and the literal:

set_diagnosis_options –abort_diagnose_many_faulty_chains ON | OFF

The default value is "ON".

By combining these two set_diagnosis_options switches, you can have the following situations:

1. If set_diagnosis_options –abort_diagnose_many_faulty_chains OFF
   a. When the #failed_chains is greater than *N*, the tool runs diagnosis on clock tree and scan_enable tree.
   b. When the #failed_chains is less than or equal to *N*, then the tool assumes the defects are on each of the failed chains and runs diagnosis for each failing chain.
2. If set_diagnosis_options –abort_diagnose_many_faulty_chains ON
   a. When the #failed_chains is greater than *N*, the tool aborts with the following message:

```
// Note: The number of faulty chains XX in failure log XXXX
// exceeds the limit N. Abort diagnosing this case.
// Recommendation: Please use 'set_diagnosis_options
// -max_faulty_chains <N> to change this limit.
```

   b. When the #failed_chains is less than or equal to *N*, the tool assumes the defects are on each failed chain and runs diagnosis for each failing chain.

Too Few Failing Cycles

If the chain failing probability is too low, the tool aborts running chain diagnosis.

Chain failing probability is too low if at least one of the following two conditions is satisfied:

1. For a chain pattern that observes the faulty chain, #failing_bits/ chain_length is less than 1%
2. For scan patterns, #failing_scan_patterns / total_number_patterns_applied_on ATE is less than 0.1%

You can change the default by using the following two commands:

> **set_diagnosis_options –abort_diagnose_minimum_chain_failing_probability min_chain_fail_prob**

Where the *min_chain_fail_prob* is 1. The minimum is 0 (no abort due to this issue). The maximum is 100.

This other command is as follows:

> **set_diagnosis_options –abort_diagnose_minimum_scan_pattern_failing_probability min_scan_fail_prob**

The *min_scan_fail_prob* default is 0.1. The minimum is 0 (no abort due to this issue). The maximum is 100.

In this case, the tool aborts with the following message:

```
// Note: The scan chain failing probability is too low to resume chain
// diagnosis in failure log XXX.
```

Compound Diagnosis Abort Logic Diagnosis Part

In the default chain diagnosis flow, the tool does not diagnose the logic part of compound defects. Compound defects are chain defects that can cause both chain failures and logic failures. The chain diagnosis portion still runs after masking all failing bits caused by logic defects.

In this case, the tool issues the following messages and recommendations:

- Tessent Scan Diagnosis

  ```
  // Note: Failure log XXXX contains chain as well as system logic
  // defects. Only chain defect is diagnosed.
  // Use ‘set_diagnosis_options -abort_diagnose_compound_faults OFF’
  // to diagnose logic part as well.
  ```

- Tessent Diagnosis Server

  ```
  // Note: Failure log XXXX contains chain as well as system
  // logic defects. Only chain defect is diagnosed.
  // Use “set_diagnosis_options <MONITOR>
  // -abort_diagnose_compound_faults OFF” to diagnose the logic part
  // as well.
  ```

You can change this by using the following set_diagnosis_options switch:

> **set_diagnosis_options –abort_diagnose_compound_faults ON | OFF**

The default is set to “ON”, meaning the tool does not diagnose the logic portion of compound defects.

No Failing 1-hot Chain Masking Patterns

Chain diagnosis requires failed 1-hot chain masking patterns to identify faulty chains.

If there are no failing 1-hot chain masking patterns, the tool reports the following error message:

```
// Error: Chain diagnosis requires failed 1-hot chain masking patterns
//        to identify faulty chain(s).
//        Missing failing chain patterns could be the result of an
//        intermittent chain defect. In some cases, re-running the chain
//        tests at a different test corner could result in more
//        predictable defect behavior.
```

If the tool determines that the 1-hot chain masking patterns were truncated, the tool reports the following error message:

```
// Error: Chain diagnosis requires all 1-hot chain patterns to be tested
//        to correctly identify faulty chain(s).
//        Failure log file contains truncated chain test that does not
//        meet this criteria.
```

Long Logic Diagnosis Runtimes

For logic diagnosis, several factors can account for long runtimes.

These factors are:

- Gate count.
- Number of patterns read in for diagnosis.
- Length of the failure file.
- Split capture and clock-off simulation enabled during ATPG, which may double the diagnosis processing time.
- Capture has long sequences of sequential events.

To minimize runtimes, consider the following:

- If you are not performing chain diagnosis, you may not need a long failure file. Check the failure file to see how many failing cycles it contains.
- During the test, you may know that you can only capture failures up to a certain pattern. In the failure file, use a keyword to define the last pattern tested (for pattern-based failure files) or last cycle applied (for cycle-based failure files). Without keywords, the tool simulates all the remaining patterns and assumes they passed the test.
- Use the Tessent Diagnosis pattern sampling feature.
- On the ATE, first apply a stuck-at pattern and then an at-speed pattern.

- For processing volume diagnosis, use Tessent Diagnosis server to process patterns in parallel.

For chain diagnosis, consider the following factors for runtimes:

- Diagnosing intermittent faults takes longer than diagnosing permanent faults.
- Diagnosing multiple faulty chains takes longer than diagnosing single fault chains.
- Diagnosing multiple faults per chain takes longer than diagnosing single faults per chain.
- Diagnosing scan chains with clock/scan_enable takes longer than diagnosing scan chains only.
- Diagnosing long chains takes longer than diagnosing short chains.

To minimize runtimes, consider the following:

- Do not skip pattern verification. Pattern verification not only guarantees the correct patterns, it also precalculates some data that can speed up chain diagnosis. Additionally, you can use a startup cache to save time upon reinvocation of the tool.
- Preferably, run chain diagnosis with EDT enabled rather than in bypass mode.
- Use the appropriate number of scan patterns (between 100 and 1000). If the pattern number is too small, the tool may have to simulate too many cells. If the pattern number is too large, the tool has to simulate too many patterns.

Chapter 3
Layout-Aware Diagnosis and Reporting

The layout-aware diagnosis flow uses your design's LEF/DEF files to create a Layout Database (LDB) on which you perform diagnosis and reporting. Using the layout-aware flow, you can view the diagnosis results in Calibre DESIGNrev, or in a third-party layout viewer.

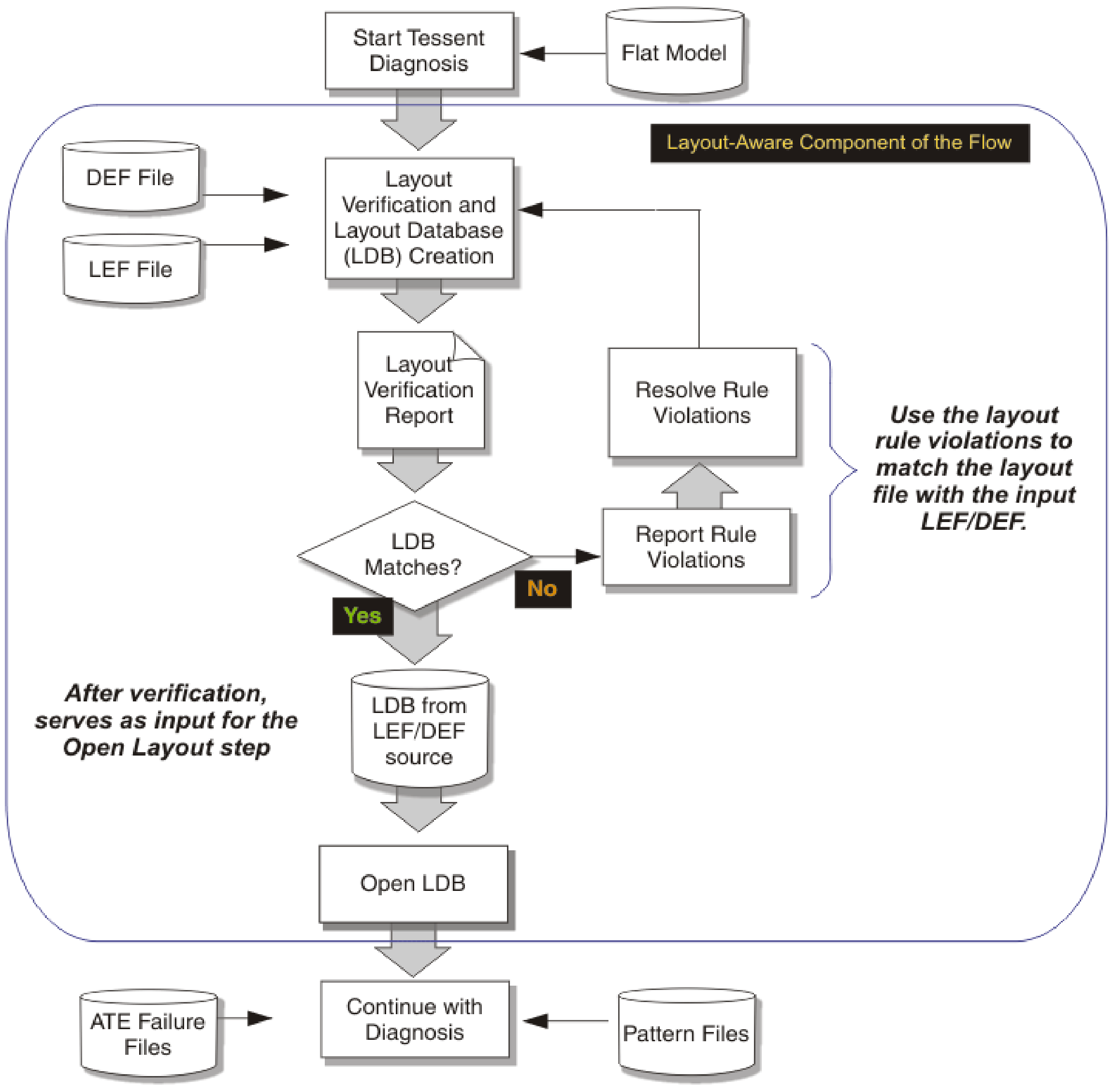
Layout-Aware Diagnosis Flow

The layout-aware diagnosis flow consists of a series of sequential tasks starting with layout verification and LDB creation, proceeding to layout verification reporting and mismatch debugging, and ending with layout-aware diagnosis and reporting.

Figure 3-1 shows a high-level view of the following sequential tasks:

1. Layout Verification and Layout Database Creation Process
2. Layout Verification Reporting
3. Layout and Design Mismatch Debugging
4. Layout-Aware Diagnosis
5. Layout-Aware Diagnosis Reporting

Figure 3-1. Layout-Aware Diagnosis Flow


Start Tessent Diagnosis
Flat Model
Layout-Aware Component of the Flow
DEF File
LEF File
Layout Verification and Layout Database (LDB) Creation
Layout Verification Report
Resolve Rule Violations
Use the layout rule violations to match the layout file with the input LEF/DEF.
LDB Matches?
Report Rule Violations
No
Yes
After verification, serves as input for the Open Layout step
LDB from LEF/DEF source
Open LDB
ATE Failure Files
Continue with Diagnosis
Pattern Files

Layout Database

The LDB you create is a representation of the layout based on the input LEF and DEF files, and your design's flat model that the Tessent Diagnosis tool uses for diagnosis. You use this LDB with Tessent Diagnosis in all subsequent layout-aware diagnosis steps.

By default, the LDB includes the following pre-extracted information for all nets in the design:

- All physical neighbors of the net.
- Net topology (net segments of the net).

Note

To ensure proper net topology extraction for net VIAs, during LEF/DEF generation specify "do not flatten" for net VIAs.

For layout-aware diagnosis, the pre-extracted information stored in the LDB improves performance and memory usage, particularly for extreme corner cases. Rather than recurring the cost of processing physical information on the fly during volume diagnosis, you have a one-time upfront cost for pre-extracting and storing this information.

By creating or opening an LDB with Tessent Diagnosis, you automatically enable layout-aware diagnosis and reporting.

It is best practice to keep the LDB that Tessent Diagnosis generates on the local hard drive of the workstation where the layout-aware diagnosis is running. There is a time penalty if Tessent Diagnosis has to access the LDB over the network. This penalty applies to both the creation of the LDB and its later usage.

LDB Encryption

You can encrypt the design and layout data within an existing LDB so that you can securely send it to third parties (such as to foundries). The data within the LDB is encrypted such that it cannot be decrypted. The only use should be for layout-aware diagnosis in the Tessent environment.

Refer to the encrypt_layout command in the *Tessent Shell Reference Manual* for usage details.

Note

The encryption process is destructive in nature because once the names are removed, they cannot be returned to the LDB. Keep a golden original copy of the LDB for archival purposes.

Layout-Aware Diagnosis Requirements and Limitations

To use layout-aware diagnosis, you must provide the standard diagnosis input files and LEF/DEF files. Layout-aware diagnosis does not support at-speed diagnosis or MBIST diagnosis.

Requirements

Layout-aware diagnosis requires the following files:

- Standard diagnosis input files — Includes design netlist, test patterns, and ATE failure logs. See "Input File Requirements" for complete information.
- LEF/DEF files — Version 5.3 through 5.8, inclusive. You use the LEF/DEF files to create the LDB for subsequent diagnosis by the tool. The tool creates the LDB from these LEF and DEF files.

Limitations

The following Tessent Diagnosis functionality is not supported with layout-aware diagnosis:

- At-speed diagnosis — Layout-aware diagnosis does not apply to at-speed diagnosis. The LEF/DEF-based layout marker file generation, however, does support at-speed—see "Considerations for At-Speed Diagnosis."
- MBIST diagnosis — No support.

Diagnosis Output Files

After performing layout-aware diagnosis, you can write out files that are compatible with Calibre, Tessent Visualizer, and Tessent YieldInsight.

- Layout-aware report — An enhanced diagnosis report containing the specific layout information of each suspect. See "Layout-Aware Diagnosis Reporting."
- Suspect layout viewing and cross probing — Using Calibre® DRC result file syntax for use with Calibre® DRC/Calibre® RVE™ and other layout tools and viewers to overlay the Tessent Diagnosis results with the layout. You must have valid Calibre licenses in order to view the layout markers in Calibre® DESIGNrev. See "Guidelines for Viewing the Diagnosis Results in Calibre DESIGNrev."

  In addition, the Camelot CAD Navigation tool understands the diagnosis report. Special options ('-SHORT -XMAP v2lvs') must be set in Tessent Diagnosis when writing out diagnosis results to include the data needed by Camelot. These options are not active by default.
- Schematic viewing — Tessent Visualizer can read the Tessent Diagnosis layout-aware diagnosis report. Using this tool you can visualize the diagnosis report in Tessent's schematic viewing environment.
- Tessent YieldInsight — Tessent YieldInsight can read the Tessent Diagnosis layout-aware diagnosis reports and display the results in a polygon viewer. See "Polygon Layout Viewer Pane" in the *Tessent YieldInsight User's Manual*.

Layout Verification and Layout Database Creation Process

Layout verification and LDB creation is a three-step phase Tessent Diagnosis performs when you enter the create_layout command.

1. Verification phase — Verifies the LEF/DEF files and the design netlist before creating the LDB. The tool reports the results of the verification in a summary, including specific rule violations, just prior to creating the LDB.

   If the layout and design percentage match is less than the abort threshold percentage that you specified with the create_layout command, or less than the default percentage of 85%, you must manually address and fix the verification mismatches. For an extensive discussion, see "Layout Verification Reporting."

   For net topology, the tool calculates the percentage of nets that it could successfully trace. If the percentage is less than the abort threshold percentage that you specified with the create_layout command, or less than the default percentage of 90%, you must fix the net tracing issues.

   In addition, the tool analyzes the vias and merges all LEF and DEF vias with the same physical information under one via name. This results in consistent via naming in the reports, which can improve the accuracy of RCD results. For more information, refer to the applicable example for create_layout -adjust_vias in the *Tessent Shell Reference Manual*.

   To debug your verification results, see "Layout and Design Mismatch Debugging." To debug errors in net tracing, see "Net Topology Extraction Debugging."

2. LDB Creation phase — Once validation is complete and the mismatch and net topology percentages meet the requirements for the design, the tool creates a LDB.

3. Pre-extraction phase — After the tool has successfully created the LDB, it proceeds to computing the bridge and net topology information, and writes out the pre-extracted bridge and open defect information to the LDB.

Performing Layout Verification and LDB Creation

During layout verification and LDB creation, you invoke Tessent Diagnosis with the flat model of your design, and verify and create a LDB from your design's LEF and DEF files. You subsequently use this LDB during the normal diagnosis process.

For the following procedural example, suppose you have the following input files:

- *my_flat_model* — The flat model of the design.
- *my_design_lef.lef* — The LEF file for the design.
- *my_design_def.def* — The DEF file for the design.

Tip

During the design process, validate the LEF/DEF against the flat models of the cores prior to generating the LDB. This makes it easier to debug any mismatch errors in the design's flat model, especially for large designs.

Tip

Prior to performing the process described in this section, run the analyze_layout_hierarchy command as a one-time setup step. Refer to "Estimation of Resources Required for Generating LDBs" on page 185.

Prerequisites

- You need a flat model and the LEF/DEF files.

Procedure

1. Invoke Tessent Shell, set the context to scan diagnosis, and specify the flat model.

```
Tessent_Tree_Path/bin/tessent -shell -logfile my_logfile

set_context patterns -scan_diagnosis

read_flat_model my_flat_model
```

2. Enter the create_layout command and specify the name you want to call the LDB, and the input LEF and DEF files.

```
FAULT> create_layout my_layout_database -lef my_design_lef.lef
        -def my_design_def.def
```

   You can improve runtime performance by specifying the -threads and -min_threads options. These options enable you to check out multiple licenses so that Tessent Diagnosis can perform some create_layout tasks in parallel.

   During verification, the tool tolerates the differences in escapes in instance pathnames and net pathnames. To turn off tolerant name matching, specify create_layout with the "-do_tolerant_match off" switch.

   In addition, you can specify the create_layout "-compact on" switch to produce a compact LDB. Refer to create_layout for important considerations.

Tip

For maximum efficiency, run the create_layout command on a disk that is local to the machine hosting the process. Additionally, because by default this command reads the entire physical layout information into memory, you should run it on a machine with sufficient physical memory.

3. Depending on the results as described below, you may need to debug errors before the tool generates the LDB. For more information, refer to "Layout and Design Mismatch Debugging" and "Net Topology Extraction Debugging."

   After debugging, you can proceed to "Layout-Aware Diagnosis."

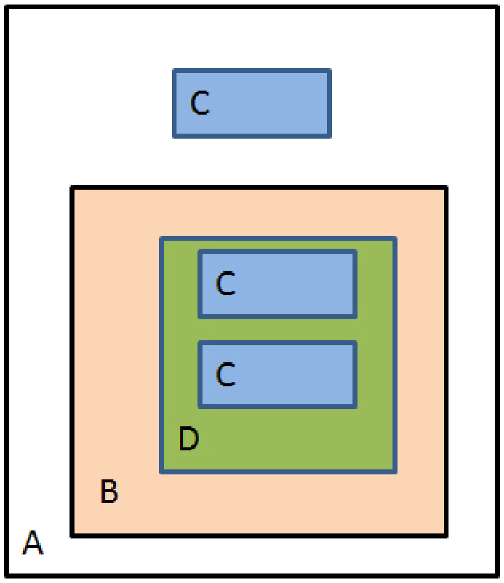
Results

The tool begins the verification process before creating the LDB:

```
//  Note: Processing syntax check for LEF file my_design_lef.lef
//  Note: Processing syntax check for DEF file my_design_def.def
//  Note: Processing up-front validation for LEF file my_design_lef.lef
//  Note: Processing up-front validation for DEF file my_design_def.def
//  Note: Pre-DB Verifying Layout
```

The tool then returns information about the DEF hierarchy structure. For example, given a hierarchy structure for the DEF files as shown in Figure 3-2, the transcript output is as shown below.

Figure 3-2. DEF Hierarchy Tree Example



```
// Start parsing DEF file 'a.def' (1. of 4) in mode \
// COMPLETE_DEF_READ_MODE
// DieArea: minx=-1.000000, miny=-1.000000, maxx=757.000000, \
// maxy=252.000000 in microns
//  ---------------------------------------------------------------------
//  The TOP DESIGN A (file=a.def)
//  DESIGN=A DIEAREA x1=-1000, y1=-1000, x2=757000, y2=252000, \
//  file=a.def
//  ---------------------------------------------------------------------
//
//  ---------------------------------------------------------------------
//  HIERARCHY STRUCTURE (COMPLETE_DEF_READ_MODE):
//  ---------------------------------------------------------------------
//   DESIGN=A DIEAREA x1=-1000,y1=-1000,x2=757000,y2=252000,file=a.def
//   DESIGN=B DIEAREA x1=0, y1=0, x2=367000, y2=180000, file=b.def
//   DESIGN=C DIEAREA x1=0, y1=0, x2=187000, y2=70000,  file=c.def
//   DESIGN=D DIEAREA x1=0, y1=0, x2=330000, y2=140000, file=d.def
//  ---------------------------------------------------------------------
//   SUB DEF   B is placed in   A as instance   t1   (100000,0)      N
//   SUB DEF   C is placed in   A as instance   t2   (500000, 66000) W
//   SUB DEF   C is placed in   D as instance   t1d  (50000, 0)      N
//   SUB DEF   C is placed in   D as instance   t2   (100000, 70000) S
//   SUB DEF   D is placed in   B as instance   t1b  (20000, 30000)  N
//  --------------------------------------------------------------------
```

At the conclusion of verification, the tool reports a summary of rule violations:

```
//  Layout Rule Violation Summary
//  Warning: Rule DesignNetMatch violated 2 times out of 5937 checks.
//  Note: The command 'report_layout_rules' can be used for detailed
//  information on rule violations
```

The tool generates an error if the match percentage match is less than the abort threshold percentage that you specified with the create_layout command, or less than the default percentage of 85%:

```
//  Note: Processing LEF file my_design_lef.lef
//  Note: Processing DEF file my_design_def.def
//  DieArea: minx=-4.000000, miny=-4.000000, maxx=8881.687000,
//  maxy=9328.000000 in microns
//   10% of the layout data has been processed, estimated remaining
//   effort: 39.780 sec
...
//  100% of the layout data has been processed
```

Next, the tool extracts the bridges and net topology:

```
//----------------------------------------------------------------------
//  Reading physical information from LDB
//----------------------------------------------------------------------
//  Step 1 of 10 complete
//  Step 2 of 10 complete
//  …
//  Completed reading physical information
// ---------------------------------------------------------------------
//  Starting bridge calculation (number of layers: 11)
// ---------------------------------------------------------------------
//  10% of bridge calculation is complete
//  20% of bridge calculation is complete
//  …
//  Bridge calculation complete, writing results into layout defects
// database…
//  Successfully calculated bridges for 11 layers
// ---------------------------------------------------------------------
//  Starting net topology calculation (number of nets: 10987)
//  --------------------------------------------------------------------
//  10% of net topology calculation is complete
//  20% of net topology calculation is complete
//  …
//  Net topology calculation complete, writing results into layout defects
// database…
//  Successfully calculated net topology for 10985 out of 10987 nets
SETUP>
```

Estimation of Resources Required for Generating LDBs

Prior to specifying the create_layout command to create an LDB, you can estimate the resources required for the LDB generation process. Do this by specifying the analyze_layout_hierarchy command.

Specify the analyze_layout_hierarchy command once as a setup step prior to loading the flat model and pattern files, and specifying create_layout. The command generates a layout hierarchy database that stores the layout hierarchy of the LEF/DEF designs.

After analyzing the layout hierarchy, the tool can estimate the resources required to create the LDB—full design LDB or chip-mapped core LDB—based on the information extracted from the LEF/DEF files. This helps you ensure that you have the proper resources for LDB creation.

To view the resource estimation, specify the report_layout_hierarchy command. The generated report includes a section with the resource estimations. For example:

```
//  LDB resource estimation for design ls169 with 75 nets.
//  --------------------------------------------------------------
//  LDB resource estimation              |  Verbose  |    Compact
//  -------------------------------------+-----------+--------------
//  Estimated LDB creation time (hrs)    |   0.20    |      0.10
//  Estimated LDB file size (GB)         |   3.00    |      1.50
//  Estimated peak RAM (GB)              |   2.50    |      2.50
//
//  Note: Estimation calculation assumes 'create_layout -threads 16' to
create layout database (recommended).
```

In addition to resource estimation, use the analyze_layout_hierarchy command for the following tasks:

- Debugging DEF files — During analysis of the LEF/DEF designs, the analyze_layout_hierarchy issues an error if there is more than one top DEF module. Refer to "Multiple Top DEF Files Debug" on page 202.
- Optimizing chip-mapped core-level LDB generation time — The existence of the layout hierarchy database enables the tool to generate LDBs for cores without reprocessing all of the LEF/DEF designs. Refer to "Core-Level Layout-Aware Diagnosis" on page 290.

Layout Database Compression and Decompression

By default, Tessent Diagnosis creates an uncompressed LDB when you create the database with the create_layout command. Several create_layout options allow you to control Tessent Diagnosis LDB compression operations.

Use the following create_layout options to control LDB compression operations:

Note

Compressed LDBs may increase processing times when you are running many parallel diagnosis jobs that are accessing the same LDB. Some jobs may hang when querying the LDB.

- -temp_directory *directory* option — Specifies an alternative location to store the uncompressed LDB during the database creation. The default location is the working directory. When you specify this switch and associated string, the tool writes the uncompressed LDB to the location and, after successful compression, deletes the uncompressed LDB.
- -compression on | off option — Specifies whether the tool compresses the LDB. The default is OFF. With LDB compression, the size of the database on disk can be reduced up to 70 percent. You can subsequently use the open_layout command to access the compressed LDB for running layout-aware diagnosis.

- -keep_uncompressed_database option — Specifies that the tool retains a copy of the uncompressed LDB as well as a compressed version of the LDB.

Database compression is not synonymous with file compression—for example, using the UNIX gzip command. Database compression is a fundamental re-structuring of the database, producing a file that is smaller but contains the same level of information. Its use is recommended when large file sizes pose a concern.

If you compress the files within an LDB using a file compression utility such as gzip, Tessent Diagnosis cannot open the database and it turns off the layout-aware diagnosis.

Tessent Diagnosis provides the following two commands you can use to compress or decompress a LDB outside of the layout-aware database creation operation:

- compress_layout
- uncompress_layout

Note
To use these commands, you must first open the LDB using the open_layout command.

Parallel Operations With the Same LDB

After you have created a LDB, you can perform several operations in parallel in different tool sessions.

These operations are:

- Verify various flat models.
- Create designs constants for root cause deconvolution (RCD) for different flat model/ pattern set pairs.
- Verify a flat model and run layout aware diagnosis on a different flat model.

Create design constants for RCD and run layout aware diagnosis on a different flat model/ pattern set pair.

Layout Verification Reporting

Layout verification automatically occurs when you create a new LDB with the create_layout command or open a previously-created LDB with the open_layout command.

Before creating the LDB, Tessent Diagnosis performs layout verification using a rules-based approach. Tessent Diagnosis creates the LDB if the design and layout match percentage is greater than the abort threshold percentage that you specified with the create_layout command, or greater than the default percentage of 85%. Otherwise, the tool halts processing.

The tool checks for and reports rule violations and mismatches. In addition, the tool checks the consistency of the LEF/DEF files. The following provides a summary of the major rule categories the tool uses to check the consistency of the LEF/DEF files before creating the LDB.

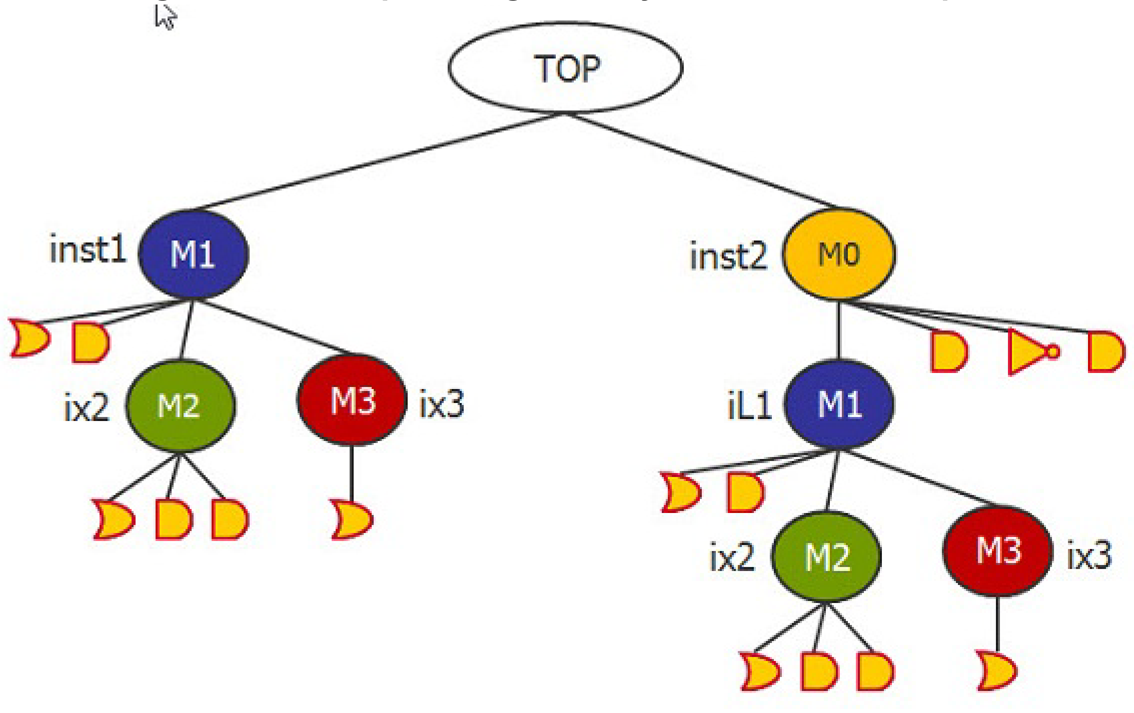
- Chip Boundary Rules
- Instance Rules
- Layer Definition Rules
- Net Rules
- Taper Rules
- Via Definition Rules
- Macro Definition Rules

Example Layout Verification Report Format

Upon completion of the layout-verification step, Tessent Diagnosis prints to stdout summaries of the library cell instance mismatches and layout rule violations, and a mismatch report.

Consider a design with the following hierarchy that consists of four modules: M0, M1, M2, and M3. The smaller yellow symbols represent library cell instances.

Figure 3-3. Example Design for Layout Verification Report



Example 1

Suppose you have specified:

```
FAULT> create_layout MY_LDB
        -def top.def M1.def M2.def       <-- M3 DEF file not provided
        -lef or.lef and.lef              <-- LEF for not cell not provided
```

The resulting report looks as follows. Each section is described in section "Layout Verification Report Details," but at a high-level you can see that the Mismatch Report shows that there are 3 undefined design cell instances that, according to the Layout Rule Violation Summary, are due to two DesignInstanceMatch rule violations and one MacroExistence rule violation error.

In addition, the "Design Cell Instance Mismatches Summarized by Design Modules" section provides a breakdown of the mismatches. In this section you can see that module M0 is the source of the MacroExistence error, and module M3 is the source of the two DesignInstanceMatch errors.

```
//DesignCell Instance Mismatches Summarized by Design Modules
// (Note: Each row in the table breaks down the design cell instance
// (  count in a module into various buckets)
// (Note: The counts in the various columns are aggregated over all
//   instances of the module in the design)
//  Module Name, Number of Instances of Module in the Design, Design Cell
Instances Included in Reporting, Mismatch due to DesignInstanceMatch,
Mismatch due to MacroExistence
// TOP,0,0,0,0
// M1,2,4,0,0
// M0,1,3,0,1
// M2,2,6,0,0
// M3,2,2,2,0
//  Column Sums,-,15,2,1
//  End of Design Cell Instance Mismatches Summarized by Design Module
//
…
//  Layout Rule Violation Summary
//  Warning: Rule DesignInstanceMatch violated 2 times out of 15 checks.
//  Warning: Rule MacroExistence violated 1 out 15 checks.
//
// Mismatch Report
//  3  (  20.00%)     design cell instances undefined (DesignInstanceMatch
rule, MacroExistence rule)
// 12  (  80.00%)     design cell instances matched with layout (common
area)
// ------------------------
// 15                    total number of design cell instances
//
//  6  ( 37.50%)      nets at the boundary of common area (MacroExistence
rule, DesignInstanceMatch rule)
//  10  ( 62.50%)     nets matched with layout
// ------------------------
//  16                total number of design nets
//
//  Warning: Design and layout only match to 62.50%
//           Mismatch may result in incomplete layout aware diagnosis.
//
```

Example 2

Now suppose you have specified the following:

```
FAULT> create_layout MY_LDB
          -def top.def M3.def        <-- M1 and M2 DEF files not provided
          -lef M1.lef M2.lef         <-- M1 and M2 are LEF macros instead
          -lef or.lef and.lef not.lef
```

The resulting report looks as follows. You can see that there is a new data point in the "Design Cell Instance Mismatches Summarized by Design Modules" section called "Mismatches due to DesignModuleCell" with a total of 12 mismatches that correspond to the 12 undefined design cell instances in the mismatch report.

DesignModuleCell mismatches generate an additional section in the report called “Information mismatches due to DesignModuleCell violations” that is described in detail in “Layout Verification Report Details.”

```
//DesignCell Instance Mismatches Summarized by Design Modules
//(Note:Each row in the table breaks down the design cell instance
//(  count in a module into various buckets)
//(Note:The counts in the various columns are aggregated overall
//   instances of the module in the design)
//  Module Name, Number of Instances of Module in the Design, Design Cell
Instances Included in Reporting,Mismatch due to DesignModuleCell
// TOP,0,0,0
// M1,2,4,4
// M0,1,3,0
// M2,2,6,6
// M3,2,2,2
//  Column Sums,-,15,12
//  End of Design Cell Instance Mismatches Summarized by Design Module
/
//  Information on mismatches due to
DesignModuleCell violations
//  Module Name with Design Module Cell Violation, Mismatch due to this
module
//  M1,6
//  M2,6
//  Column Sum,12
//  End of Information on mismatches due to DesignModuleCell violations
//
…
//  Layout Rule Violation Summary
//  Warning: Rule DesignModuleCell violated 2 times out of 4 checks.
//  Warning: Rule DesignInstanceMatch violated 12 times out of 15 checks.
//
//  Mismatch Report
// 12  ( 80.00%)    design cell instances undefined (DesignInstanceMatch
rule)
//   3  ( 20.00%)    design cell instances matched with layout (common
area)
// ------------------------
//  15                         total number of design cell instances
//
//  10  ( 62.50%)    nets outside of common area (DesignInstanceMatch
rule)
//   4  ( 25.00%)    nets at the boundary of common area (MacroExistence
rule, DesignInstanceMatch rule)
//   2  ( 12.50%)    nets matched with layout
// ------------------------
//  16                 total number of design nets
//
//  Warning: Design and layout only match to 12.50%
//           Mismatch may result in incomplete layout aware diagnosis.
//
…
```

Layout Verification Report Details

The layout verification report consists of five sections that provide information about design-versus-layout mismatches and rule violations.

As shown in section "Example Layout Verification Report Format," the main elements of the report, in order, are:

1. Design Cell Instance Mismatches Summarized by Design Modules
2. Information on Mismatches Due to DesignModuleCell Violations
3. Layout Rule Violation Summary
4. Mismatch Report
5. Design and Layout Match Percentage

Design Cell Instance Mismatches Summarized by Design Modules

The first section in the layout verification report is titled "Design Cell Instance Mismatches Summarized by Design Modules." This section provides details about library cell instance mismatches broken down by root cause and module.

The report presents this information in a CSV-style table. Each row in the table breaks down the design cell instance count in a module into various buckets. The counts in the various columns are aggregated over all instances of the module in the design.

The Example 1 report in section "Example Layout Verification Report Format" shows:

```
//Design Cell Instance Mismatches Summarized by Design Modules
// (Note: Each row in the table breaks down the design cell instance
//        count in a module into various buckets)
// (Note: The counts in the various columns are aggregated over all
//        instances of the module in the design)
// Module Name, Number of Instances of Module in the Design, Design Cell
Instances Included in Reporting, Mismatch due to DesignInstanceMatch,
Mismatch due to MacroExistence

// TOP,0,0,0,0
// M1,2,4,0,0
// M0,1,3,0,1
// M2,2,6,0,0
// M3,2,2,2,0
// Column Sums,-,15,2,1
// End of Design Cell Instance Mismatches Summarized by Design Modules
```

As shown in bold above, the report lists the column headers for the comma-separated data that follows. Each row in the table applies to one higher level module. The Column Sums line in the table lists the mismatch count due to each root cause summed over all the modules.

The following table describes the columns that can appear in a report. If a rule is not violated, its corresponding column does not appear in the report.

Table 3-1. Design Cells Instance Mismatches Summarized by Design Modules Table Columns

Column Name	Column Description
Module Name	Name of the non-library cell module for which data is reported in the current line.
Number of Instance of Modules in the Design	The number of times the current module is instantiated in the entire design. The top module is given an instance count of 0. For example, M1 is instantiated twice in the design shown in Figure 3-3. Module M0 is instantiated once.
Design Cell Instances Included in Reporting	The number of library cell instances in the current module that are matched against LEF/DEF given that: • This number is counted only for the current module and not for any other non-library cell modules instantiated in the current module. • The count is aggregated over all the instances of the current module in the design. For example, in Figure 3-3 the counts are: M0=3, M1 =2x2=4, M2=2x3=6, and M3=2x1=2.

Table 3-1. Design Cells Instance Mismatches Summarized by Design Modules Table Columns (cont.)

Column Name	Column Description
Mismatch due to DesignInstanceMatch	The number of library cell instances in the current module that are not found in LEF/DEF. The tool aggregates this count over all the instances of the current module in the design.
Mismatch due to MacroExistence	The number of library cell instances in the current module that match with DEF but the corresponding LEF macro for the instance is not defined. The tool aggregates this count over all the instances of the current module in the design.
Mismatch due to DesignModuleCell	The number of library cell instances in the current module that do not match DEF because of a DesignModuleCell violation. A DesignModuleCell violation occurs when a non-library cell module in the design is defined as a LEF macro instead of a DEF design in LEF/DEF. The tool aggregates this count over all the instances of the current module in the design.

Missing DEF Files

The Design Cell Instance Mismatches Summarized by Design Modules section of the layout verification report can help you quickly identify missing DEF files. For example, consider module M3 in the Example 1 report.

```
// Module Name, Number of Instances of Module in the Design, Design
Cell Instances Included in Reporting, Mismatch due to
DesignInstanceMatch, Mismatch due to MacroExistence
...
// M3,2,2,2,0
...
```

The data tells you that there are two instances of this module in the design, a total of two library cell instances included in the report (because M3 contains one instance of a library cell), and that neither of these instances were found in the LEF/DEF (DesignInstanceMatch violations). Given this, there is a high probability that the entire definition of M3 is missing.

Missing LEF Macros

Now consider the scenario in which a LEF macro that is specified for a gate type is not specified in the create_layout command. In the Example 1 report, this applies to the not gate and the column header “Mismatch due to MacroExistence” is displayed in the report. The report shows that this results in a library mismatch error for a M0 library cell instance.

```
// Module Name, Number of Instances of Module in the Design, Design Cell
Instances Included in Reporting, Design due to DesignInstanceMatch,
Mismatch due to DesignModuleCell,Mismatch due to MacroExistence
...
// M0,1,3,0,1
...
```

Information on Mismatches Due to DesignModuleCell Violations

Mismatches can occur when a non-library cell module in the design is defined as a LEF macro instead of a DEF design in LEF/DEF. These violations are called DesignModuleCell violations. The second section of the layout verification report, "Information on mismatches due to DesignModuleCell violations," reports the contribution of each DesignModuleCell violation to the overall undefined cell instance count.

This section displays in the report only when layout verification results in DesignModuleCell violations.

As shown below, the snippet from the Example 2 report in section "Example Layout Verification Report Format" consists of the following two columns:

- Module Name with Design Module Cell Violation — Specifies the name of the design module that is specified as a LEF macro instead of a DEF design.
- Mismatch due to this module — Lists the library cell instances that do not match in the LEF/DEF due to the DesignModuleCell violation of the current module. The tool arrives at this number by first summing the library cell instance counts in the current module and recursively traversing down the hierarchy to all its children, except those that themselves have a DesignModuleCell violation. Next it multiplies this sum by the number of instances of the current module in the entire design.

```
//Information on mismatches due to DesignModuleCell violations
// Module Name with Design Module Cell Violation, Mismatch due to this
module
// M1,6
// M2,6
// Column Sum,12
// End of Information on mismatches due to DesignModuleCell violations
```

Design Modules Defined as LEF Macros

The Example 2 report shows that all 12 undefined design cell instances are due to DesignModuleCell violations. The table shown above breaks down the violations by each violation of the DesignModuleCell rule. Modules M1 and M2 contain this violation. The M1 violation results in 6 cell instances being undefined (2 cell instances in M1 plus 1 cell instance in M3 multiplied by 2, the number of instances of M1 in the entire design).

Note

The library cell instances in M2 are not included in the table because M2 itself has a DesignModuleCell violation.

Similarly, the M2 violation also results in 6 undefined cells (3 cell instances in M2 multiplied by 2, the number of instances of M2 in the entire design). From these results, you can conclude that the design modules for M1 and M2 were defined as LEF macros rather than DEF designs.

Layout Rule Violation Summary

The third section of the layout verification report consists of the Layout Rule Violation Summary.

Consider the Example 1 report in section "Example Layout Verification Report Format." The Layout Rule Violation Summary displays as follows:

```
//  Layout Rule Violation Summary
//  Warning: Rule DesignInstanceMatch violated 2 times out of 15 checks.
//  Warning: Rule MacroExistence violated 1 out 15 checks.
```

The summary correlates to the Column Sums line near the end of the "Design Cell Instance Mismatches Summarized by Design Modules" section as follows:

```
...
//  Module Name, Number of Instances of Module in the Design, Design Cell
Instances Included in Reporting, Mismatch due to DesignInstanceMatch,
Mismatch due to MacroExistence
// TOP,0,0,0,0
// M1,2,4,0,0
// M0,1,3,0,1
// M2,2,6,0,0
// M3,2,2,2,0
// Column Sums,-,15,2,1
// End of Design Cell Instance Mismatches Summarized by Design Modules
...
```

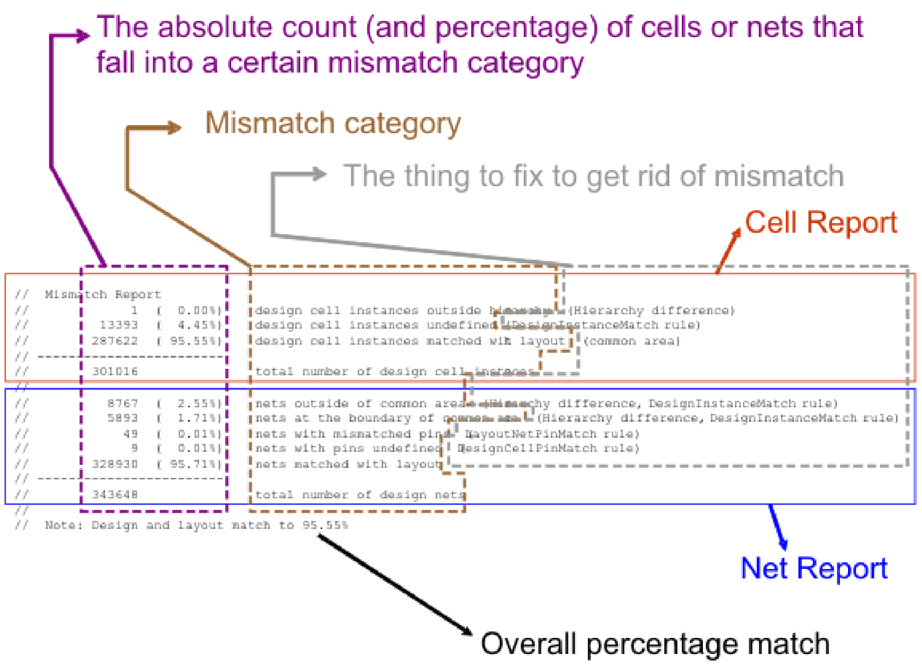
- The column sum for Mismatch due to DesignInstanceMatch shows 2, which displays as the first warning in the summary.
- The column sum for Mismatch due to MacroExistence shows 1, which displays as the second warning in the summary.

Mismatch Report

After the layout rule violation summary, the tool issues a layout verification mismatch report to stdout.

The following figure illustrates a layout verification mismatch report. (This example does not correspond to a previously illustrated example.) The layout verification mismatch report summarizes the contribution of the missing design net or library cell instances creating an overall mismatch between design and layout.

Figure 3-4. Layout Verification Mismatch Report Example


The absolute count (and percentage) of cells or nets that fall into a certain mismatch category
Mismatch category
The thing to fix to get rid of mismatch
Cell Report
Net Report
Overall percentage match

```
//  Mismatch Report
//          1  (  0.00%)  design cell instances outside hierarchy (Hierarchy difference)
//      13393  (  4.45%)  design cell instances undefined (DesignInstanceMatch rule)
//     287622  ( 95.55%)  design cell instances matched with layout (common area)
// -----------------------------
//     301016             total number of design cell instances
//
//       8767  (  2.55%)  nets outside of common area (Hierarchy difference, DesignInstanceMatch rule)
//       5893  (  1.71%)  nets at the boundary of common area (Hierarchy difference, DesignInstanceMatch rule)
//         49  (  0.01%)  nets with mismatched pins (LayoutNetPinMatch rule)
//          9  (  0.01%)  nets with pins undefined (DesignCellPinMatch rule)
//     328930  ( 95.71%)  nets matched with layout
// -----------------------------
//     343648             total number of design nets
//
//  Note: Design and layout match to 95.55%
```

The mismatch report contains the following sections:

- The first section provides details about the library cell instance mismatches.
- The second section contains information about net mismatches.
- The end of the mismatch report displays the overall percentage match between the design and layout. Refer to "Design and Layout Match Percentage" for details and "Layout and Design Mismatch Debugging" for information about how to raise this percentage.

In general, if the layout information for a design net or library cell instance is missing, then this directly impacts diagnosis.

Layout Rule Violations Related to Library Cells

The verification process checks that all the library cell instances in the design can be found in the layout. If the tool cannot find a cell instance in the layout then it is considered a mismatch.

Library cell instances in the design do not have corresponding layout information if:

- They are outside the hierarchy (specifically, outside of the topmost common module). This is due to underlying hierarchical factors.

- They are undefined in the layout. This can be due to two reasons:
    - If a design cell instance is missing from the layout as indicated by DesignInstanceMatch violations.
    - If a design cell instance exists in the layout, however, the corresponding LEF macro is not defined as indicated by violations of the MacroExistence rule.

Layout Rule Violations Related to Nets

Several layout rule violations related to nets can impact diagnosis results.

- Nets outside of common area — The common area is defined as that part of the design that is in common with the layout. Ideally, the common area should be the entire design; however, in practice, the common area may be a subset of the design.

  If the common area is smaller than the design, then no layout information is available for nets outside of the common area. The common area is generally smaller than the design due to hierarchy level differences and design instances missing in the layout (specifically, violations of the DesignInstanceMatch or MacroExistence rule).
- Nets at the boundary of common area — If a net is connected to a pin that is not in the common area, then the layout information on such a net is incomplete. The common area is generally smaller than the design due to hierarchy level differences and design instances missing in the layout (specifically, violations of the DesignInstanceMatch or MacroExistence rule).
- Nets not found in layout — This is indicated by violations of the DesignNetMatch rule.
- Nets with pin mismatch — If the pins connected to a net in design do not match the pins connected to the corresponding net in the layout then the layout information for this net cannot be used. This situation is indicated by violations of the DesignNetPinMatch or LayoutNetPinMatch rule.
- Nets with pins undefined — Nets that are connected to library cell instances that have a DesignCellPinMatch violation have incomplete layout information.

Design and Layout Match Percentage

The last line of the mismatch report shows the design and layout match as a percent.

For example:

```
//  Note: Design and layout match to 99.97%
```

This number tells you how well the design in the flat model and the LEF/DEF physical layout match each other. The lower the percentage match, the greater the chance that the diagnosis results may not correlate with physical data. For this reason, Tessent Diagnosis automatically aborts LDB creation if this percentage is less than the abort threshold percentage that you specified with the create_layout command, or less than the default percentage of 85%.

If the match percentage is less than 100 percent, you could end up with suspects that do not have any layout data. Therefore, you must ensure that the design and layout match is as close to 100 percent as possible.

Note

After the create_layout command's layout verification stage is complete, you can interrupt the LDB creation by issuing a **Ctrl-C** in the invocation shell.

See section "Layout and Design Mismatch Debugging" for debugging tips that can help you improve your match percentage.

Rules That Directly Impact the Match Percentage

Several layout rules directly impact the layout and design match percentage, and you must ensure they are clean before proceeding with diagnosis.

For details about debugging mismatch errors, see "Layout and Design Mismatch Debugging."

Figure 3-1 lists the layout rules that directly impact the match percentage.

Table 3-2. Layout Rules that Matter for Diagnosis

Rule	Comments
DesignInstanceMatch	All instances in the design (library cell instances as well as higher level instances) must exist in the layout.
MacroExistence	A specified component must use a macro that is defined in the specified LEF files.
DesignNetMatch	All design nets must exist in layout
	The net layout information must be complete (no boundary nets)
DesignNetPinMatch	All pins connected to a net in design must match those in layout
LayoutNetPinMatch	
DesignCellPinMatch	

lists the layout rules that directly impact the match percentage.

Refer to "Layout-Aware Diagnosis Layout Verification Rules" for detail information about the layout verification rules.

Layout and Design Mismatch Debugging

The basic debugging strategy is to fix rule violations and exclude LEF/DEF design blocks that are missing from the physical layout.

Fix Rule Violations. In general, you use the following approach to debug, fix, and verify mismatches caused by rule violations:

- View mismatch report to determine the specific rule violations—see "Layout-Aware Diagnosis Layout Verification Rules" for complete rule descriptions.
    - First debug low instance match number.
    - Then debug any remaining net problems.
- Look at other rule violations for clues.
- LEF/DEF is normally generated automatically; look for systemic issues.
- Repeat the LDB verification and LDB creation step using the create_layout command.

See "Layout Verification Examples" for step-by-step examples of debugging verification issues.

Exclude LEF/DEF Design Blocks. In some cases, the LEF/DEF may not exist for some design blocks. In these cases, you have no option but to exclude these design blocks from verification. The report_layout_rule command enables you to exclude design blocks that do not have a match in the physical layout.

In the following example, Tessent Diagnosis stops processing because the match percentage is only 82.03%, which is below the default threshold of 85%. After reviewing the mismatch report, you exclude the cpu_edt and the block06 design blocks from verification, which improves the match percentage up to 99.85%. Then, you re-run create_layout with a lower threshold setting, knowing that although the match for the entire design is low, it is acceptable when you exclude blocks you have determined can be excluded.

```
ANALYSIS> create_layout design.ldb -def design_top.def -lef lib.lef
...
//  Mismatch Report
// ...
//  Warning: Design and layout only match to 82.03%
//           Mismatch may result in incomplete layout aware diagnosis.
//
//  Error: The percentage match between the design and layout is lower
//  than the threshold of 85.00%
//         Due to this low match between design and layout verification
//         will abort.
//         This threshold can be lowered using the -threshold switch.
// ...
ANALYSIS> report_layout_rules -mismatch_report
// ...
//  Warning: Design and layout only match to 82.03%
//           Mismatch may result in incomplete layout aware diagnosis.
//
ANALYSIS> report_layout_rules -mismatch_report -exclude cpu_edt
// ...
//  Warning: Design and layout only match to 90.82%
//           Mismatch may result in incomplete layout aware diagnosis.
//
ANALYSIS> report_layout_rules -mismatch_report -exclude cpu_edt block06
// ...
//  Note: Design and layout match to 99.84%
//
ANALYSIS> create_layout design.ldb -def design_top.def -lef lib.lef -
threshold 70 -replace
// ...
//  Warning: Design and layout only match to 82.03%
//           Mismatch may result in incomplete layout aware diagnosis.
//
// ...
//  Note: Violations database opened successfully.
//  Note: Created layoutDB ( layout.ladb )
```

Suggested Flow for Debugging Layout Verification Report Undefined Cell Instances

The main debugging strategy consists of debugging and fixing mismatches caused by rule violations and excluding (or including) missing design blocks from the reporting.

You can use the following flow to help you debug undefined cell instances in the layout verification report:

1. Under “Design Cell Instance Mismatches Summarized by Design Modules,” review the breakdown of the undefined cell instances by root cause: MacroExistence, DesignModuleCell or DesignInstanceMatch in the Column Sums line.

2. For MacroExistence violations, retrieve the details using the following command:

   report_layout_rules MacroExistence

   Find and fix the unspecified LEF macros, prioritizing the macros according to those with the most number of violations.

3. Under "Information on mismatches due to DesignModuleCell violations," review the breakdown for the DesignModuleCell violations and specify the DEF files for violating modules, prioritizing them by their impact on the undefined cell count. Correct the violations in the LEF files and identify the missing DEFs. For details, use report_layout_rule designModuleCell.

4. For DesignInstanceMatch violations, check the modules listed under "Design Cell Instance Mismatches Summarized by Design Modules" for those in which the number of Mismatch due to DesignInstanceMatch violations is equal to the number of Design Cell Instances Included in Reporting. This indicates that the DEF files corresponding to the modules could be missing.

   See section "Guidelines for Including or Excluding Design Modules From Mismatch Reporting" for more about including or excluding missing design blocks.

5. Once the missing DEF files have been identified, debug the remaining DesignInstanceMatch violations in various modules prioritized by their contribution to the total undefined cell count.

Multiple Top DEF Files Debug

To create an LDB you should only have one top DEF module. The existence of multiple top modules causes the tool to stop the verification process and indicates an error in the list of DEF files you specified with the create_layout command.

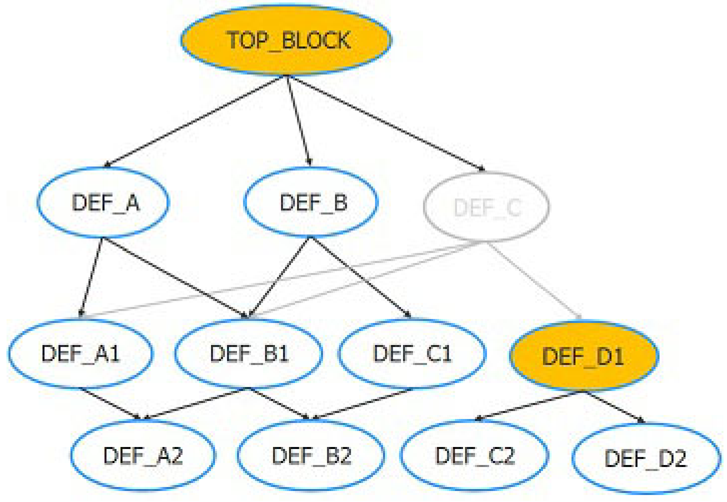
To verify that you have only one top DEF module, prior to executing create_layout, specify the analyze_layout_hierarchy command. This command generates a layout hierarchy database. However, the layout hierarchy database is only created if one top DEF module exists. When there are multiple top DEF modules, the tool produces reports for each top module and issues an error.

To fix the issue, you must identify the DEF files that are missing or unnecessary. The following sections describe scenarios that could result in multiple top module definitions.

Missing DEF Files

In the figure below, the DEF_C file was missing from the list of DEF files, which results in a secondary top file, DEF_D1. To complete the design, you would need to provide the DEF_C file.

Figure 3-5. Missing DEF Files

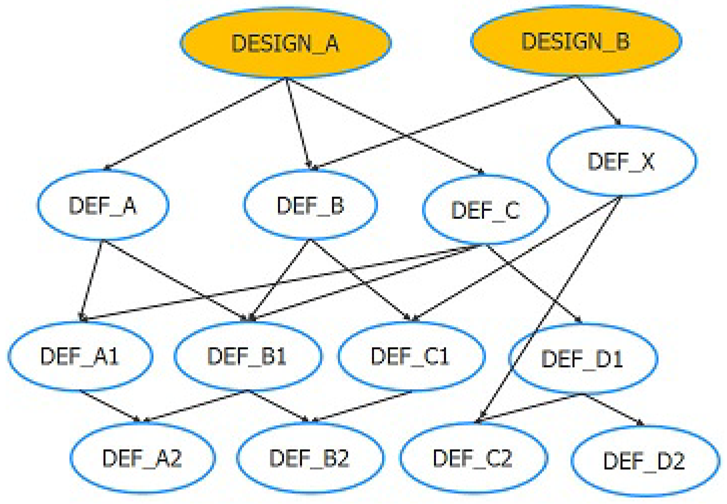


In this case, allowing the tool to ignore all but the TOP_BLOCK would lead to an incomplete LDB. This is especially true in cases where each module also has a corresponding LEF macro definition (that should have been overridden by the missing DEF).

Unnecessary DEF Files

In the following figure, DESIGN_A and DESIGN_B use a common set of sub-DEF files. Each top module is valid, but for the purposes of creating the LDB you need to remove one top design by removing the DESIGN_B.def and DEF_X.def files.

Figure 3-6. Multiple Top DEF Files



The Top Module Report

The top module report provides information about the DEF files that are associated with each top design. You can use this information to help you pinpoint missing DEF files or extra DEF files that are causing you to have multiple top-level DEF modules.

For each top DEF design that the tool finds, it reports a list of sub-DEFs instantiated by the top DEFs, the full pathnames for the DEF modules, and a list of DEFs that you can use with create_layout to instantiate only that DEF. Because this report is produced for every top module, you can decide whether the DEF files for the desired top module are relevant and complete with respect to the design.

The report is divided into three sections. The first section reports the unique modules at each level of hierarchy and the number of instantiations of that module within its immediate parent. For example, in the following report, TOP_DEF_A instantiates DEF_A_3 three times, and DEF_A_3 instantiates DEF_A1_6 six times.

```
//  --------------------------------------------------------------------------------
//  --------------------------------------------------------------------------------
//  Top module (1 of 2) : TOP_DEF_A
//     DIEAREA x1=-1000, y1=-1000, x2=757000, y2=600000
//  --------------------------------------------------------------------------------
//  ------------------------------------------------
//  MODULE NAME TREE   | #INSTANCES IN PARENT MODULE
//  -------------------+----------------------------
//  TOP_DEF_A          |                          --
//     DEF_A_3         |                           3
//        DEF_A1_6     |                           6
//           DEF_A2_17 |                          17
//        DEF_B1_5     |                           5
//           DEF_A2_17 |                          17
//           DEF_B2_3  |                           3
//     DEF_B_4         |                           4
//        DEF_B1_5     |                           5
//           DEF_A2_17 |                          17
//           DEF_B2_3  |                           3
//        DEF_C1_9     |                           9
//           DEF_B2_3  |                           3
//           DEF_C2_9  |                           9
//     DEF_C_2         |                           2
//        DEF_A1_6     |                           6
//           DEF_A2_17 |                          17
//        DEF_B1_5     |                           5
//           DEF_A2_17 |                          17
//           DEF_B2_3  |                           3
//        DEF_D1_4     |                           4
//           DEF_C2_9  |                           9
//           DEF_D2_5  |                           5
```

The second section of the report lists the DEF modules that comprise the top block and paths to their DEF file definitions. For each DEF module, the tool also reports the number of instantiations of the module in the design, the number of nets and components for each module, and the die area for each module.

```
// ------------------------------------------------------------------------------------
// MODULE NAME |     TOTAL         | #COMPONENTS |  #NETS      | DIEArea     | FILENAME
//             | #INSTANTIATIONS   |  IN MODULE  | IN MODULE   | in um       |
//             |   IN DESIGN       |             |             |             |
// ------------+-------------------+-------------+-------------+-------------+---------------
//  TOP_DEF_A  |        1          |      1      |     3       |  50 x 60    | TOP_DEF_A.def
//  DEF_A_3    |        3          |      1      |     3       |  50 x 60    | DEF_A_3.def
//  DEF_A1_6   |       30          |      3      |     3       |  50 x 60    | DEF_A1_6.def
//  DEF_A2_17  |     1275          |      0      |     3       |  50 x 60    | DEF_A2_17.def
//  DEF_B1_5   |       45          |     60      |     3       |  50 x 60    | DEF_B1_5.def
//  DEF_B2_3   |      243          |     15      |     3       |  50 x 60    | DEF_B2_3.def
//  DEF_B_4    |        4          |      1      |     3       |  50 x 60    | DEF_B_4.def
//  DEF_C1_9   |       36          |      1      |     3       |  50 x 60    | DEF_C1_9.def
//  DEF_C2_9   |      396          |     10      |     3       |  50 x 60    | DEF_C2_9.def
//  DEF_C_2    |        2          |    180      |     3       |  50 x 60    | DEF_C_2.def
//  DEF_D1_4   |        8          |      1      |     3       |   3 x 3     | DEF_D1_4.def
//  DEF_D2_5   |       40          |      1      |     3       |   3 x 3     | DEF_D2_5.def
```

The third section of the report provides a DEF file list of the modules that make up the designated top module. Assuming that the report correctly describes the top module, you can specify the DEF file list verbatim with the create_layout command.

```
// If the module hierarchy described above accurately represents the top module 'TOP_DEF_A',
// you can specify the DEF file list with the 'create_layout' command using the '-def' switch
// as follows:-def ../data/TOP_DEF_A.def \
     ../data/DEF_A_3.def \
     ../data/DEF_A1_6.def \
     ../data/DEF_A2_17.def \
     ../data/DEF_B1_5.def \
     ../data/DEF_B2_3.def \
     ../data/DEF_B_4.def \
     ../data/DEF_C1_9.def \
     ../data/DEF_C2_9.def \
     ../data/DEF_C_2.def \
     ../data/DEF_D1_4.def \
     ../data/DEF_D2_5.def
```

Net Topology Extraction Debugging

After debugging and fixing mismatch errors, the tool extracts bridge and net topology. For net topology, it calculates the percentage of nets that could be successfully physically traced from their drivers to their sinks. If this percentage falls below 90%, the tool issues an error.

Regardless of the calculation percentage, the tool reports the results of the net topology extraction. You can use the information reported in the transcript to help you debug electrical issues with the nets (that is, broken nets).

The Net Topology Extraction Transcript

The net topology extraction results transcript is composed of two main sections. The first section summarizes all the layout rule violations that directly result in topology errors. The second section provides a detailed categorization of nets into five issue classes, each one representing a possible reason for unsuccessful topology.

The tool reports the percentage of nets that could not be electrically traced followed by the summary of the layout rule violations that directly impact topology calculation. For each layout rule, the tool reports the first violation that would be displayed in the full report generated by the report_layout_rule command. For example:

```
Error: Net topology could only be calculated for 43,75% of the nets (7 out of 16 nets)
     This means that 56,25% of nets could not be electrically traced from the driver gate
     to the sink gate(s).

     Please review the violations of the following layout rules for possible causes:
     (the first rule violation is reported)

LayerExistenceNet: #fails=3 #checks=119 handling=warning (layer in net not exist).
//  Warning: LayerExistenceNet-1: The specified NET net_unknown_layer1 in DEF file.def uses
//  a LAYER route_55 which is not specified in the LEF files.

          More information: report_layout_rules LayerExistenceNet
PinExistenceMacro: #fails=5 #checks=301 handling=warning (pin not exist in macro).
//  Warning: PinExistenceMacro-1: The specified NET net_unknown_layer1 in DEF file.def is
//  connected to MACRO ao32 which uses a PIN B11 which is not defined in the LEF files.

          More information: report_layout_rules PinExistenceMacro
```

Next, the tool reports on physical issues that could be causing the net-tracing problems. These potential problems are divided into five issue classes:

- No driving chip/port pin found: The net is not connected to any macro port that has direction OUTPUT or INOUT, or to any chip pin with direction INPUT or INOUT. Therefore, no driver for the net can be found and topology is aborted.

- No net segments available: The net has no physical polygons defined in the LEF/DEF files.
- Minimum of two connections are required: There must be at least two connections to chip pins or macro ports.
- Macro pin does not exist: The net is connected to at least one macro port that is undefined in the LEF files. This issue is also reported as a PinExistenceNetMacro layout rule violation. Resolving all violations of that rule automatically fixes this issue.
- Physical path-tracing errors: The net is physically broken and cannot be traced from driver to sink cells.

If there is more than one net with the same issue, the tool reports the five nets with the lowest number of polygons so that it is easier to debug the failing net. Generally, examining the listed nets can help you discover systemic issues related to the DEF file. To change the number of displayed nets per issue, specify the create_layout -topology_error_count option.

The following example shows how the tool reports the five issue classes in the event of physical net-tracing errors. The example displays one net for each issue class.

```
//  --------------------------------------------------------------------------------
//  Topology problem route causes:
//  --------------------------------------------------------------------------------
//  Issue-1: Physical path tracing error. Number nets having this issue: 2, e.g. failing
net=cpu_i/nx1265
// =======================================================================
//  The signal net=cpu_i/nx1265 is connected to 4 components:
//      (1) component=cpu_i/ix1264 portName=Y
//           The macro=inv02 has 5 ports
//           This output port=Y is a possible driver
//      (2) component=cpu_i/ix1266 portName=A
//           The macro=inv02 has 5 ports
//           This is an input port=A
//      (3) component=cpu_i/ix1268 portName=A
//           The macro=inv02 has 5 ports
//           This is an input port=A
//      (4) component=cpu_i/ix1270 portName=A
//           The macro=inv02 has 5 ports
//           This is an input port=A
//    Number connections=4, number of chip pins=0, number of segments=12, number placed
vias=16
//  GDS2, layerprops and marker files for net=cpu_i/nx1265 (netID=10) has been created.
//      GDS2       filename: …/inmemory_ladb/10.gds
//      layerprops filename: …/inmemory_ladb/10.gds.layerprops
//      Marker     filename: …/inmemory_ladb/10.marker
//  Analyze failing net with: calibredrv -m …/inmemory_ladb/10.gds -l …/inmemory_ladb/
10.gds.layerprops -rve …/nmemory_ladb/10.marker

// Issue-2: No driving chip/port pin found. Number nets having this issue: 1, e.g. failing
net=cpu_i/nx1229
// ================================================================================
//  The signal net=cpu_i/nx1229 is connected to 3 components:
//     (1) component=cpu_i/ix1230 portName=A
//          The macro=inv02 has 5 ports
//          This is an input port=A
//     (2) component=cpu_i/ix1232 portName=A
//          The macro=inv02 has 5 ports
//          This is an input port=A
//     (3) component=cpu_i/ix1234 portName=A
//          The macro=inv02 has 5 ports
//          This is an input port=A
//   Number connections=3, number of chip pins=0, number of segments=33, number placed
vias=26

//  Issue-3: No net segments available. Number nets having this issue: 1, e.g. failing
net=no_segments
// =======================================================================
//  The signal net=no_segments is connected to 1 component:
//     (1) component=cpu_i/uALU/ix318 portName=xxxx
//          The macro=ao21 has 7 ports
//          The macro port=xxxx does not exist
//  Number connections=1, number of chip pins=0, number of segments=0, number placed vias=0
//  Issue-4: Minimum of two connections are required. Number nets having this issue: 3,  e.g.
failing net=net_one_inst
// ========================================================================
//  The signal net=net_one_inst  is connected to 1 component:
//     (1) component=cpu_i/uALU/ix349 portName=B13
//          The macro=ao32  has 11 ports
//  Number connections=1, Number of chip pins=0, number of segments=1, number placed vias=0

//  Issue-5: Macro pin does not exist. Number nets having this issue: 2, e.g. failing
net=net_unknown_layer1
// ========================================================================
//  The signal net=net_unknown_layer1 is connected to 2 components:
//     (1) component=cpu_i/uALU/ix349 portName=B11
//          The macro=ao32 has 9 ports
//          The macro port=B11 does not exist
//    (2) component=cpu_i/uALU/ix345 portName=Y1
//          The macro=ao21 has 7 ports
//          The macro port=Y1 does not exist
//  Number connections=2, number of chip pins=0, number of segments=1, number placed vias=0
```

Debugging Net Topology Extraction Failures

When debugging net topology extraction failures, you begin by fixing the layout rule violations listed in the first section of the net topology extraction results transcript to yield a high topology success rate. Once the layout rule violations are resolved, there could still be some nets that fail topology because the tool does not perform physical net tracing during layout rule checking.

Prerequisites

- You have run create_layout and successfully resolved mismatch errors.
- This procedure assumes you have experience with the Calibre DESIGNrev (Calibre DRV) layout viewer. For more information, refer to the Calibre DESIGNrev documentation.

Procedure

1. From the first section of the net topology extraction transcript, resolve the layout rule violations.
2. From the second section of the net topology extraction transcript, determine the issue classes that cause the most number of nets to fail topology extraction. This number displays as a “Number of nets having this issue: <count>” statement for each issue class. These issues should get priority over others to resolve. The instructions below can help you debug and resolve the various issue classes:

   a. No driving chip/port pin found: Fix the LEF/DEF files to ensure that each net is connected to at least one MACRO port that has direction OUTPUT/INOUT, or at least one chip PIN that has direction INPUT or INOUT. The transcript one example of a net that does not have any such port/pin connected to it, and there the tool cannot determine the driver.

b. No net segments available: Fix the LEF/DEF files so that all nets have physical polygons associated with them.

c. Minimum of two connections are required: Tessent Diagnosis cannot calculate topology for dangling nets that are connected to only one MACRO port or chip pin. Check the LEF/DEF files to discover why dangling nets are present.

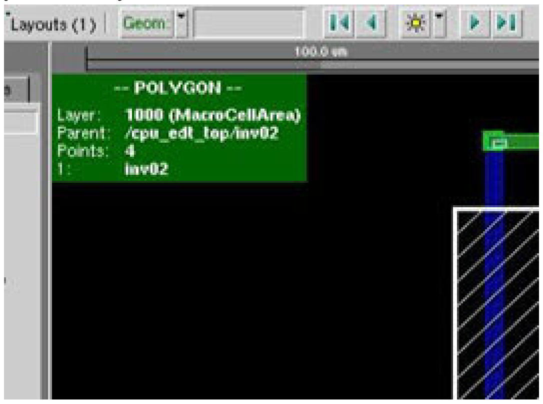
d. Macro pin does not exist: Same as the PinExistenceNetMacro layout rule violation.

e. Physical path-tracing errors: Refer to the rest of this procedure for details.

3. In Calibre DRV, as a one-time setup process, activate the following settings:

   a. **Preserve properties**: In the **Choose Layout Files** dialog box, click **Options**, and then click **Preserve properties.**

   b. **Show object info popup window**: On the **Options** menu, click **Objects**, and then under **Selection**, click **Show object info popup window**. The object info popup window displays when you select GDS elements.



-- POLYGON --
Layer: 1000 (MacroCellArea)
Parent: /cpu_edt_top/inv02
Points: 4
1: inv02

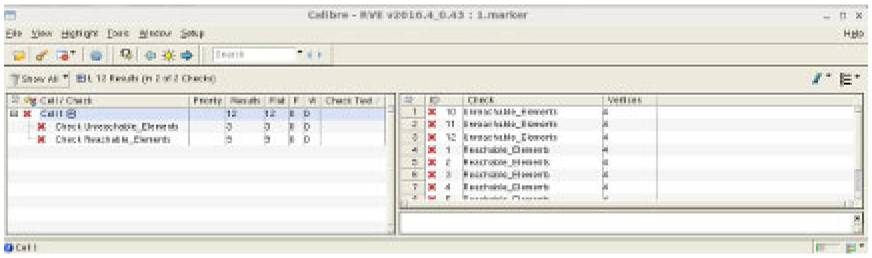
4. Resolve physical path-tracing errors. The tool generates GDS2 and marker files for the failed nets, which enables you to view the nets in Calibre DRV. For example:

```
//  Analyze failing net with: calibredrv -m …/inmemory_ladb/10.gds -
l …/inmemory_ladb/10.gds.layerprops -rve …/nmemory_ladb/10.marker
```

   When you run the calibredrv command, the tool opens the GDS2 file and the RVE marker file with Unreachable_Elements and Reachable_Elements on the net displayed.

   - Unreachable_Elements: These are the polygons that are physically connected to the driver of the net.

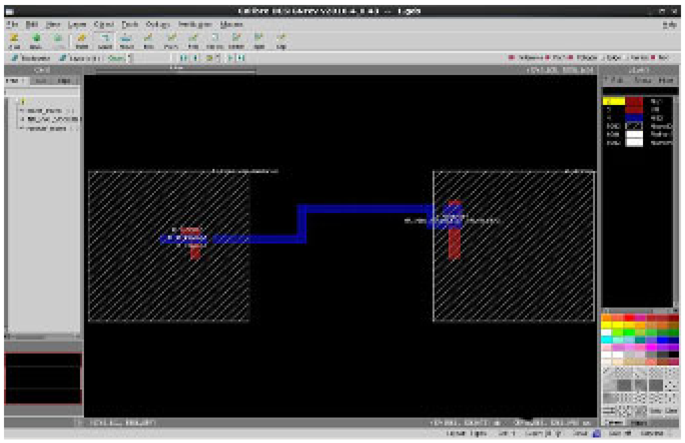
- Reachable_Elements: These are the polygons that cannot be physically disconnected from the driver.


Calibre - RVE v2016.4_0.43 : 1.marker
File View Highlight Tools Window Setup Help
Show All
12 Results (in 2 of 2 Checks)

Cell / Check	Priority	Results	Flat	F	W	Check Text
Cell t		12	12	8	0	
Check Unreachable_Elements		3	3	8	0	
Check Reachable_Elements		9	9	8	0	

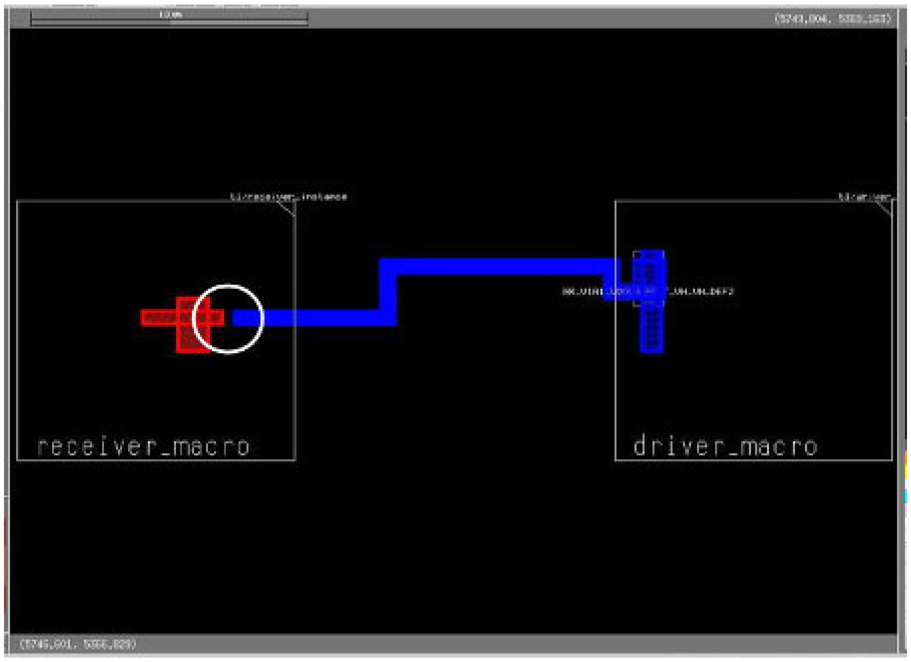
	ID	Check	Vertices
1	10	Unreachable_Elements	4
2	11	Unreachable_Elements	4
3	12	Unreachable_Elements	4
4	1	Reachable_Elements	4
5	2	Reachable_Elements	4
6	3	Reachable_Elements	4
7	4	Reachable_Elements	4

CalibreDRV displays the failing net with its connected pins, macros, and vias viewed as polygons, and the cell boundaries. If the number of display levels is 0, increase the number of display levels to 1 so that you see the contents of the macros and vias.



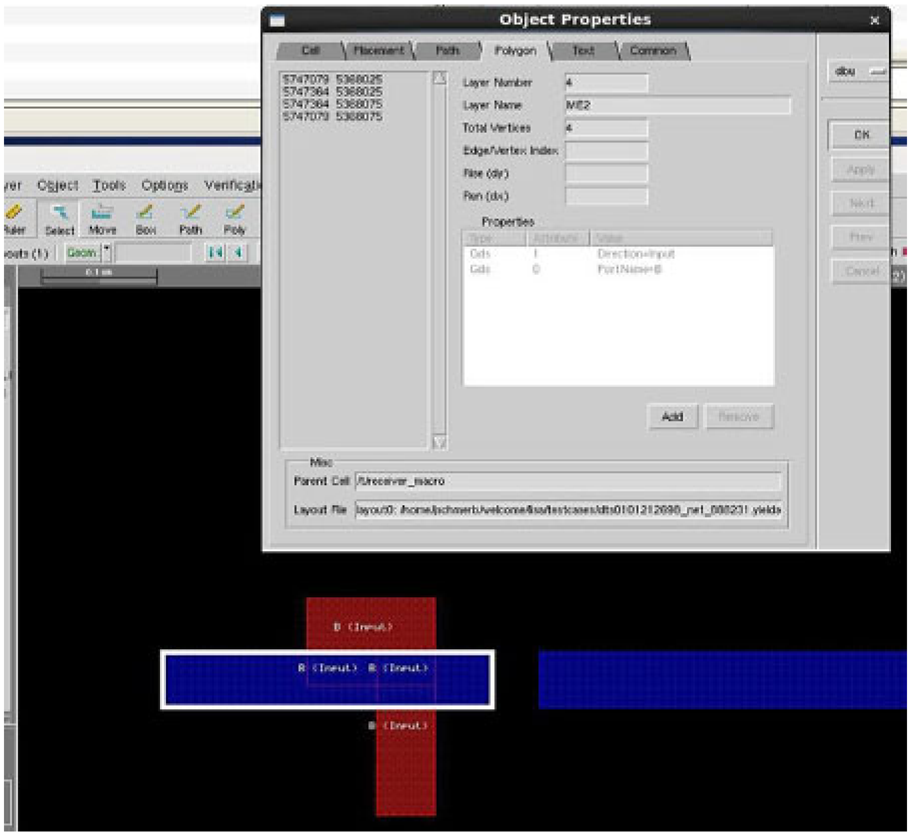
5. In Calibre RVE (the layout marker window), right-click on the top-level cell name (Cell t), and then click **Highlight**.

Calibre DRV highlights the reachable and unreachable net segments in two different colors, with blue reachable and red unreachable.You can see that there is a gap.



6. Turn off highlighting, right-click on the unreachable segment of the net, and click **Properties**.

In this example, the unreachable net segment is the input of port B and you can see the gap, on the other side of which starts the reachable segment of the net.



```
5747079  5368025
5747364  5368025
5747364  5368075
5747079  5368075
```

Layer Number: 4
Layer Name: ME2
Total Vertices: 4
Edge/Vertex Index:
Rise (dy):
Run (dx):

Properties

Type	Attribute	Value
Gds	1	Direction=Input
Gds	0	PortName=B

Misc
Parent Cell: /t/receiver_macro
Layout File: layout0: /home/jschmerb/welcome/lsa/testcases/dts0101212090_net_888231.yields

Likewise, you can right-click on the reachable segment of the net to view its properties. To tool outlines the reachable segment of the net starting from the driver up to the gap. The net name is t1/n_888231.

In this example, you learn that the unreachable segment is a port segment (LEF) and the reachable segment is a net segment (DEF). The gap could be caused by:

- Wrong DEF net coordinates.
- Wrong port polygons.
- Wrong macro placement.
- Layer width is too small if all pins, macros/pins, and vias exist.

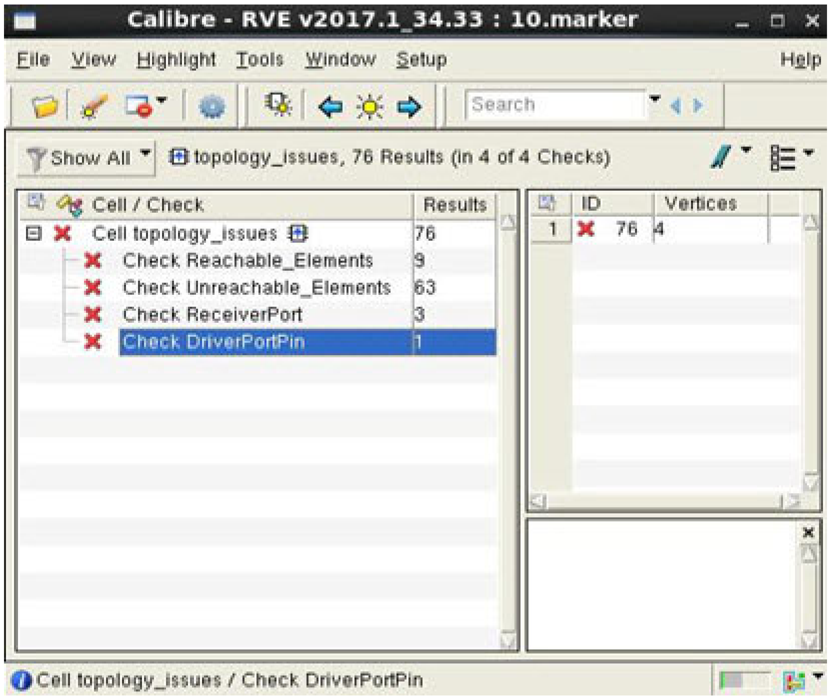
Results

After debugging net topology extraction failures, your net topology calculation should be greater than 90%. You can proceed to "Layout-Aware Diagnosis."

Examples

Besides the Unreachable_Elements and Reachable_Elements, CalibreDRV may also display the following elements on a net:

- DriverPortPin: The port/pin on the driver.
- ReceiverPort: The port on the receiver.
- ReceiverPin: The pin on the receiver.

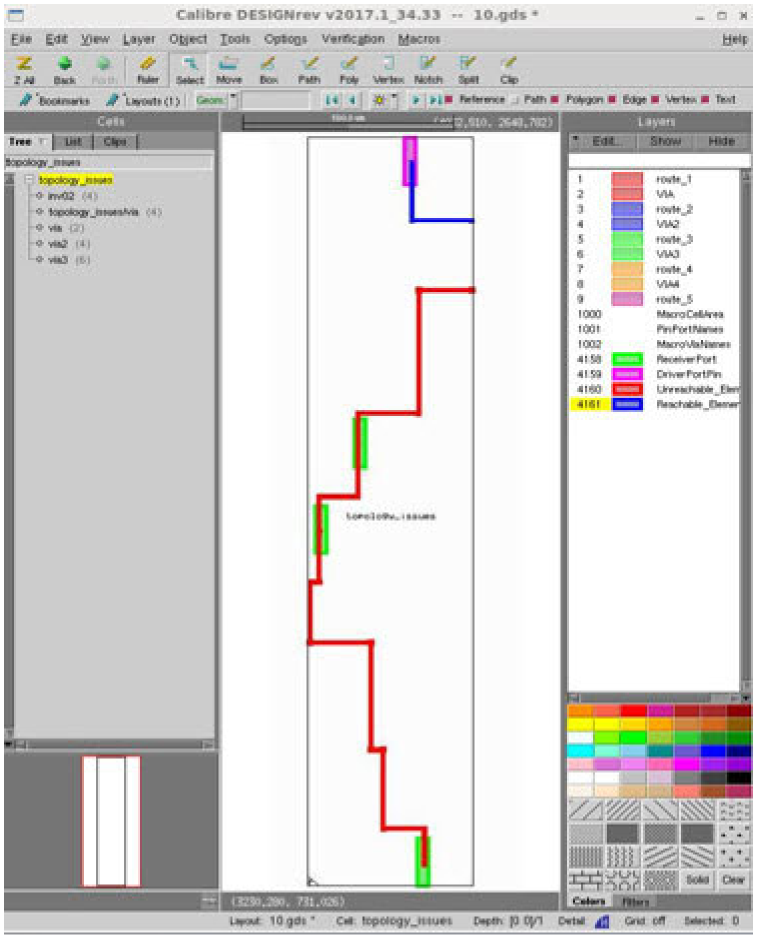

Calibre - RVE v2017.1_34.33 : 10.marker
File View Highlight Tools Window Setup Help
Search
Show All
topology_issues, 76 Results (in 4 of 4 Checks)

Cell / Check	Results
Cell topology_issues	76
Check Reachable_Elements	9
Check Unreachable_Elements	63
Check ReceiverPort	3
Check DriverPortPin	1

	ID	Vertices
1	76	4

Cell topology_issues / Check DriverPortPin

You can highlight these net elements the same you would highlight reachable and unreachable elements. The following figure shows the highlighted polygons that represent the receiver ports (green) and the driver port/pin (pink) in addition to the reachable and unreachable elements (blue and red, respectively).


Calibre DESIGNrev v2017.1_34.33 -- 10.gds *
File Edit View Layer Object Tools Options Verification Macros Help
Z All Back Forth Ruler Select Move Box Path Poly Vertex Notch Split Clip
Bookmarks Layouts (1) Geom Reference Path Polygon Edge Vertex Text
Cells
Tree List Clips
topology_issues
topology_issues
inv02 (4)
topology_issuesVia (4)
via (2)
via2 (4)
via3 (6)
topology_issues
(3230.280, 731.026)
Layers
Edit... Show Hide
Colors Fills
Solid Clear
Layout: 10.gds * Cell: topology_issues Depth: [0 0]/1 Detail Grid: off Selected: 0

1	🟥	route_1
2	🟥	VIA
3	🟦	route_2
4	🟦	VIA2
5	🟩	route_3
6	🟩	VIA3
7	🟧	route_4
8	🟧	VIA4
9	🟪	route_5
1000		MacroCellArea
1001		PinPortNames
1002		MacroViaNames
4158	🟩	ReceiverPort
4159	🟪	DriverPortPin
4160	🟥	Unreachable_Elem
4161	🟦	Reachable_Elemen

Layout-Aware Diagnosis

With layout-aware diagnosis, you can perform diagnosis for root cause deconvolution (RCD) analysis and design for manufacturability (DFM) analysis.

To use the Tessent Diagnosis Server for layout-aware diagnosis, refer to "Layout-Aware Diagnosis with the Tessent Diagnosis Server."

Performing Layout-Aware Diagnosis with Tessent Diagnosis

When performing layout-aware diagnosis, use the create_layout command to generate the LDB.

Prerequisites

- Flat model of your design
- LEF/DEF files
- Test patterns
- ATE failure log files

Procedure

1. Start scan diagnosis in the Tessent Shell environment. From a Linux/UNIX shell, enter:

   Tessent_Tree_Path/bin/tessent -shell -logfile logfile_name

   where *Tessent_Tree_Path* is the path to where the Tessent Shell application tree is installed.

   For example:

   ```
   % tessent -shell-logfile my_log_file
   ```

2. Set the context. After Tessent Shell has started, enter:

   set_context patterns -scan_diagnosis

3. Specify the flat netlist. Use the read_flat_model command to specify the flat netlist.

   read_flat_model flat_model

   where *flat_model* is the pathname of the flat design netlist.

4. Perform verification and create the LDB. For example:

   create_layout ./src/design.dft.ldb -lef ./src/design.lef -def ./src/design.def

   Where:

   - *./src/design.dft.ldb* — The name of the LDB to create.
   - -lef *./src/design.lef* — A required switch, and location and name of the source LEF file.
   - -def *./src/design.def* — A required switch, and location and name of the source DEF file.

   This step first performs layout verification that checks the consistency of LEF/DEF files and matches them against the design, and then it proceeds to creating the LDB if the percentage match is greater than the abort threshold percentage that you specified with the create_layout command, or greater than the default percentage of 85%.

   During verification, the tool tolerates the differences in escapes in instance pathnames and net pathnames. To turn off tolerant name matching, specify create_layout with the "-do_tolerant_match off" switch.

   For verification troubleshooting and information, see "Layout Verification Reporting."

5. Open the LDB. Enter the open_layout command and specify the location and name of the LDB you created.

   open_layout ./src/design.dft.ldb

   Where:

   - *design.dft.ldb* — The pathname and name of the layout-aware LDB to open.

6. Load the test pattern file. Enter the read_patterns command and specify the pathname of the test patterns:

   read_patterns./src/patterns.ascii

   Where:

   - *./src/patterns.ascii* — The pathname of the test pattern file to load.

7. Run layout-aware diagnosis on the failure file. Enter the diagnose_failures command and specify the pathname of the failure file similar to:

   diagnose_failures ./tester_files/file1.flog

   Where:

   - *./tester_files/file1.flog* — The pathname of the properly formatted failure file to diagnose.

8. Write results to the disk. Enter the write_diagnosis command and specify the type of report or reports you want from Tessent Diagnosis:

   ```
   write_diagnosis -format text layout -file ./results/file1 -replace
   ```

   Where:

   - -format *format_choice* — A switch specifying the format of the report from the following formats:
     - TEXT— Specifies ASCII text format. This is the default.
     - CSV — Specifies comma-separated value (CSV) format.
     - LAYout_marker — Specifies layout coordinate format.
   - -file *base_file_name* — A switch specifying the name of the report. The *base_file_name* value is used for the report filename, regardless of format, and identifies the format using the following filename suffixes:
     - *name*.csv — CSV format diagnosis report
     - *name*.diag — ASCII text format diagnosis report
     - *name*.lay — Layout markers
     - *name*.enc.csv — Encoded CSV diagnosis report
     - *name*.enc.diag — Encoded ASCII text diagnosis report
     - *name*.enc.lay — Encoded layout markers

     Note
     For backwards compatibility with layout viewers, use the write_diagnosis command's -SHORT switch.

9. Optionally, close the LDB. Enter the close_layout command and close the LDB.
10. View the layout-aware diagnosis results. See "Layout-Aware Diagnosis Reporting" for complete information.

Results

Tessent Diagnosis generates the layout-aware diagnosis results in the report format you specified.

Examples

The following dofile example creates a LDB and runs diagnosis. The current directory is *layout_aware_demo*.

```
// *********************************************************************
//
// LDB creation is necessary only on the first invocation.
// All later invocations can use the generated file directly.
//
// *********************************************************************

create_layout ./src/design.dft.ldb \
-lef ./src/design.lef \
-def ./src/design.def

// *********************************************************************
//
// Open the LDB
//
// *********************************************************************

open_layout ./src/design.dft.ldb

// *********************************************************************
//
// Start the diagnosis session
// You first need to load the pattern set
//
// *********************************************************************

read_patterns ./data/patterns.ascii

// *********************************************************************
//
// Optionally perform a non-layout-aware diagnosis first
//
// *********************************************************************

diagnose_failures ./tester_files/file1.flog
write_diagnosis -file ./results/file1.nolayout -replace
write_diagnosis -encoded -file ./results/file1.nolayout -replace

diagnose_failures ./tester_files/file2.flog
write_diagnosis -format text -file ./results/file2.nolayout
write_diagnosis -xmap v2lvs -format text -file ./results/file2x.nolayout

// *********************************************************************
//
// Now load the LDB and repeat the diagnosis runs
// Save the results in new files for later comparison.
//
// Notice that the options '-format layout' and '-short' of the command
// 'write_diagnosis' became available
//
// *********************************************************************

open_layout ./src/design.dft.ldb

diagnose_failures ./tester_files/file1.flog
write_diagnosis -format text layout -file ./results/file1.layout
write_diagnosis -encoded -format text layout -file ./results/file1.layout
```

```
diagnose_failures ./tester_files/file2.flog
write_diagnosis -format text layout -file ./results/file2.layout
write_diagnosis -short -format text -file ./results/file2s.layout
write_diagnosis -short -xmap v2lvs -format text -file \
./results/file2x.layout

// **********************************************************************
//
// exit Tessent Diagnosis
//
// **********************************************************************

exit
```

Related Topics

Layout Database Compression and Decompression

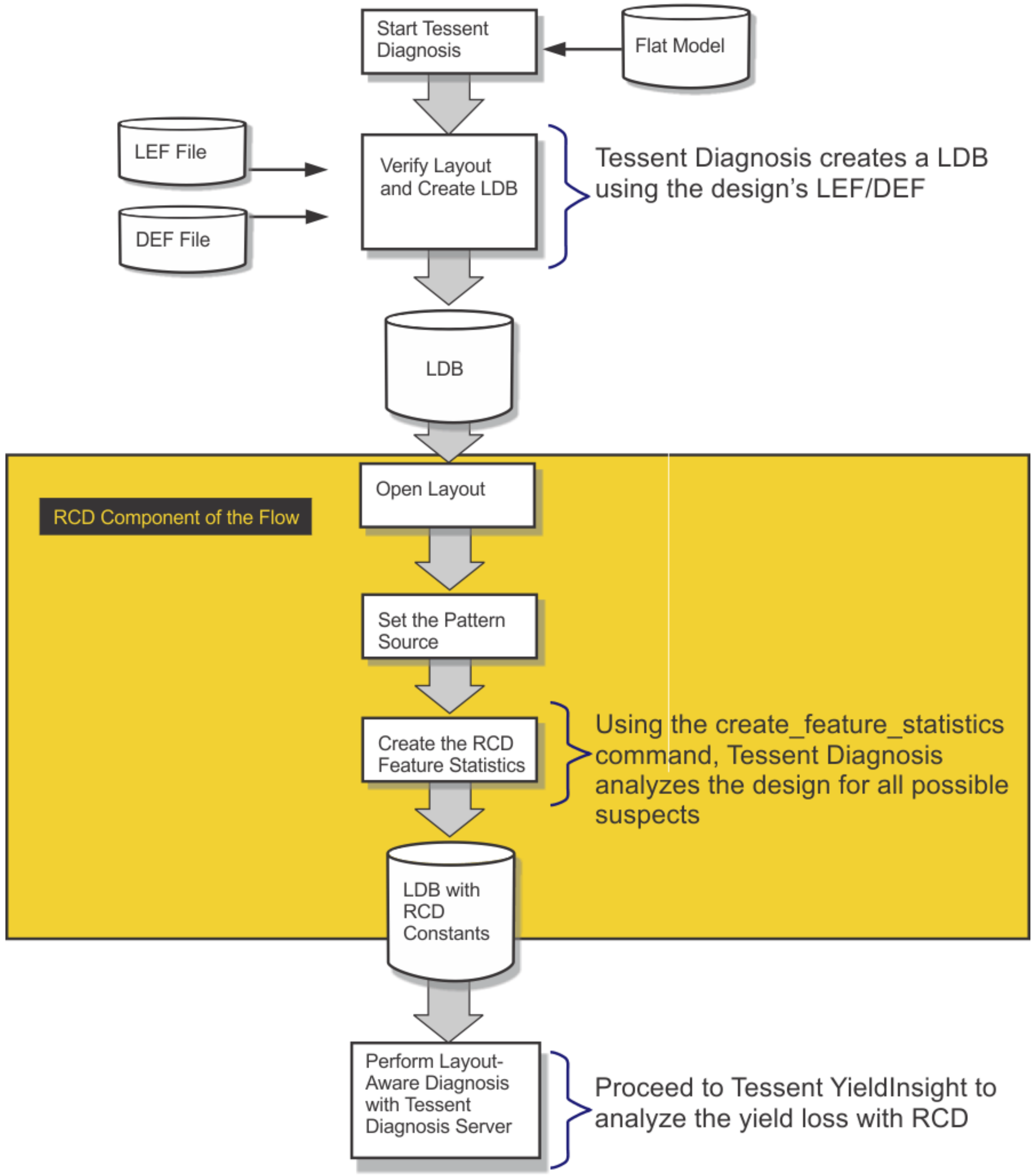
Diagnosis Output Files

Diagnosis for Root Cause Deconvolution Analysis

Root cause deconvolution (RCD) is the process of differentiating possible root causes in a population of failing devices from diagnosis ambiguity (or noise) in a high-volume scan diagnosis production environment. Tessent YieldInsight provides a signature that identifies the root causes—for example, M3 open and M5 bridge—of a population of die, which can help you accurately understand the true root causes leading to device failure, and, through subsequent drill-down analysis in Tessent YieldInsight, to significantly improve device selection for physical failure analysis (PFA).

For information about RCD analysis in Tessent YieldInsight, see Root Cause Deconvolution Analysis.”

Figure 3-7. RCD Diagnosis Flow


Start Tessent Diagnosis
Flat Model
LEF File
DEF File
Verify Layout and Create LDB
Tessent Diagnosis creates a LDB using the design's LEF/DEF
LDB
RCD Component of the Flow
Open Layout
Set the Pattern Source
Create the RCD Feature Statistics
Using the create_feature_statistics command, Tessent Diagnosis analyzes the design for all possible suspects
LDB with RCD Constants
Perform Layout-Aware Diagnosis with Tessent Diagnosis Server
Proceed to Tessent YieldInsight to analyze the yield loss with RCD

Preparing for RCD Analysis in Tessent YieldInsight

To perform RCD analysis in Tessent YieldInsight, you must have previously calculated the RCD feature statistics as described in this section.

See "Layout-Aware Diagnosis Flow" for an overview of the layout-aware diagnosis flow, steps for creating a LDB, and other requirements.

Note

You must create the LDB populated with RCD constants by using the Tessent Diagnosis point tool.

Prerequisites

- Flat model of your design
- LEF/DEF design files
- Test patterns
- ATE failure log files

Procedure

1. Start scan diagnosis in the Tessent Shell environment. From a Linux/UNIX shell, enter:

   ```
   Tessent_Tree_Path/bin/tessent -shell -logfile logfile_name
   ```

2. Set the context. After Tessent Shell has started, enter:

   ```
   set_context patterns -scan_diagnosis
   ```

3. Specify the flat netlist as follows:

   ```
   read_flat_model flat_model
   ```

   where *flat_model* is the pathname of the flat design netlist.

4. Perform verification and create the LDB. For example:

   ```
   create_layout ./src/design.dft.ldb -lef ./src/design.lef -def ./src/design.def
   ```

   For verification troubleshooting and information, see "Layout Verification Reporting."

5. Open the LDB. For example:

   ```
   open_layout ./src/design.dft.ldb
   ```

6. Load the test pattern source file. For example:

   ```
   read_patterns ./src/patterns.ascii
   ```

   You can add RCD constants to existing diagnosis reports. See the Examples section below for more information.

7. Perform the RCD feature statistics calculation by executing the create_feature_statistics command as follows:

   create_feature_statistics

   Tessent Diagnosis determines the bridge pairs, net segments, and cells that are tested, and it extracts the relevant physical information such as the critical area for each possible suspect. The RCD constants statistics are stored in dedicated .rcddb files in the LDB directory. The tool creates one .rcddb file per flat model. If an .rcddb file exists in the LDB, it indicates that the feature statistics calculation has been performed.

   You must create the RCD constants statistics for every flat model and pattern combination. To do this, you can process the combinations in parallel in different tool sessions.

   If the RCD constants have been created and the .rcddb file exists in the LDB, Tessent Diagnosis populates the diagnosis reports with RCD constants in the RCD_CONSTANTS section during diagnosis. To inhibit the population of constants in the diagnosis report, specify the following command:

   set_diagnosis_options -include_rcd_constants off

   If you do not want to populate the LDB with the RCD constants, set the include_rcd_constants variable to false.

8. In Tessent Diagnosis server, add monitors, the design and layout, patterns, and analyzers in accordance with a typical Tessent Diagnosis server run. Refer to "Setting Up the Tessent Diagnosis Server" for details.

   Note
   You can also use the Tessent Diagnosis point tool to run the diagnosis.

9. In Tessent Diagnosis server, perform the diagnosis as described in "Running the Diagnosis."

Results

The RCD constants data appear as follows in the XMAP_TABLE within the diagnosis report:

```
RCD_CONSTANTS_BEGIN (rcd_version)
rcd_feature rcd_constant
rcd_feature rcd_constant
rcd_feature rcd_constant
rcd_feature rcd_constant
...
RCD_CONSTANTS_END
```

If you have run create_feature_statistics multiple times on different combinations of flat models and pattern sets, you can use the report_layout_files command to view the flat models and pattern sets associated with the RCD constants stored in the LDB.

In Tessent YieldInsight, you can use RCD constants data to identify the root cause distribution in a population of devices. The *rcd_feature* is the same root cause designation that shows up in the RCD - Sum of Probability signature in Tessent YieldInsight.

The following example shows the RCD_CONSTANTS output.

```
RCD_CONSTANTS_BEGIN ( 3.4.1.1.2 )
{Area CELL}                    6787.98
{Count CELL and02}             0.0280805
{Count CELL and03}             0.00502934
{Count CELL and04}             0.00775356
{Count CELL oai33}             0.000838223
{Count CELL oai332}            0.000209556
{Count CELL or02}              0.00880134
{Count CELL sffr}              0.000628667
{Count CELL sffs}              0.000628667
{Count CELL xnor2}             0.0148785
{Count CELL xor2}              0.0108969
{Count ViaMacro via}           0.641782
{Count ViaMacro via3}          0.37276
{Count ViaMacro via4}          0.3977
{CritArea OPEN VIA}            6.15225
{CritArea OPEN VIA2}           5.38636
{CritArea OPEN VIA4}           2.25199
{CritArea OPEN route_1}        150.435
{CritArea OPEN route_3}        216.29
{CritArea OPEN route_4}        213.319
{CritArea OPEN route_5}        60.964
{CritArea SHORT route_1}       15.1244
{CritArea SHORT route_2}       30.9221
{CritArea SHORT route_3}       23.5965
{CritArea SHORT route_5}       6.22894
{CritArea SHORT_GROUND route_1} 4.01231
{CritArea SHORT_GROUND route_2} 0.00180776
{CritArea SHORT_POWER route_1} 5.01358
{CritArea SHORT_POWER route_2} 0.00350706
...
RCD_CONSTANTS_END
```

Examples

Adding RCD Constants to Existing Diagnosis Reports

You can add RCD constants to existing diagnosis reports using the annotate_diagnosis command. To annotate a diagnosis report, you must load the same pattern set and flat model that you specified with the read_patterns command prior to running diagnosis. The patterns must be the same as used for diagnosis and creating the RCD constants.

```
# Specify the same pattern set that was used to run diagnosis
# and create RCD constants
read_patterns /src/patterns.ascii

# Open the LDB
open layout design.dft.ldb
annotate_diagnosis small_diags_rcd_gz
```

Note

You can set the set_diagnosis_options -missing_rcd_action to warning or error if you want to be forewarned that an existing LDB does not contain RCD constants for the current flat model and pattern set.

Related Topics

Performing Layout-Aware Diagnosis with Tessent Diagnosis

Diagnosis for Design for Manufacturability Analysis

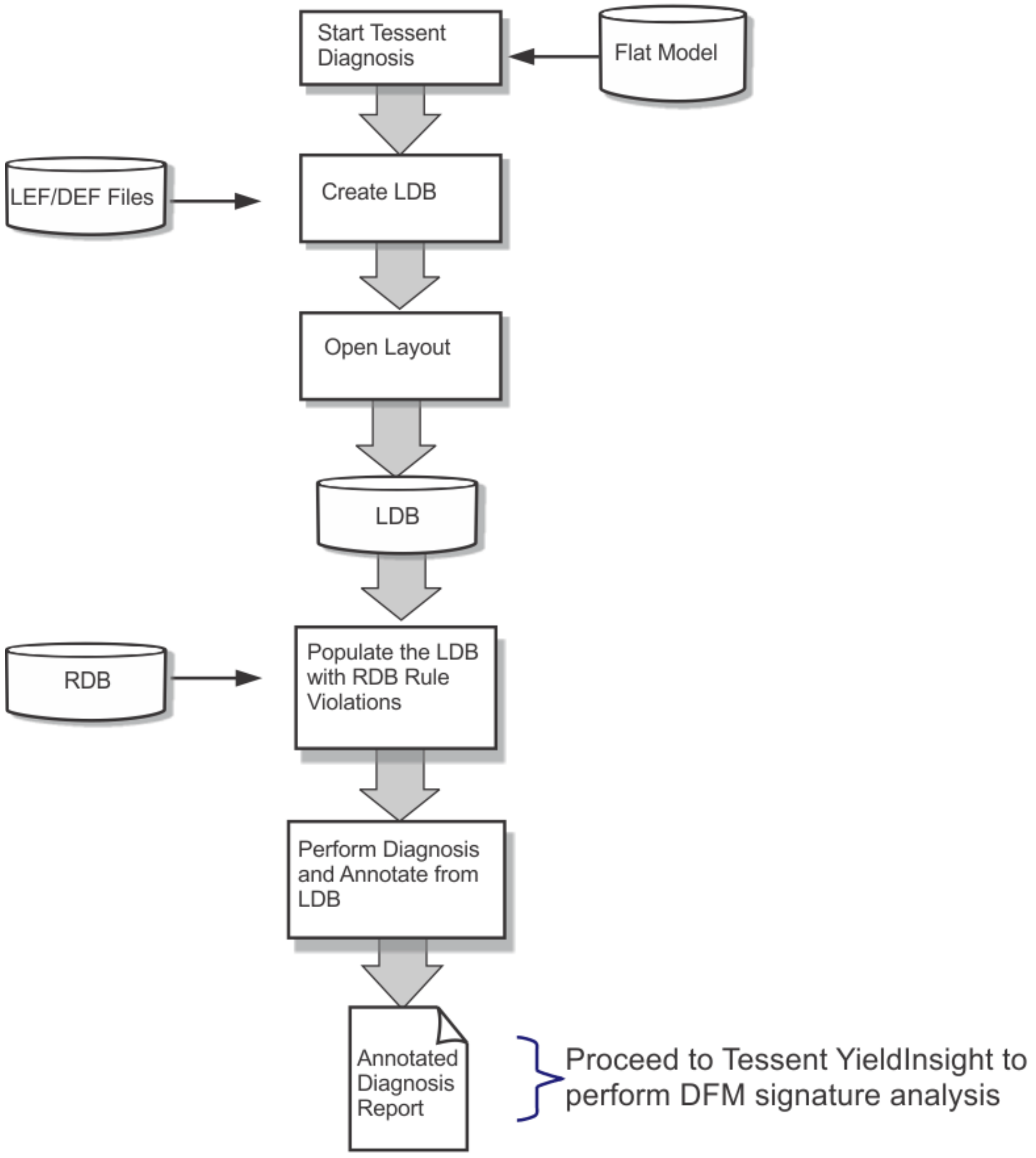
Design for Manufacturability (DFM) techniques optimize the physical layout of ICs to improve their yield. With a Calibre®-compatible DFM Results Database (RDB) as input, Tessent Diagnosis enables you to compare RDB rule violations against layout-aware diagnosis report defect bounding boxes. The reported failures—or DFM hits—indicate areas where the overlaps occur.

Note

For information on Calibre DFM, consult your Siemens EDA Calibre documentation.

Figure 3-8 shows the DFM diagnosis flow.

Figure 3-8. DFM Diagnosis Flow


Start Tessent Diagnosis
Flat Model
LEF/DEF Files
Create LDB
Open Layout
LDB
RDB
Populate the LDB with RDB Rule Violations
Perform Diagnosis and Annotate from LDB
Annotated Diagnosis Report
Proceed to Tessent YieldInsight to perform DFM signature analysis

For information about DFM signature analysis in Tessent YieldInsight, see ‘Design for Manufacturability Signature Analysis.”

During the DFM diagnosis flow, Tessent Diagnosis stores the RDB rule violations that it imports from the RDB. Layout-aware diagnosis then uses the violations to generate a diagnosis report that includes the DFM hits.

By default, Tessent Diagnosis performs DFM diagnosis. To inhibit DFM annotation, set the following command:

```
set_diagnosis_options
-include_dfm_rules off
```

Supported DFM RDB Violation Types

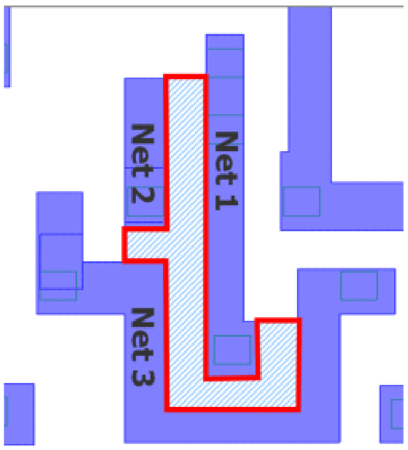
Tessent Diagnosis supports DFM violation types in three categories: interconnect, cell, and cell-aware.

Interconnect Violations

For interconnect violations, you must ensure the layer name matches the layer name in the LDB.

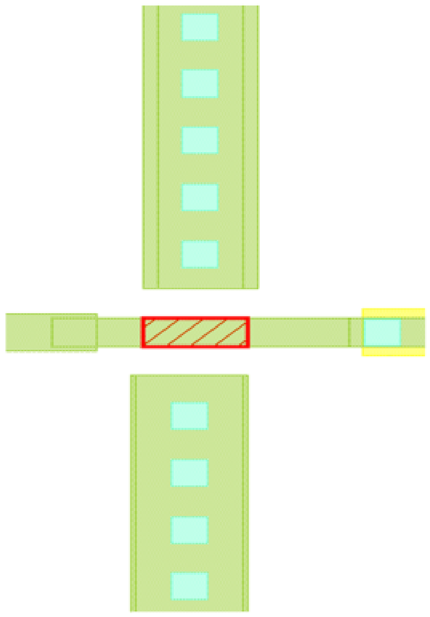
- Interconnect bridge: BRIDGE

Figure 3-9. Interconnect Bridge DFM Violation



- Interconnect open: OPEN

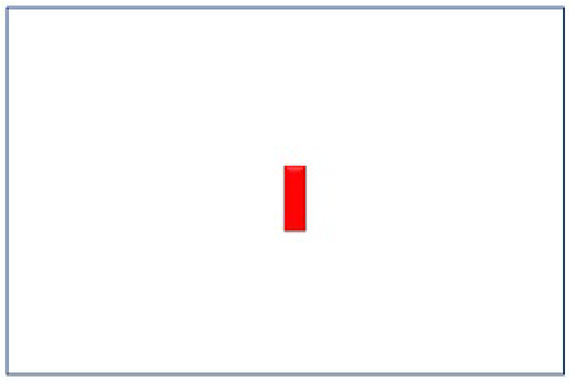
Figure 3-10. Interconnect Open DFM Violation



Cell Violations

By default, rule violations within cells are identified as cell violations and the bounding box for the cell is flagged. These types of violations are named “CELL.”

Figure 3-11. Cell DFM Violation



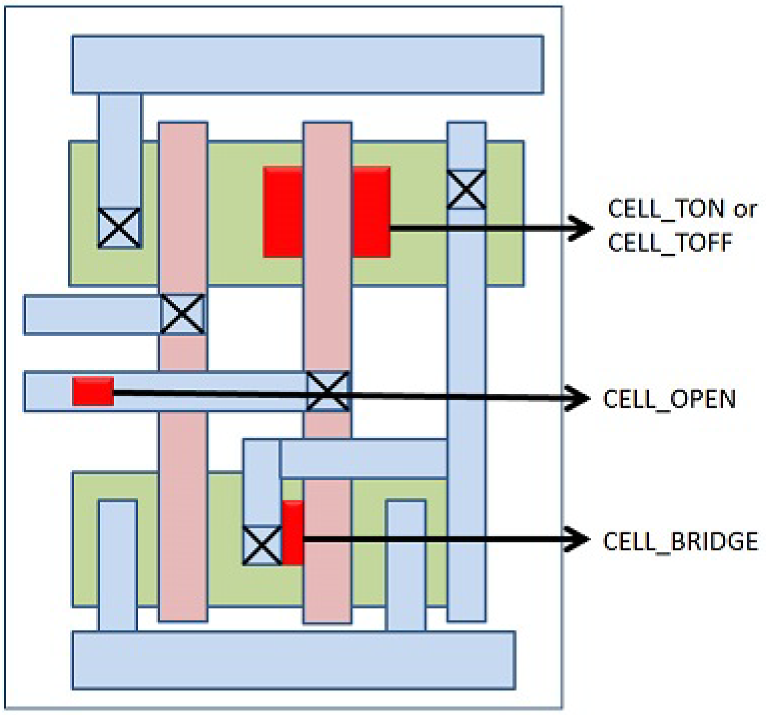
Cell-Aware Violations

When you have a UDFM file with Diagnosis Views as generated by Tessent CellModelGen, the tool is aware of the layout within the cell, thus it can flag open, bridge, and transistors defects particular to the cell. These are called CELL_OPEN, CELL_BRIDGE, CELL_TOFF, and CELL_ON defects.

CELL_BRIDGE violations can include bridge violations between layers, called interlayer bridge violations.

For cell-aware violations, you must ensure the layer name matches the layer name in the UDFM.

Figure 3-12. Cell-Aware DFM Violations

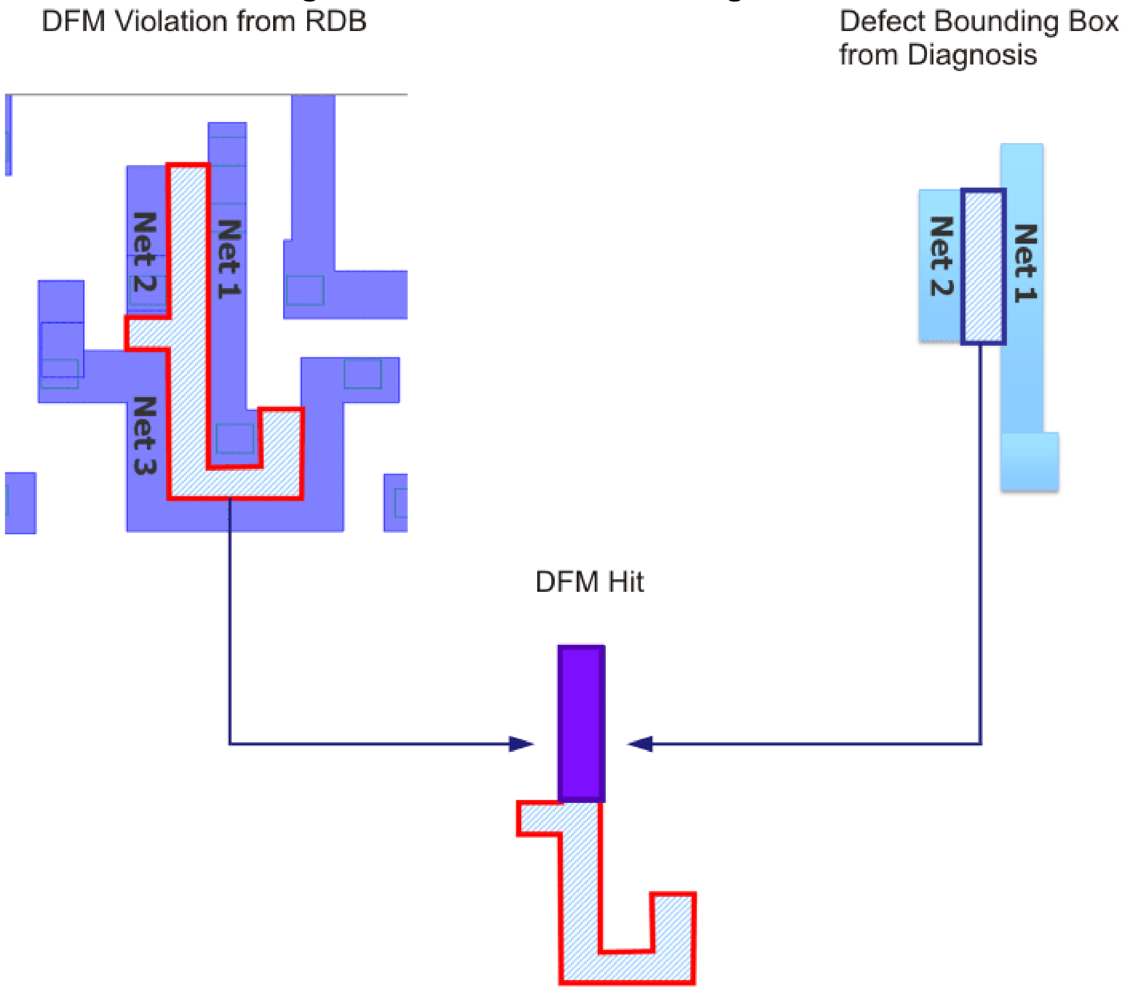


What Constitutes a DFM Hit

DFM hit reporting in Tessent Diagnosis identifies RDB rule violations that overlap diagnosis defect bounding boxes for bridges, opens, and cells. These overlaps are called DFM hits.

Figure 3-13 illustrates a DFM hit for an interconnect bridge DFM violation.

Figure 3-13. Interconnect Bridge DFM Hit



RDB-to-Layout Database Verification Results

During RDB importation, Tessent Diagnosis verifies the DFM violations in the RDB against the layout shapes in the LDB. The tool reports the percentage of DFM violations for each rule that satisfy certain verification conditions.

Note

Verification is intended for reporting purposes only. All violations are imported into the LDB whether they pass or fail verification.

Figure 3-14 illustrates cell DFM hits and their verification results. Cell DFM hits pass verification when a RDB cell violation is inside, partially inside, or the same size as a LDB cell shape. Cell DFM hits fail verification when a RDB cell violation and a LDB cell shape do not touch at all.

Figure 3-14. Cell DFM Hits and Their Verification Results

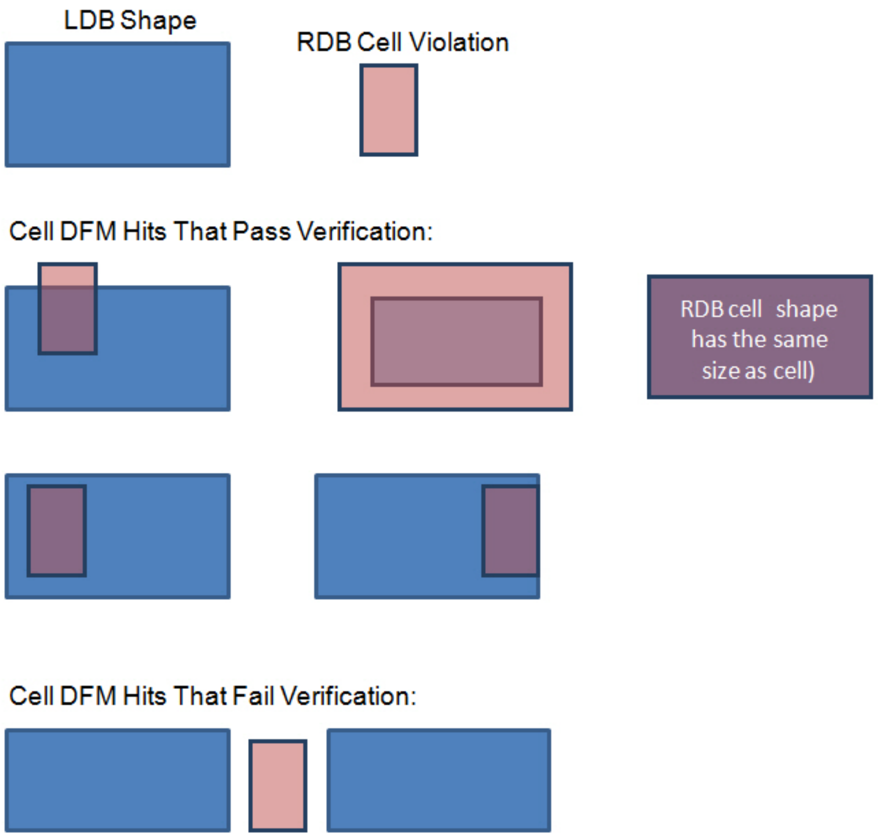

LDB Shape
RDB Cell Violation
Cell DFM Hits That Pass Verification:
RDB cell shape has the same size as cell)
Cell DFM Hits That Fail Verification:

Figure 3-15 illustrates bridge DFM hits and their verification results. All bridge DFM hits pass verification.

Figure 3-15. Bridge DFM Hits and Their Verification Results

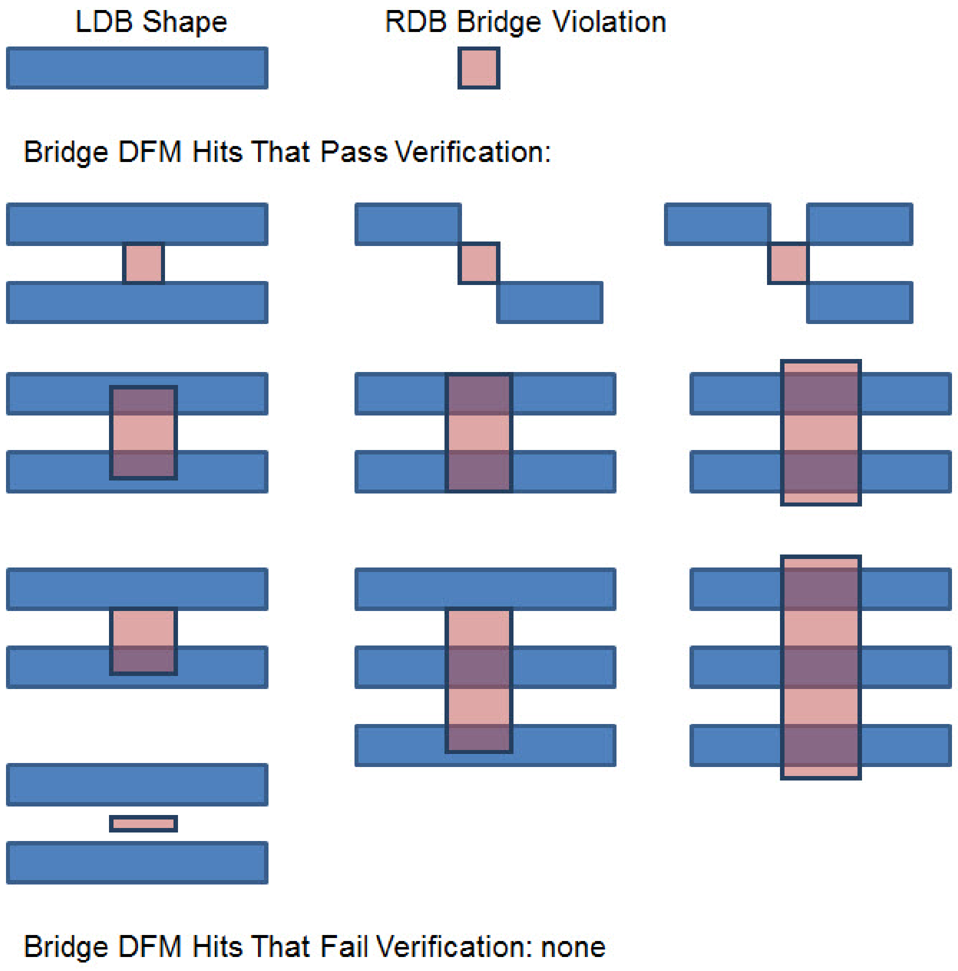
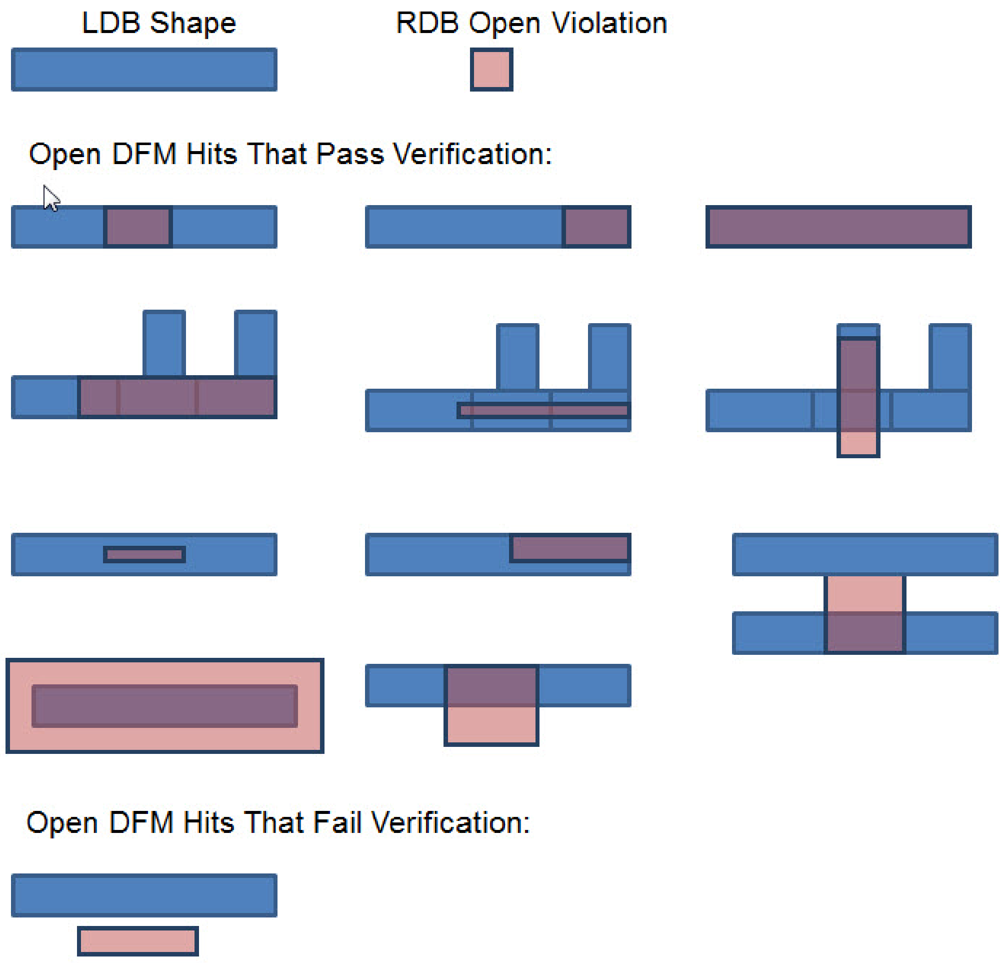

LDB Shape
RDB Bridge Violation
Bridge DFM Hits That Pass Verification:
Bridge DFM Hits That Fail Verification: none

Figure 3-16 illustrates open DFM hits and their verification results. Open DFM hits pass verification when a RDB open violation overlaps a LDB shape in any way. Open DFM hits fail verification when they do not touch the LDB shape at all.

Figure 3-16. Open DFM Hits and Their Verification Results


LDB Shape
RDB Open Violation
Open DFM Hits That Pass Verification:
Open DFM Hits That Fail Verification:

Tool Messaging During Verification

The following examples show the messages the tool produces during verification:

import_dfm -rdb RDB/route1.rdb -layer route_1 -type bridge

```
// The specified rule 'rule_cell_1' will be imported into the LDB
// The RDB 'RDB/route.rdb' was loaded successfully
```

import_dfm -rdb RDB/route1.rdb -layer route_1 -type bridge

```
// Error: The specified rule 'rule_cell_1' already exist in the LDB file
// and will be ignored
// The RDB 'RDB/route.rdb' was loaded successfully
```

The tool also checks the RDB design name against the LDB design name. If these names do not match, then the tool issues a warning.

Verification Results for Cell-Aware Violations

At the beginning of the diagnosis process, the tool verifies the UDFM file and then performs verification against the LDB's DFM rule layer names. The tool returns a warning when it finds DFM rule layers that do not match those in the UDFM file. For example:

```
diagnose_failure failure.log -output cell_diag.rpt -replace
//  Reading UDFM File: ../src/cells.udfm
//     UDFM delay definitions read: Total=2, Cells=0, Modules=2,
Instances=0, No Match=0, Static&Delay faults=623

//  Warning: DFM rules match some layers (1 out of 3) mentioned in UDFM
file.
//           Use 'report_dfm_rules' to see dfm rules and layers.
//           Cell internal layers are:
//              M0-M1
//              M1
//              PS
```

Note

You can override the default layer names used in the UDFM file by using the -layer_map or -original_layer_transfer options as described in the *Tessent CellModelGen Tool Reference.*

Performing DFM Diagnosis

When performing DFM diagnosis, you must import the Calibre RDB file into the LDB.

Refer to "Layout-Aware Diagnosis Flow" for an overview of the layout-aware diagnosis flow, steps for creating an LDB, and other requirements.

Note

You can also perform this procedure in Tessent Diagnosis server. See "Running Tessent Diagnosis Server" for details about performing layout-aware diagnosis with Tessent Diagnosis server.

Prerequisites

- Flat model of your design
- LEF/DEF design files
- Test patterns
- ATE failure log files
- Calibre-compatible RDB
- Optimally, for cell-aware diagnosis, you need a UDFM with Diagnosis View. Refer to "Cell-Aware Diagnosis" for details.

Procedure

1. Start scan diagnosis in the Tessent Shell environment. From a Linux/UNIX shell, enter:

   ```
   Tessent_Tree_Path/bin/tessent -shell -logfile logfile_name
   ```

2. Set the context. After Tessent Shell has started, enter:

   ```
   set_context patterns -scan_diagnosis
   ```

3. Specify the flat netlist as follows:

   ```
   read_flat_model flat_model
   ```

   where *flat_model* is the pathname of the flat design netlist.

4. Perform verification and create the LDB. For example:

   ```
   create_layout ./src/design.dft.ldb -lef ./src/design.lef -def ./src/design.def
   ```

   For verification troubleshooting and information, see "Layout Verification Reporting." If you specify the create_layout command with the "-compact on" switch, the tool does not perform DFM rule verification.

   To view the DFM rules contained in the LDB, use the get_dfm_rules or report_dfm_rules command.

5. Open the LDB. For example:

   ```
   open_layout ./src/design.dft.ldb
   ```

6. Load the test pattern source file. For example:

   ```
   read_patterns ./src/patterns.ascii
   ```

7. Import the Calibre RDB file into the LDB. For example:

   ```
   import_dfm -rdb src/VIA3_VIA4.rdb -layer VIA3 -type OPEN
   ```

   The import_dfm command verifies that some percentage of the DFM violations in the RDB match the layout geometries in the LDB. It then populates with LDB with the DFM rule violations.

   The following example shows how to specify the layer for interlayer CELL_BRIDGE violations.

   Note

   In the UDFM file, the interlayer CELL_BRIDGE name, for example "M0-M1", is equivalent to layer name "M1-M0". Tessent Diagnosis also considers these layer names equivalent for the purposes of DFM hit analysis.

   ```
   import_dfm -rdb inter_layer_bridge_M0_M1.rdb -layer M0-M1 \
       -type CELL_BRIDGE -replace
   ```

   In the import_dfm command, the -rdb, -layer, and -type switches are required.

When using the -layer switch, you must specify an existing layer in the layout-aware diagnosis layout file; otherwise the tool issues the following error message:

```
Could not match RDB's layer in the LayoutDB. The RDB seems to be
invalid for this design.
```

Tessent Diagnosis generates an error message if you use import_dfm and have not opened the LDB with the open_layout command.

8. Perform diagnosis.

   diagnose_failures

Note

Use the delete_dfm [-rule *rule_name*] command if you need to eliminate some or all DFM rules from the LDB.

Results

Tessent Diagnosis creates a diagnosis report that contains the results for DFM hit reporting. In Tessent YieldInsight, you import the diagnosis results into the ADB to perform DFM signature analysis.

During diagnosis, cell-aware suspects are compared to previously imported DFM violations. Any DFM violations that overlap with cell-aware suspects are reported in the DFM_RULE section of the diagnosis report.

The following example shows the DFM diagnostics results that appear in the diagnosis report.

Note

Due to space constraints, the following results example does not show the x_coord2, y_coord2, and dfm_properties columns.

```
DFM_RULE_BEGIN
  rule_id hits  violations layout_layer  category      rule_name
  1       1     1          M2             OPEN         rule_SUSPECT_OPEN_m2
  2       3     1          M3             OPEN         rule_SUSPECT_OPEM_m3
  3       2     2          M2-M3          OPEN         rule_SUSPECT_OPEN_m2_m3
  4       1     1          NA             CELL_TON     rule_SUSPECT_1.2_CELL_TON.D66
  5       1     1          NA             CELL_TON     rule_SUSPECT_1.5_CELL_TON.D80
  6       1     1          M1             CELL_BRIDGE  rule_SUSPECT_1.3_CELL_BRIDGE.D25
  7       1     1          PS             CELL_BRIDGE  rule_SUSPECT_1.4_CELL_BRIDGE.D42
DFM_RULE_END
DFM_RULE_DESCRIPTION_BEGIN
  rule 1 {
      1 1 2 Jun 12 14:42:19 2013
      CELL ICV_2522 1 0 0 1 0 0
      IL: distxy_1021
    }
    rule 2 {
      1 1 2 Jun 12 14:42:27 2013
      CELL ICV_2522 1 0 0 1 0 0
      IL: distxy_1023
    }
    rule 3 {
      2 2 2 Jun 12 14:42:14 2013
      CELL ICV_2522 1 0 0 1 0 0
      IL: distxy_1022
    }
...
DFM_RULE_DESCRIPTION_END

DFM_RULE_HIT_BEGIN
 symptom suspect rule_id x_dfm1    y_dfm1     x_dfm2     y_dfm2    x_coord1    y_coord1 ...
 1       1.1     1       2543.7450  4106.8250  2543.9450  4107.5250  2543.7450  4106.8250 ...
 1       1.1     3       2543.7550  4106.8800  2543.9350  4107.0600  2543.7550  4106.8800 ...
...
 1       1.2     4       -1949.440  -2208.881  -1949.280  -2208.38  -1949.440  -2208.881 ...
 1       1.3     6       -1947.516  -2207.160  -1947.325  -2206.23  -1947.516  -2207.160 ...
 1       1.4     7       -1948.180  -2207.810  -1948.030  -2207.65  -1948.180  -2207.810 ...
 1       1.5     5       -1949.440  -2206.220  -1949.280  -2205.44  -1949.440  -2206.220 ...
DFM_RULE_HIT_END
```

The DFM_RULE_HIT table lists the DFM hits. The following table describes the columnized data provided by the DFM_RULE_HIT table.

Table 3-3. DFM_RULE_HIT Fields

Rule	Description
symptom	The symptom number to which the matching suspect belongs.
suspect	Suspect ID for the matching suspect.
rule_id	ID of the rule of the violation in the current row. This ID references into the DFM_RULE table.
x_dfm1	X coordinate of the lower left corner of the violation box.
y_dfm1	Y coordinate of the lower left corner of the violation box.

Table 3-3. DFM_RULE_HIT Fields (cont.)

Rule	Description
x_dfm2	X coordinate of the upper right corner of the violation box.
y_dfm2	Y coordinate of the upper right corner of the violation box.
x_coord1	X coordinate of the lower left corner of the defect bounding box that matches the rule violation box.
y_coord1	Y coordinate of the lower left corner of the defect bounding box that matches the rule violation box.
x_coord2	X coordinate of the upper right corner of the defect bounding box that matches the rule violation box.
y_coord2	Y coordinate of upper right corner of the defect bounding box that matches the rule violation box.
dfm_properties	The DFM properties associated with the violation in the current row as read in from the RDB file. The DFM properties are specified as a comma separated list with alternating property_name and property_value fields. When there are no properties associated with the violation in the current row, this column contains "-".

Related Topics

Performing Layout-Aware Diagnosis with Tessent Diagnosis

Dofile Examples for DFM Diagnosis

You can update DFM information in existing diagnosis reports and add DFM information to non-annotated diagnosis reports.

Example 1: Create LDB and Import RDBs to Prepare for DFM Diagnosis

The following dofile shows the configuration step in preparation for running the diagnosis.

```
#CREATE AN LDB FROM DEF AND LEF
create_layout ./results/ya_demo6.ldb -def ./src/design.def.gz \
      -lef ./src/design.lef.gz -rep

#OPEN THE NEWLY CREATED LDB INTO WHICH THE RDB RULES VIOLATIONS
#ARE WRITTEN
open_layout ya_demo6.ldb

#IMPORT THE RDB FILES
import_dfm -rdb id_route_2_100.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_201.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_1210.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_1001.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_2002.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_2000.rdb -layer route_2 -type OPEN -sample 0
```

Example 2: Perform DFM Diagnosis with Tessent Diagnosis Server

After you have imported the RDB files, you can perform DFM diagnosis as shown in the following dofile.

```
#SETUP THE MONITOR
add_monitor design1 ../6.diag_server_and_annotate/flogs -results ../
results/flogs.ya
add_design  design1 ../src/design.flat.gz
add_pattern design1 ../src/design.bin.gz

#ADD THE LDB
add_layout design1 -dft ./results/ya_demo6.ldb

#USE MULTIPLE CPUS FOR THE LOCAL MACHINE
add_analyzer localhost:6

#BEGIN
start_diag

#PROCESS UNTIL THERE ARE NO FAIL LOGS LEFT IN THE QUEUE
while { [check -queued] != 0 && [check -analyzers] > 0 && ![abort] } { }
```

Example 3: Update DFM Information in Diagnosis Reports

In the process of running trial and error on the DFM rules, you may find that you no longer need some rules. Rather than starting over, you can delete the particular rules and use the annotate_diagnosis command to update the diagnosis reports.

When you use the annotate_diagnosis command, Tessent Diagnosis updates the diagnosis report and performs diagnosis. Use the report_dfm_rules command to view a list of the DFM rules.

```
#OPEN THE LDB INTO WHICH THE RULES ARE WRITTEN IF THEY
#DO NOT ALREADY EXIST
open_layout ya_demo6.ldb

#DELETE A RULE FROM THE LDB
delete_dfm -rule "metal_cross_edge_route1"

#GO THROUGH ALL THE EXISTING DIAGNOSIS RESULTS IN THE flogs.ya
#DIRECTORY AND UPDATE THEM
annotate_diagnosis flogs.ya
```

Example 4: Add DFM Information to Non-Annotated Diagnosis Reports

In addition, you can use the annotate_diagnosis command to add DFM data to older non-annotated diagnosis reports or to add new DFM rules to an existing diagnosis report (as shown below).

```
#OPEN THE LDB INTO WHICH THE RULES ARE WRITTEN IF THEY
#DO NOT ALREADY EXIST
open_layout ya_demo6.ldb

#IMPORT THE RDB FILES
import_dfm -rdb id_route_2_212.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_21.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_1110.rdb -layer route_2 -type OPEN -sample 0
import_dfm -rdb id_route_2_1111.rdb -layer route_2 -type OPEN -sample 0

#GO THROUGH ALL THE EXISTING DIAGNOSIS RESULTS IN THE flogs.ya
#DIRECTORY AND ANNOTATE THEM WITH THE DFM RULES
annotate_diagnosis flogs.ya
```

Cell-Aware Diagnosis

During a typical Tessent Diagnosis run, the tool reports cell failures at the cell boundaries. These cell-internal defects are reported as bounding boxes for the suspect cells. The cell-aware diagnosis functionality enables you to diagnose defects within the cells, called *cell-aware defects*.

Cell-aware diagnosis uses a user-defined fault model (UDFM) library file as input. UDFM files provide transistor-level fault simulation data for all library cells. Specifically, you generate a UDFM with a Diagnosis View file that contains information suitable for diagnosis, such as the electrical behavior and physical location of each defect. Tessent Diagnosis compares these defects against the failure file; the best matching defects are reported as cell-aware defect suspects. Also, design for manufacturability (DFM) violations that overlap the physical locations of cell-aware defect suspects are reported.

To get the cell-aware defect data, use Tessent CellModelGen to generate UDFM files with Diagnosis Views. Tessent CellModelGen generates one UDFM file (with Diagnosis View) per cell. Use the run_export script to concatenate the UDFM files into one library file, which then becomes the input for cell-aware diagnosis.

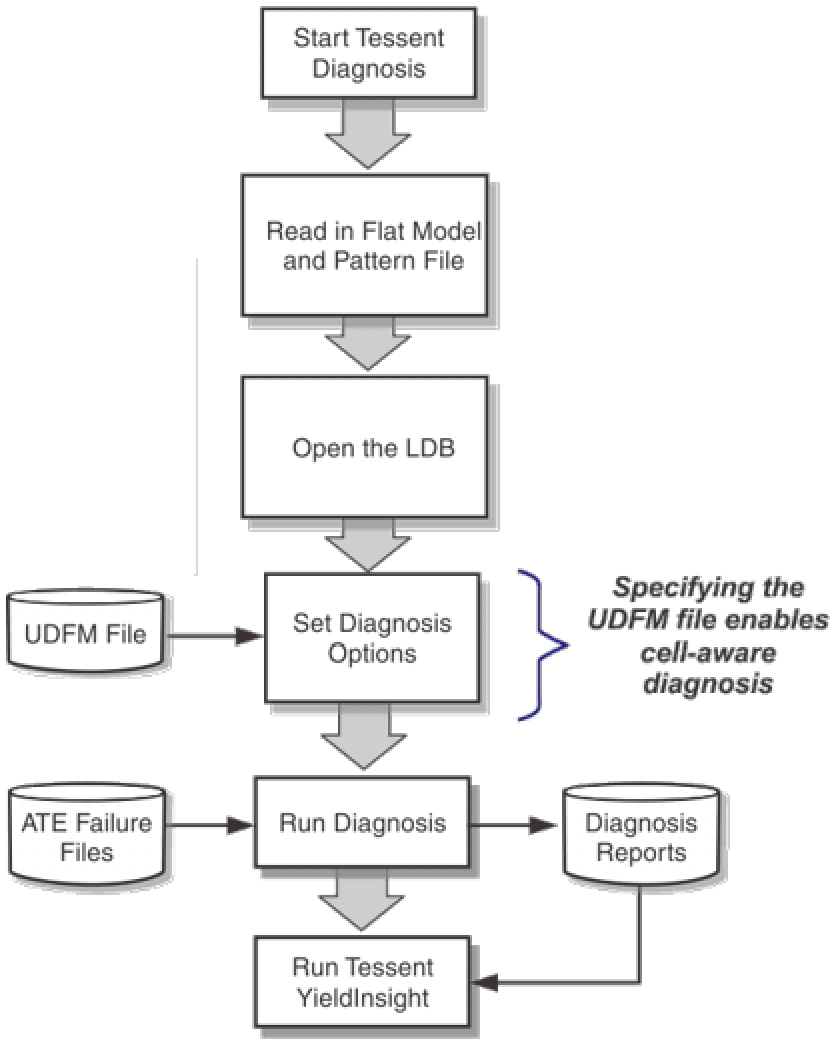
For more information about Tessent CellModelGen and how to create UDFM files with Diagnosis Views, refer to -layout_aware_analysis in the *Tessent CellModelGen Tool Reference*.

Figure 3-17 shows that the cell-aware diagnosis flow is similar to the layout-aware Tessent Diagnosis flow. You read in the design's flat model and pattern files, open the LDB, and then diagnose failure files. You must also specify the UDFM file with the set_diagnosis_options command before performing diagnosis.

Note

Tessent Diagnosis does not require automotive-grade UDFM files and does not support them in diagnosis.

Figure 3-17. Cell-Aware Diagnosis Flow



Running Cell-Aware Diagnosis

Use the set_diagnosis_options -cell_faults option to automatically enable cell-aware diagnosis. When you set this option, the tool performs a series of UDFM checks and provides warning information about the Spice optimizations that were enabled during UDFM creation with Tessent CellModelGen.

When cell-aware diagnosis completes, the tool reports cell-aware defects as follows:

- CELL_OPEN: open defect within a cell.
- CELL_BRIDGE: short defect within a cell.
- CELL_TON: defect that causes a transistor to behave as stuck-on.
- CELL_TOFF: defect that causes a transistor to behave as stuck-off.

Restrictions and Limitations

- Cell-aware diagnosis is supported only for flat model files that were created using Tessent Shell version 2013.2 or later with new kernel enabled.
- Cell-aware diagnosis is supported only for layout-aware diagnosis (server or point tool), which includes gross-delay defect diagnosis. Because at-speed diagnosis is not layout aware, it is not supported in the cell-aware diagnosis flow.
- Cell-aware diagnosis only considers defects included in the UDFM with Diagnosis View files.
- Bit analysis and chain diagnosis are not supported for cell-aware suspects.
- If you have MBFF cells with optimized stimuli in the specified UDFM file, the tool reports a warning message if you have the following:
    - An MBFF has fewer than expected stimuli for 1TF.
    - The MBFF has up to 4 bits.

      For example:

      ```
      // Warning: The UDFM for the following cells were created using
      optimized stimuli that could impact diagnosis accuracy.
      // Please re-create the UDFM for these cells without the
      multi-bit definitions:
      // SFF_TRAY2_VM_X1 ( 2 bits )
      ```

Prerequisites

- As with the layout-aware diagnosis flow, you need a layout database (LDB), design netlist, test pattern files, and ATE failure files.
- You need a UDFM file with Diagnosis View that contains layout information for the cell-aware defects. To generate this file, in Tessent CellModelGen version 2016.2 or later, enable -layout_aware_analysis before generating the UDFM library files. See

"Layout-Aware Analyze Settings" in the *Tessent CellModelGen Tool Reference* for information about the layout-aware analysis feature.

- As an optional setup step for performing DFM analysis, you have previously imported DFM violations with the import_dfm command. For more information, see "Performing DFM Diagnosis" on page 236.

Note

For cell-aware suspects, you must specify the import_dfm command with the layer names specified in the UDFM file.

Procedure

1. From a Linux/UNIX shell, enter:

   Tessent_Tree_Path/**bin/tessent -shell**

2. After Tessent Shell has started, enter:

   set_context patterns -scan_diagnosis

3. Enter:

   read_flat_model *flat_model*

   For more information about invoking Tessent Diagnosis, see the tessent shell command in the *Tessent Shell Reference Manual*.

4. Load the test pattern file. Enter the read_patterns command and specify the pathname of the final test patterns similar to:

   read_patterns *my_patterns*.**stil.giz**

   See "Preparing the Test Patterns" for more information.

5. Open the LDB, as follows:

   open_layout *design*.**ldb**

6. Specify the UDFM with Diagnosis View file, as follows:

   ```
   # When using the point tool:
   set_diagnosis_options -cell_faults name.udfm
   # When using the server:
   set_diagnosis_options monitor_id -cell_faults name.udfm
   ```

   Specifying the -cell_faults option automatically enables cell-aware diagnosis. Tessent Diagnosis uses the fault models listed in the specified UDFM file to perform cell-aware diagnosis.

   Note

   You can only specify one UDFM file. In Tessent CellModelGen, use the run_export script to merge the cell-related UDFM files it produces (one per cell) into one file.

7. Run diagnosis on the failure file. Enter the diagnose_failures command and specify the pathname of the failure file similar to:

   diagnose_failures *failure_file* -output *ascii_report*

   Tessent Diagnosis uses the UDFM faults and their tests to diagnose the failure files.

Results

Tessent Diagnosis generates diagnosis reports as described in "Cell-Aware Diagnosis Report."

Performing Cell-Aware Diagnosis With RCD

The process for performing cell-aware diagnosis that includes RCD constants follows the regular cell-aware diagnosis process with the addition of specifying the set_diagnosis_options -rcad on option.

For information about RCD, refer to "Diagnosis for Root Cause Deconvolution Analysis" on page 221.

For the best results, use the following versions:

- UDFM generated using Tessent CellModelGen version 2018.1 or later.
- Diagnosis reports generated using Tessent Diagnosis version 2018.4 or later.
- RCD constants generated using Tessent Diagnosis version 2019.1 or later.

Prerequisites

- Ensure that the set_diagnosis_options -include_rcd_constants is set to on. This is the default behavior.

Procedure

1. In Tessent Shell, in the diagnosis context, create RCAD constants and store them in an existing LDB.

   RCAD constants refers to the set of RCD constants plus constants for the cell-internal layers, which are sourced from the specified UDFM file.

   The following sample dofile illustrates this task.

```
set_context patterns -scan_diagnosis
read_flat_model flat_model
create_layout ./src/design.dft.ldb -lef ./src/design.lef -def ./src/
design.def
read_patterns ./src/patterns.ascii

// Enable RCAD constant creation
set_diagnosis_options -cell_faults udfmFile -rcad on

open_layout ./src/design.dft.ldb

// Generate RCAD constants
create_feature_statistics -add_processors host:16
```

Tip

To minimize RCD constant creation processing times, turn on multi-processing using the create_feature_statistics -add_processors option. The recommended number of processors is 16.

2. Perform RCAD-enabled diagnosis to produce diagnosis reports that include regular and cell-internal RCD constants. For example:

```
set_context patterns -scan_diagnosis
read_flat_model flat_model
read_patterns ./src/patterns.ascii

// Enable RCAD, when -rcad is off tool performs regular CAD
set_diagnosis_options -cell_faults udfmFile -rcad on

open_layout ./src/design.dft.ldb
diagnose_failure failure_log
```

If you are using the Tessent Diagnosis server, specify:

```
set_diagnosis_options monitor -cell_faults udfmFile -rcad on
```

Results

Tessent Diagnosis generates diagnosis reports as described in "Cell-Aware Diagnosis Report."

Cell-Aware Diagnosis Report

The cell-aware diagnosis report designates cell-aware defect suspects with the following designators: CELL_BRIDGE, CELL_OPEN, CELL_TON, and CELL_TOFF.

Tessent Diagnosis generates cell-aware diagnosis reports similar to the following example. In addition, if you have imported DFM data as described in "Performing DFM Diagnosis," the cell-aware diagnosis report includes a DFM_RULE section that lists DFM hits for cell-aware defect suspects.

Note

For cell-aware defects, the value column is not applicable, and the layout_layer and critical_area are not used.

```
               fail_  pass_
suspect  score match  mismatch type   value  pin_pathname     cell_name  net_pathname
layout_status
------------------------------------------------------------------------------------
1       100    2      0       STUCK    0   /cpu_i/uPMI/ix1420/A1 aoi22 /cpu_i/uPMI/nx1781
INPUT_PIN_FAULT
  #potential_open_segments=1,#total_segments=15,#potential_bridge_aggressors=0,#total_neigh
bors=na
   #cell_internal_suspects=4,#total_cell_internal_faults=56
  suspect score fail_match pass_mismatch type     value location  layout_layer critical_area
  ----------------------------------------------------------------------------------
  1.1      100     2          0       CELL          0   /cpu_i/uPMI/ix420
  1.2    100    2          0       CELL_BRIDGE  -  D2/NET7_1--C_3/1.0
  1.3    100    2          0       CELL_OPEN    -  D10/NET7_3--NET7_4/1.0G
  1.4    95     2          1       CELL_TON     -  D22/M1_1--M1_2/1.0
  1.5    95     2          1       CELL_TOFF    -  D28/M4_1--M4_2/1.0G
  1.6      95      2          1       OPEN          0  B6 route_2     1.85E+02
                                                       VIA2        9.59E+00
                                                       route_3     1.34E+01
-----------------------------------------------------------------------------------
```

In this example, the tool identified four cell-aware defects for cell /cpu_i/uPMI/ix1420. The summary line indicates that these four cell-aware defects were called out by cell-aware diagnosis out of a total of 56 cell-aware faults defined for the specific cell instance in the UDFM file you specified with the set_diagnosis_options command. All cell-aware suspects belonging to the same cell instance are included in the same table.

The location string describes the suspect information, and it consists of three parts: defect ID, net1, and net2, and the resistor value. The string is formatted:

```
DefectID/Net1--Net2/ResistorValue
```

- Defect ID: a unique ID created by Tessent CellModelGen for each defect for a specific cell.
- Net1--Net2
    - For cell bridge suspects, the nodes that are shorted by the specified defect. For example, suspect 1.2 in the example is a bridge suspect on layer M1 between node NET7_1 and C_3.
    - For cell open suspects, the nodes indicate where a large resistor is inserted to mimic the open defect. For example, suspect 1.3 is an open suspect on net NET7 located between nodes NET7_3 and NET7_4.

    - For both cell TON and cell TOFF suspects, the nodes indicate where the drain and source terminal of that transistor are connected. For example, suspect 1.4 is a TON suspect for transistor M1, and suspect 1.5 is a TOFF suspect for transistor M4.

- ResistorValue: a small value (1.0) is typically used for bridge suspects and TON suspects, and a large value (1.0G) is used for open suspects and TOFF suspects.

If the layout information is defined for each UDFM fault in the provided UDFM file, the tool adds a CELL_DEFECT_LOCATION XMAP table to the layout-aware diagnosis report.

Note

Your layout-aware diagnosis report could show that there are no cell-aware defects yet greater than zero total cell-aware faults. For example: #cell_internal_suspects=0, #total_cell_internal_faults=207. This result means that 207 UDFM faults were simulated and rejected as suspects because they did not match the defect well enough.

Note

If the UDFM file does not contain the cell type that is identified by cell internal diagnosis, then the total_cell_internal_faults are reported as =0. For example:

#cell_internal_suspects=0, #total_cell_internal_faults=0

The following example shows how the CELL_DEFECT_LOCATION XMAP table reports the layout information for cell-aware suspects:

```
CELL_DEFECT_LOCATION_BEGIN

  symptom suspect layout_layer category critical_area  x_coord1  y_coord1  x_coord2 y_coord2

  1       1.2     M1            BR.D2    NA          178.3370  748.1600  178.4330  748.1920
  1       1.3     M1            OP.D10   NA          178.4300  748.2540  178.5200 748.2850
  1       1.3     COD           OP.D10   NA          178.2170  748.3020  178.2830 748.3920
  1       1.4     NA            TON.D22  NA          178.2890  748.3250  178.3020 748.3570
  1       1.5     NA            TOFF.D28 NA          178.2890  748.4240  178.3020 748.4440

CELL_DEFECT_LOCATION_END
```

For example, the first line shows that bridge suspect 1.2 locates within a box with bottom left corner at (178.3370, 748.1600) and top-right corner at (178.4330, 748.1920) on layer M1. For open suspect 1.3, two possible locations are reported: the first one is on layer M1, and the second one is on layer COD (contact to diffusion). For TON/TOFF suspects, you get a bounding box to show the location of the corresponding transistor without specific layer information.

Note

The tool does not generate the CELL_DEFECT_LOCATION table if the layout information is not defined in the UDFM file. If this is the case, you need to manually map the cell suspect based on the defect information in the report (type, Net1, and Net2).

For cell_bridge suspects, the tool indicates inter-layer defects as shown below. The layout layer for suspect 1.2 shows “PS-DI.” This means that the cell_bridge defect spans layers PS (poly) and DI (diffusion).

```
CELL_DEFECT_LOCATION_BEGIN
 symptom suspect layout_layer  category  critical_area x_coord1 y_coord1  x_coord2 y_coord2
 1       1.2     PS-DI         BR.D34    NA            387.6840 166.4070  387.6940 166.5410
 1       1.4     NA            TON.D41   NA            387.8460 166.5410  387.8660 166.6800
```

Layout information for cell-aware suspects is also reported in the CSV reports and chip-level layout marker files.

In addition to the CELL_DEFECT_LOCATION table, the tool provides a CELL_AREA table as shown in the example below. (This example does not follow from the previous examples.) For the cell areas described by the coordinates in the CELL_DEFECT_LOCATION table, the CELL_AREA table provides a summary of the areas aggregated by symptom/ layer, where a layer could contain multiple suspects.

```
CELL_AREA_BEGIN
  symptom suspect layout_layer    category                area      ratio
  1       1       CELL            arfadd_01_std_thk_dnd   55987.2   100%
  1       1       CO-PS           arfadd_01_std_thk_dnd   724.348   1.3%
  1       1       PS              arfadd_01_std_thk_dnd   450.400   0.8%
  1       1       PS-DI           arfadd_01_std_thk_dnd   215.964   0.4%
CELL_AREA_END
```

This example shows the results for one symptom with several layers. The table provides a correct summation of the area of the rectangles on a given layer by taking into account overlapping rectangles. That is, areas that overlap are not double-counted. The ratio column lists the ratio of the cell area for the suspect layer against the overall cell area. The overall cell area is the first entry in the table, so its ratio is 100%. You can use this table to scan for the largest ratio so you can investigate those suspects first.

Chip-Level Layout Marker File Results Viewing

One of the results of the cell-aware diagnosis flow is the chip-level layout marker file. Unlike the cell-level layout marker file produced by Tessent CellModelGen, the chip-level layout marker file enables you to view the cell-aware suspects against their surrounding layouts or backgrounds at the chip level.

By using the chip-level layout marker file, you can use CalibreDESIGNrev to view cell-aware defects inside the cells with chip-level coordinates. You provide the chip-level GDSII and the chip-level layout marker file.

Refer to “The Calibre GDS Viewer” in the *Tessent CellModelGen Tool Reference* for detailed information about viewing the cell-level results in CalibreDESIGNrev. Refer to “Guidelines for Viewing the Diagnosis Results in Calibre DESIGNrev” in the *Tessent Diagnosis User’s Manual* for information about viewing the diagnosis layout marker file.

Considerations for At-Speed Diagnosis

The layout-aware diagnosis flow does not support at-speed diagnosis. You can, however, perform at-speed diagnosis with Tessent Diagnosis and create layout markers with a diagnosis LDB.

The following dofile example illustrates the command sequence you use to create a layout-marker file:

```
create_layout ./src/design.dft.ldb -lef ./src/design.lef -def ./src/design.def
open_layout ./src/design.dft.ldb
read_patterns my_patterns.asc
set_diagnosis_options -at_speed on
diagnose_failures ./tester_files/file1.flog
write_diagnosis -format layout_marker text csv -file results/file1 -replace
write_diagnosis -short -format layout_marker text csv -file results/file2 -replace
write_diagnosis -encoded -format layout_marker text csv -file results/file3 -replace
report_diagnosis
close_layout
exit
```

Related Topics

At-Speed Failure Diagnosis

Source/Sink Polygon Layout Markers for Open Diagnosis Suspects

By default, Tessent Diagnosis generates layout marker files for Open suspects in the diagnosis. The information includes polygons for the sink and source cells for the defect. Calibre RVE may be used to view a layout marker file.

Open Suspect Diagnosis and Layout Marker File Generation

The source and sink cell Open suspect information within the layout marker files is available by default when the tool generates a layout marker file.

Layout Marker File Generation

The "set_diagnosis_options -include_src_sink_cells_in_marker" command controls the generation of the Open suspect information in the layout marker files. For example:

```
set_diagnosis_options -include_src_sink_cells_in_marker on
diagnose_failures -o phyb2_adom_0.diag ../flogs/phyb2_adom_o \
                  -output my_diag
write_diagnosis -format layout -file phyb2_adom_0
```

By default, the tool generates the markers.

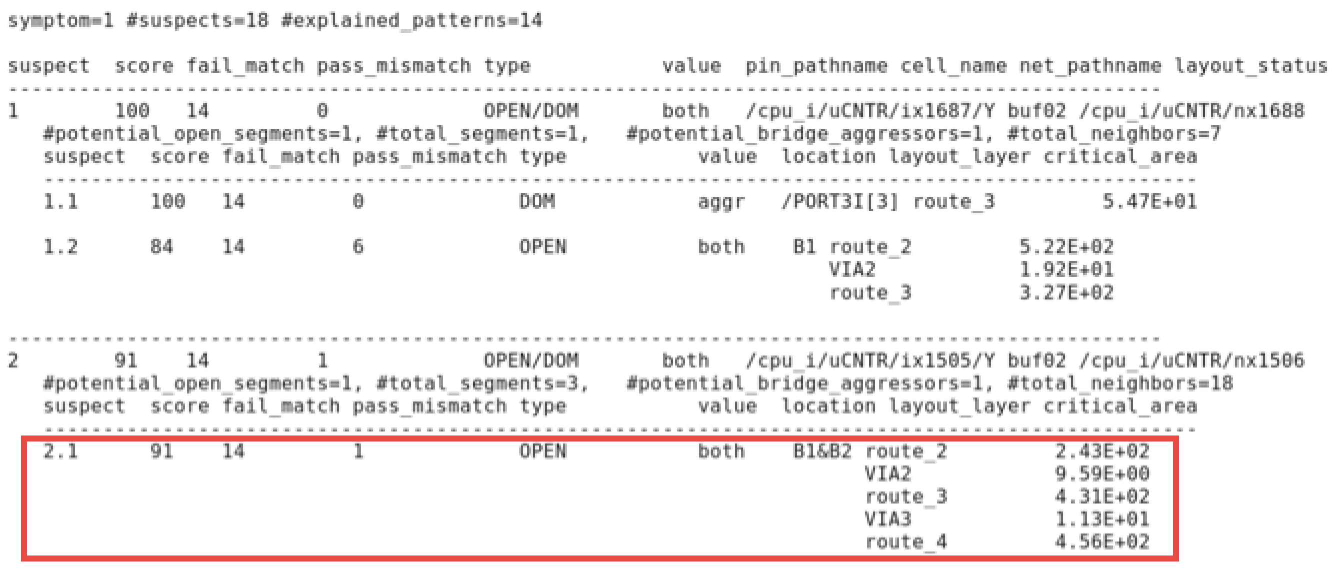
Layout Marker Files Keywords

The resulting layout marker file contains the sink and source suspect data located at the ends of the Open suspect nets. The source and sink cell suspect information is identifiable by the SOURCE CELL and SINK CELL keywords. For example:

```
phyb2_adom_0.lay

SUSPECT-1-2.1_SINK CELL 12
1 1 0   Mon Nov 18  04:11:44 GMT 2019
p 1 4
ALL 12
SYMPTOM-1 12
SUSPECT-1-2 12
SUSPECT-1-2.1 12
4513000 1170500
4562000 1170500
4562000 1290500
4513000 1290500
```

```
SUSPECT-1-2.1_SINK CELL 12
1 1 0   Mon Nov 18  04:11:44 GMT 2019
p 1 4
ALL 12
SYMPTOM-1 12
SUSPECT-1-2 12
SUSPECT-1-2.1 12
4570000 1170500
4611000 1170500
4611000 1290500
4570000 1290500

SUSPECT-1-2.1_SOURCE CELL 12
1 1 0   Mon Nov 18  04:11:44 GMT 2019
p 1 4
ALL 12
SYMPTOM-1 12
SUSPECT-1-2 12
SUSPECT-1-2.1 12
4339000 745500
4372500 745500
4372500 865500
4339000 865500
```

The suspects in the layout marker file correspond to the physical attributes of suspect 2.1 of symptom 1 in the diagnosis report shown below (highlighted by the red box):



```
symptom=1 #suspects=18 #explained_patterns=14

suspect  score fail_match pass_mismatch type          value  pin_pathname cell_name net_pathname layout_status
------------------------------------------------------------------------------------------------------
1        100   14         0             OPEN/DOM      both   /cpu_i/uCNTR/ix1687/Y buf02 /cpu_i/uCNTR/nx1688
   #potential_open_segments=1, #total_segments=1,   #potential_bridge_aggressors=1, #total_neighbors=7
   suspect  score fail_match pass_mismatch type          value  location layout_layer critical_area
   ------------------------------------------------------------------------------------------------------
   1.1      100   14         0             DOM           aggr   /PORT3I[3] route_3         5.47E+01

   1.2      84    14         6             OPEN          both    B1 route_2         5.22E+02
                                                                    VIA2            1.92E+01
                                                                    route_3         3.27E+02

------------------------------------------------------------------------------------------------------
2        91    14         1             OPEN/DOM      both   /cpu_i/uCNTR/ix1505/Y buf02 /cpu_i/uCNTR/nx1506
   #potential_open_segments=1, #total_segments=3,   #potential_bridge_aggressors=1, #total_neighbors=18
   suspect  score fail_match pass_mismatch type          value  location layout_layer critical_area
   ------------------------------------------------------------------------------------------------------
   2.1      91    14         1             OPEN          both    B1&B2 route_2         2.43E+02
                                                                       VIA2            9.59E+00
                                                                       route_3         4.31E+02
                                                                       VIA3            1.13E+01
                                                                       route_4         4.56E+02
```

The diagnosis OPEN_LOCATION section shows the polygons for the defect bounding boxes. The source and sink cell polygons in the layout marker file contain the beginning and ending segments.

```
OPEN_LOCATION_BEGIN
  symptom suspect layout_layer    category    critical_area x_coord1     y_coord1     x_coord2     y_coord2
  1       2.1     route_2         OP          2.17E+02      4363.0000    796.0000     4366.0000    898.5000
  1       2.1     route_2         OP          1.34E+01      4362.5000    895.0000     4366.5000    899.0000
  1       2.1     route_2         OP          1.34E+01      4362.5000    795.5000     4366.5000    799.5000
  1       2.1     VIA2            via2        9.59E+00      4363.5000    896.0000     4365.5000    898.0000
  1       2.1     route_3         OP          2.99E+02      4363.0000    895.5000     4506.5000    898.5000
  1       2.1     route_3         OP          7.77E+01      4503.5000    1104.0000    4536.5000    1107.0000
  1       2.1     route_3         OP          1.97E+01      4536.5000    1104.0000    4540.5000    1107.0000
  1       2.1     route_3         OP          1.34E+01      4362.5000    895.0000     4366.5000    899.0000
  1       2.1     route_3         OP          1.06E+01      4502.5000    1103.0000    4507.5000    1108.0000
  1       2.1     route_3         OP          1.06E+01      4502.5000    894.5000     4507.5000    899.5000
  1       2.1     VIA3            via3        5.66E+00      4503.5000    1104.0000    4506.5000    1107.0000
  1       2.1     VIA3            via3        5.66E+00      4503.5000    895.5000     4506.5000    898.5000
  1       2.1     route_4         OP          4.35E+02      4503.5000    895.5000     4506.5000    1107.0000
  1       2.1     route_4         OP          1.06E+01      4502.5000    1103.0000    4507.5000    1108.0000
  1       2.1     route_4         OP          1.06E+01      4502.5000    894.5000     4507.5000    899.5000
```

Using Calibre RVE to View Source/Sink Cell Polygons

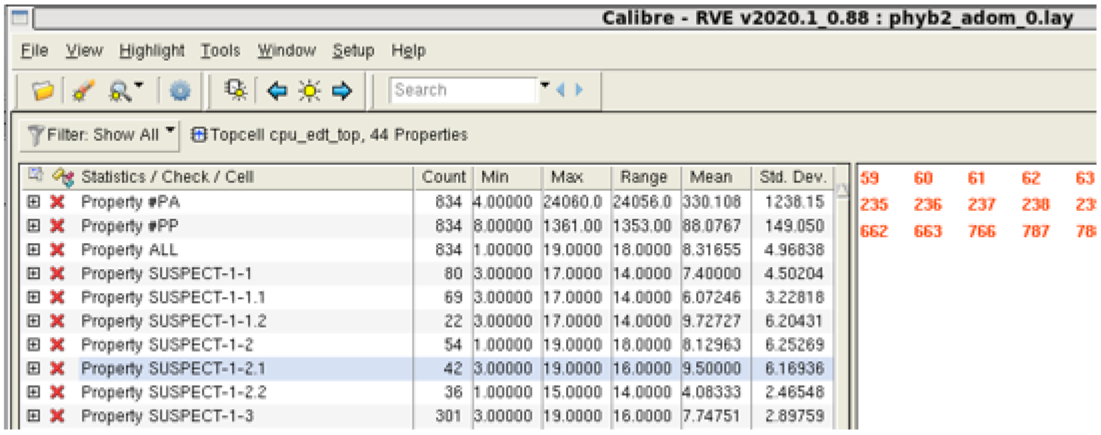
You can use Calibre RVE to highlight specific source/sink diagnosis defect and landmark information contained within a layout marker file. The following procedure assumes that you have some experience with Calibre RVE.

1. Open Calibre RVE. For example:

   ```
   $CALIBRE_HOME/bin/calibredrv -64 -m ya_demo.oas
   ```

2. To start RVE, click **Verification** in the menu bar, and then click **Start RVE**.
3. In the Calibre RVE dialog box, under Database, select the layout marker file you want to view.

The RVE control window displays. The following figure shows that Property SUSPECT-1-2.1 is select; this corresponds to diagnosis coordinate 1.2.1.

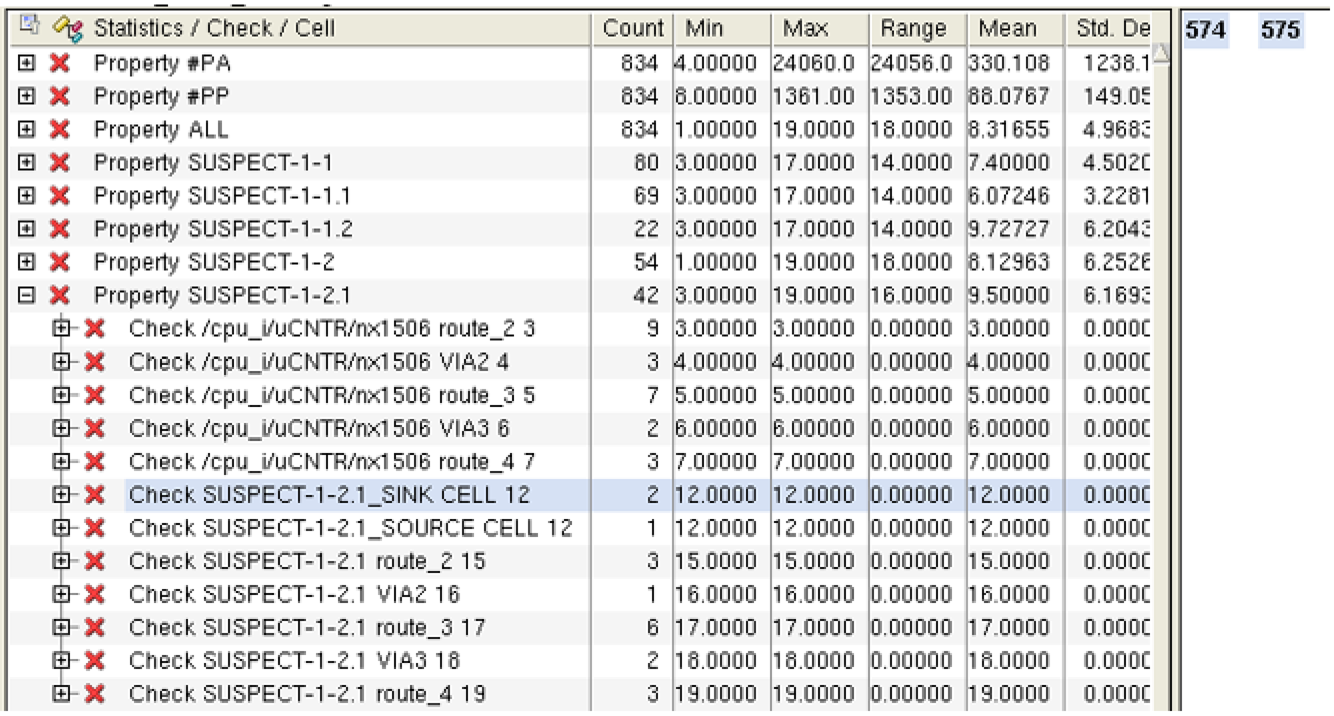

Calibre - RVE v2020.1_0.88 : phyb2_adom_0.lay
File View Highlight Tools Window Setup Help
Search
Filter: Show All
Topcell cpu_edt_top, 44 Properties

Statistics / Check / Cell	Count	Min	Max	Range	Mean	Std. Dev.
Property #PA	834	4.00000	24060.0	24056.0	330.108	1238.15
Property #PP	834	8.00000	1361.00	1353.00	88.0767	149.050
Property ALL	834	1.00000	19.0000	18.0000	8.31655	4.96838
Property SUSPECT-1-1	80	3.00000	17.0000	14.0000	7.40000	4.50204
Property SUSPECT-1-1.1	69	3.00000	17.0000	14.0000	6.07246	3.22818
Property SUSPECT-1-1.2	22	3.00000	17.0000	14.0000	9.72727	6.20431
Property SUSPECT-1-2	54	1.00000	19.0000	18.0000	8.12963	6.25269
Property SUSPECT-1-2.1	42	3.00000	19.0000	16.0000	9.50000	6.16936
Property SUSPECT-1-2.2	36	1.00000	15.0000	14.0000	4.08333	2.46548
Property SUSPECT-1-3	301	3.00000	19.0000	16.0000	7.74751	2.89759

59	60	61	62	63
235	236	237	238	23
662	663	766	787	78

4. Click the "+" symbol next to the property of interest to expand its hierarchy tree.

   The following figure shows the SINK CELL and SOURCE CELL bounding boxes within the list of defect suspects.

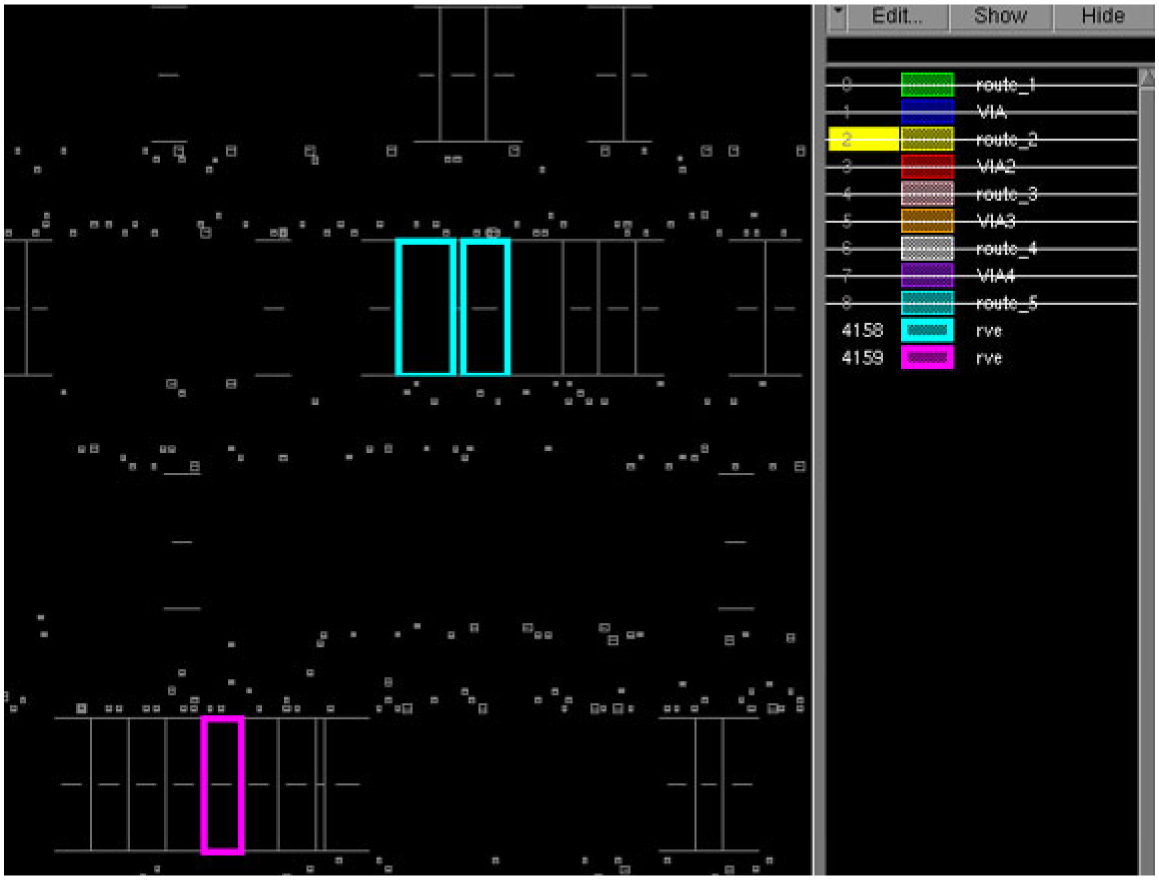


Statistics / Check / Cell	Count	Min	Max	Range	Mean	Std. De
Property #PA	834	4.00000	24060.0	24056.0	330.108	1238.1
Property #PP	834	8.00000	1361.00	1353.00	88.0767	149.05
Property ALL	834	1.00000	19.0000	18.0000	8.31655	4.9683
Property SUSPECT-1-1	80	3.00000	17.0000	14.0000	7.40000	4.502C
Property SUSPECT-1-1.1	69	3.00000	17.0000	14.0000	6.07246	3.2281
Property SUSPECT-1-1.2	22	3.00000	17.0000	14.0000	9.72727	6.2043
Property SUSPECT-1-2	54	1.00000	19.0000	18.0000	8.12963	6.252E
Property SUSPECT-1-2.1	42	3.00000	19.0000	16.0000	9.50000	6.1693
Check /cpu_i/uCNTR/nx1506 route_2 3	9	3.00000	3.00000	0.00000	3.00000	0.000C
Check /cpu_i/uCNTR/nx1506 VIA2 4	3	4.00000	4.00000	0.00000	4.00000	0.000C
Check /cpu_i/uCNTR/nx1506 route_3 5	7	5.00000	5.00000	0.00000	5.00000	0.000C
Check /cpu_i/uCNTR/nx1506 VIA3 6	2	6.00000	6.00000	0.00000	6.00000	0.000C
Check /cpu_i/uCNTR/nx1506 route_4 7	3	7.00000	7.00000	0.00000	7.00000	0.000C
Check SUSPECT-1-2.1_SINK CELL 12	2	12.0000	12.0000	0.00000	12.0000	0.000C
Check SUSPECT-1-2.1_SOURCE CELL 12	1	12.0000	12.0000	0.00000	12.0000	0.000C
Check SUSPECT-1-2.1 route_2 15	3	15.0000	15.0000	0.00000	15.0000	0.000C
Check SUSPECT-1-2.1 VIA2 16	1	16.0000	16.0000	0.00000	16.0000	0.000C
Check SUSPECT-1-2.1 route_3 17	6	17.0000	17.0000	0.00000	17.0000	0.000C
Check SUSPECT-1-2.1 VIA3 18	2	18.0000	18.0000	0.00000	18.0000	0.000C
Check SUSPECT-1-2.1 route_4 19	3	19.0000	19.0000	0.00000	19.0000	0.000C

574	575

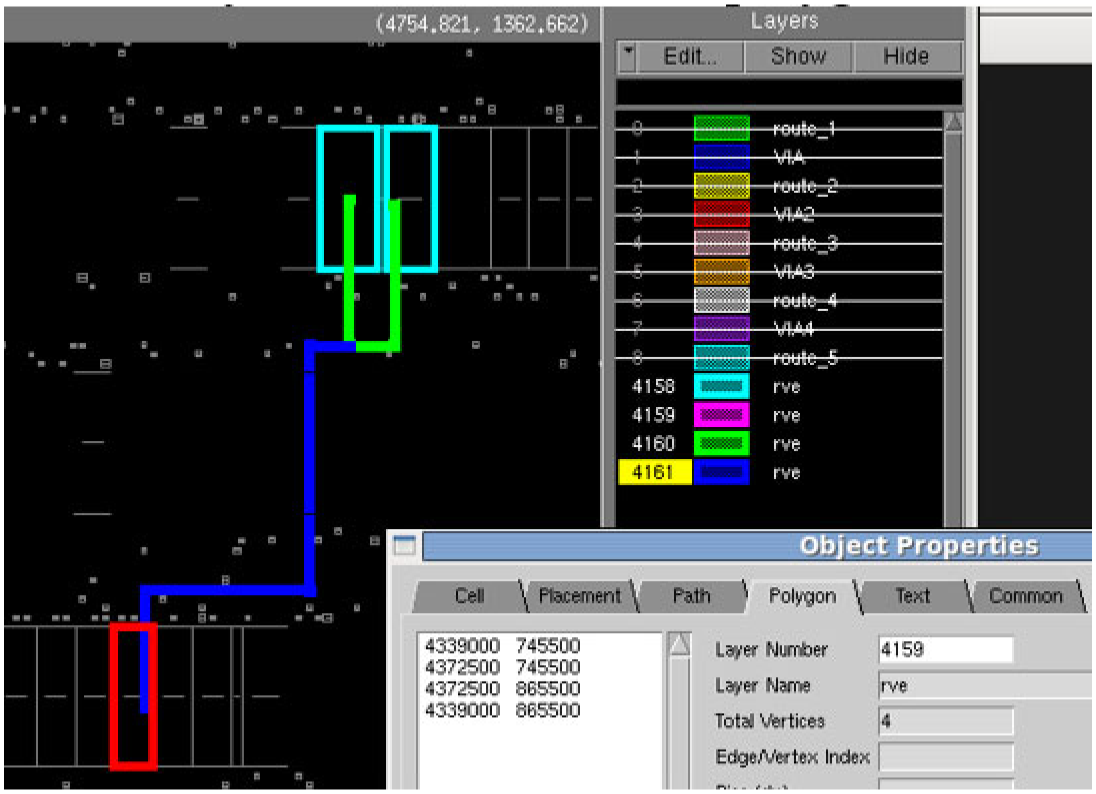
5. Select the SINK CELL, and then click the **Highlight** button in the standard RVE tool bar. Perform the same task with the SOURCE CELL.

In the resulting layout view, the sink cells and source cells display with their own layer color attributes in the layer map area:



6. As needed, select and highlight any of the suspects to view them in RVE.

You can view the polygon coordinate data for any selected layer by accessing the Object Properties dialog box.



```
4339000  745500
4372500  745500
4372500  865500
4339000  865500
```

Layer Number	4159
Layer Name	rve
Total Vertices	4
Edge/Vertex Index	

Guidelines for Viewing the Diagnosis Results in Calibre DESIGNrev

To view the Tessent Diagnosis layout-area results in Calibre DESIGNrev, you must use a layout marker file instead of the Tessent diagnosis report. A layout marker file contains coordinate information that Calibre DESIGNrev uses for displaying the layout-aware results. In general, you create a layout marker file when you write out the Tessent Diagnosis report.

To view the layout-aware diagnosis results in Calibre DESIGNrev, choose one of the following topics depending on the Calibre version you are using:

- Viewing Results in Pre-Calibre 2010.1 Software
- Viewing Results in Calibre 2010.1 or Newer Software

To use Calibre DESIGNrev, you must:

- Have access to a Siemens EDA Calibre software tree (version Calibre 2007.3 or newer) and the following Calibre licenses:
    - caldesignrev
    - calibreqdb

  For complete Calibre licensing information, see the *Calibre Administrator's Guide* in your Calibre software tree or on Siemens EDA Support Center.

  For more information about using Calibre DESIGNrev, refer to the *Calibre DESIGNrev Layout Viewer User's Manual*.

- Have generated a Calibre tool-compatible layout marker file as follows;
    - Tessent Diagnosis — Use the write_diagnosis -format layout_marker command and arguments.
    - Tessent Diagnosis Server — Use the add_reporting_format -layout_marker command and argument.

Viewing Results in Pre-Calibre 2010.1 Software

Use the following procedure to view results in Calibre software prior to 2010.1.

Prerequisites

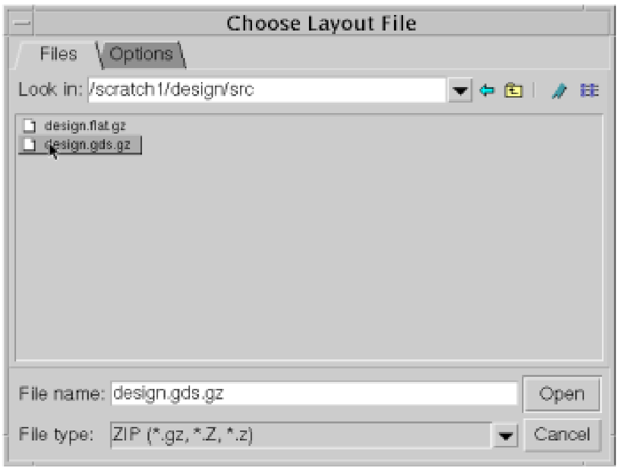
- A layout marker file. You cannot use the diagnosis report. Consult "Guidelines for Viewing the Diagnosis Results in Calibre DESIGNrev" before you begin.

Procedure

1. Invoke Calibre DESIGNrev from a UNIX/Linux shell using the following syntax:

   calibredrv

2. Select **File** > open_layout, select the layout, and click **Open**.



Look in: /scratch1/design/src

design.flat.gz
design.gds.gz

File name: design.gds.gz

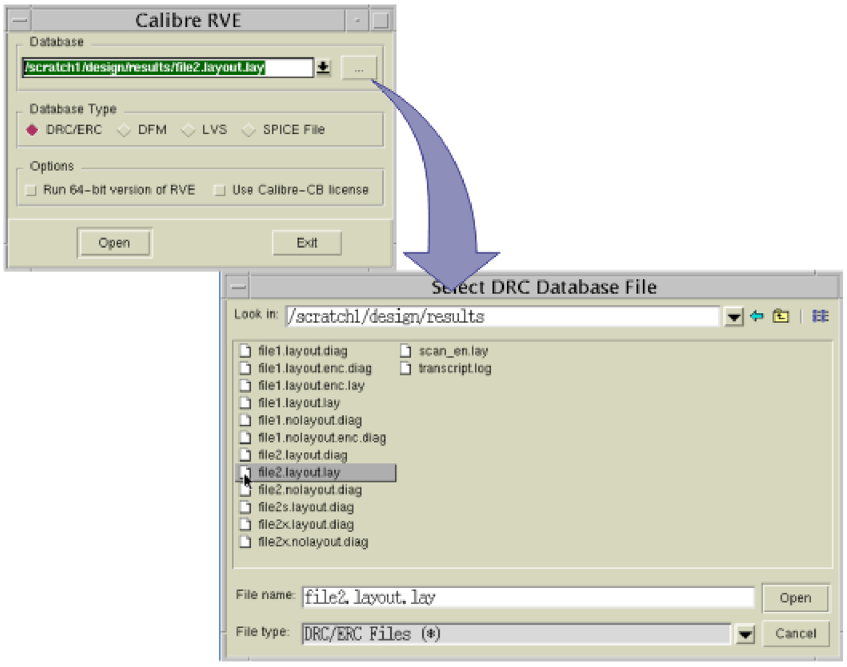
File type: ZIP (*.gz, *.Z, *.z)

   You must open the same layout and coordinate space as the LEF/DEF files. In some cases, you must create a new GDSII or OASIS® layout using the **File** > **Open Database** option in Calibre DESIGNrev.

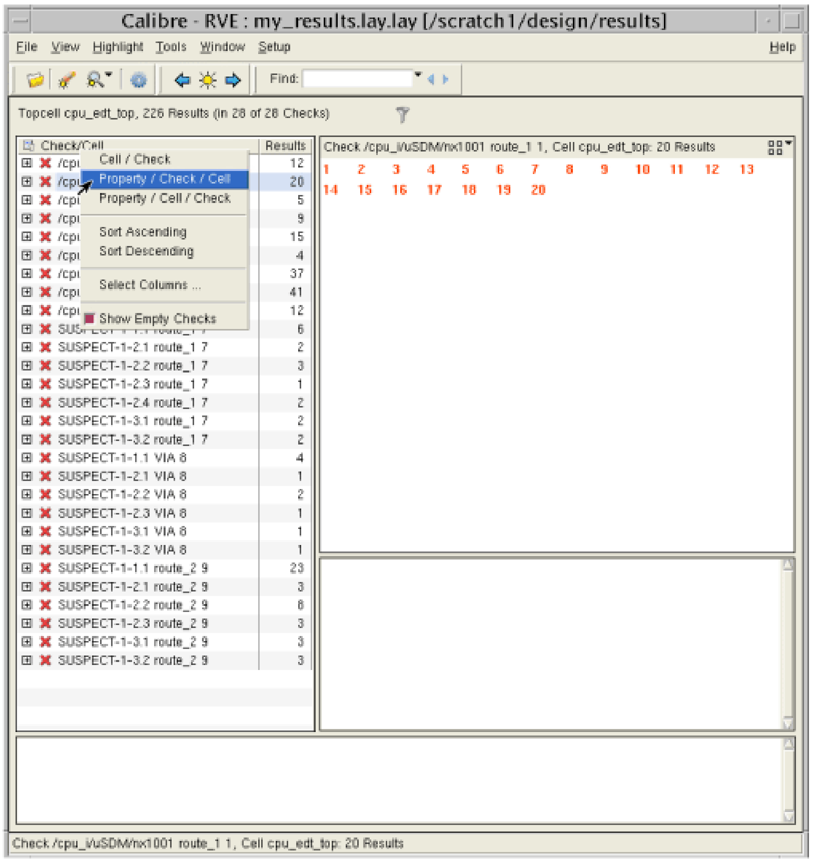
3. To enable highlighting in different layers/colors (recommended), select **Tools** > **RVE/CI Setup**. In the RVE/CI Setup dialog box, click the **Multi-layerHighlights** radio button if not activated, and then click **OK**.

4. Select **Tools** > **Start RVE**

5. Navigate to and select the Tessent Diagnosis layout marker file. Click **Open**.



6. In the left-hand pane, move the mouse cursor over the **Check/Cell** column heading, and click and hold the mouse menu button, and choose **Property / Check / Cell** option.



Topcell cpu_edt_top, 226 Results (in 28 of 28 Checks)

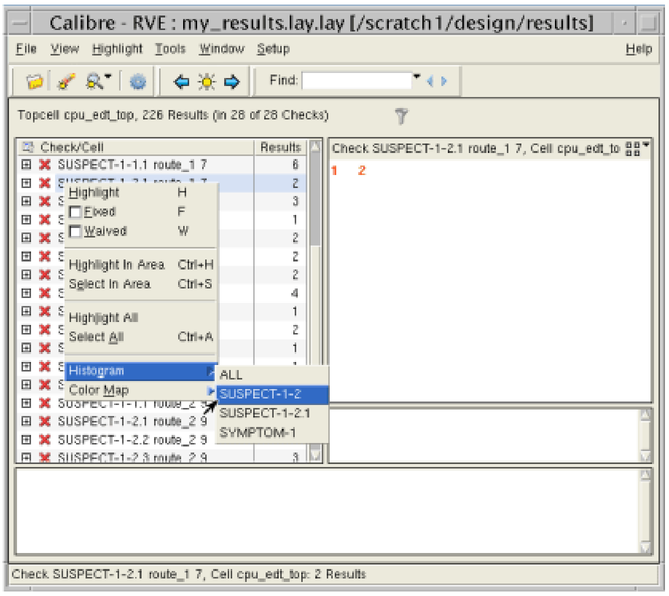
Check/Cell	Results
/cpu	12
/cpu	20
/cpu	5
/cpu	9
/cpu	15
/cpu	4
/cpu	37
/cpu	41
/cpu	12
SUS	6
SUSPECT-1-2.1 route_1 7	2
SUSPECT-1-2.2 route_1 7	3
SUSPECT-1-2.3 route_1 7	1
SUSPECT-1-2.4 route_1 7	2
SUSPECT-1-3.1 route_1 7	2
SUSPECT-1-3.2 route_1 7	2
SUSPECT-1-1.1 VIA 8	4
SUSPECT-1-2.1 VIA 8	1
SUSPECT-1-2.2 VIA 8	2
SUSPECT-1-2.3 VIA 8	1
SUSPECT-1-3.1 VIA 8	1
SUSPECT-1-3.2 VIA 8	1
SUSPECT-1-1.1 route_2 9	23
SUSPECT-1-2.1 route_2 9	3
SUSPECT-1-2.2 route_2 9	8
SUSPECT-1-2.3 route_2 9	3
SUSPECT-1-3.1 route_2 9	3
SUSPECT-1-3.2 route_2 9	3

Check /cpu_i/uSDM/nx1001 route_1 1, Cell cpu_edt_top: 20 Results

1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20

Check /cpu_i/uSDM/nx1001 route_1 1, Cell cpu_edt_top: 20 Results

7. In the left-hand pane, click a suspect of interest, click and hold the mouse menu button, and choose **Histogram > suspect**.



Topcell cpu_edt_top, 226 Results (in 28 of 28 Checks)

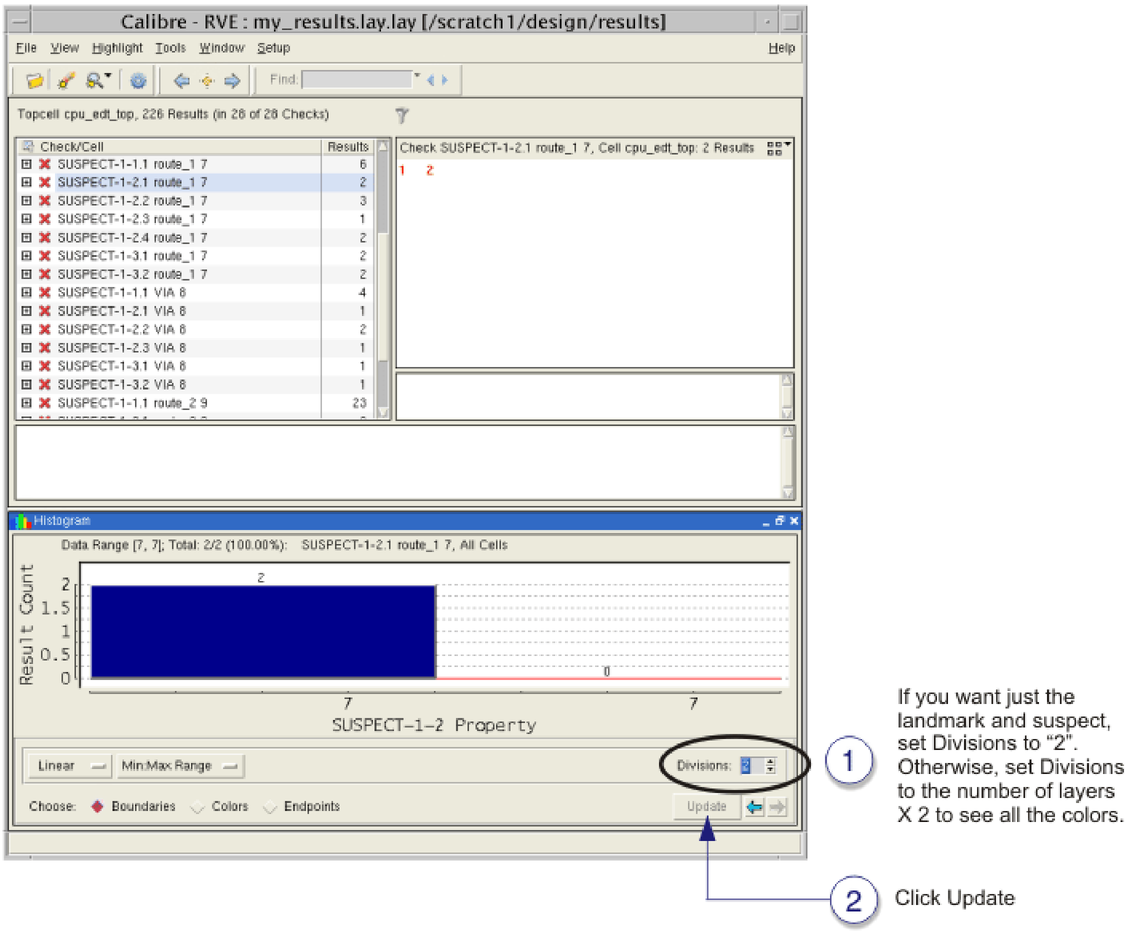
Check/Cell	Results
SUSPECT-1-1.1 route_1 7	6
SUSPECT-1-2.1 route_1 7	2
S	3
S	1
S	2
S	2
S	2
S	4
S	1
S	2
S	1
S	1
S	
SUSPECT-1-1.1 route_2 9	
SUSPECT-1-2.1 route_2 9	
SUSPECT-1-2.2 route_2 9	
SUSPECT-1-2.3 route_2 9	3

Check SUSPECT-1-2.1 route_1 7, Cell cpu_edt_to

1 2

Check SUSPECT-1-2.1 route_1 7, Cell cpu_edt_top: 2 Results

8. In the **Histogram Pane**, select the number of divisions and click **Update**.


Calibre - RVE : my_results.lay.lay [/scratch1/design/results]
File View Highlight Tools Window Setup Help
Find:
Histogram
Result Count
2
1.5
1
0.5
0
SUSPECT-1-2 Property
Linear
Min:Max Range
Divisions:
Update
1
2

Topcell cpu_edt_top, 226 Results (in 28 of 28 Checks)

Check/Cell	Results
SUSPECT-1-1.1 route_1 7	6
SUSPECT-1-2.1 route_1 7	2
SUSPECT-1-2.2 route_1 7	3
SUSPECT-1-2.3 route_1 7	1
SUSPECT-1-2.4 route_1 7	2
SUSPECT-1-3.1 route_1 7	2
SUSPECT-1-3.2 route_1 7	2
SUSPECT-1-1.1 VIA 8	4
SUSPECT-1-2.1 VIA 8	1
SUSPECT-1-2.2 VIA 8	2
SUSPECT-1-2.3 VIA 8	1
SUSPECT-1-3.1 VIA 8	1
SUSPECT-1-3.2 VIA 8	1
SUSPECT-1-1.1 route_2 9	23

Check SUSPECT-1-2.1 route_1 7, Cell cpu_edt_top: 2 Results

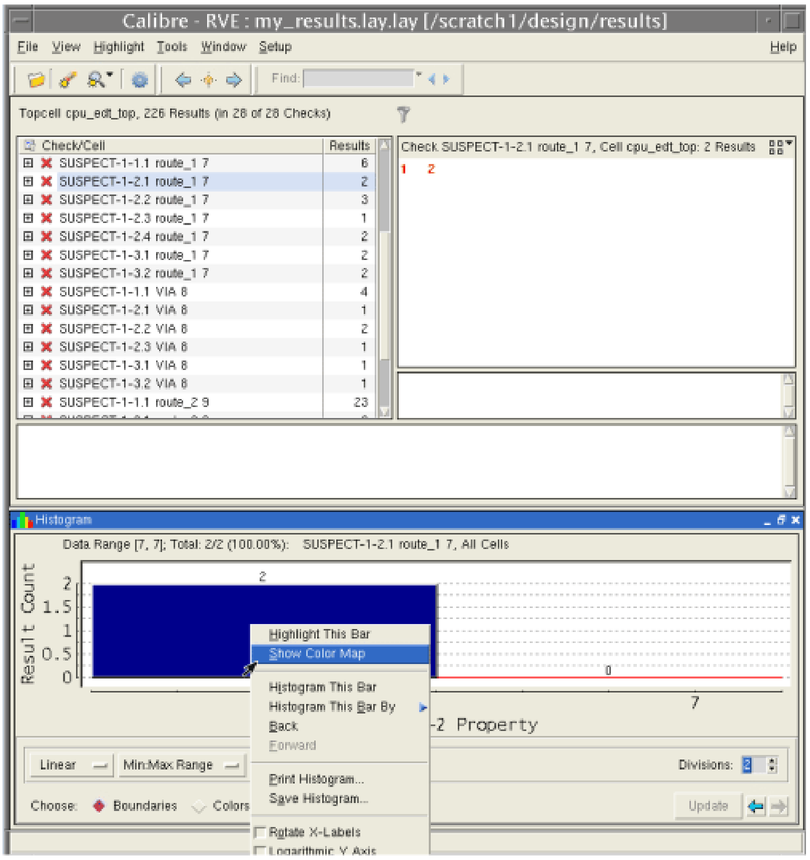
Data Range [7, 7]; Total: 2/2 (100.00%): SUSPECT-1-2.1 route_1 7, All Cells

Choose: Boundaries Colors Endpoints

If you want just the landmark and suspect, set Divisions to “2”. Otherwise, set Divisions to the number of layers X 2 to see all the colors.

Click Update

9. In the **Histogram Pane**, move the cursor over the colored histogram area, click and hold the menu mouse button, and select **Show Colormap**.


Calibre - RVE : my_results.lay.lay [/scratch1/design/results]

Topcell cpu_edt_top, 226 Results (in 28 of 28 Checks)

Check/Cell	Results
SUSPECT-1-1.1 route_1 7	6
SUSPECT-1-2.1 route_1 7	2
SUSPECT-1-2.2 route_1 7	3
SUSPECT-1-2.3 route_1 7	1
SUSPECT-1-2.4 route_1 7	2
SUSPECT-1-3.1 route_1 7	2
SUSPECT-1-3.2 route_1 7	2
SUSPECT-1-1.1 VIA 8	4
SUSPECT-1-2.1 VIA 8	1
SUSPECT-1-2.2 VIA 8	2
SUSPECT-1-2.3 VIA 8	1
SUSPECT-1-3.1 VIA 8	1
SUSPECT-1-3.2 VIA 8	1
SUSPECT-1-1.1 route_2 9	23

Check SUSPECT-1-2.1 route_1 7, Cell cpu_edt_top: 2 Results

Data Range [7, 7]; Total: 2/2 (100.00%): SUSPECT-1-2.1 route_1 7, All Cells

Results

Selecting **Show Colormap** sends the result to Calibre DESIGNrev.

Viewing Results in Calibre 2010.1 or Newer Software

Use the following procedure to view results in Calibre 2010.1 or newer software.

Prerequisites

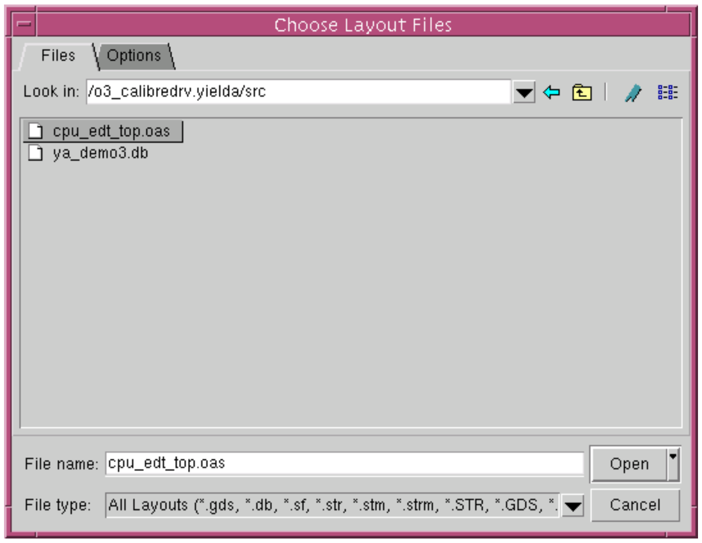
- A layout marker file. You cannot use the diagnosis report. Consult "Guidelines for Viewing the Diagnosis Results in Calibre DESIGNrev" before you begin.

Procedure

1. Invoke Calibre DESIGNrev from a UNIX/Linux shell using the following syntax:

```
calibredrv
```

2. Select **File > open_layout Files**, select the layout, and click **Open**.



Look in: /o3_calibredrv.yielda/src

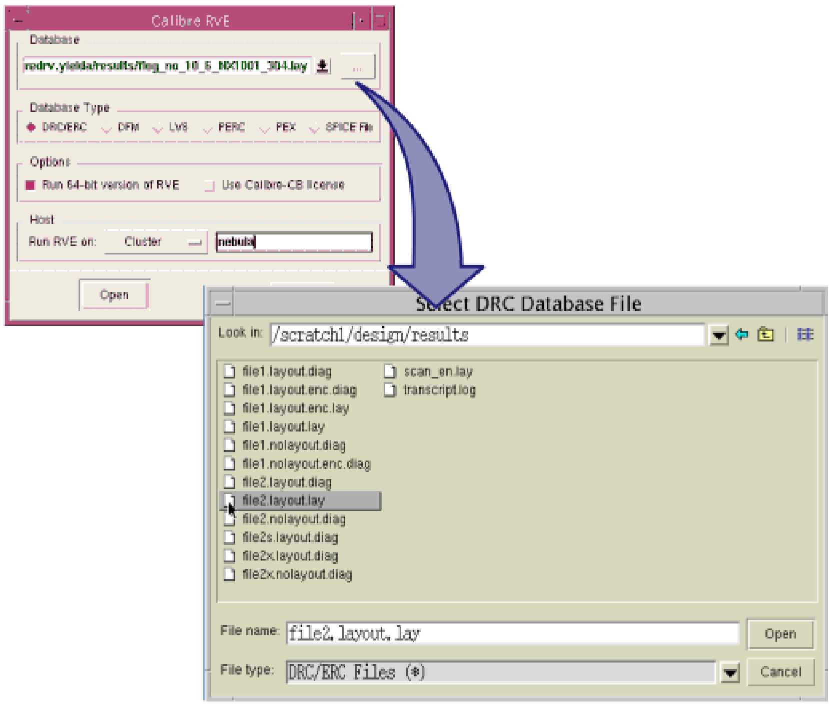
cpu_edt_top.oas
ya_demo3.db

File name: cpu_edt_top.oas
File type: All Layouts (*.gds, *.db, *.sf, *.str, *.stm, *.strm, *.STR, *.GDS, *.

You must open the same layout and coordinate space as the LEF/DEF files. In some cases, you must create a new GDSII or OASIS® layout using the **File > Open Database** option in Calibre DESIGNrev.

3. Select **Verification > Start RVE**.

4. Navigate to and select the Tessent Diagnosis layout marker file. Click **Open**.



Database: rcdrv.yielda/results/flog_no_10_6_NX1001_304.lay

Database Type: [✓ DRC/ERC]

[✓ Run 64-bit version of RVE] Use Calibre-CB license

Run RVE on: Cluster nebula

Look in: /scratch1/design/results

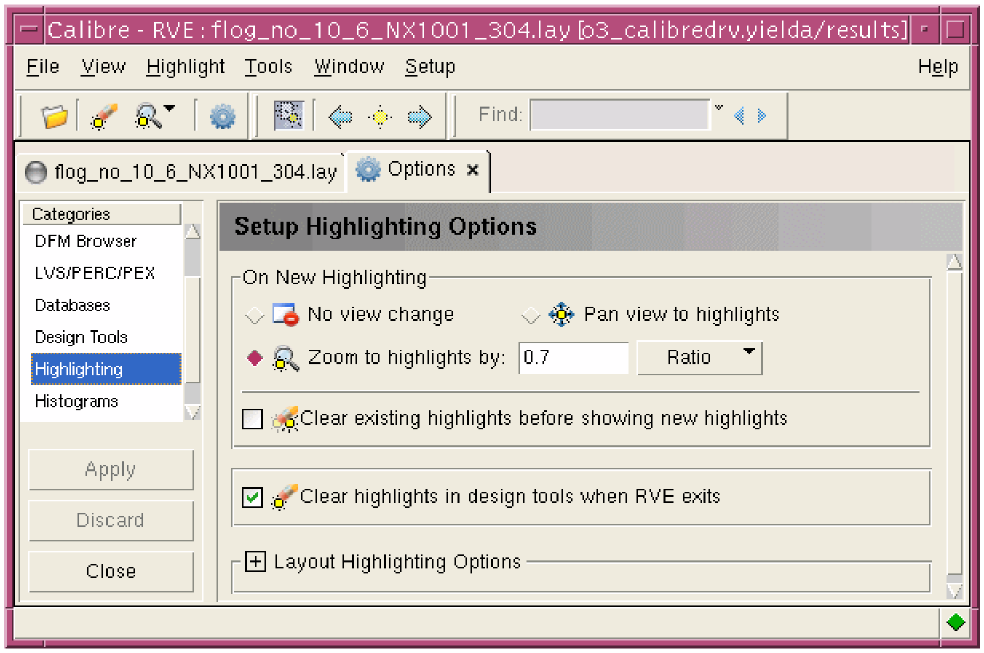
file1.layout.diag
file1.layout.enc.diag
file1.layout.enc.lay
file1.layout.lay
file1.nolayout.diag
file1.nolayout.enc.diag
file2.layout.diag
file2.layout.lay
file2.nolayout.diag
file2s.layout.diag
file2x.layout.diag
file2x.nolayout.diag
scan_en.lay
transcript.log

File name: file2.layout.lay
File type: DRC/ERC Files (*)

5. In RVE, choose **Setup > Options**.

On the RVE **Options** tab, choose the **Highlighting** category, enable the **Zoom to highlights by** option, and set the zoom value to 0.7.

Unselect **Clear Existing Highlights Before ShowingNew Highlights**.


Calibre - RVE : flog_no_10_6_NX1001_304.lay [o3_calibredrv.yielda/results]
File View Highlight Tools Window Setup Help
Find:
flog_no_10_6_NX1001_304.lay
Options
Categories
DFM Browser
LVS/PERC/PEX
Databases
Design Tools
Highlighting
Histograms
Apply
Discard
Close

Setup Highlighting Options

On New Highlighting
- No view change
- Pan view to highlights
- [✓ Zoom to highlights by: 0.7 Ratio]

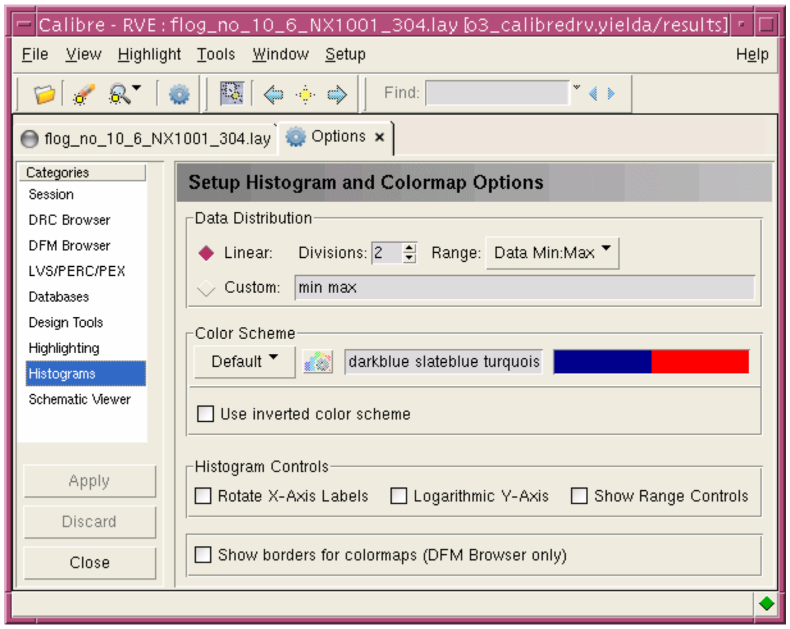
☐ Clear existing highlights before showing new highlights

☑ Clear highlights in design tools when RVE exits

⊞ Layout Highlighting Options

6. In the Histogram Pane, select **Setup > Options**. On the RVE Options tab, choose the **Histograms** category, and set the Data Distribution Divisions to "2".

   Click **Apply**.


Calibre - RVE : flog_no_10_6_NX1001_304.lay [o3_calibredrv.yielda/results]
File View Highlight Tools Window Setup Help
Find:
flog_no_10_6_NX1001_304.lay
Options
Categories
Session
DRC Browser
DFM Browser
LVS/PERC/PEX
Databases
Design Tools
Highlighting
Histograms
Schematic Viewer
Apply
Discard
Close

Setup Histogram and Colormap Options

Data Distribution
- [✓ Linear:] Divisions: 2 Range: Data Min:Max
- Custom: min max

Color Scheme
Default darkblue slateblue turquois

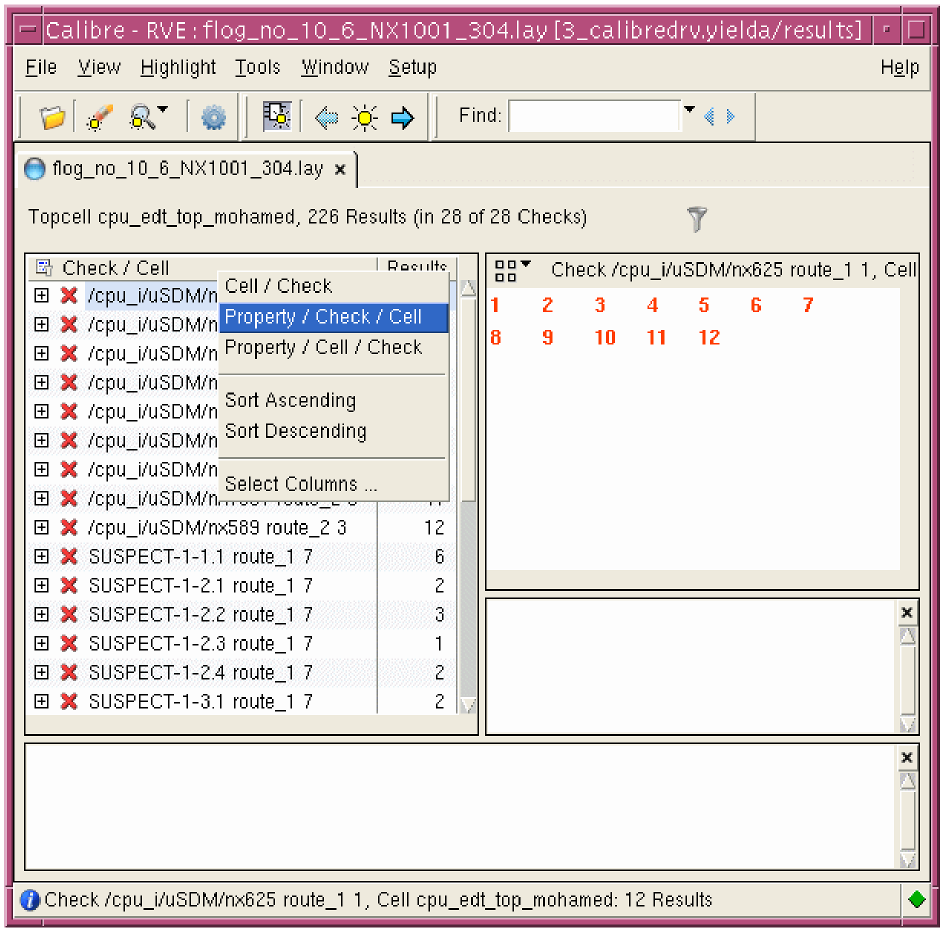
☐ Use inverted color scheme

Histogram Controls
☐ Rotate X-Axis Labels ☐ Logarithmic Y-Axis ☐ Show Range Controls

☐ Show borders for colormaps (DFM Browser only)

Click the "**x**" on the **Options** tab to close the tab.

7. In the left-hand pane, move the mouse cursor over the **Check / Cell** column heading, and click the mouse menu button, and choose the **Property / Check / Cell** option.


Calibre - RVE : flog_no_10_6_NX1001_304.lay [3_calibredrv.yielda/results]
File View Highlight Tools Window Setup Help
Find:
flog_no_10_6_NX1001_304.lay
Check / Cell
Results
Cell / Check
Property / Check / Cell
Property / Cell / Check
Sort Ascending
Sort Descending
Select Columns ...
Check /cpu_i/uSDM/nx625 route_1 1, Cell

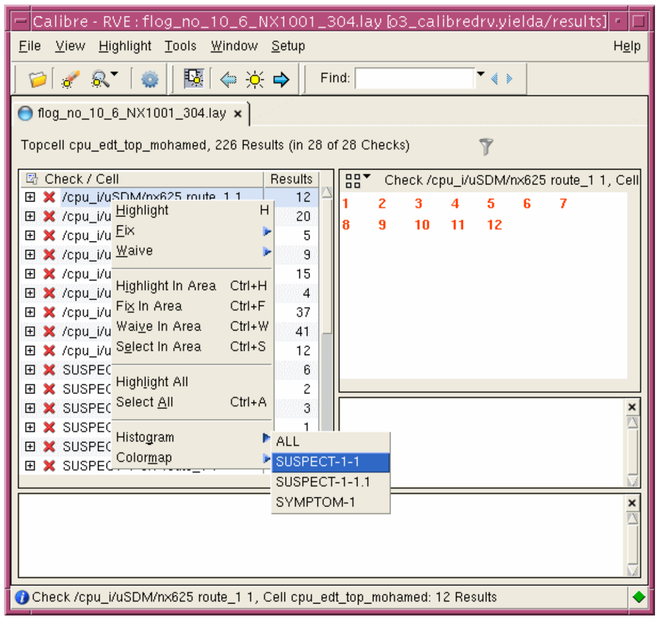
Topcell cpu_edt_top_mohamed, 226 Results (in 28 of 28 Checks)

Check / Cell	Results
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/n	
/cpu_i/uSDM/nx589 route_2 3	12
SUSPECT-1-1.1 route_1 7	6
SUSPECT-1-2.1 route_1 7	2
SUSPECT-1-2.2 route_1 7	3
SUSPECT-1-2.3 route_1 7	1
SUSPECT-1-2.4 route_1 7	2
SUSPECT-1-3.1 route_1 7	2

1 2 3 4 5 6 7
8 9 10 11 12

Check /cpu_i/uSDM/nx625 route_1 1, Cell cpu_edt_top_mohamed: 12 Results

8. In the left-hand pane, click a suspect of interest, click the mouse menu button, and choose **Histogram** > **suspect**.



Topcell cpu_edt_top_mohamed, 226 Results (in 28 of 28 Checks)

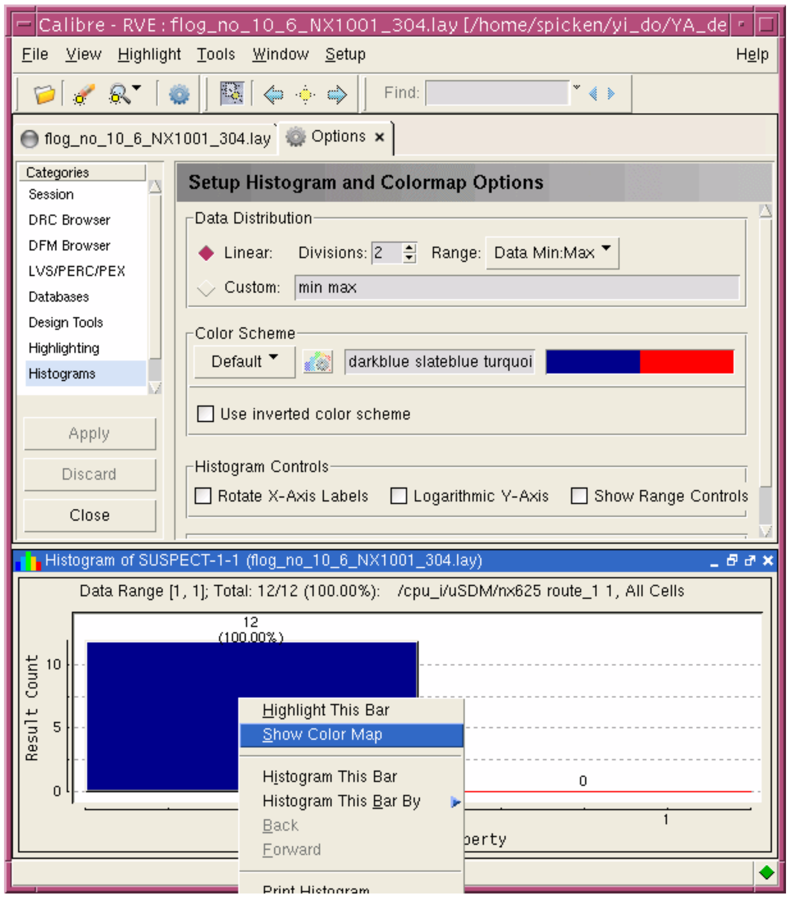
Check / Cell	Results
/cpu_i/uSDM/nx625 route_1 1	12
/cpu_i/u	20
/cpu_i/u	5
/cpu_i/u	9
/cpu_i/u	15
/cpu_i/u	4
/cpu_i/u	37
/cpu_i/u	41
/cpu_i/u	12
SUSPEC	6
SUSPEC	2
SUSPEC	3
SUSPEC	1
SUSPEC	
SUSPEC	

Check /cpu_i/uSDM/nx625 route_1 1, Cell

1 2 3 4 5 6 7
8 9 10 11 12

Check /cpu_i/uSDM/nx625 route_1 1, Cell cpu_edt_top_mohamed: 12 Results

9. In the **Histogram Pane**, move the cursor over the colored histogram area, click and hold the menu mouse button, and select **Show Colormap**.


Calibre - RVE : flog_no_10_6_NX1001_304.lay [/home/spicken/yi_do/YA_de
File View Highlight Tools Window Setup Help
Find:
flog_no_10_6_NX1001_304.lay
Options
Categories
Session
DRC Browser
DFM Browser
LVS/PERC/PEX
Databases
Design Tools
Highlighting
Histograms
Apply
Discard
Close
Setup Histogram and Colormap Options
Data Distribution
Linear: Divisions: 2 Range: Data Min:Max
Custom: min max
Color Scheme
Default
darkblue slateblue turquoi
Use inverted color scheme
Histogram Controls
Rotate X-Axis Labels
Logarithmic Y-Axis
Show Range Controls
Histogram of SUSPECT-1-1 (flog_no_10_6_NX1001_304.lay)
Data Range [1, 1]; Total: 12/12 (100.00%): /cpu_i/uSDM/nx625 route_1 1, All Cells
Result Count
12
(100.00%)
10
5
0
0
1
perty
Highlight This Bar
Show Color Map
Histogram This Bar
Histogram This Bar By
Back
Forward
Print Histogram

Results

Clicking **Show Colormap** for this suspect in Calibre RVE displays the suspect in Calibre DESIGNrev.

]]>

Layout-Aware Diagnosis Reporting

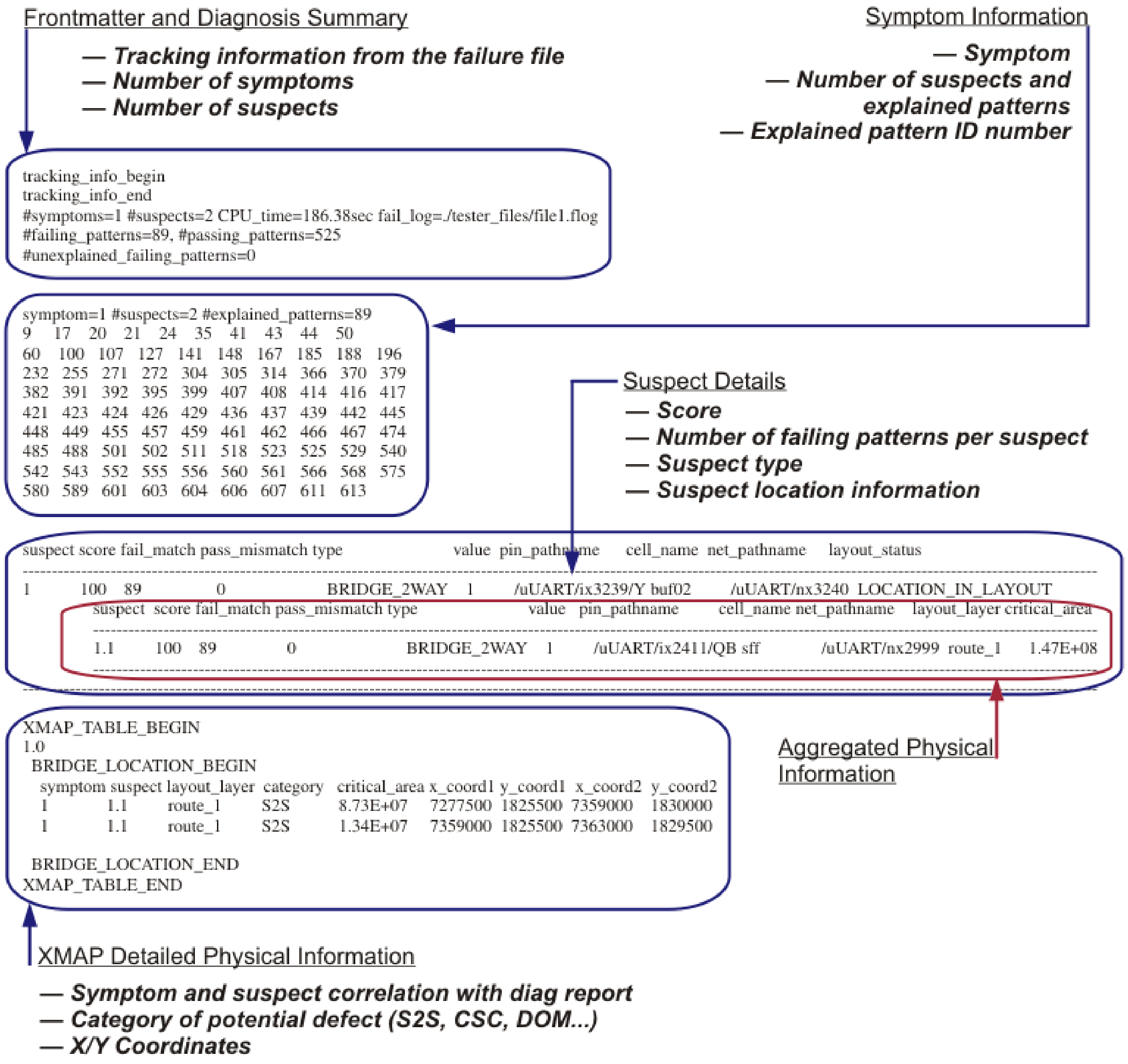
The diagnosis report for layout-out aware diagnosis displays additional, layout-related data. This enhanced reporting is activated automatically when you open the layout in Tessent Diagnosis. Tessent Diagnosis provides one reporting schema for cell, open, and bridges.

The Layout-Aware Diagnosis Report

As compared to the regular diagnosis report, the layout-aware diagnosis report enhances the suspect details section and adds a new section. It follows the principle to add only as much layout-related information into the suspect section to enable the failure analysis engineer to gain an overview of the layout-related defects. In this section, only the combined critical area of all potential defect locations per layer and the name of the layer are listed right after each suspect.

Figure 3-18 illustrates the contents of a diagnosis report showing a typical layout-aware diagnosis report. In the figure, the report shows a 2-way bridge, with both nets present in the layout (note the "LOCATION_IN_LAYOUT" comment). Right after both nets of this 2-way bridge are listed, an indented table reports the combined critical area of all locations in the two listed layers, where these two nets might bridge.

Figure 3-18. Example Layout-Aware Diagnosis Report


Frontmatter and Diagnosis Summary
— Tracking information from the failure file
— Number of symptoms
— Number of suspects
Symptom Information
— Symptom
— Number of suspects and explained patterns
— Explained pattern ID number
Suspect Details
— Score
— Number of failing patterns per suspect
— Suspect type
— Suspect location information
Aggregated Physical Information
XMAP Detailed Physical Information
— Symptom and suspect correlation with diag report
— Category of potential defect (S2S, CSC, DOM...)
— X/Y Coordinates

```
tracking_info_begin
tracking_info_end
#symptoms=1 #suspects=2 CPU_time=186.38sec fail_log=./tester_files/file1.flog
#failing_patterns=89, #passing_patterns=525
#unexplained_failing_patterns=0
```

```
symptom=1 #suspects=2 #explained_patterns=89
9    17   20   21   24   35   41   43   44   50
60   100  107  127  141  148  167  185  188  196
232  255  271  272  304  305  314  366  370  379
382  391  392  395  399  407  408  414  416  417
421  423  424  426  429  436  437  439  442  445
448  449  455  457  459  461  462  466  467  474
485  488  501  502  511  518  523  525  529  540
542  543  552  555  556  560  561  566  568  575
580  589  601  603  604  606  607  611  613
```

```
suspect score fail_match pass_mismatch type          value  pin_pathname      cell_name  net_pathname    layout_status
------------------------------------------------------------------------------------------------------------------------
1       100   89         0             BRIDGE_2WAY   1      /uUART/ix3239/Y   buf02      /uUART/nx3240   LOCATION_IN_LAYOUT
      suspect  score fail_match pass_mismatch type          value  pin_pathname      cell_name net_pathname   layout_layer critical_area
      -----------------------------------------------------------------------------------------------------------------------------------
      1.1      100   89         0             BRIDGE_2WAY   1      /uUART/ix2411/QB  sff       /uUART/nx2999  route_1      1.47E+08
      -----------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
```

```
XMAP_TABLE_BEGIN
1.0
 BRIDGE_LOCATION_BEGIN
  symptom suspect layout_layer  category  critical_area x_coord1 y_coord1 x_coord2 y_coord2
  1       1.1     route_1       S2S       8.73E+07      7277500  1825500  7359000  1830000
  1       1.1     route_1       S2S       1.34E+07      7359000  1825500  7363000  1829500

 BRIDGE_LOCATION_END
XMAP_TABLE_END
```

Via Naming Convention. For two DEF files with local and global via names as follows:

- DEF design "top": local name, viax, and global name, via1.
- DEF design "core": local name, viax, and global name, via2.

The reported names are:

```
top.viax                // design name "top" appended with local via name
via1                    // global LEF via name
core.viax               // design name "core" appended with local via name
via2                    // global LEF via name
```

Layout Status Column. 273

Layout Status Column

The layout status column in the layout-aware diagnosis report provides the status of input pins, nets, and net names for suspects.

Table 3-4 provides the descriptions of the layout_status column of the report.

Table 3-4. layout_status Column Descriptions

layout_status	Description
INPUT_PIN_FAULT	Only used for STUCK/INDETERMINATE suspects that are on an input pin of a gate/library cell. The tool is unable to assign a physical meaning to the suspect. These are logical failing suspects similar to STUCK/INDETERMINATE on the input pin of the gate, the physical defect condition is unknown.
LOCATION_IN_LAYOUT	The suspect's net or nets are available in the layout.
MISSING_FROM_LAYOUT	Only used for missing design net names in the layout and vice versa. No layout-aware features, however, are available for these nets.

Additional detail about each individual potential defect location is shown in the "BRIDGE_LOCATION", "CELL_LOCATION", and "OPEN_LOCATION" tables of the cross map table towards the end of the report.

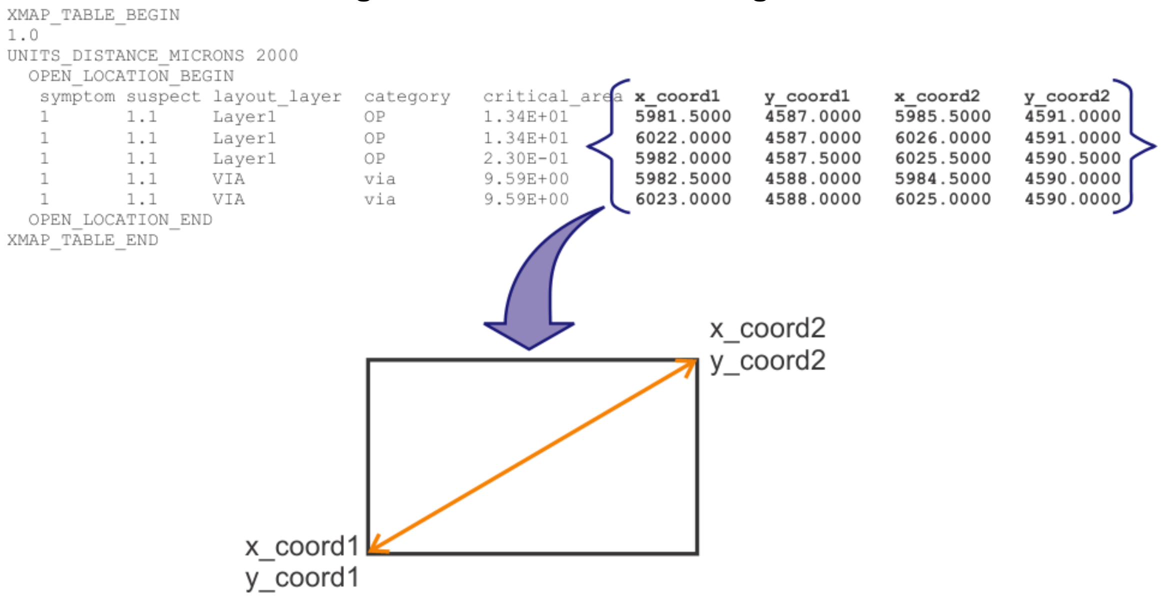
Defect Location Information

The layout-aware diagnosis report contains physical defect location information of each suspect contained in the report.

The diagnosis report lists both the bounding box coordinates (x_coord1, y_coord1, x_coord2, and y_coord2) and the enclosing circle in the XMAP_TABLE section, as follows:

- Bounding box — A rectangular area on a given layer in the layout containing a particular physical defect. Figure 3-19 shows an example bounding box.

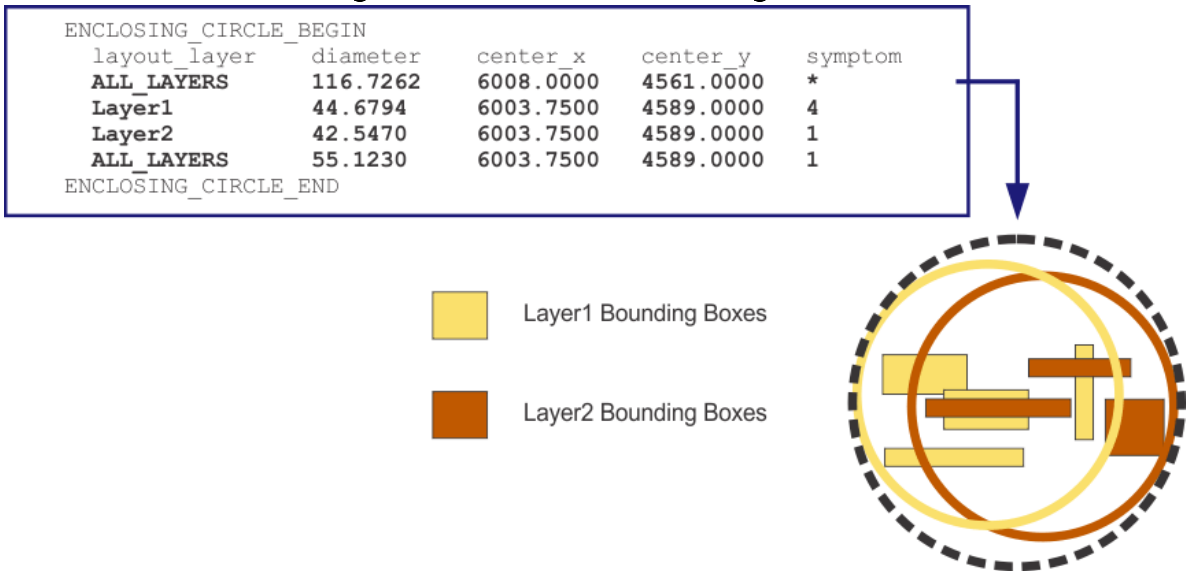
Figure 3-19. Defect Bounding Box



```
XMAP_TABLE_BEGIN
1.0
UNITS_DISTANCE_MICRONS 2000
  OPEN_LOCATION_BEGIN
   symptom suspect layout_layer category   critical_area x_coord1  y_coord1  x_coord2  y_coord2
   1       1.1     Layer1       OP         1.34E+01      5981.5000 4587.0000 5985.5000 4591.0000
   1       1.1     Layer1       OP         1.34E+01      6022.0000 4587.0000 6026.0000 4591.0000
   1       1.1     Layer1       OP         2.30E-01      5982.0000 4587.5000 6025.5000 4590.5000
   1       1.1     VIA          via        9.59E+00      5982.5000 4588.0000 5984.5000 4590.0000
   1       1.1     VIA          via        9.59E+00      6023.0000 4588.0000 6025.0000 4590.0000
  OPEN_LOCATION_END
XMAP_TABLE_END
```

- Enclosing circle — A diameter around one or more bounding boxes. Figure 3-20 shows an example enclosing circle. The ENCLOSING_CIRCLE section of the diagnosis report shows the per symptom minimal enclosing circles for all of the defect bounding boxes associated with each layout layer. The asterisk in the symptom column indicates all symptoms. The special name ALL_LAYERS indicates a circle that encompasses all the bounding boxes on all layers.

Figure 3-20. Defect Enclosing Circle



```
ENCLOSING_CIRCLE_BEGIN
  layout_layer    diameter    center_x    center_y    symptom
  ALL_LAYERS      116.7262    6008.0000   4561.0000   *
  Layer1          44.6794     6003.7500   4589.0000   4
  Layer2          42.5470     6003.7500   4589.0000   1
  ALL_LAYERS      55.1230     6003.7500   4589.0000   1
ENCLOSING_CIRCLE_END
```

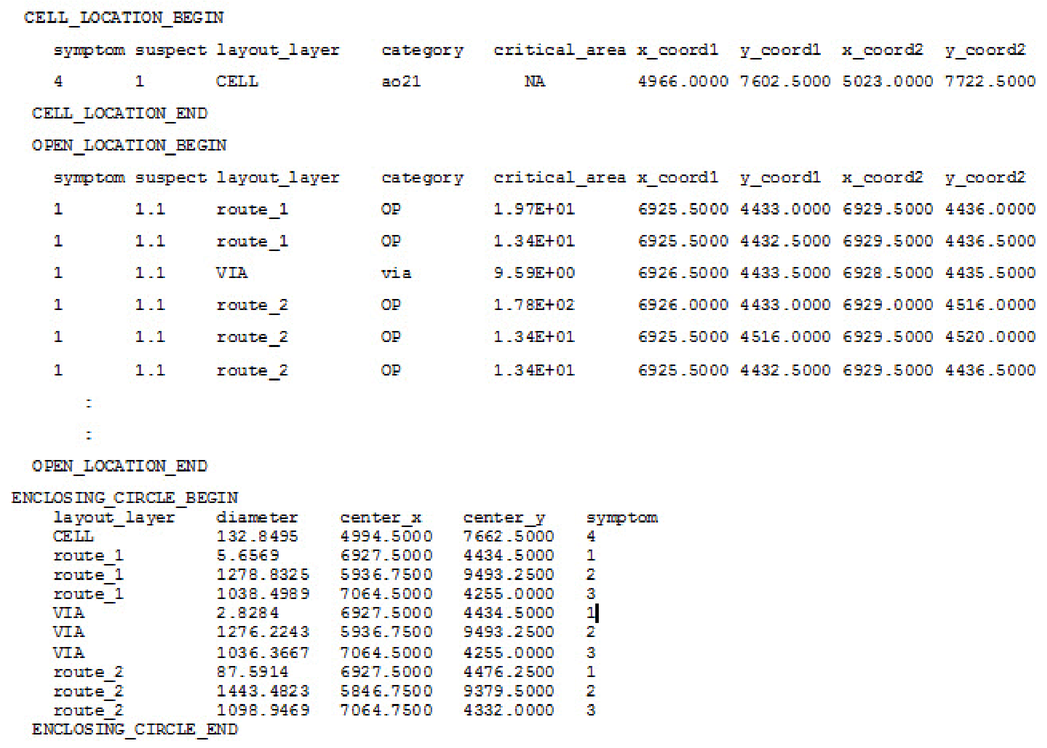
The ENCLOSING_CIRCLE table consists for the following columns:

Table 3-5. ENCLOSING_CIRCLE Table

Heading	Description
layout_layer	Layout layer name that the circle encompasses. The special name ALL_LAYERS indicates a circle that encompasses all the bounding boxes on all layers.
diameter	Diameter of the enclosing circle.
center_x	X coordinate of the circle.
center_y	Y coordinate of the circle.
symptom	Symptom number for which the enclosing circle was calculated. The special symbol "*" indicates a tally across all symptoms.

Figure 3-21 shows the two sections of the diagnosis report that itemize the layout defect bounding boxes.

Figure 3-21. Diagnosis Report: Layout Defect Bounding Boxes



```
CELL_LOCATION_BEGIN
   symptom suspect layout_layer    category   critical_area x_coord1  y_coord1  x_coord2  y_coord2
   4       1       CELL            ao21          NA         4966.0000 7602.5000 5023.0000 7722.5000
 CELL_LOCATION_END
 OPEN_LOCATION_BEGIN
   symptom suspect layout_layer    category   critical_area x_coord1  y_coord1  x_coord2  y_coord2
   1       1.1     route_1         OP         1.97E+01      6925.5000 4433.0000 6929.5000 4436.0000
   1       1.1     route_1         OP         1.34E+01      6925.5000 4432.5000 6929.5000 4436.5000
   1       1.1     VIA             via        9.59E+00      6926.5000 4433.5000 6928.5000 4435.5000
   1       1.1     route_2         OP         1.78E+02      6926.0000 4433.0000 6929.0000 4516.0000
   1       1.1     route_2         OP         1.34E+01      6925.5000 4516.0000 6929.5000 4520.0000
   1       1.1     route_2         OP         1.34E+01      6925.5000 4432.5000 6929.5000 4436.5000
       :
       :
 OPEN_LOCATION_END
ENCLOSING_CIRCLE_BEGIN
   layout_layer    diameter    center_x    center_y    symptom
   CELL            132.8495    4994.5000   7662.5000   4
   route_1         5.6569      6927.5000   4434.5000   1
   route_1         1278.8325   5936.7500   9493.2500   2
   route_1         1038.4989   7064.5000   4255.0000   3
   VIA             2.8284      6927.5000   4434.5000   1
   VIA             1276.2243   5936.7500   9493.2500   2
   VIA             1036.3667   7064.5000   4255.0000   3
   route_2         87.5914     6927.5000   4476.2500   1
   route_2         1443.4823   5846.7500   9379.5000   2
   route_2         1098.9469   7064.7500   4332.0000   3
 ENCLOSING_CIRCLE_END
```

The single symptom 4 CELL layer has a defect bounding box of:

```
4   1   CELL   ao21   NA   4966.0000   7602.5000  5023.0000  7722.5000
```

The calculated enclosing circle associated with CELL is:

```
CELL            132.8495    4994.5000    7662.5000    4
```

There are two symptom 1 route_1 open features with defect bounding boxes:

```
1   1.1   route_1   OP1.34E+01   5981.5000   4587.0000   5985.5000   4591.0000
1   1.1   route_1   OP1.34E+01   6022.0000   4587.0000   6026.0000   4591.0000
```

Together, they are contained by the enclosing circle:

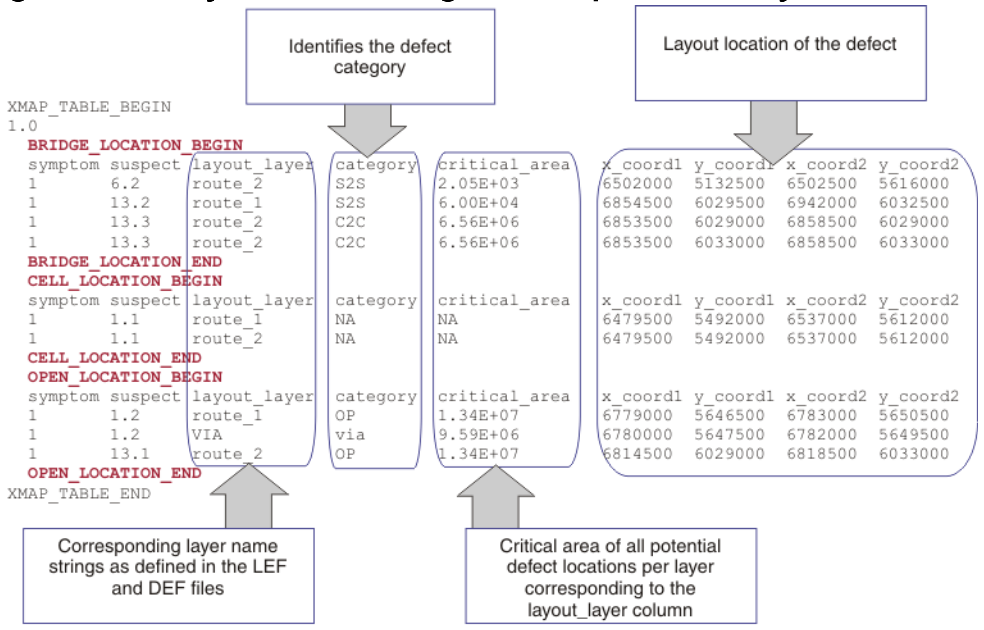
```
route_1          44.6794     6003.7500    4589.0000    1
```

The XMAP Table

The entries in the XMAP table provide detailed defect information. The tool indexes the entries by the symptom and suspect numbers so you can cross-reference back to the pathnames and cell names reported in the chain and logic diagnosis sections.

Figure 3-22 shows an example XMAP table containing layout details reported for each defect location. The tool writes the XMAP table to the diagnosis report after it writes all the symptoms and suspects of the chain and logic diagnosis sections.

Figure 3-22. Layout-Aware Diagnosis Report With Layout Locations


Identifies the defect category
Layout location of the defect
Corresponding layer name strings as defined in the LEF and DEF files
Critical area of all potential defect locations per layer corresponding to the layout_layer column

```
XMAP_TABLE_BEGIN
1.0
  BRIDGE_LOCATION_BEGIN
  symptom suspect layout_layer  category  critical_area  x_coord1  y_coord1  x_coord2  y_coord2
  1       6.2     route_2       S2S       2.05E+03       6502000   5132500   6502500   5616000
  1       13.2    route_1       S2S       6.00E+04       6854500   6029500   6942000   6032500
  1       13.3    route_2       C2C       6.56E+06       6853500   6029000   6858500   6029000
  1       13.3    route_2       C2C       6.56E+06       6853500   6033000   6858500   6033000
  BRIDGE_LOCATION_END
  CELL_LOCATION_BEGIN
  symptom suspect layout_layer  category  critical_area  x_coord1  y_coord1  x_coord2  y_coord2
  1       1.1     route_1       NA        NA             6479500   5492000   6537000   5612000
  1       1.1     route_2       NA        NA             6479500   5492000   6537000   5612000
  CELL_LOCATION_END
  OPEN_LOCATION_BEGIN
  symptom suspect layout_layer  category  critical_area  x_coord1  y_coord1  x_coord2  y_coord2
  1       1.2     route_1       OP        1.34E+07       6779000   5646500   6783000   5650500
  1       1.2     VIA           via       9.59E+06       6780000   5647500   6782000   5649500
  1       13.1    route_2       OP        1.34E+07       6814500   6029000   6818500   6033000
  OPEN_LOCATION_END
XMAP_TABLE_END
```

In the report's category column, you can find additional defect-related location information.

- BRIDGE_LOCATION — Identifies the following categories of bridges:
    - S2S — Side-to-side bridges. For a S2S bridge, the coordinates mark the lower left corner of the bridge location.
    - C2C — Corner-to-corner bridges. For C2C bridges, the corner with the smaller y-value is chosen for reporting, and if these y-values are equal, the one with the smaller x-value is selected.
    - DOM — Dominant bridges. These are bridge faults where only one of the two nets, the 'victim', is observable at an observation point (scan flop or primary output). The other net, the 'aggressor', never carries a difference to any observation point, and thus remains unknown to a strictly logical-based diagnosis too. The reported suspect, however, is only the victim net, specifically, the observable net. In the indented table, each remaining layout-derived potential aggressor candidate is listed. With its net name also the combined critical area of all locations in all listed layers, where these two nets, the victim and the aggressor, might bridge is reported. DOM entries are in the indented suspect sub-table below the primary suspect.
- CELL_LOCATION — Always NA.
- OPEN_LOCATION — Identifies the following categories of opens:
    - OP — Opens on metal layers.
    - *via_type_name* — For opens on via layers, Tessent Diagnosis reports the type name of the via instance at this location. This via type name was defined in the LEF file.
- CELL_DEFECT_LOCATION — Identifies the following category of cell-aware defects.
    - *defect_location_string* — Identifies the defect ID, defect location, and resistor value. For example: D37/6:73--CIN:15/1.0. Refer to "Cell-Aware Diagnosis" for details.

Pin Location Table

You can include an optional table of pin coordinates in the report. This PIN_LOCATION table is not present by default. Use the following command to add it.

```
set_diagnosis_options -report_pin_locations ON
```

You can also use the diagnosis server version of the command.

```
set_diagnosis_options Monitor-ID -report_pin_locations ON
```

The resulting PIN_LOCATION table is similar to the following:

```
PIN_LOCATION_BEGIN
symptom suspect layout_layer pin_location_x pin_location_y
1       1       metal1       7566.0400      2698.3600
1       2       metal1       7566.0400      2698.4350
2       1       metal1       7566.0400      2698.4350
2       2       metal2       7566.1200      2698.5310
2       2.1     metal2       7566.1250      2698.5300
PIN_LOCATION_END
```

Cell Defect Location Table

The CELL_DEFECT_LOCATION table provides a list of defect locations from the cell models. It begins with the keyword CELL_DEFECT_LOCATION_BEGIN and ends with the keyword CELL_DEFECT_LOCATION_END. This table has the following columns by default.

- **symptom** — The symptom number.
- **suspect** — The suspect number of the corresponding symptom.
- **layout_layer** — Layer name from the LEF and DEF files.
- **category** — Defect type and the defect ID.
- **critical_area** — Critical area of all potential defect locations per layer corresponding to the layout_layer column.
- **x_coord1** — The lower left x-coordinate of the defect bounding box.
- **y_coord1** — The lower left y-coordinate of the defect bounding box.
- **x_coord2** — The upper right x-coordinate of the defect bounding box.
- **y_coord2** — The upper right y-coordinate of the defect bounding box.

An example dataset of this table is similar to the following:

```
1  1.3  TRAN   TON.D23  5.73E-03  107.6850  120.5205  107.7350  120.6550
1  1.4  PS     OP.D27   5.02E-03  107.6545  121.3075  107.7350  121.3450
1  1.4  PS     OP.D27   8.50E-03  107.6850  121.3450  107.7350  121.5900
1  1.4  PS     OP.D27   4.62E-03  107.6850  121.5450  107.7350  121.6355
1  1.5  CO     OP.D28   2.77E-03  107.6600  121.2750  107.7250  121.3405
1  1.5  M1     OP.D28   4.78E-03  107.6350  121.2400  107.6925  121.3800
1  1.6  TRAN   TOFF.D33 4.62E-03  107.6850  121.5450  107.7350  121.6355
1  1.7  PS-DI  BR.D19   5.33E-03  107.6625  121.5450  107.7070  121.6355
1  1.8  M1     BR.D5    5.26E-04  107.6500  120.5580  107.7700  120.6180
1  1.8  CO     BR.D5    1.01E-03  107.6500  120.5655  107.7690  120.6105
1  1.8  DI     BR.D5    5.57E-03  107.6840  120.5205  107.7350  120.6550
1  1.9  M1     BR.D6    3.85E-04  107.6195  121.3810  107.6650  121.5980
1  1.9  CO-PS  BR.D6    4.58E-03  107.6445  121.5660  107.6895  121.6310
1  1.9  CO-PS  BR.D6    2.63E-04  107.6295  121.3465  107.6745  121.5660
```

You can optionally add a column of UDFM layer names from the data that is available in the UDFM file. Use the "set_diagnosis_options -add_spice_layers_to_diagnosis_report on" command, which adds a column named spice_layer after the y_coord2 column. The results rely on the UDFM to SPICE translation data available in the UDFM file. If data is not available (such as a missing cell definition or an incomplete mapping) the tool prints "NA". An example spice_layer dataset is similar to the following:

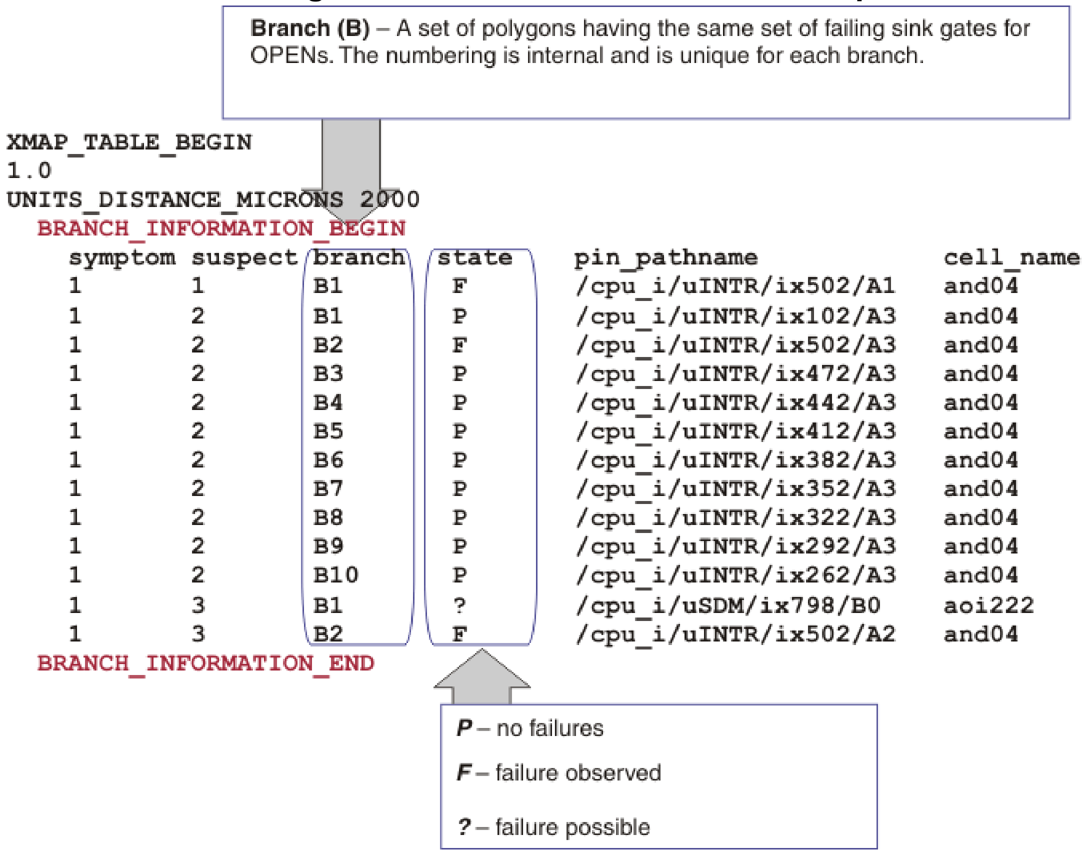
```
NA
{poly}
{poly}
{poly}
{contact}
{metal1}
NA
{poly}-{nsrcdrn,psrcdrn}
{metal1}
{contact}
{nsrcdrn,psrcdrn}
{metal1}
{contact}-{poly}
{contact}-{poly}
```

The Branch Information Table

The branch information table displays a set of polygons that have the same set of failing sink gates for OPEN defects. The numbering is internal and unique for each branch.

Figure 3-23 shows an example branch information table from a layout-aware diagnosis run. The tool produces this table only for OPEN defects.

Figure 3-23. Branch Information Example



Branch (B) – A set of polygons having the same set of failing sink gates for OPENs. The numbering is internal and is unique for each branch.

```
XMAP_TABLE_BEGIN
1.0
UNITS_DISTANCE_MICRONS 2000
  BRANCH_INFORMATION_BEGIN
    symptom suspect branch  state    pin_pathname           cell_name
    1       1       B1       F       /cpu_i/uINTR/ix502/A1  and04
    1       2       B1       P       /cpu_i/uINTR/ix102/A3  and04
    1       2       B2       F       /cpu_i/uINTR/ix502/A3  and04
    1       2       B3       P       /cpu_i/uINTR/ix472/A3  and04
    1       2       B4       P       /cpu_i/uINTR/ix442/A3  and04
    1       2       B5       P       /cpu_i/uINTR/ix412/A3  and04
    1       2       B6       P       /cpu_i/uINTR/ix382/A3  and04
    1       2       B7       P       /cpu_i/uINTR/ix352/A3  and04
    1       2       B8       P       /cpu_i/uINTR/ix322/A3  and04
    1       2       B9       P       /cpu_i/uINTR/ix292/A3  and04
    1       2       B10      P       /cpu_i/uINTR/ix262/A3  and04
    1       3       B1       ?       /cpu_i/uSDM/ix798/B0   aoi222
    1       3       B2       F       /cpu_i/uINTR/ix502/A2  and04
  BRANCH_INFORMATION_END
```

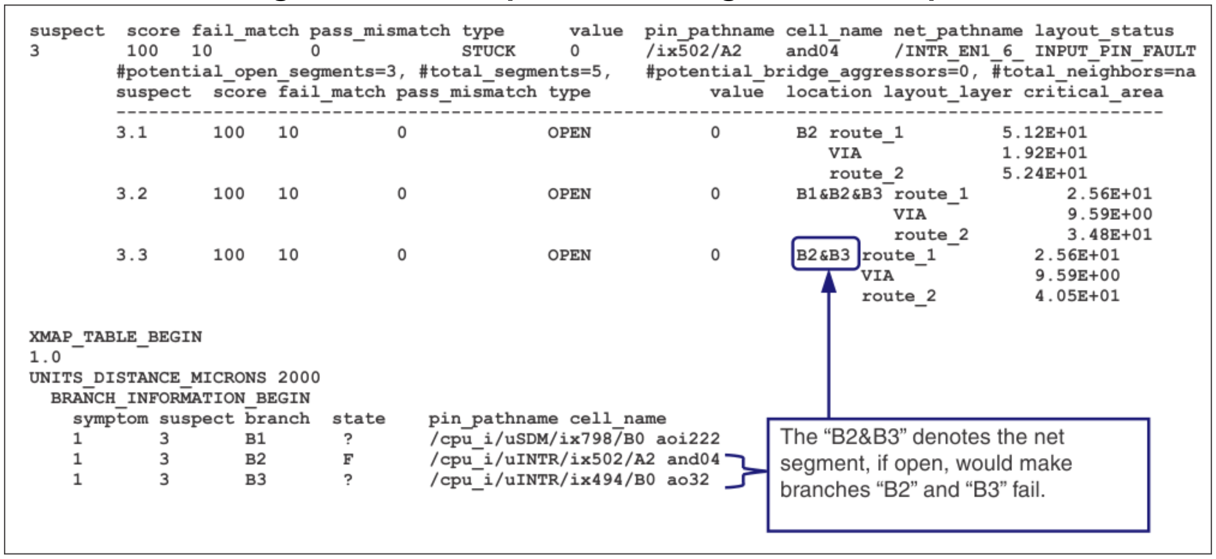
P – no failures

F – failure observed

? – failure possible

In Figure 3-24, the main report uses "B<*numbers*>" in the suspect detail section and corresponding location information in the BRANCH_INFORMATION section of the report.

Figure 3-24. Multiple Branch Segments Example



```
suspect  score fail_match pass_mismatch type      value  pin_pathname cell_name net_pathname layout_status
3         100   10          0            STUCK     0      /ix502/A2    and04     /INTR_EN1_6_ INPUT_PIN_FAULT
        #potential_open_segments=3, #total_segments=5,   #potential_bridge_aggressors=0, #total_neighbors=na
        suspect  score fail_match pass_mismatch type            value  location layout_layer critical_area
        ----------------------------------------------------------------------------------------------------
        3.1      100   10          0            OPEN            0      B2 route_1         5.12E+01
                                                                          VIA             1.92E+01
                                                                          route_2         5.24E+01
        3.2      100   10          0            OPEN            0      B1&B2&B3 route_1         2.56E+01
                                                                                VIA             9.59E+00
                                                                                route_2         3.48E+01
        3.3      100   10          0            OPEN            0      B2&B3 route_1         2.56E+01
                                                                             VIA             9.59E+00
                                                                             route_2         4.05E+01

XMAP_TABLE_BEGIN
1.0
UNITS_DISTANCE_MICRONS 2000
  BRANCH_INFORMATION_BEGIN
    symptom suspect branch  state    pin_pathname cell_name
    1       3       B1       ?       /cpu_i/uSDM/ix798/B0 aoi222
    1       3       B2       F       /cpu_i/uINTR/ix502/A2 and04
    1       3       B3       ?       /cpu_i/uINTR/ix494/B0 ao32
```

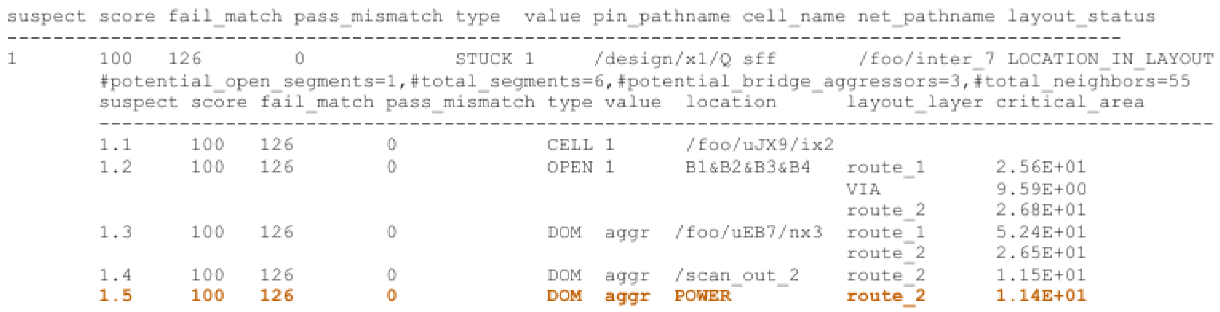
The "B2&B3" denotes the net segment, if open, would make branches "B2" and "B3" fail.

Power and Ground Bridge Reporting

For STUCK suspects, Tessent Diagnosis layout-aware diagnosis analyzes and reports potential bridging defects between the STUCK suspect net, and a power or ground line.

Figure 3-25 demonstrates a Tessent layout-aware diagnosis report containing a STUCK suspect potentially bridging to the power net.

Figure 3-25. Example Diagnosis Report With STUCK Suspect Bridging to Power Net



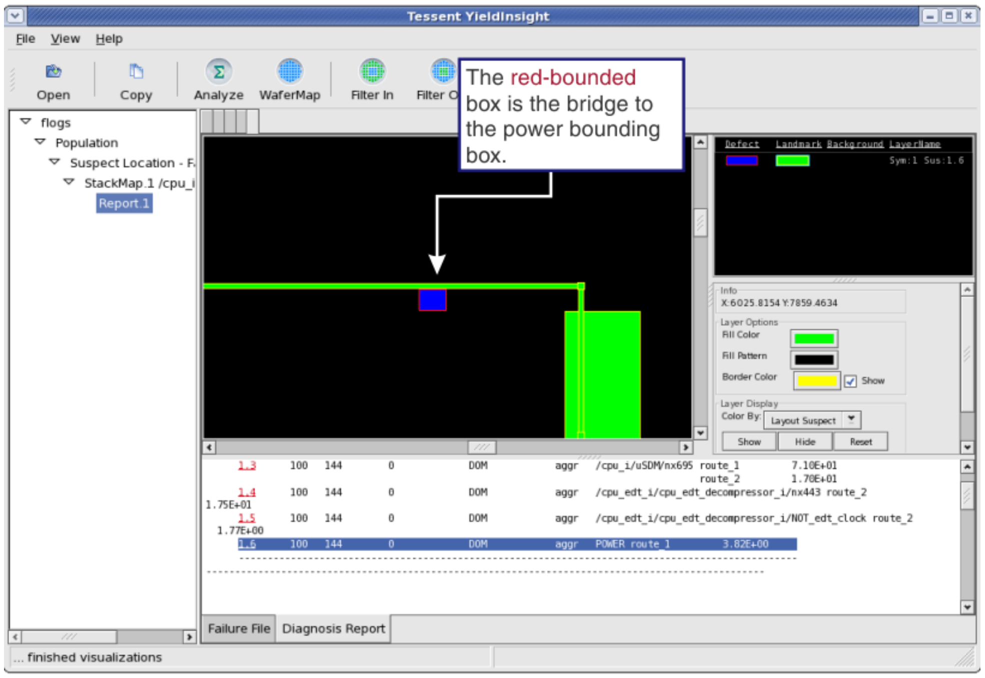
```
suspect score fail_match pass_mismatch type  value pin_pathname cell_name net_pathname layout_status
--------------------------------------------------------------------------------------------------
1       100   126         0             STUCK 1     /design/x1/Q sff       /foo/inter_7 LOCATION_IN_LAYOUT
        #potential_open_segments=1,#total_segments=6,#potential_bridge_aggressors=3,#total_neighbors=55
        suspect score fail_match pass_mismatch type value  location      layout_layer critical_area
        ---------------------------------------------------------------------------------------------
        1.1     100   126         0             CELL 1      /foo/uJX9/ix2
        1.2     100   126         0             OPEN 1      B1&B2&B3&B4   route_1      2.56E+01
                                                                          VIA          9.59E+00
                                                                          route_2      2.68E+01
        1.3     100   126         0             DOM   aggr /foo/uEB7/nx3  route_1      5.24E+01
                                                                          route_2      2.65E+01
        1.4     100   126         0             DOM   aggr /scan_out_2    route_2      1.15E+01
        1.5     100   126         0             DOM   aggr POWER          route_2      1.14E+01
```

For these potential power or ground bridging suspects, Tessent Diagnosis reports the suspect's type as DOM (see Table 3-25), and POWER or GROUND as the netname. The report also contains the secondary suspect's critical area and bounding box for the bridge—see "Defect Location Information."

You can turn off this analysis and reporting by setting -INCLUDE_BRIDGE_to_power switch to "OFF" for the set_diagnosis_options (scan diagnosis) or set_diagnosis_options (server) command.

You can also view these potential power or ground bridging suspects in Tessent YieldInsight as shown in Figure 3-26. See "Polygon Layout Viewer Pane" in the *Tessent YieldInsight User's Manual*.

Figure 3-26. Visualization of Potential Bridging Defect to a Power Line



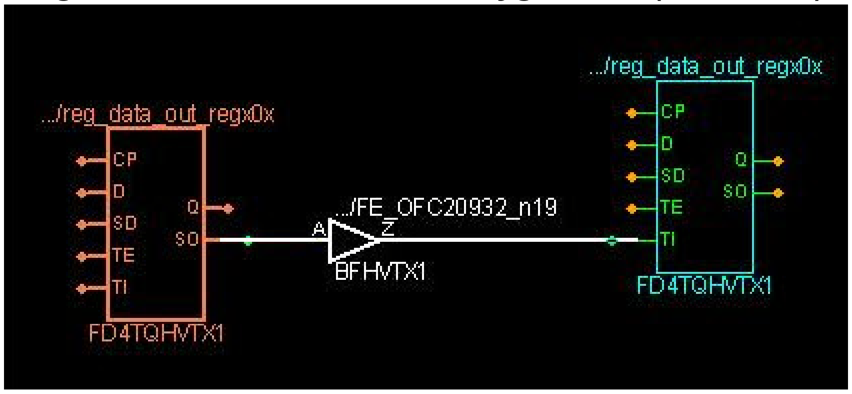
```
    1.3    100  144     0        DOM        aggr   /cpu_i/uSDM/nx695 route_1        7.10E+01
                                                                     route_2        1.70E+01
    1.4    100  144     0        DOM        aggr   /cpu_edt_i/cpu_edt_decompressor_i/nx443 route_2
1.75E+01
    1.5    100  144     0        DOM        aggr   /cpu_edt_i/cpu_edt_decompressor_i/NOT_edt_clock route_2
 1.77E+00
    1.6    100  144     0        DOM        aggr   POWER route_1        3.82E+00
    ----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
```

Inter-Scan Cell Polygon Reporting for Chain Diagnosis

The Tessent Diagnosis chain diagnosis report includes CELL and OPEN defect polygons for logic gates located between the scan cells, or *inter-scan cell polygons*. These polygons also appear in any associated layout marker files and CSV files.

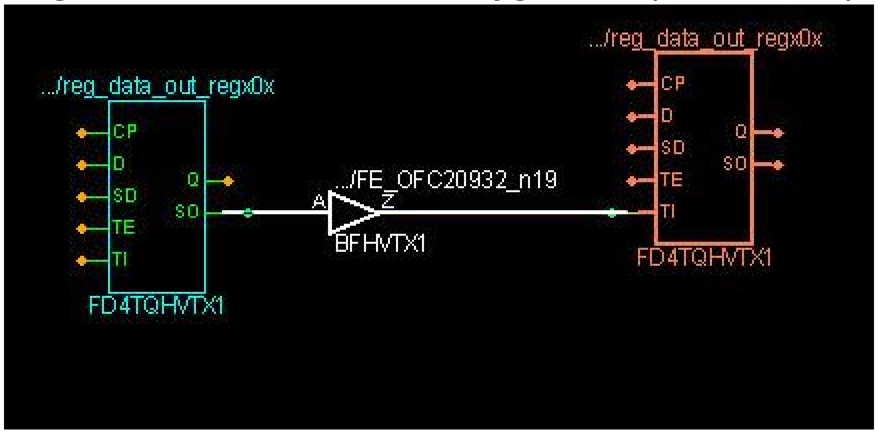
For example, Figure 3-27 shows a suspect cell with a fault type that includes (IN+CELL). The tool traces from the scan cell's fan-in to the previous scan cell. If it encounters any logic gates along the trace (such as the buffer shown below), it adds that logic's polygons to the CELL_LOCATION table of the diagnosis report. In addition, it includes the logic's ouput net polygon to the OPEN_LOCATION table.

Figure 3-27. Inter-Scan Cell Polygons for (IN + CELL)



If the fault type includes (CELL+OUT) such as shown in Figure 3-28, the tool traces the scan cell's fanout to the next scan cell and includes polygons for any logic gates found along the trace to the CELL_LOCATION and OPEN_LOCATION tables.

Figure 3-28. Inter-Scan Cell Polygons for (CELL + OUT)



If the fault type includes (IN+CELL+OUT), the tool traces both from the suspect scan cell's fan-in to the previous scan cell and the suspect scan cell's fanout to the next scan cell.

To report the inter-scan cell logic, use the report_scan_polygons command. For example, suppose your chain diagnosis report includes the following for suspect 16:

```
16    65  STUCK(CELL+OUT) 1
/top_dut/io/data_out/SO FFTX1 \       /top_dut/io/n18 333 chain78
```

This suspect is associated with chain 78 at scan cell 333. Fault type CELL+OUT means that the tool searches for logic gates from cell 333 to cell 334 and includes any logic gate's cell and net defect polygons in the diagnosis report.

To verify the inter-scan cell logic on the trace from scan cell 333 to scan cell 334 and report the net and gate polygons associated with the selected scan chain segment, specify the following command:

report_scan_polygons -chain chain78 -start 333 -stop 334

See report_scan_polygons for more information and for an example of a report.

Note

When you have a chip-mapped core-level LDB from hierarchical layout-aware diagnosis, prior to specifying the following command and option, you must specify the set_layout_core_instance command to set the current layout core instance for reporting purposes.

For hierarchical layout-aware diagnosis, the report_scan_polygons command generates reports with scanned net pathnames at the core level, by default. To generate reports with the scanned net pathnames for each instance of a given core reported at the chip level, specify:

```
report_scan_polygons -all_core_instances
```

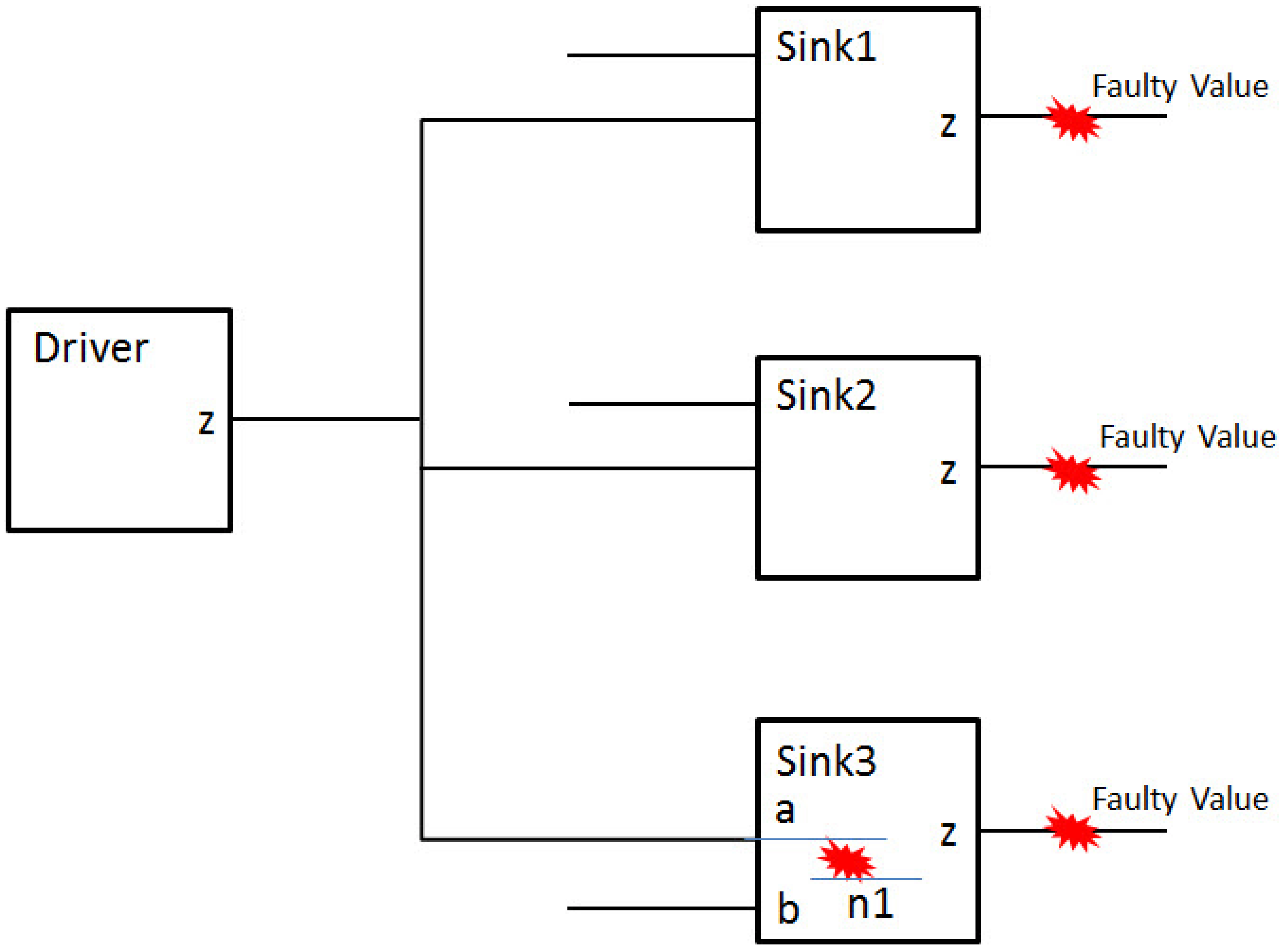
Refer to "Diagnosis for Hierarchical Designs" for information about performing hierarchical layout-aware diagnosis.

Cell Bridge Port Diagnosis Reporting

Cell bridge port defects can cause a propagating fault effect called *fault effect back propagation*. This fault effect occurs when a bridge port defect on one cell effects sibling cells on the same net.

In the following figure, Net1 has one driver cell and three sink cells. All sink cells for the same net are called sibling cells. Suppose that there is a bridge defect in cell Sink3 between its port a and an internal signal n1. Because all of Net1 may be affected by this bridge defect, the fault effect may propagate to not only the output port z of Sink3, but also to the output ports of its sibling cells, Sink1 and Sink2.

Figure 3-29. Cell Bridge Port Defect, Propagating



To diagnose and report fault effect back propagation issues arising from cell bridge port defects, specify the following command:

```
set_diagnosis_options -cell_port_bridge_analysis on
# For Tessent Diagnosis server:
set_diagnosis_options monitor -cell_port_bridge_analysis on
```

By default, the -cell_port_bridge_analysis option is set to “auto,” which means that it is off for layout-aware diagnosis and on for cell-aware diagnosis. Cell port bridge diagnosis requires an LDB. For cell-aware diagnosis, you should use the most current UDFM file.

Layout-Aware Diagnosis Example

The following example shows a layout-aware diagnosis report with cell port bridge diagnosis turned off.

```
symptom=1 #suspects=1 #explained_patterns=23
13     16     28     81     89     118    157    162    174    191
195    201    203    208    221    246    335    377    406    412
479    502    506
suspect  score fail_match pass_mismatch type
value  pin_pathname cell_name net_pathname layout_status
------------------------------------------------------------------------------------------
1        100    23          0             OPEN/DOM       0      /…/U471/o1 INV_X1 /…/n265
LOCATION_IN_LAYOUT
   #potential_open_segments=1, #total_segments=3,   #potential_bridge_aggressors=0,
   #total_neighbors=na
   suspect  score fail_match pass_mismatch type           value  location layout_layer
   critical_area
   ------------------------------------------------------------------------------------------
   1.1      100    23         0            OPEN           0       B1&B2 m1             7.84E-03
                                                         v1              1.22E-03
                                                         m2              1.14E-02
------------------------------------------------------------------------------------------
```

The following example shows the same layout-aware diagnosis report but with cell port bridge diagnosis enabled. Now two extra CELL_PB suspects are called out for two sink cells, U87 and U101, for the potential port bridge defect.

```
symptom=1 #suspects=1 #explained_patterns=23
13     16     28     81     89     118    157    162    174    191
195    201    203    208    221    246    335    377    406    412
479    502    506
suspect  score fail_match pass_mismatch type
value  pin_pathname cell_name net_pathname layout_status
------------------------------------------------------------------------------------------
1        100    23          0             OPEN/DOM       0      /…/U471/o1 INV_X1 /…/n265
LOCATION_IN_LAYOUT
   #potential_open_segments=1, #total_segments=3,   #potential_bridge_aggressors=0,
   #total_neighbors=na
   suspect  score fail_match pass_mismatch type           value  location layout_layer
   critical_area
------------------------------------------------------------------------------------
1.1      100    23         0             OPEN           0       B1&B2 m1             7.84E-03

v1              1.22E-03

m2              1.14E-02
------------------------------------------------------------------------------------------
2        80     23          19            CELL_PB        0      /…/U87/a NAND2_X1 /…/n265
LOCATION_IN_LAYOUT
3        79     23          24            CELL_PB        0      /…/U101/a NAND2_X2 /…/n265
LOCATION_IN_LAYOUT
------------------------------------------------------------------------------------------
```

Cell-Aware Diagnosis Example

The following cell-aware diagnosis report shows 2 CELL_PB suspects for the same failure file. For the CELL_PB suspect 1.2, 14 bridge faults associated with port U87/a are called out as cell-aware suspects. For CELL_PB suspect 1.3, 11 bridge faults associated with port U101/a are called out as cell aware suspects.

```
symptom=1 #suspects=1 #explained_patterns=23
13    16    28    81    89    118   157   162   174   191
195   201   203   208   221   246   335   377   406   412
479   502   506

suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
net_pathname layout_status
-----------------------------------------------------------------------------------------
1        100   23         0             OPEN/DOM       0      /…/U471/o1 INV_X1 /…/n265
LOCATION_IN_LAYOUT
   #potential_open_segments=1, #total_segments=3,   #potential_bridge_aggressors=0,
   #total_neighbors=na
   suspect  score fail_match pass_mismatch type           value  location layout_layer
   critical_area
   --------------------------------------------------------------------------------------
   1.1      100   23         0             OPEN           0      B1&B2 m1            7.84E-03
                                                         v1              1.22E-03
                                                         m2              1.14E-02
   --------------------------------------------------------------------------------------
2        80    23         19            CELL_PB        0      /…/U87/a NAND2_X1 /…/n265
LOCATION_IN_LAYOUT
   #cell_internal_suspects=14, #total_cell_internal_faults=137
   suspect  score fail_match pass_mismatch type           value  location layout_layer
   critical_area
   -------------------------------------------------------------------------------------
   2.1      80    23         19            CELL_BRIDGE    -      D11/a_33--o1_27/1.0
   2.2      80    23         19            CELL_BRIDGE    -      D29/a_29--vcc_28/1.0
   2.3      80    23         19            CELL_BRIDGE    -      D30/a_18--b_24/1.0
   2.4      80    23         19            CELL_BRIDGE    -      D14/o1_27--a_14/1.0
...
   2.12     80    23         19            CELL_BRIDGE    -      D61/o1_18--a_34/1.0
   2.13     80    23         19            CELL_BRIDGE    -      D70/a_18--o1_18/1.0
   2.14     80    23         19            CELL_BRIDGE    -      D72/o1_1--a_1/1.0
   --------------------------------------------------------------------------------------
3        79    23         24            CELL_PB        0      /…/U101/a NAND2_X2 /…/n265
LOCATION_IN_LAYOUT
   #cell_internal_suspects=11, #total_cell_internal_faults=123
   suspect  score fail_match pass_mismatch type           value  location layout_layer
critical_area
   -------------------------------------------------------------------------------------
   3.1      79    23         24            CELL_BRIDGE    -      D10/a_26--o1_13/1.0
   3.2      79    23         24            CELL_BRIDGE    -      D29/a_25--vcc_44/1.0
   3.3      79    23         24            CELL_BRIDGE    -      D30/a_15--b_14/1.0
   3.4      79    23         24            CELL_BRIDGE    -      D13/o1_13--a_11/1.0
...
   3.9      79    23         24            CELL_BRIDGE    -      D36/a_15--g1.n1_7/1.0
   3.10     79    23         24            CELL_BRIDGE    -      D46/a_18--o1_16/1.0
   3.11     79    23         24            CELL_BRIDGE    -      D53/a_21--o1_36/1.0
   ---------------------------------------------------------------------------------------
```

Chapter 4
Diagnosis for Hierarchical Designs

For large designs, it is typical to use the Tessent Shell hierarchical RTL and scan DFT insertion flow, which results in ATPG test patterns for core-level blocks that you retarget to the top level to create top-level ATPG test patterns. This chapter describes how to perform layout-aware diagnosis on the core-level test patterns and the top-level test patterns.

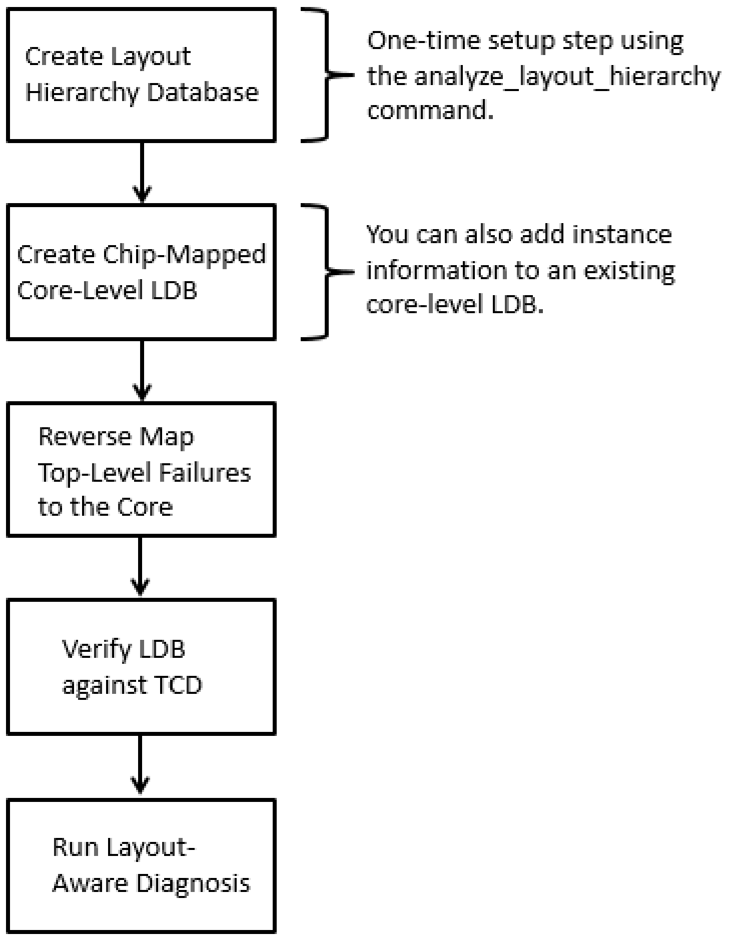
Refer to "Tessent Shell Flow for Hierarchical Designs" in the *Tessent Shell User's Manual* for information about the hierarchical DFT insertion flow in Tessent Shell.

Core-Level Layout-Aware Diagnosis

The hierarchical DFT insertion flow results in the top-level failure files, which are obtained by applying the top-level retargeted patterns on the testers. However, Tessent Diagnosis requires core-level flat models as inputs. This means that to diagnose failures at the core level, you must translate—or reverse map—the top-level failure files back to chip- mapped core-level failure files after creating core-level LDBs.

For each wrapped core in the design, the high-level flow is:

Figure 4-1. Chip-Mapped Core-Level LDB Layout-Aware Diagnosis Flow


Create Layout Hierarchy Database
One-time setup step using the analyze_layout_hierarchy command.
Create Chip-Mapped Core-Level LDB
You can also add instance information to an existing core-level LDB.
Reverse Map Top-Level Failures to the Core
Verify LDB against TCD
Run Layout-Aware Diagnosis

For hierarchical designs, each core is a sub-portion of the complete design, which means the core LDB is smaller than the LDB for the entire design because it only stores the net, pin, cell, and core instance locations for a given core LDB. All other data pertaining to top level and other cores is discarded. The layout hierarchy database contains the hierarchy tree of the entire design along with the LEF/DEF files required for each core. This enables Tessent Diagnosis to generate chip-mapped core-level LDBs for any core in a design without re-processing the LEF/DEF for the entire design.

Additionally, from the information stored in the layout hierarchy database, the tool can estimate the size of any chip-mapped core LDB and the amount of memory required for processing.

If the same core is instantiated in different chips, you can add the instance information from both of these chips to the same core-level LDB.

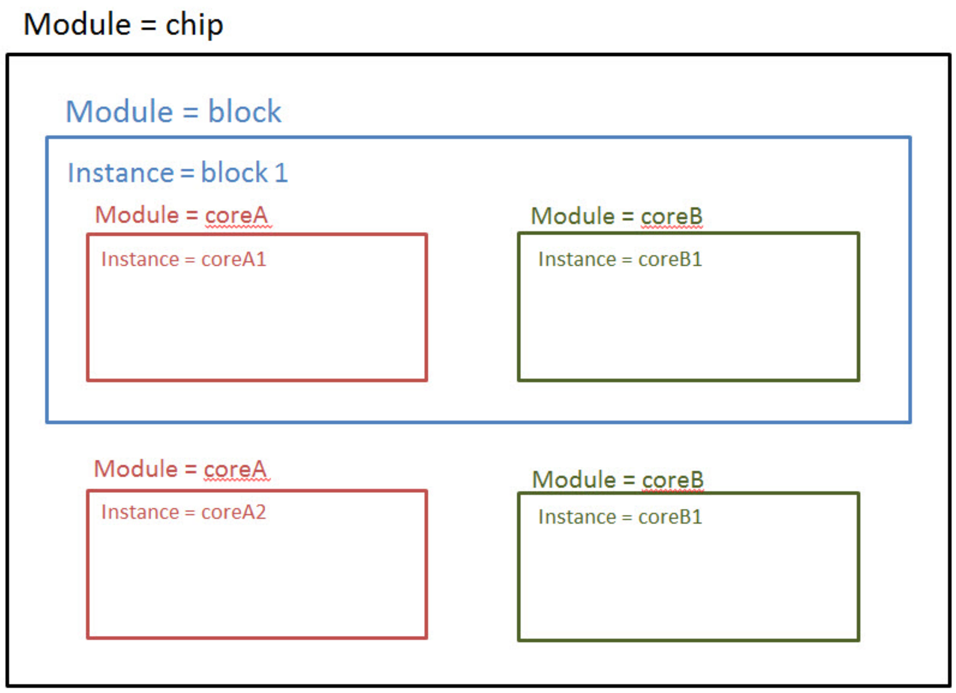
Note
For STDF-V4 2007(.1)-formatted files, you must first use the dlogutil utility to extract the failures into files formatted for Tessent Diagnosis. Refer to "dlogutil Utility" for details.

Generating Chip-Mapped Core-Level LDBs

For each core, generate chip-mapped core-level LDBs that store the coordinates, including placement and orientation in reference to the top-level chip, of all instances of a specified core. Tessent Diagnosis derives this information from the layout hierarchy database that you specify. For a specified core, the tool reads only the core LEF/DEF information stored in the database.

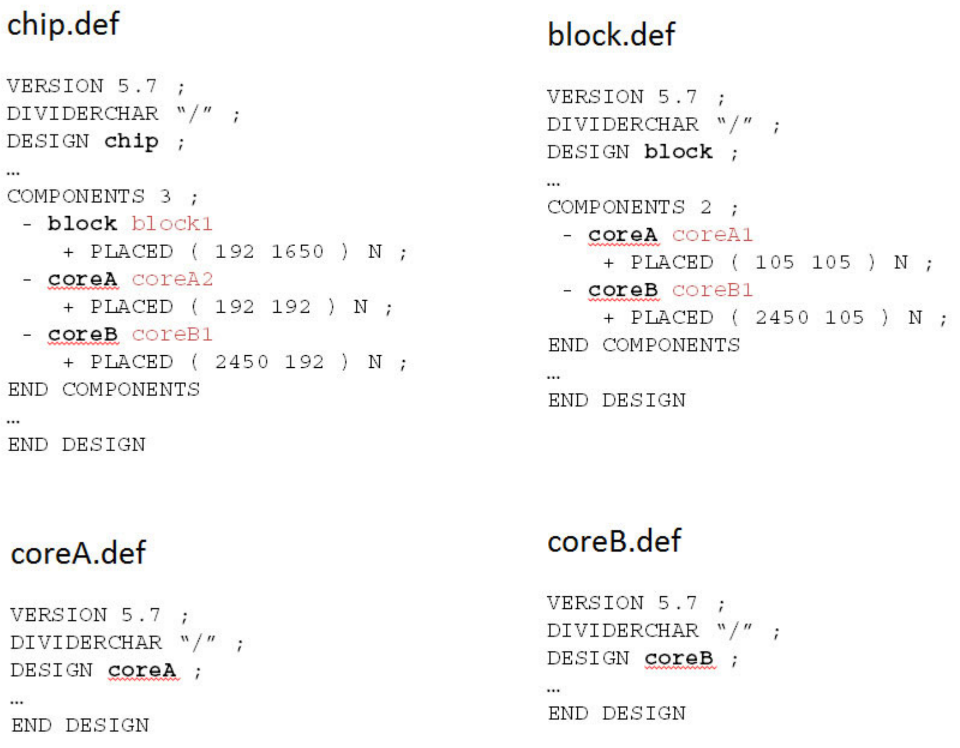
Consider the following design with two pairs of identical cores. Two of the cores are instantiations of CoreA and the other two are instantiations of CoreB. The core-level flat model files for the two cores are coreA.flat.gz and coreB.flat.gz.

Figure 4-2. Design With Two Pairs of Identical Cores



The corresponding DEF files are:

Figure 4-3. DEF Corresponding to Design With Two Pairs of Identical Cores



chip.def

```
VERSION 5.7 ;
DIVIDERCHAR "/" ;
DESIGN chip ;
…
COMPONENTS 3 ;
 - block block1
     + PLACED ( 192 1650 ) N ;
 - coreA coreA2
     + PLACED ( 192 192 ) N ;
 - coreB coreB1
     + PLACED ( 2450 192 ) N ;
END COMPONENTS
…
END DESIGN
```

block.def

```
VERSION 5.7 ;
DIVIDERCHAR "/" ;
DESIGN block ;
…
COMPONENTS 2 ;
 - coreA coreA1
     + PLACED ( 105 105 ) N ;
 - coreB coreB1
     + PLACED ( 2450 105 ) N ;
END COMPONENTS
…
END DESIGN
```

coreA.def

```
VERSION 5.7 ;
DIVIDERCHAR "/" ;
DESIGN coreA ;
…
END DESIGN
```

coreB.def

```
VERSION 5.7 ;
DIVIDERCHAR "/" ;
DESIGN coreB ;
…
END DESIGN
```

Prerequisites

- Core-level flat models.
- You have a layout hierarchy database as previously created by the analyze_layout_hierarchy command. Refer to "Estimation of Resources Required for Generating LDBs" on page 185.

Procedure

Note

If you already have core-level LDBs, you can add instance information to them as described in "Adding Instance Information to an Existing Core-Level LDB" on page 301.

1. The following flow creates a core-level LDB for coreA. Note the following for the create_layout command:

   - The -core option specifies to create an LDB for the core.
   - The tool uses the hierarchical LEF/DEF structural information stored in the layout hierarchy database, which means that you do not have to specify LEF/DEF files. If you specify both the layout hierarchy database with the -hierdb option and LEF/DEF files, the tool issues an error.

   ```
   > tessent -shell
   SETUP> set_context pattern -scan_diagnosis
   SETUP> read_flat_model coreA.flat.gz
   ...
   ANALYSIS> create_layout coreA.ldb -core coreA -hierdb hierdb_name
   ...
   ANALYSIS> open_layout coreA.ldb -chip_design_name chip
   ANALYSIS> report_layout_core_information
   ...
   ANALYSIS>
   ```

2. The following flow creates a core-level LDB for coreB:

   ```
   > tessent -shell
   SETUP> set_context pattern -scan_diagnosis
   SETUP> read_flat_model coreB.flat.gz
   ...
   ANALYSIS> create_layout coreB.ldb -core coreB -hierdb hierdb_name \
   ...
   ANALYSIS> open_layout coreB.ldb
   ANALYSIS> report_layout_core_information
   ...
   ANALYSIS>
   ```

Results

The tool generates LDBs that include the polygons defined in the DEF file for specified cores plus sub-DEF files instantiated within the cores. The tool ignores polygon interactions with

DEF files located at higher or parallel locations. Thus, some bridge defect inaccuracies might occur for bridge defects between these polygons and the core polygons.

An error occurs if the DEF design name specified with the -core option does not match a design name specified with the create_layout command.

By default, core name and core instance path mismatches between the tracking information in the failure file and the LDB generate errors. You can use the set_diagnosis_options -core_instance_error option to change the default behavior to warnings. In addition, when opening an LDB that contains core instances, the layout fails to open with an error if the -chip_design_name option is unspecified.

Reverse Mapping Top-Level Failures to the Core

The top-level failures come from one or more test suites derived from retargeted patterns associated with the same test configuration.

The following considerations also apply to reverse mapping top-level failures:

- For repeated top-level patterns that stem from the same core pattern, Tessent Shell considers only the failures for the first pattern.
- Failures corresponding to different top-level Tessent Core Description (TCD) files should be logged in separate failure files and reverse mapped separately. Do not merge failures from multiple core-level patterns that correspond to multiple top-level core description files together into one failure file with multiple test suites.

Note

Prior to performing reverse mapping of actual silicon top-level failures to the core, you can create emulated open-socket failure files to test that your hierarchical design diagnosis with reverse mapping performs correctly. For details, refer to "Validating Reverse Mapping Prior to Core-Level Layout-Aware Diagnosis" on page 305.

Prerequisites

- Top-level TCD file that resulted from scan pattern retargeting.
- Top-level parallel STIL or parallel WGL test pattern files.
- Top-level failure files.

Procedure

1. In Tessent Shell, set the context as follows:

   set_context patterns -failure_mapping

2. Specify the top-level TCD file. For example:

   read_core_description top.tcd

3. Specify the current design. For example:

   set_current_design top

   This step is optional if you read only one retargeted TCD file into the tool.

4. Set the system mode to analysis, as follows:

   set_system_mode analysis

5. Specify the top-level STIL/WGL test pattern files. For example:

   read_patterns top.stil

   read_patterns top2.stil -append

   The pattern files contain the information that the tool requires to reverse map the failing cycles. The tool extracts only the information that it needs for reverse mapping. Automatic consistency checking ensures that the top-level TCD matches the top-level patterns.

   To load multiple pattern files, use the -append argument to specify additional pattern files after the first specified STIL/WGL file. This argument is required if the retargeted patterns were written to multiple STIL/WGL files, typically by using the write_patterns -maxloads or -begin/-end arguments.

6. Read the top-level failure file. For example:

   read_failures fail1.top

7. Reverse map the top-level failure file into core-level failure files, and write them to a specified directory. For example:

   write_failures ./core_flogs/fail1

   By default, the tool verifies the failure file expected values against the top-level pattern data before generating a core-level failure file. If the tool detects a mismatch, it issues an error. If the failure file does not have expected values, the tool issues a warning.

   Optionally, before writing out the failure files, you can adjust some aspects of the tool's reverse mapping behavior with the set_failure_mapping_options command.

   Note

   The generated filename for the core-level failure log contains escaped characters if the core instance name contains escapes. In this case, you must specify additional escape characters when you load this failure log with either the read_failures or diagnose_failures commands. For example, you specify the filename for an escaped log file named *test.flog__c2_1\[3\]__core2__internal* as:

   ```
   read_failures {test.flog__c2_1\\\[3\\\]__core2__internal}
   ```

8. Report the unmapped failures. For example:

   report_failures -unmapped

Results

Reverse mapping the top-level failure files for the retargeted test patterns results in a set of core-level failure files that you can diagnose with Tessent Diagnosis. After you reverse map the failures back to the core level, the tool uses them in conjunction with the ASCII patterns that were used for retargeting and the core-level flat model to perform diagnosis at the core level.

The following example for a converted core-level failure file named "top_flog1___core_3.core_2.test__piccpu_maxlen16_1__internal" shows the core description added into core-level failure files.

```
// top_flog1___core_3.core_2.test__piccpu_maxlen16_1__internal
format pattern

tracking_info_begin
core_instance "core_3/\core_2.test "
core_name piccpu_maxlen16_1
core_mode internal
core_id core_3.core_2.test__piccpu_maxlen16_1__internal
tracking_info_end

scan_test

failures_begin

  0 blk1_edt_channels_out1 2 L H
  3 blk1_edt_channels_out1 2 H L
  4 blk1_edt_channels_out1 2 L H
  7 blk1_edt_channels_out1 2 H L
…

failures_end

failure_buffer_limit_reached none
failure_file_end
```

Examples

Sometimes when the ATE reports failures, there is an offset between the reported cycles and the cycle number that diagnosis expects from the ATPG pattern. You can correct this during reverse mapping by using the set_failure_mapping_options -cycle_offset command that realigns the failures to the pattern. For example:

```
proc findValidCycleOffset { flog cycle_offset_lb cycle_offset_ub } {
   set result 0;  // return value
   set cycle_offset_list { } ;
   set verbose 1 ;     // print out detailed results
   set cycle_offset_identified 0 ;    // true if one valid offset is found
// Scan the offset range to find valid offset value(s)
 if { $verbose == 0 }  { set_screen_display off }
 for { set offset $cycle_offset_lb } { $offset <= $cycle_offset_ub } {
incr offset } {
      puts "*** Try cycle offset ( $offset ) with failure file ( $flog )
***"
         set_failure_mapping_options -cycle_offset $offset

         read_failure $flog;

        if { [ catch_output { write_failure core_$flog -replace } -output
oVar -result rVar ]  } {
            puts $oVar;
            puts $rVar;
            puts "Found one invalid cycle offset ( $offset )" ;
         } else {
           puts "Found one possible valid cycle offset ( $offset )" ;
           puts $oVar
           lappend cycle_offset_list $offset;
         }
   }

   // Report the valid cycle offset identified
   set_screen_display on
      set num_valid_offset [ llength $cycle_offset_list ];
   if { $num_valid_offset == 0 } {
      puts "//  Error: Failed to identify any valid cycle offset within \
[$cycle_offset_lb, $cycle_offset_ub\]"
         set result 1;
   } else {
      if { $num_valid_offset == 1 } {
         puts "// Found $num_valid_offset valid cycle offset within \
[$cycle_offset_lb, $cycle_offset_ub\] : $cycle_offset_list";

set cycle_offset_identified 1;
         set final_offset [ lindex $cycle_offset_list 0 ];
         set_failure_mapping_options -cycle_offset $final_offset;
         puts "// set_failure_mapping_options '-cycle_offset' to value (
$final_offset )"
            //read_failure $flog
      } else {
         puts "// Found $num_valid_offset valid cycle offsets within \
[$cycle_offset_lb, $cycle_offset_ub\] : $cycle_offset_list";
        puts "// Further investigation is needed to determine which offset
value should be used." ;
         puts "// Hint: Another failure file with more failure data may
help identify the right offset value." ;
         set result 2;
      }
   }

   return $result;
}
```

Running Layout-Aware Diagnosis Using a Core-Level LDB

Use a core-level LDB to automatically produce chip-level suspect bounding box coordinates in the diagnosis report while still running diagnosis at the core level.

Prerequisites

- For Tessent Diagnosis and Tessent Diagnosis Server to automatically report chip-level suspect coordinates in the diagnosis report, you must:
  - Provide the pathname of the core instance associated with the failure log by using the core_instance keyword in the tracking information section of the failure log.
  - Ensure that the core instance pathname you specify matches one of the core instance pathnames stored in the LDB.
  - When opening the LDB using the open_layout command, use the -chip_design_name option to specify the top design that the tool should consider. If the LDB contains a single chip_design_name, the tool automatically sets the top design.
- Preparation steps:
  - You have performed reverse-mapping as described in "Reverse Mapping Top-Level Failures to the Core."

    Note

    When you use the Tessent Shell pattern retargeting flow to create core-level failure logs, the tool automatically populates the core_instance keyword with the correct core instance pathname. In the pattern retargeting flow, the core instance pathnames in the TCD come from the Verilog design, while the core instance pathnames in a core-level LDB come from the DEF files. This implies that the pathname in the Verilog should match the pathname in the DEF files.

  - You have verified that the core instance pathnames and hierarchy in the TCD matches the core instance pathnames and hierarchy in the LDB. Use the report_tcd_ldb_validation command to perform this verification. If the names do not match, the tool issues errors.

Procedure

1. Suppose the failure log for coreA in "Generating Chip-Mapped Core-Level LDBs" on page 291, displays a tracking information section as follows.

```
format cycle
tracking_information_begin
   lot_id 1
   wafer_id 1
   chip_id Lot1Wafer1~X12Y17
   x_coord 012
   y_coord 017
   core_instance coreA2
tracking_information_end
```

The core_instance keyword shows that this failure log originated from the coreA2 instance. The following flow runs layout-aware diagnosis on this failure log.

```
> tessent -shell
…
SETUP> set_context pattern -scan_diagnosis
SETUP> read_flat_model coreA.flat.gz
…
ANALYSIS> read_patterns coreA.wgl.gz
…
ANALYSIS> open_layout coreA.ldb -chip_design_name chip
ANALYSIS> report_layout_core_information
ANALYSIS> diagnose_failures coreA.fail_log
…
ANALYSIS> write_diagnosis -format text csv layout -file \
   coreA.fail_log
```

For the diagnosis report, CSV file, and layout marker file, the tool automatically translates the coreA layout coordinates that are stored in the LDB (which came from the top.def file as shown "Generating Chip-Mapped Core-Level LDBs") to 192, 192, rotated north.

2. The following tracking information section applies to coreB, with the failure log originating from the block1/coreB1 instance:

```
format cycle
tracking_information_begin
   lot_id 1
   wafer_id 1
   chip_id Lot1Wafer1~X34Y11
   x_coord 034
   y_coord 011
   core_instance block1/coreB1
tracking_information_end
```

3. Given this failure log, you would perform diagnosis as follows:

   ```
   > tessent -shell
   …
   SETUP> set_context pattern -scan_diagnosis
   SETUP> read_flat_model coreB.flat.gz
   …
   ANALYSIS> read_patterns coreB.wgl.gz
   …
   ANALYSIS> open_layout coreB.ldb -chip_design_name chip
   ANALYSIS> report_layout_core_information
   ANALYSIS> diagnose_failures CoreB.fail_log
   …
   ANALYSIS> write_diagnosis -format text csv layout -file \
      CoreB.fail_log
   ```

   The tool automatically translates the layout coordinates to 2642, 1755, rotated north. The tool derives these coordinates from the coordinates for block1 (top.def file) and coreB1 (block.def file).

   x = 192 + 2450 = 2643

   y = 1650 + 105 = 1755

Results

By default, the tool returns the core-level pin and net names for both core-level and chip-level diagnoses. For example:

```
...
suspect  score fail_match pass_mismatch typevalue  pin_pathname cell_name
net_pathname layout_status
-----------------------------------------------------------------------------------------
1        100   118        0             BRIDGE_2WAY    0      /cpu_i/uPMI/ix2092/Y inv02
/cpu_i/uPMI/nx2093  LOCATION_IN_LAYOUT

   suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
   net_pathname layout_layer critical_area
   -------------------------------------------------------------------------------------
   1.1      100   118        0             BRIDGE_2WAY    0      /cpu_i/uPMI/ix1024/Y inv02
   /cpu_i/uPMI/nx1023 route_3     1.30E+02
   -------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
...
```

Tessent Diagnosis can generate reports with pin names and net names corresponding to the chip level when you specify the following command:

```
set_diagnosis_options -include_core_instance_name on
```

The following example shows the results with chip-level names:

```
...
suspect  score fail_match pass_mismatch typevalue  pin_pathname cell_name
net_pathname layout_status
-------------------------------------------------------------------------------------------
1        100   118        0             BRIDGE_2WAY    0      /tree_branch1/coreB/cpu_i/uPMI/
ix2092/Y inv02 /tree_branch1/coreB/cpu_i/uPMI/nx2093  LOCATION_IN_LAYOUT

   suspect  score fail_match pass_mismatch type           value  pin_pathname cell_name
   net_pathname layout_layer critical_area
   ----------------------------------------------------------------------------------------
   1.1      100   118        0             BRIDGE_2WAY    0      /tree_branch1/coreB/cpu_i/
uPMI/ix1024/Y inv02 /tree_branch1/coreB/cpu_i/uPMI/nx1023 route_3     1.30E+02
   ----------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
...
```

You may receive the following warnings when you specify “set_diagnosis_options -include_core_instance_name on”. Ensure that you have satisfied the conditions listed in “Prerequisites” for reporting chip-level suspect coordinates.

```
Warning: Diagnosis option -include_core_instance_name is set but the layout DB was not opened
with -chip_name. Due to this the suspect layout co-ordinates will be reported at the core
level in the diagnosis results.

Warning: Diagnosis option -include_core_instance_name is set but the tracking info in the
fail log does not specify a core instance name using the ‘core_instance’ keyword. Without
this the pathnames in the diagnosis results will be reported at the core level.
```

Examples

When you have multiple designs that include instantiations of the same core—such as coreB within both design top and design top_other as described in the previous section—you must specify the design when you run diagnosis. Do this by using the open_layout -chip_design_name option as shown below.

```
> tessent -shell
…
SETUP> set_context pattern -scan_diagnosis
SETUP> read_flat_model coreB.flat.gz
…
ANALYSIS> open_layout coreB.ldb -chip_design_name chip_other
…
ANALYSIS> report_layout_core_information
```

Adding Instance Information to an Existing Core-Level LDB

You can add core instance information to an existing core-level LDB by using the add_layout_core_information command.

Prerequisites

- The add_layout_core_information command requires a valid LDB opened with the open_layout command, top-level DEF files that instantiate the core represented by the currently opened LDB. The core DEF file is optional.

Procedure

Given the design in "Generating Chip-Mapped Core-Level LDBs" on page 291, suppose you already have a core-level LDB located in the coreB_existing.ldb directory. To automatically add core instance information to this LDB, use the add_layout_core_information command as shown below:

```
> tessent -shell
…
SETUP> set_context pattern -scan_diagnosis
SETUP> read_flat_model coreB.flat.gz
…
ANALYSIS> open_layout coreB_existing.ldb
ANALYSIS> add_layout_core_information -def chip.def block.def

# Optionally, you can include the core DEF file as show below.
# ANALYSIS> add_layout_core_information -def chip.def block.def
# coreB.def
…
```

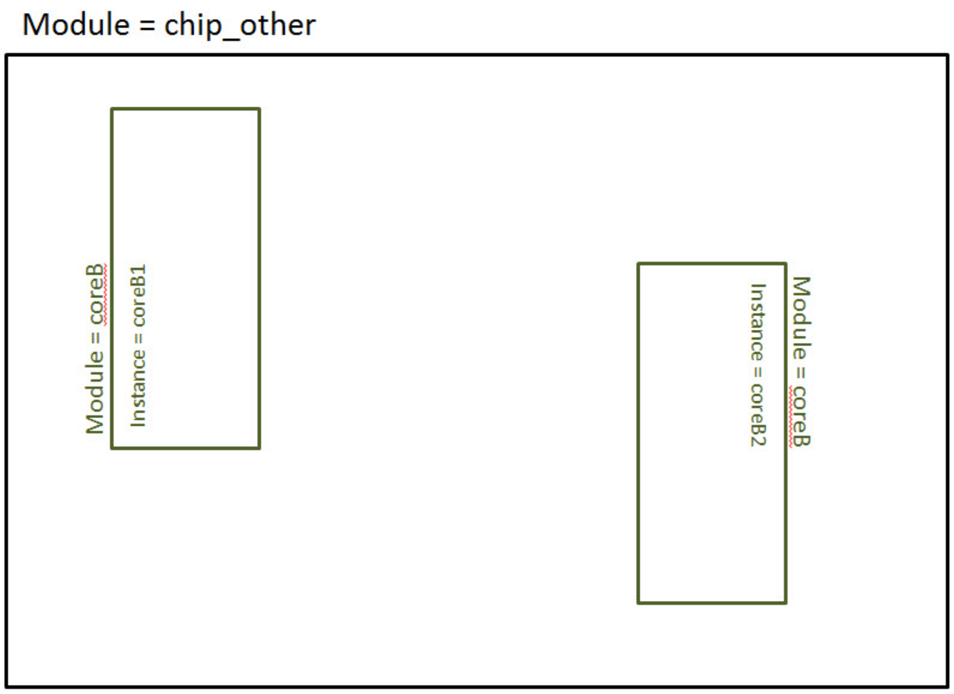
Note

If you only have core-level DEF files, you can manually add chip-mapped core instance information. Refer to "add_layout_core_information" in the *Tessent Shell Reference Manual*.

Examples

You may have a scenario in which the same core is instantiated in multiple designs. For example, suppose coreB from "Generating Chip-Mapped Core-Level LDBs" is also instantiated in a design called chip_other.

Figure 4-4. coreB Instantiation in Design chip_other



Assuming, you have already created the core-level LDB for coreB (coreB.ldb). You can add the core instance information from chip_other to the coreB.ldb as shown below:

```
> tessent -shell
…
SETUP> set_context pattern -scan_diagnosis
SETUP> read_flat_model coreB.flat.gz
…
ANALYSIS> open_layout coreB.ldb
ANALYSIS> report_layout_core_information
// Top design name: 'chip'
// Instances of 'coreB':
// Instances:
// ---------------------------------------------------------------
// Instance Path       PlaceX    PlaceY     Orient
// ---------------------------------------------------------------
//   block1/coreB1     2642      1755       N
//   coreB1            2450      192        N
// ---------------------------------------------------------------
// Current top design not set

ANALYSIS> add_layout_core_information -def chip_other.def
//  Note: Processing chip DEF files to load core "coreB"
//  Note: Loading core instances from DEF file chip_other.def
//  Note: Adding core def file design.def
//  Note: Instances:
//  Note: ---------------------------------------------------
//  Note:   Instance Path       PlaceX    PlaceY     Orient
//  Note: ---------------------------------------------------
//  Note:   coreB1             192       2450       W
//  Note:   coreB2             2450      192        E
//  Note: ---------------------------------------------------
…
//  Note: Loaded top design 'chip_other' with 2 instances of core 'coreB '

ANALYSIS> report_layout_core_information

// Top design name: 'chip'
// Instances of 'coreB':
// Instances:
// ---------------------------------------------------------------
//   Instance Path       PlaceX    PlaceY     Orient
// ---------------------------------------------------------------
//   block1/coreB1       2642      1755       N
//   coreB1              2450      192        N
//  ---------------------------------------------------------------
// Top design name: 'chip_other'
// Instances of 'coreB':
// Instances:
// ---------------------------------------------------
//   Instance Path       PlaceX    PlaceY     Orient
// ---------------------------------------------------
//   coreB1              192       2450       W
//   coreB2              2450      192        E
// ---------------------------------------------------
…
```

Validating Reverse Mapping Prior to Core-Level Layout-Aware Diagnosis

Prior to performing reverse mapping of actual silicon top-level failures to the core, you can create emulated open-socket failure files to test that your hierarchical design diagnosis with reverse mapping performs correctly. You can use this as a final preparatory step before going to silicon; open-socket failure files enable you to discover issues that would otherwise be unknown until diagnosis of actual silicon failures.

Prerequisites

- Top-level TCD files that resulted from scan pattern retargeting.
- Top-level parallel STIL or parallel WGL test pattern files.

Procedure

1. Set the Tessent Shell context to patterns -failure_mapping.
2. For all top-level TCD file and test pattern file pairs, run the following command:

   ```
   create_open_socket_failure failure_filename
   ```

   This command creates emulated top-level, open-socket failure files that contain a failing bit for each valid measure cycle in the top-level pattern. Because there is no chip, all the cycles show as failing. However, you can use these sample cycle-based failure files to validate your reverse-mapping setup.
3. Perform reverse mapping as described in "Reverse Mapping Top-Level Failures to the Core" on page 294".
4. Using the read_failures command, validate the core-level failure log using the core-level flat model and patterns to confirm that the failing bits are reverse-mapped correctly.

Top-Level Layout-Aware Diagnosis

When performing layout-aware diagnosis at the top level, Tessent Diagnosis can automatically preserve only the external mode layout information of the wrapped cores—that is, the graybox models. In the bottom-up hierarchical DFT process, graybox models retain the minimum logic required to generate ATPG patterns for the internal modes of the parent physical blocks. Likewise, for diagnostic purposes, Tessent Diagnosis does not need to know about the layout information completely internal to the cores when you are performing layout-aware diagnosis on parent blocks or the top-level chip.

In the DFT insertion flow, graybox models allow Tessent Shell to connect the wrapped cores at the top level for logic testing of the chip. However, for layout-aware diagnosis at the top level, to create an LDB that includes layout information for logic in the graybox model, you would either use full-chip DEF files or specifically generate DEF files that only include the graybox layout information. Tessent Diagnosis enables you to perform top-level diagnosis without producing a graybox DEF model. The LDB that is created from the DEF files skips the data from the grayboxed cores but includes all the PWR and GND nets from the grayboxed cores as well as all the physical data from the top level. This may increase the LDB by including some nets but increases the diagnosis accuracy.

Running graybox-aware layout-aware diagnosis on parent- or top-level designs minimizes the disc space required for layout data as well as the run time to complete LDB creation and pre-extraction of bridge and topology features.

Running Layout-Aware Diagnosis Using a Graybox-Aware Top-Level LDB

Layout-aware diagnosis when using a graybox-aware parent-level or top-level LDB follows the same flow as traditional layout-aware diagnosis with the exception of specifying create_layout with the -gray_box_detection switch.

Prerequisites

- The same prerequisites for layout-aware diagnosis as described in "Performing Layout-Aware Diagnosis with Tessent Diagnosis" on page 216.
- A graybox flat model that Tessent Diagnosis can use as a reference.

Procedure

The following dofile shows a simple flow for performing layout-aware diagnosis with a graybox-aware LDB. When you specify create_layout -gray_box_detection, the tool uses the graybox flat model to populate the LDB from the full-chip LEF/DEF. Without the -gray_box_detection option, the tool creates a full-chip LDB that could be much larger in size than a graybox-aware LDB. The run time to create a full-chip LDB is typically significantly higher than that of a greybox-aware LDB.

```
set_context patterns -scan_diagnosis
read_flat_model top_model
create_layout -gray_box_detection ./src/design.dft.ldb \
      -lef ./src/top_hier.lef -def ./src/core1.def
      -def ./src/core2.dft -def ./src/top_hier.def
open_layout ./src/design.dft.ldb
read_patterns./src/patterns.ascii
diagnose_failures ./tester_files/file1.flog
write_diagnosis -format text layout -file ./results/file1 -replace
```

Chapter 5
Diagnosis for Tessent LogicBIST Designs

Tessent Diagnosis can perform diagnosis on Tessent LogicBIST designs with an existing LVDB and design workspace with a corresponding ATPG library. This diagnosis flow supports logic diagnosis, layout-aware diagnosis, and at-speed diagnosis, including the capability to write out failing paths.

Overview

The diagnosis flow you use for performing Tessent LogicBIST diagnosis depends on whether you use signatureAnalyze or Tessent FastScan for fault simulation.

Note

As of the version 2015.1 release, signatureAnalyze fault simulation is no longer available. The signatureAnalyze Fault Simulator diagnosis flow is intended for users of the pre-2015.1 LV flow.

- signatureAnalyze fault simulation — Select this flow if you are using signatureAnalyze as the fault simulator. See "signatureAnalyze Fault Simulator Diagnosis Flow" for complete usage.
- Tessent FastScan fault simulation — Select this flow if you are using Tessent FastScan as the fault simulator. See "Tessent FastScan Fault Simulator Diagnosis Flow" for complete usage.

In general, these flows consist of the following aspects:

- Tessent FastScan as the simulator to create a MISR signature and save the flat model at the core level.

  Note
  This step is mandatory regardless of which flow you are using.

  This flat model includes the core Tessent LogicBIST netlist, the MISR signatures, and associated settings. Tessent FastScan requires an ATPG library.

- The dlogutil tool, a Tessent Diagnosis utility, is used to translate the top-level failure file created by Tessent SiliconInsight™ into a core-level failure file.
- The core-level failure file and flat model are used by Tessent Diagnosis for running logic BIST diagnosis. You can optionally include the LEF/DEF layout data for layout-aware diagnosis.

  Note
  If your design contains multiple cores (for example USB and PCI for a design, ROUTER, and so forth) and each core has its own logicTest controller, the diagnosis flow must treat each BISTed core individually.

Limitations

The following limitations apply to diagnosis for Tessent LogicBIST:

- There is no support for scan chain diagnosis. If you have chain failures, then Tessent SiliconInsight cannot create a top-level design failure file.
- Special handling is required for static chain masks. See "Special Handling for Static Chain Masks."

Tessent FastScan ATPG Library Requirements

The following ATPG library requirements apply, depending on your diagnosis flow:

- signatureAnalyze diagnosis flow — Tessent FastScan requires an ATPG library that matches the scan models in your existing LVDB. These scan models are normally located in the UserScanModels sub-directory of the *LVDB* directory.
- Tessent FastScan diagnosis flow — You must have completed the ATPG library verification.

You must use the LibComp utility to create the ATPG library or use the lcVerify utility to verify an existing ATPG library before using Tessent FastScan. Both of these utilities are available in your Tessent software tree.

Refer to the following sections in the *Tessent Cell Library Manual*:

- LibComp — “Create Tessent Simulation Models Using LibComp”
- lcVerify — “Verification of Tessent Simulation Models”

signatureAnalyze Fault Simulator Diagnosis Flow

In the LV Flow, Tessent LogicBIST uses the signatureAnalyze command to simulate logic BIST patterns and generate the MISR signatures.

Note

As of the version 2015.1 release, signatureAnalyze fault simulation is no longer available. This section is intended for users of the pre-2015.1 LV flow.

The signatureAnalyze fault simulation diagnosis flow consists of the following high-level steps:

1. Using an existing LVDB and a matched ATPG library, generate Tessent FastScan files that include the flat model, Tessent FastScan dofile, and MISR signatures. This step is required. Refer to "Creating the Tessent FastScan Flat Model and Verifying the MISR Signatures."

   Note

   Tessent FastScan requires an ATPG library that matches the scan models in your existing LVDB. These scan models are normally located in the *UserScanModels* sub-directory of the *LVDB* directory.

2. If you are using a LV flow version earlier than v9.5, then you must use ETVerify to create a logic BIST mapping file if you have static chain masks. Refer to "Preparing the Logic BIST Chain Mapping File."

   For version v9.5 or later, you can omit this step of the flow.

3. Use Tessent SiliconInsight to create a top-level failure file from the LVDB. Refer to "Failure File Generation."

4. Using the dlogutil utility, convert the top-level failure file to a core-level failure file. Refer to "Converting the Top-Level Failure File to a Core-Level Failure File."

5. With the core-level failure file and core-level flat model as inputs, use Tessent Diagnosis to perform the diagnosis. Refer to "Performing Logic BIST Diagnosis in Tessent Diagnosis."

   Optionally, you can input layout data (LEF/DEF) to enable layout-aware diagnosis. Refer to "Layout-Aware Diagnosis Flow" for complete information.

signatureAnalyze Flow Requirements

To run logicBIST diagnosis using signatureAnalyze as the fault simulator, you must prepare various input files: the ATPG library, LVDB MISR signatures and production patterns, design image, and design workspace.

Prepare the following files before running logic BIST diagnosis with the signatureAnalyze fault simulator:

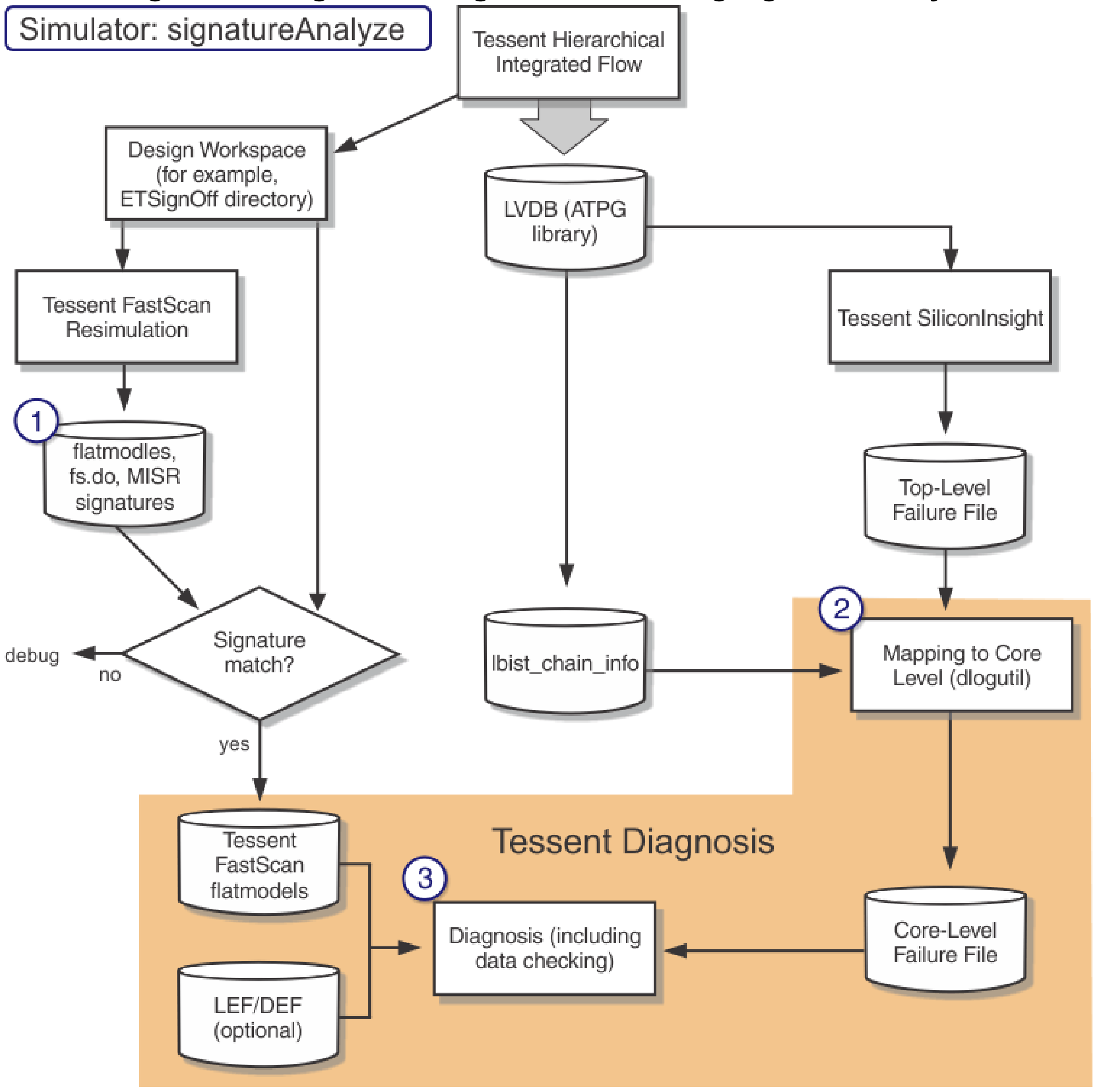
- ATPG library — When using this flow, you must use Tessent FastScan to simulate the MISR signatures. Consequently, you need an ATPG library in place that matches the netlist in the existing LVDB.
- LVDB MISR signatures and production patterns — If you are not using Tessent FastScan as the default simulator, then you must use signatureAnalyze to create and simulate the MISR signatures and production patterns in the LVDB. Using the LV Flow tool-produced dofile, you use Tessent FastScan as the simulator to re-generate the MISR signatures and compare these signatures with those signatureAnalyze had created. If you are using Tessent FastScan as the default simulator, then you can omit this requirement.
- Design image — Tessent FastScan archives the design image at the core level. You must convert, using the dlogutil utility, the top-level failure file to a core-level failure file in order to run logic BIST diagnosis in Tessent Diagnosis.
- Design workspace — In order to create the flat model, you must have a design workspace.

  For example, a typical design directory after final LVDB is ready may have the following structures:

  - final LVDB and it contains:
    - ROUTER.lvdb (top level)
    - USB.lvdb (core level)
    - PCI.lvdb (core level)
  - ROUTER_LVWS (top level workspace)
  - USB_LVWS (core level workspace) containing
    - ETSignOff
  - PCI_LVWS (core level workspace) containing
    - ETSignOff

The following figure illustrates the signatureAnalyze fault simulation diagnosis flow.

Figure 5-1. LogicBIST Diagnosis Flow Using signatureAnalyze


Simulator: signatureAnalyze
Tessent Hierarchical Integrated Flow
Design Workspace (for example, ETSignOff directory)
LVDB (ATPG library)
Tessent FastScan Resimulation
Tessent SiliconInsight
1
flatmodles, fs.do, MISR signatures
Top-Level Failure File
2
Signature match?
debug
no
yes
lbist_chain_info
Mapping to Core Level (dlogutil)
Tessent Diagnosis
Tessent FastScan flatmodels
3
Diagnosis (including data checking)
Core-Level Failure File
LEF/DEF (optional)

Note

As of the v9.5 Tessent Release, the lbist_chain_info is included in the LVDB by default. Consequently, there is no requirement to regenerate this information using ETVerify.

Creating the Tessent FastScan Flat Model and Verifying the MISR Signatures

In this step of the flow, you create a flat model (for use by Tessent Diagnosis) and verify the MISR signatures using either signatureAnalyze or Tessent FastScan.

- signatureAnalyze — See “signatureAnalyze Fault Simulator Diagnosis Flow.”
- Tessent FastScan — See “Tessent FastScan Fault Simulator Diagnosis Flow.”

By default, the LVDB contains 1024 diagnostic vectors with expected values. When you use Tessent SiliconInsight, the tool-produced failure file can contain up to 1024 vectors with expected and actual values. If the vector ID is higher than 1024, the failure file contains record raw data.

If you need to change the number of diagnosis trials with expected value, then you can change the StoredDiagnosticVectors property value (the default is 1000), re-run signatureAnalyze, and update the LVDB.

Prerequisites

- A Tessent FastScan ATPG library.
- *<moduleName>.fastscan_GenSimPatterns_lbist* — The Tessent FastScan dofile created by the LV Flow. This file is located in the *<moduleName>_LVWS/ETSignOff* directory.

Procedure

1. In a Linux/UNIX shell, go to your design’s *ETSignOff* directory and run the following target:

   make fastscan_lbist_faultsim

   This step creates the flat model of the design.

2. From within *ETSignOff* directory, run the following target:

   make compare_fs_lv_sigs

   This compares the first 256 MISR signatures generated by Tessent FastScan with those by signatureAnalyze.

Results

The corresponding logic BIST Tessent FastScan flat model is saved as follows:

```
./LV_WORKDIR/<moduleName> .fastscan_flat_model_lbist.
```

If the signatures match, then the flat model is correct and ready for you to use.

Related Topics

Special Handling for Static Chain Masks

Preparing the Logic BIST Chain Mapping File

By default, as of the v9.5 Tessent release, the logic BIST fail mapping file (*.lbist_chain_info*) is included in LVDB. There is no need to generate this file using ETVerify. The dlogutil requires the logic BIST mapping file in order to convert the top-level failure file to a core-level failure file.

Note

You must ensure you use the Tessent v9.3 (or later) release version of ETVerify, otherwise the Tessent FastScan chain name information is not included in fail map that may be needed for static chain masking—see "Special Handling for Static Chain Masks."

Prerequisites

- An existing LVDB
- If required, modifications for static chain masking. See "Special Handling for Static Chain Masks"

Procedure

1. In a Linux/UNIX shell, navigate to the top level of the LVDB.
2. Generate or re-generate the logic BIST fail mapping file using ETVerify using the following syntax:

```
etv <designName>
   -inputLVDBName <pathToLVDB> \
   -configFile <pathToCfgFile> \
   -lbistVectorDump <ON |OFf |ChainInfo>
```

   For example:

```
etv top
      -inputLVDBName ./designs/top.lvdb \
       -configFile ./designs/top.lvdb/top.etManufacturing \
        -lbistVectorDump chaininfo
```

Results

ETVerify generates the logic BIST mapping file and writes the file to the following location in the LVDB:

./LV_WORKDIR/<LBIST controller id>.lbist_chain_info

This file defines the configuration for mapping the failure file from chip level to core level as in the following example:

```
lbist_chain_stitching_configuration {
   version            : 0.2;

   controller_id      : BP0.WPPPP1; // TAP port id for LBIST controller
   total_chains       : 6;          // total number of scan chains
   total_misr_segments: 3;          // total number of MISR segments

   chain (chain5) {
      length                 : 0;      // length of scan chain
      sci_to_sco_inv         : T;      // inversion between SCI and SCO
      static_mask            : F;      // indication of static masking
     misr_segment          : misr_s1; // name of MISR segment that chain is
                                       // connected to the
      fastscan_chain_name   : lbist4; // corresponding FastScan chain name
   }
   chain (chain6) {
      length                 : 90;
      sci_to_sco_inv         : T;
      static_mask            : T;
      misr_segment           : misr_s1;
      fastscan_chain_name    : lbist;
   }
   chain (chain4) {
      length                 : 94;
      sci_to_sco_inv         : T;
      static_mask            : F;
      misr_segment           : misr_s2;
      fastscan_chain_name    : lbist3;
   }
   chain (chain3) {
      length                 : 0;
      sci_to_sco_inv         : F;
      static_mask            : F;
      misr_segment           : misr_s2;
      fastscan_chain_name    : lbist2;
   }
   chain (chain2) {
      length                 : 98;
      sci_to_sco_inv         : T;
      static_mask            : F;
      misr_segment           : misr_s3;
      fastscan_chain_name    : lbist1;
   }
   chain (chain1) {
      length                 : 99;
      sci_to_sco_inv         : T;
      static_mask            : F;
      misr_segment           : misr_s3;
      fastscan_chain_name    : lbist0;
   }
}
```

Tessent FastScan Fault Simulator Diagnosis Flow

To use Tessent FastScan instead of signatureAnalyze for the fault simulator, you can select optional targets in ETSignOff. These targets fault grade the logic LBIST patterns with Tessent FastScan and generate the flat model.

See "Dofile Usage" in the *LV Flow User's Manual* for more information.

The Tessent FastScan fault simulation diagnosis flow consists of the following high-level steps:

1. Using an existing LVDB and a matched ATPG library, generate Tessent FastScan files that include the flat model, Tessent FastScan dofile, and MISR signatures. This step is required. Refer to "Creating the Tessent FastScan Flat Model and Verifying the MISR Signatures."

   In addition, refer to "Special Handling for Static Chain Masks."

   Note
   You must have completed the ATPG library verification.

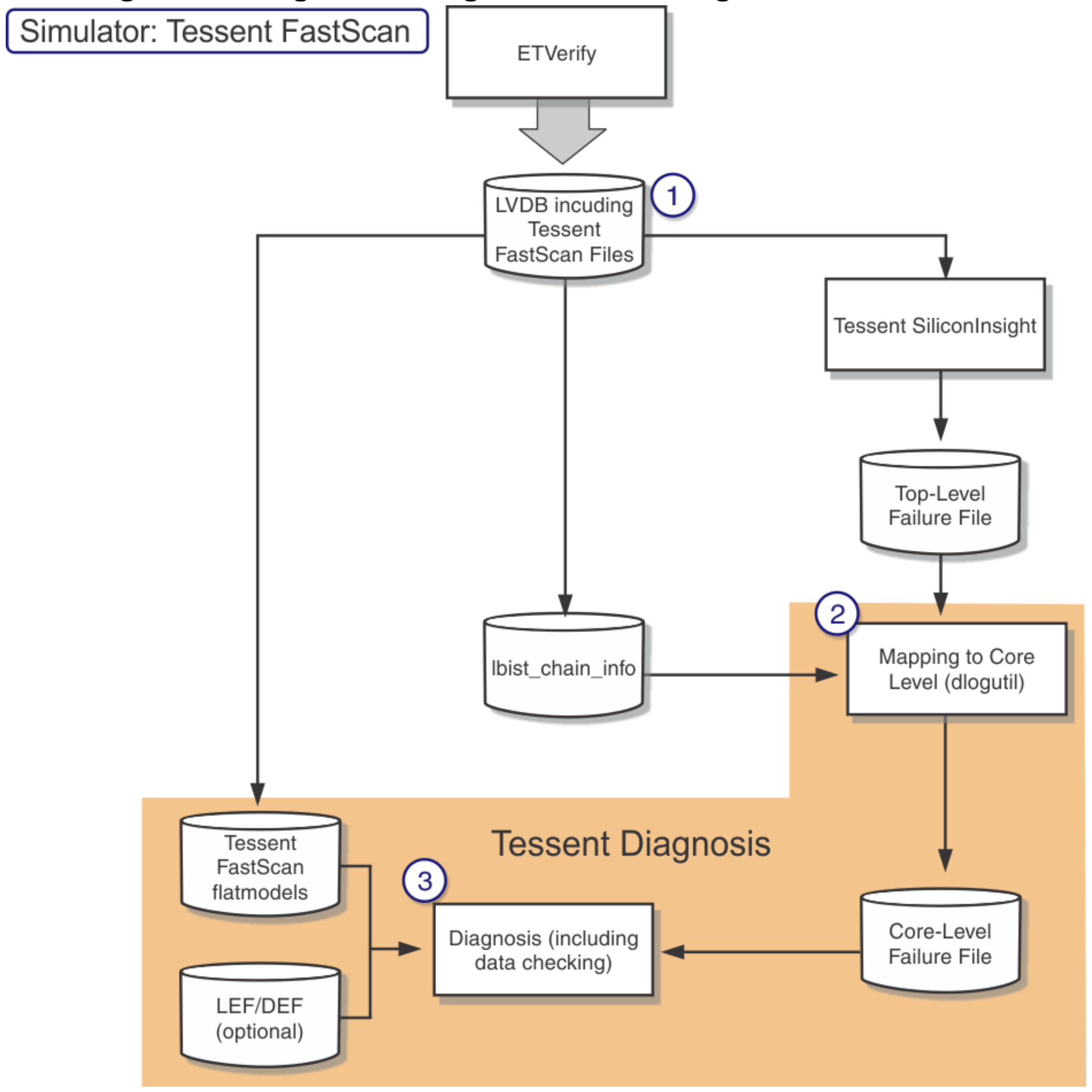
2. Using the dlogutil, convert the top-level failure file to a core-level failure file. Refer to "Converting the Top-Level Failure File to a Core-Level Failure File."

3. With the core-level failure file and core-level flat model as inputs, use Tessent Diagnosis to perform the diagnosis. Refer to "Performing Logic BIST Diagnosis in Tessent Diagnosis."

   Optionally, you can input layout data (LEF/DEF) to enable layout-aware diagnosis. Refer to "Layout-Aware Diagnosis Flow" for complete information.

Figure 5-2 illustrates this flow.

Figure 5-2. LogicBIST Diagnosis Flow Using Tessent FastScan


Simulator: Tessent FastScan
ETVerify
1
LVDB incuding Tessent FastScan Files
Tessent SiliconInsight
Top-Level Failure File
lbist_chain_info
2
Mapping to Core Level (dlogutil)
Tessent Diagnosis
Tessent FastScan flatmodels
3
Diagnosis (including data checking)
Core-Level Failure File
LEF/DEF (optional)

As of the v9.5 Tessent Release, the lbist_chain_info is included in the LVDB by default. Consequently, there is no requirement to regenerate this information using ETVerify.

Tessent FastScan Flow Requirements

To run logic BIST diagnosis using Tessent FastScan as the fault simulator, you must prepare the design image and design workspace files.

Prepare the following files before running logic BIST diagnosis with the Tessent FastScan fault simulator:

- Design image — Tessent FastScan archives the design image at the core level. You must convert, using the dlogutil utility, the top-level failure file to a core-level failure file in order to run logic BIST diagnosis in Tessent Diagnosis.
- Design workspace — In order to create the flat model, you must have a design workspace.

  For example, a typical design directory after final LVDB is ready may have the following structures:

  - Final LVDB and it contains:
    - ROUTER.lvdb (top level)
    - USB.lvdb (core level)
    - PCI.lvdb (core level)
  - ROUTER_LVWS (top level workspace)
  - USB_LVWS (core level workspace) containing
    - ETSignOff
  - PCI_LVWS (core level workspace) containing
    - ETSignOff

Creating the Tessent FastScan Flat Model and Verifying the MISR Signatures

In this step of the flow, you create a flat model (for use by Tessent Diagnosis) and verify the MISR signatures.

By default, the LVDB contains 1024 diagnostic vectors with expected values. When you use Tessent SiliconInsight, the tool-produced failure file can contain up to 1024 vectors with expected and actual values. If the vector ID is higher than 1024, the failure file contains record raw data.

If you need to change the number of diagnosis trials with expected value, then you can change the StoredDiagnosticVectors property value (the default is 1000), re-run signatureAnalyze, and update the LVDB.

Prerequisites

- A Tessent FastScan ATPG library

- *<moduleName>.fastscan_GenSimPatterns_lbist* — The Tessent FastScan dofile created by the LV Flow. This file is located in the *<moduleName>_LVWS/ETSignOff* directory.

Procedure

In a Linux/UNIX shell, go to your design's *ETSignOff* directory and run the following target:

make fastscan_lbist_sim_vectors

Results

The corresponding logic BIST Tessent FastScan flat model is saved as follows:

```
./LV_WORKDIR/<moduleName> .fastscan_flat_model_lbist.
```

Related Topics

Special Handling for Static Chain Masks

Special Handling for Static Chain Masks

Under certain circumstances, unexpected unknown values can be captured by scan cells due to design overlook or systematic fabrication errors. In this event, the MISR signatures are corrupted and, consequently, certain scan chains must be masked at the post-silicon phase.

The default Tessent FastScan flat model generated by the "make fastscan_lbist_faultsim" target does not include the chain mask information. To obtain the correct flat model with the chain mask information, you must make manual edits to the following configuration files and then generate the new test vectors:

- The <moduleName>.etSignOff file to specify the chain mask file.
- The <manualMaskingConfigFile>.etChainMask.tpl file to specify the failing flops and chains.

When you have completed this process, you can generate the flat model with chain mask information.

<moduleName>.etSignOff

To add static chain mask, you must edit the *<moduleName>.etSignOff* ETVerify configuration file in the *ETSignOff* directory.

In this file, specify the chain masking file using the following syntax:

```
etv (<designName>) {
LogicTestVectors {
      LogicBist {
            ChainMaskingFile :
                    <manualMaskingConfigFile>.etChainMasks.tpl;
     }
}
}
```

<manualMaskingConfigFile>.etChainMask.tpl

In the *<manualMaskingConfigFile>.etChainMasks.tpl* file, you can specify both failing flops and chains. When a flop is specified, then its corresponding scan chain is masked.

This file's syntax is as follows:

```
MaskData {
:      FailingFlops {
             <hierarchicalFlipFlopPath> ;
             ...
     }
     ChainMasking {
            Chain (#)  : Output | InputOutput;
     }
}
```

where:

- Chain (#) — The chain number to be masked. For example, Chain (23)
- Output — All logic BIST chains inputs are normally fed by the PRPG, but the output of the masked chains is forced to zero (0) before going into the MISR. As a result, the output of these masked chains does not contribute to the signature computation.
- InputOuput — The input and output of the masked chains are forced to zero (0) so that they not only do not contribute to the signature computation, but also, they are filled with zeros.

Note
All chains being masked must have the same chain masking mode. The combination is not allowed that some chains are masked with InputOutput while others are masked with Output only.

ChainMaskingFile Example

The following example of ChainMaskingFile specifies one failing flop and two chains masked in InputOutput mode:

```
MaskData {
:      FailingFlops {
:             MY_TOP.inst_CIRC0_REG_0.m4.\s_reg[0] ;
:             ...
:      }
:      ChainMasking {
:             Chain (23): InputOutput;
:             Chain (14): InputOutput;
     }
}
```

Refer to the LV Flow Usage Guide and *ETAnalysis Tools Reference* for specific information on chain masking.

Generating the New Test Vectors

After manually editing the *<moduleName>.etSignOff* and *<manualMaskingConfigFile>.etChainMasks.tpl* configuration files, you must re-run the "make rulea" and "make logictest_vectors" targets to generate new test vectors.

Prerequisites

- You have added the chain mask information to the *<moduleName>.etSignOff* and *<manualMaskingConfigFile>.etChainMasks.tpl* configuration files.

Procedure

1. Re-run the following targets:
    - make rulea
    - make logictest_vectors
2. Update the LVDB to include those new test vectors. See the *ETVerify Tool Reference* and *ETAnalysis Tools Reference* for more information.

Results

You are now ready to generate the flat model with chain mask information.

Related Topics

Generating the Flat Model With Chain Mask Information

Generating the Flat Model With Chain Mask Information

To generate a Tessent FastScan flat model with chain mask information, you must identify the chain IDs of the chains to be masked, and then edit the Tessent FastScan doile to include the chain mask definition.

Prerequisites

- You have completed the process as described in "Generating the New Test Vectors."

Procedure

1. Identify the chain IDs of the chains to be masked by opening the following file:

    ./LV_WORKDIR/<moduleName>.ruleainfo_lbist

    This file includes definitions for all scan chains and scan flops in logic BIST mode. Each scan chain is represented with a chain ID (from 1 to maximum number of scan chains) with all the scan flops listed. For example:

    ```
    CHAIN 6 LENGTH 17 FREQUENCY 1 PRPG 1 18 MISR 1 18 prpgPhase 0 \
       misrPhase 0 ;
    ```

In this example, the chain ID is 6. For MaskData defined with ChainMasking, the chain ID is given as it is. Additionally, in the example shown in "<manualMaskingConfigFile>.etChainMask.tpl," the scan chains with ID 14 and 23 are masked in InputOutput mode.

For MaskData defined with Failing Flop, do the following in the *<moduleName>.ruleainfo_lbist* file:

a. Search for the failing flop name and find its corresponding chain.

b. Write down the chain ID.

In the example, assume the following failing flop:

```
MY_TOP.inst_CIRC0_REG_0.m4.\s_reg[0]
```

comes from scan chain with ID 15. It means chain 15 is also masked in InputOutput mode. In this example, there are a total of three chains (14, 15, and 23) being masked.

2. Edit the Tessent FastScan *ETSignOff/<moduleName>.fastscan_GenSimPatterns_lbist* dofile to include the chain mask definition.

   The Tessent FastScan chain names for each chain ID are located in the logic BIST fail mapping file. For example, for scan chain with ID 6, the following definition can be found where "lbist2" is the corresponding Tessent FastScan chain name.

```
chain (chain6) {
length              : 90;
sci_to_sco_inv      : T;
static_mask         : F;
misr_segment        : misr_s1;
fastscan_chain_name : lbist2;
}
```

   Assume for chain IDs 14, 15, and 23, "lbist7", "lbist15", and "lbist16" are identified as the corresponding Tessent FastScan chain names. These are the chains that are going to be masked in InputOutput mode.

   In the *ETSignOff/<moduleName>.fastscan_GenSimPatterns_lbist* file, add the chain mask definition just prior to the fault command, as follows:

   - For Output chain masking, add the following command:

     add_chain_masks <chain_name> … -unload_value 0

   - For InputOutput chain masking, add the following command:

     add_chain_masks <chain_name> … -load_value 0 –unload_value 0

   Continuing with the example, the following represents the entry:

   add_chain_masks lbist7 lbist15 lbist16 -load_value 0 –unload_value 0

Or alternatively:

```
add_chain_masks lbist7 –load_value 0 –unload_value 0
FAULT> add_chain_masks lbist15 –load_value 0 –unload_value 0
FAULT> add_chain_masks lbist16 –load_value 0 –unload_value 0
```

3. Run the following targets in the *ETSignOff* directory:
   - make fastscan_lbist_faultsim
   - make compare_fs_lv_sigs

   This ensures the Tessent FastScan flat model with chain mask information is generated and MISR signatures match those generated by signatureAnalyze.

Results

The corresponding logic BIST Tessent FastScan flat model is saved as the following:

./LV_WORKDIR/<moduleName> .fastscan_flat_model_lbist

Failure File Generation

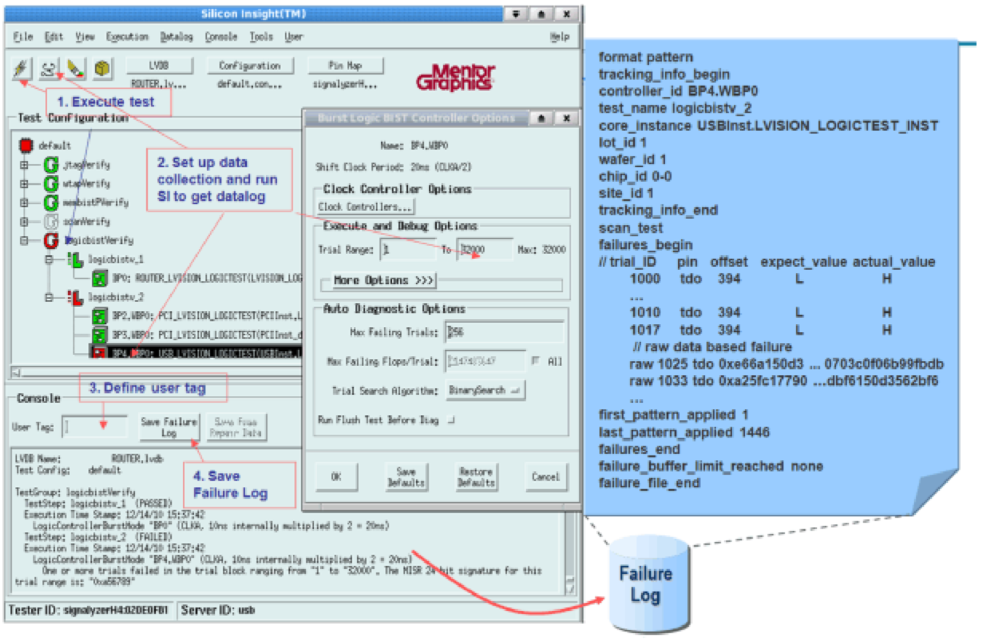
To generate a failure file for use with Tessent Diagnosis, you must use Tessent SiliconInsight ATE or Tessent SiliconInsight Desktop to generate the logic BIST top-level failure file, and then convert the top-level failure file to a core-level failure file.

Generating the Logic BIST Top-Level Failure File

You run Tessent SiliconInsight at the top design level; consequently, you must use the default top-level configuration file, *<moduleName>.lvdb/default.config_eta*.

Generate the failure file per device and per controller as shown in Figure 5-3:

Figure 5-3. Generating a Logic BIST Top-Level Failure File


Silicon Insight(TM)
File Edit View Execution Datalog Console Tools User Help
1. Execute test
2. Set up data collection and run SI to get datalog
3. Define user tag
4. Save Failure Log
Burst Logic BIST Controller Options
Save Failure Log
OK
Save Defaults
Restore Defaults
Cancel
Failure Log

```
LVDB Name:          ROUTER.lvdb
Test Config:     default

TestGroup: logicbistVerify
  TestStep: logicbistv_1  (PASSED)
  Execution Time Stamp: 12/14/10 15:37:42
    LogicControllerBurstMode "BP0" (CLKA, 10ns internally multiplied by 2 = 20ns)
  TestStep: logicbistv_2  (FAILED)
  Execution Time Stamp: 12/14/10 15:37:42
    LogicControllerBurstMode "BP4.WBP0" (CLKA, 10ns internally multiplied by 2 = 20ns)
      One or more trials failed in the trial block ranging from "1" to "32000". The MISR 24-bit signature for this
trial range is: "0xa56789"
```

```
format pattern
tracking_info_begin
controller_id BP4.WBP0
test_name logicbistv_2
core_instance USBInst.LVISION_LOGICTEST_INST
lot_id 1
wafer_id 1
chip_id 0-0
site_id 1
tracking_info_end
scan_test
failures_begin
// trial_ID    pin   offset   expect_value actual_value
      1000    tdo    394          L              H
      ...
      1010    tdo    394          L              H
      1017    tdo    394          L              H
       // raw data based failure
      raw 1025 tdo 0xe66a150d3 ... 0703c0f06b99fbdb
      raw 1033 tdo 0xa25fc17790 ...dbf6150d3562bf6
      ...
first_pattern_applied 1
last_pattern_applied  1446
failures_end
failure_buffer_limit_reached  none
failure_file_end
```

Prerequisites

- An existing LVDB.

Procedure

From a Linux/UNIX shell, invoke Tessent SiliconInsight to create the top-level failure file using the following syntax:

```
tessent -siliconinsight -cable signalyzerH4 \
     -lvdb  finalLVDB/<moduleName>.lvdb \
     -configfile finalLVDB/<moduleName>.lvdb/default.config_eta \
     -pinmapfile signalyzerH4.pinmap \
     -serverExtraArgs "-allowRawFlopDataCollection TRUE" \
     -outdir ./outDir
```

If you use Tessent SiliconInsight ATE, then the tool turns on the raw datalog generation by default without "-serverExtraArgs."

Results

Example 5-4 provides an example top-level logic BIST failure file produced by Tessent SiliconInsight. Note the following two keywords:

- *controller_id* — Indicates the LBIST controller to which this failure file belongs.

  The corresponding fail map file can be located here:

  ```
  ./LV_WORKDIR/<controller id>.lbist_chain_info.
  ```

- *core_instance* — Corresponding instance of the core under test. You can use this to help locate the design files Tessent FastScan used to generate the flat model.

Figure 5-4. Top-Level Logic BIST Failure File

```
format pattern

tracking_info_begin

controller_id BP4.WBP0 // TAP port id for LBIST controller
test_name logicbistv_2 // used in Tessent SiliconInsight
core_instance top.lbist_core1 //The core under test
lot_id 1
wafer_id 1
chip_id 0-0
part_id 0-0
site_id 1

tracking_info_end

scan_test
failures_begin
1002 tdo 142 H L
1003 tdo 142 H L
1008 tdo 142 H L
1013 tdo 142 H L
1019 tdo 142 H L
raw 1028 tdo
0x03af2b406c28a2958c54b6c006103fccb6994a51d2103d27004d9e03c
raw 1030 tdo
0x160366f5c75512100400240029a83f13c012404903be4ad4c637de486
raw 1031 tdo
0x0d63395273b313f16800150028a03408b9e21c5f4393f1f36edc125c8
raw 1033 tdo
0x2255a6c9220f1ed040000e408c58185a401e05ea02b9d13e37e0c8b12
raw 1037 tdo
0x0f8199a3307dcca4c0000380007f97ce8013a97293cd8d95c1d82f200
first_pattern_applied 1000
last_pattern_applied 1037

failures_end

failure_buffer_limit_reached none

failure_file_end
```

Converting the Top-Level Failure File to a Core-Level Failure File

The Tessent Diagnosis tool performs diagnosis at the core level, and the Tessent SiliconInsight failure file is generated at the top level. Consequently, you must map the failure file from the chip level to the core level using the dlogutil, a Tessent Diagnosis utility.

Note

This step demonstrates using dlogutil interactively to convert the failure file. Alternatively, you can include these commands in a dofile and pass the dofile name to the dlogutil during invocation—see "dlogutil Utility".

Prerequisites

- A logic BIST fail mapping file (*.lbist_chain_info*)—see "Preparing the Logic BIST Chain Mapping File"
- A Tessent SiliconInsight top-level failure file—see "Failure File Generation"

Procedure

1. From a Linux/UNIX shell, invoke the dlogutil utility using the following syntax:

   dlogutil

2. Set the failure type to lbist using the following command:

   set fail_type lbist

3. Load the logic BIST fail mapping file (*.lbist_chain_info*) using the load_fail_map command. For example:

   dlogutil> load_fail_map BP4.WBP0.lbist_chain_info

4. Load the Tessent SiliconInsight top-level failure file using the map_fail_log command. For example:

   dlogutil> map_fail_log top_level_failure_file.flog -out core.flog -replace

   Once you have loaded the top-level failure file, the dlogutil utility converts the file to a core-level failure file.

5. Exit the tool.

Results

The default name of the converted core-level failure file is *failure_file_name*.core.

Related Topics

Preparing the Logic BIST Chain Mapping File

Failure File Generation

Including User-Defined Auxiliary Flops in the Conversion

Including User-Defined Auxiliary Flops in the Conversion

When converting the top-level failure file into a core-level failure file, dlogutil must consider any user-defined auxiliary flops so that it can adjust the failing cycle properly and produce a valid failure file that Tessent Diagnosis can correctly validate and diagnose.

Prerequisites

- A logic BIST fail mapping file with the *.lbist_chain_info* suffix.

- A separate optional file that contains the user-defined auxiliary flops formatted as described below.

Procedure

1. Ensure that the file that contains the user-defined auxiliary flops lists the indices for all of the flops, one index per line, and that the file resides in the same directory as the logic BIST fail mapping file.
2. Append the *.lbist_masked_flop_indexes* suffix to the file that lists the auxiliary flops. For example, for the following logic BIST fail mapping file:

   ```
   ./lvdb/LV_WORKDIR/BP6.WBP0.lbist_chain_info
   ```

   Name the optional auxiliary flop file:

   ```
   ./lvdb/LV_WORKDIR/BP6.WBP0.lbist_masked_flop_indexes
   ```

Results

After loading the logic BIST fail mapping file, dlogutil automatically loads the optional auxiliary flop file if it exists and is named correctly. Tessent Diagnosis uses the auxiliary flop information to adjust the cycle offset during failure file translation.

If a .lbist_masked_flop_indexes file is not found, dlogutil assumes that there are no auxiliary flops and does not perform cycle offset adjustments during failure file translation.

Related Topics

Converting the Top-Level Failure File to a Core-Level Failure File

Performing Logic BIST Diagnosis in Tessent Diagnosis

In general, you perform logic BIST diagnosis with Tessent Diagnosis using the standard diagnosis flow.

Note
You must be sure you set the number of logic BIST patterns in the Tessent Diagnosis dofile to the number used in the LVDB. Alternatively, you can also load a LDB to perform layout-aware diagnosis, which can greatly improve diagnosis resolution.

Prerequisites

- At run time, scan diagnosis requires a design netlist (flat model), patterns, and a failure file. For logic BIST diagnosis, you use the following as input files to scan diagnosis:
  - Design netlist — Use the read_flat_model command to specify the flat Tessent FastScan model.

- Patterns — Use the set_pattern_source command's BIST literal as follows:

  ```
  set_pattern_source bist
  ```

- Failure file — Use the core-level failure file you converted with the dlogutil. See "Converting the Top-Level Failure File to a Core-Level Failure File."

Procedure

Invoke the tool in Tessent Shell, executing a dofile. For example:

```
% tessent -shell -dofile td.dofile -log td.log -replace
```

where, the contents of the td.dofile are as follows:

```
set_pattern_source bist
//optional if you have the layout-aware database
open_layout flat.v.lvdb
// -num_lbist_patterns sets up the maximum number of logic BIST patterns that
// can be used in diagnosis. It's correlated to the "max: 32000" in "Execute
// and Debug Option".
set_diagnosis_options -num_lbist_patterns 32000
// in case it is defined in FastScan dofile but not saved in flat model
set_split_capture_cycle on
diagnose_failures top_level_failure_file.flog -out diag.rep -replace
```

Chapter 6
Running Tessent Diagnosis Server

The Tessent Diagnosis server enables you to configure and run diagnosis automatically on multiple failure files produced by multiple testers while testing one or more designs and eliminates the need for scripting.

Introduction to Tessent Diagnosis Server

Tessent Diagnosis server can be set up to detect failure files in one or more directories and run diagnosis on them automatically. These are called *monitored directories*, or *monitors*. Once failure files are detected, Tessent Diagnosis server checks them for syntax errors and redundancy and then automatically runs diagnosis on them. The failure files are sorted by size, and the smallest ones are processed first.

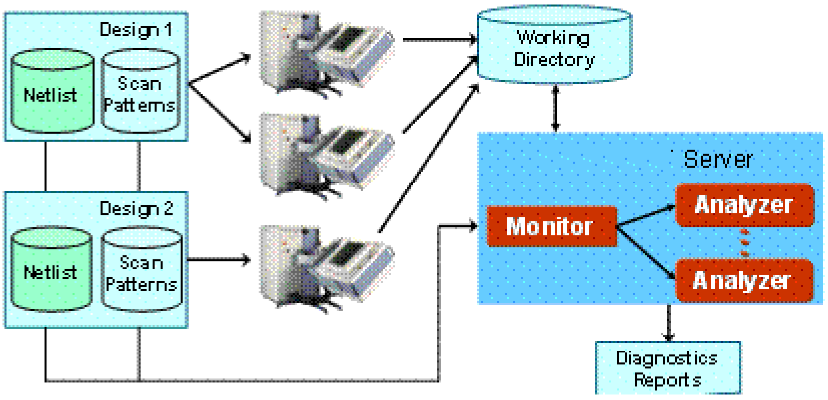
A design netlist and pattern set are associated with each monitored directory. When failure files are detected in a monitored directory, the associated design netlist and pattern set are used to diagnose them.

Depending on the available system resources, Tessent Diagnosis server can be configured to monitor multiple directories and use multiple Tessent Diagnosis server diagnosis engines that are the analyzers. Each analyzer is equivalent to one licensed copy of the Tessent Diagnosis server tool. The workload is spread equally among all available analyzers to optimize diagnosis throughput.

The diagnosis results are then output into a directory where they are available for viewing and subsequent analysis.

Figure 6-1 illustrates a typical Tessent Diagnosis server diagnosis configuration.

Figure 6-1. Tessent Diagnosis Server Configuration Example



Tessent Diagnosis Server Prerequisites

To use Tessent Diagnosis server you must satisfy several prerequisites such as system directory, input file, and licensing requirements.

The general requirements for using Tessent Diagnosis server are:

- One or more system directories exist that Tessent Diagnosis server can be set up to monitor.
- One or more test pattern file(s) associated with the design under test (DUT). For more information, see "Preparing the Test Patterns."
- Flat design netlist for each set of patterns associated with the DUT. For more information, see "Preparing the Design Netlist."
- You should successfully complete diagnosing at least one failure log using the manual mode of Tessent Diagnosis server. This is required to ensure all input files are compatible so the diagnosis can run automatically. For more information on using the manual mode of Tessent Diagnosis server, see "Running the Diagnosis."
- A Tessent Diagnosis license is available for each analyzer engine in your configuration. One license is required to set up automatic diagnosis and run one analyzer. Each additional analyzer requires an additional license.
- If you are performing cell-aware diagnosis, you also need the Tessent Diagnosis Cell-Aware Op SW license(s). The server only checks out a license when an analyzer is performing cell-aware diagnosis. The server releases the license when the cell-aware diagnosis job completes and the analyzer fall idles, the queue is empty, or the analyzer migrates to another monitor to pick up non-Cell-Aware jobs.

  Note

  The server checks out Tessent Diagnosis licenses when you specify the add_analyzer command. However, it checks out Tessent Diagnosis Cell-Aware Op SW licenses when you specify start_diagnosis later in the flow. If you have insufficient Cell-Aware licenses, the server may hang at this time.

- Before using Tessent Diagnosis server, you must have converted ATE failure logs. One of the following methods must be in place to convert ATE failure logs into a failure file format compatible with Tessent Diagnosis server:
  - Manual conversion: Manual or other method to convert the ATE failure logs. For more information, see "Guidelines for Preparing the ATE Failure File."
  - Automatic conversion: Perl script to automate the conversation as described in "Automated ATE Failure Log Conversion."

Creating the Scratch Directory

During invocation, the tool creates the *scratch* directory in the temporary directory.

It uses the following syntax to create the *scratch* directory:

```
.tessent.tmp.username.hostname.process_id
```

When you exit, the tool deletes this *scratch* directory.

By default, it stores the scratch directory in the location identified by the TMPDIR environment variable. If you have not defined the TMPDIR environment variable, the tool creates the scratch directory in the /*tmp* directory or in the current directory if the /*tmp* directory is not available.

You can explicitly specify the location where the temporary *.tessent.tmp.username.hostname.process_id* directory is created by setting the TESSENT_TMP_LOCATION environment variable. The tool creates the temporary *.tessent.tmp.username.hostname.process_id* directory in the default location if the location specified by the TESSENT_TMP_LOCATION environment variable does not exist, is not a valid directory, or does not have write permission. The tool uses the TESSENT_TMP_LOCATION environment variable when both TMPDIR and TESSENT_TMP_LOCATION exist.

Automated ATE Failure Log Conversion

The *ya_convert.pl* is a Perl script you create for automating ATE failure log conversion into a file format compatible with the Tessent Diagnosis server. When you put the *ya_convert.pl* script in the monitor directory, the Tessent Diagnosis server invokes the *ya_convert.pl* script that performs conversion on each incoming failure file.

You create the *ya_convert.pl* Perl script to automate the following handling of ATE failure logs:

- Converting ATE failure logs to a format compatible with Tessent Diagnosis—see "Guidelines for Preparing the ATE Failure File."
- Modifying ATE failure logs.

After you have created the *ya_convert.pl* script in Perl, you put the script in a monitored directory you have specified with the add_monitor command. When the Tessent Diagnosis server detects the *ya_convert.pl* script in the monitored directory, Tessent Diagnosis server invokes this script and performs the conversion on each incoming failure file using the following command line convention:

```
ya_convert.pl failure_file_name failure_file_name.ya
```

Where:

- *failure_file_name* — The name of the incoming ATE failure file. For these input files, you cannot use the .ya extension for the file name.
- *failure_file_name*.ya — The name of the resultant ATE failure file that Tessent Diagnosis server converts or that it modified using the *ya_convert.pl* script.

When you write your *ya_convert.pl* Perl script, you should adhere to the following guidance:

- Moving or deleting input failure files — There is no requirement to move or delete the input ATE failure file.
- Naming the converted failure file — You must construct your Perl script to write the converted output file as *failure_file_name.ya*
- Handling conversion errors — You should build in error handling into your Perl script. For instance if you have conversion errors, then your script should not create the output failure file.

Tessent Diagnosis Server Interface

Tessent Diagnosis server uses commands within the Tcl scripting language. When creating scripts and dofiles for Tessent Diagnosis server, or issuing instructions from the command line from within a Tessent Diagnosis server session, you must use Tcl-compliant syntax.

Refer to "The Tessent Tcl Interface" in the *Tessent Shell User's Manual* for guidelines for using the Tessent Tcl interface, include common issues you could encounter when using the Tcl programming language.

It is recommended that you take advantage of one or more of the excellent books and websites on the language. The following URL is provided to give you a place to start in your search for the reference material that works best for you. It is not an endorsement of any book or website.

`www.tcl.tk/`

Monitored Directories and Analyzers

Monitored directories are associated with the test pattern files and design netlists necessary to perform the diagnosis on failure files that are placed inside.

You can add as many analyzers as you have Tessent Diagnosis licenses; one analyzer is equivalent to one invocation of the Tessent Diagnosis scan diagnosis tool.

Guidelines for Working with Monitored Directories

You can add, delete, or report on specific monitored directories and the associated design netlists and test pattern files.

- add_monitor — Adds a monitored directory to the Tessent Diagnosis server configuration.
- delete_monitor — Deletes a monitored directory from the Tessent Diagnosis server configuration.
- add_design — Associates a specified design netlist with a specified monitored directory.
- delete_design — Deletes the associated design netlist from a specified monitored directory.
- add_pattern — Associates a specified test pattern file with a specified monitored directory.
- delete_pattern — Deletes the associated test pattern file from a specified monitored directory.
- report_monitor — Displays a report on a specified monitored directory including the associated test pattern file and design netlist. A report similar to the following displays for each monitored directory:

```
--- monitors ------------------------------
(1) Alpha
directory: /user/directory/testerAA
results: /user/directory/TESTERA_RESULTS
: adb.sql INACTIVE
design: /user/directory/design.flat.gz
pattern set 0: /user/directory/tester_pat.wgl
48 total (10 duplicates)
0 done [ 0% ]
0 errors ( 0% )
0 analyzed ( 0% )
0 diagnosed ( 0% )
48 queued failure files for 1 monitor
```

The following list describes the lines in the report in ascending order:

- (1) Alpha: indicates the order the monitored directory was defined and the user-defined alias for the directory. For example, 1 means the monitored directory was the first one added to the Tessent Diagnosis server configuration and Alpha is the monitored directory alias.
- directory: /user/directory/testerAA is the pathname of the monitored directory.
- results: /user/directory/TESTERA_RESULTS is the pathname of the directory where the associated diagnosis results are being written.
- design: /user/directory/design.flat.gz is the pathname of the design netlist associated with the monitored directory.
- pattern set 0: /user/directory/tester_pat.wgl is the pathname of the test pattern file associated with the monitored directory.
- 48 total (10 duplicates) is the total number of failure files in the monitored directory, and the number of these failure files that are duplicates.
- 0 done [0%] is the number of failure files processed.
- 0 errors (0%) is the number of errors encountered.
- 0 analyzed (0%) is the number of failure files analyzed.
- 48 queued failure files for 1 monitor is the summary of all the failure files processed by all the monitors contained in the report.

Failure File Management

By default, Tessent Diagnosis server creates and uses three subdirectories in the specified monitor directory for managing processed failure files. These directories are:

- *completed.ya*
- *aborted.ya*
- *timeout.ya*

Failure files waiting to be processed remain in the monitor directory. Upon being processed successfully, failure files are moved from the monitored directory to the *completed.ya* subdirectory. Failure files that cannot be processed successfully are moved to the *aborted.ya* subdirectory. Any failure file that exceeds the diagnosis time limit is moved to the *timeout.ya* subdirectory. The diagnosis time limit is set by using one of these methods:

- set_diagnosis_options -wall_time_limit command argument
- set_diagnosis_options -time_limit command argument
- set the analyzer_timelimit server variable

You can turn off the management of failure files using the monitor_filer variable. This alternative mode of operation leaves all failure files in their original locations.

Diagnosis Failure File Sorting

Before and during diagnosis, if you add additional failure files to the target directory, Tessent Diagnosis Server dynamically re-sorts the failure files in the queue based on user-defined criteria. By default, the failure files are sorted based on file size, and the smallest failure file is targeted first.

You can change this default by using the following Tessent Diagnosis Server variable:

set faillog_sort_criteria *sort_order_value*

where *sort_order_value* is one of the following four possible choices:

- smallest_file — A literal that specifies re-queuing the failure files with the smallest size first. This is the default.
- largest_file — A literal that specifies re-queuing the failure files with the largest size first.
- oldest_file — A literal that specifies re-queuing the failure files with the oldest timestamp first.
- newest_file — A literal that specifies re-queuing the failure files with the newest timestamp first.

For example, the following Tessent Diagnosis Server command sequence re-queues the fail logs to process the smallest file first:

```
set faillog_sort_criteria smallest_file
add_analyzer localhost:1
add_monitor flogs ../flogs/ -results ../results/flogs.ya
add_design flogs ../src/design.flat.gz
add_pattern flogs ../pat/pat.gz
start_diagnosis
```

Guidelines for Working with Analyzers

One or more analyzers can be initially bound with a specific monitored directory, but once the directory is empty, the analyzers become free to diagnose failure files in other directories.

You can add, delete, or report on specific analyzers. For more information, refer to the following commands:

- add_analyzer— Adds an analyzer to the Tessent Diagnosis server configuration.

- delete_analyzer — Deletes an analyzer from the Tessent Diagnosis server configuration.
- report_analyzer — Displays a report on a specified analyzer. A report similar to the following displays:

```
---- analyzers monitoring ------------------------------
(1) wvbld03 #1 Alpha ... idle (35 secs) waiting to start diagnosis
(2) wvbld05 #1 Alpha ... idle (33 secs) waiting to start diagnosis
(3) wvbld07 #1 Beta ... idle (31 secs) waiting to start diagnosis
(4) wvbld09 #1 Beta ... idle (29 secs) waiting to start diagnosis
```

  The following list describes the fields in the first line from left to right:

  - (1) indicates the order an analyzer was defined. For example, 1 means the analyzer was the first one added to the Tessent Diagnosis server configuration and so on.
  - wvbld03 is the name of the host computer running the analyzer. Multiple analyzers may be running on the same host.
  - #1 Alpha is the name of the monitored directory to which the analyzer is currently assigned.
  - idle (35 secs) describes the current state of the analyzer and the number of seconds the analyzer has occupied the state.
  - waiting to start diagnosis describes the state of the analyzer.

Setting Up the Tessent Diagnosis Server

You must configure Tessent Diagnosis server to automatically diagnose failure files as they are detected in specific directories. The purpose of this step is to create the LDB that you need to complete the diagnosis.

The user interface remains active during diagnosis allowing you to reconfigure settings, display session statistics, and interrupt diagnosis at any time during the session.

Prerequisites

- Refer to "Tessent Diagnosis Server Prerequisites" on page 336.

Procedure

1. Invoke Tessent Diagnosis server. From a Linux shell, enter:

```
Tessent_Tree_Path/bin/tessent -diagserver
```

   where:

   - *Tessent_Tree_Path* — The path to where the Tessent Diagnosis application tree is installed.
   - -diagserver — A required switch that invokes Tessent Diagnosis in server mode.

By default, a logfile *DiagServer.log* of the server diagnosis session is saved in the directory Tessent Diagnosis server from where it is invoked. If a default logfile from a previous session exists, it is overwritten. The logfile contains version information and the sequence of commands and their outputs for the entire session.

For more information on invoke options, see the tessent shell command in the *Tessent Shell Reference Manual*.

2. Define the monitored directories. From the Tessent Diagnosis server command prompt, enter the add_monitor command:

   add_monitor monitor_id monitored_directory

   where:

   - *monitor_id* — An alternate name you must provide for the monitored directory. This alias is used as shorthand when you refer to this monitored directory from other commands.
   - *monitored_directory* — The pathname of the directory for Tessent Diagnosis server to monitor.

   You can define multiple monitored directories. You can also specify a pathname for the diagnosis results.

3. Load the design netlist(s). From the Tessent Diagnosis server command prompt, enter the add_design command:

   add_design monitor_id flat_design_netlist

   where:

   - *monitor_id* — The alias that identifies a monitored directory. The alias is assigned with the add_monitor command in Step 2.
   - *flat_design_netlist* — The pathname of the flat design netlist to use for diagnosing the failure files found in the specified monitored directory. If a design netlist is encrypted, you are prompted for a password before the design is loaded.

   The same design netlist can be loaded for multiple monitored directories.

4. Load the test pattern file(s). From the Tessent Diagnosis server command prompt, enter the add_pattern command:

   add_pattern <monitor_id> <test_pattern_file>

   where:

   - *monitor_id* — The alias that identifies a monitored directory. The alias is assigned with the add_monitor command in Step 2.
   - *test_pattern_file* — The pathname of the test pattern file to use for diagnosing the failure files found in the specified monitored directory. The same test pattern file can be loaded for multiple monitored directories.

Refer to the following Examples section for details about loading multiple test patterns with multiple test suites.

5. Add the analyzers. From the Tessent Diagnosis server command prompt, enter the add_analyzer command:

   ```
   add_analyzer host_name
   ```

   where:

   - *host_name* — The name of the host computer to invoke and run the analyzer on. Use localhost:1 to create a single, non-floating, local, analyzer process.

   This command invokes a floating analyzer that is controlled by the workload. You can add several analyzers, specify a job scheduler, or initially bind the analyzer to a specific monitored directory with this command.

   Analyzers can be added across different operating systems. For example: Invoke Tessent Diagnosis server on a machine running the Sun operating system and then, add an analyzer that resides on a machine running the Linux operating system.

Results

You now have a LDB that you can use as input to peform diagnosis with the Tessent Diagnosis server.

Examples

If you use multiple test patterns with multiple test suites, then you must associate a specific pattern with the add_pattern command to a specific test suite.

For example, you have the following three multiple test suites entries in your failure file (*fail_file.log*):

```
...
test_suite_begin suite1
failures_begin
67386 PIN_15 H L
203562 PIN_15 H L
...
failure_buffer_limit_reached all
failures_end
total_cycles 2491916
test_suite_end

test_suite_begin suite2
failures_begin
192214 PIN_15 H L
532654 PIN_15 H L
...
failure_buffer_limit_reached all
failures_end
total_cycles 2491916
test_suite_end


test_suite_begin suite3
failures_begin
430522 PIN_15 H L
532654 PIN_15 H L
...
failure_buffer_limit_reached all
failures_end
total_cycles 2491916
test_suite_end
```

And, you have three different patterns sets, one for each of the test suites. In the server dofile, you identify these patterns with the add_pattern command using the following method:

add_monitor monitor1 fail_file.log -result fail_file.ya

add_design monitor1 my_design.flat

add_pattern monitor1 patterns/pattern_set_for_suite1.wgl.gz

add_pattern monitor1 patterns/pattern_set_for_suite2.wgl.gz

add_pattern monitor1 patterns/pattern_set_for_suite3.wgl.gz

When using multiple patterns with multiple test suites, you must list the add_pattern commands so the commands line up with the intended fail file or test suite in a failure file. For example, if your failure file contains the following test suites:

```
test_suite_begin suite1
...
test_suite_end

test_suite_begin suite2
...
test_suite_end
```

and you want to use “patterns1” with suite1 and “patterns2” with suite2, then you list the add_pattern commands in the following order in the dofile:

add_pattern monitor1 patterns1

add_pattern monitor1 patterns2

See also “Multiple Test Suite Failure Data.”

Running the Diagnosis

The start_diagnosis command assigns analyzers to the specified monitored directories and launches the automatic monitoring and diagnosis of the associated failure files.

See Table 6-13 — Tessent Diagnosis Server Variables for a list of variables that you can use with Tessent Diagnosis server.

Prerequisites

- Refer to “Setting Up the Tessent Diagnosis Server.”

Procedure

1. From the Tessent Diagnosis server command prompt, enter:

   start_diagnosis monitor_id

   The monitor_id is the alias that identifies a monitored directory. The alias is assigned with the add_monitor command in step 2 of the procedure described in “Setting Up the Tessent Diagnosis Server.” You can start diagnosis on multiple monitors by entering multiple space-separated monitor IDs.

2. As needed, suspend and resume Tessent Diagnosis server by using the following commands:

   - suspend_diagnosis — Completes any diagnosis in process and stops initiating any new diagnoses.

- resume_diagnosis — Resumes diagnosing failure files where it left off when it was suspended.

  When you resume the diagnosis, the Tessent Diagnosis server picks up where it left off. For example, if 2523 out of 5000 failure files completed diagnosis before the diagnosis was stopped, Tessent Diagnosis server resumes diagnosing failure file 2524 and a message similar to the following displays:

  ```
  //  command: add_analyzer hercules -monitor design1_fs skipped
  //  2523 previously diagnosed files.
  ```

3. Exit Tessent Diagnosis server as follows:

   exit

   When you exit Tessent Diagnosis server, all in-progress diagnoses are completed, monitoring of directories is discontinued, and analyzers are shut down.

   Depending on the active transactions, Tessent Diagnosis server shuts down after a few moments.

   If you terminate Tessent Diagnosis server without using the exit command, you must clean up any .lock files left in the monitored directories. The presence of .lock files in a directory prevents another Tessent Diagnosis server session from monitoring the directory.

Results

Refer to "The Diagnosis Results Directory" for information. You can also view the server session status by using the watch command. See "Server Session Status" for more information.

Related Topics

Running Tessent Diagnosis Server with a Local Host

Running Tessent Diagnosis Server in Batch Mode

The Tessent Diagnosis Server Daemon

Layout-Aware Diagnosis with the Tessent Diagnosis Server

You can perform layout-aware diagnosis and hierarchical layout-aware diagnosis with the Tessent Diagnosis Server using an existing LDB you have created with the Tessent Diagnosis tool.

To perform layout-aware diagnosis, you must have the following input files:

- Flat model of your design
- LDB
- Test Patterns
- ATE Failure Log Files

See "Layout-Aware Diagnosis Flow" for an overview of the layout-aware diagnosis flow and steps to create a LDB as well as other requirements. See "Diagnosis for Hierarchical Designs" for information about performing hierarchical layout-aware diagnosis.

Note
You must create the LDB with the Tessent Diagnosis tool.

Layout-Aware Diagnosis Commands

To perform layout-aware diagnosis with the Tessent Diagnosis Server, you must specify the LDB using the add_layout command with the -dft switch.

The following dofilea illustrates the layout-aware diagnosis flow:

```
add_analyzer generic:2
add_monitor monitorA my_failure_files -results results/bfails
add_design monitorA my_design.flat
add_pattern monitorA my_patterns.ascii
add_layout monitorA -dft my_layout_aware_file.ldb
start_diagnosis monitorA
```

Running Layout-Aware Diagnosis on a Local Layout Database

You can configure the Tessent Diagnosis Server to perform layout-aware diagnosis with a locally cached copy of an existing LDB. When using this method, the Tessent Diagnosis Server instructs the analyzer to copy the LDB to a host machine's local disk, and the analyzers on the host machine use this copy of the LDB for diagnosis.

Note
When using LSF as the job scheduler, the tool automatically determines the required disk space for the LDB copy and passes the proper resource requirement to LSF using the bsub command, such that only the host machine with the necessary resource is selected for this job. For example: set generic_scheduler "bsub -q normal -o /dev/null %command"

Prerequisites

- You must have a locally cached copy of the LDB.
- You have configured the Tessent Diagnosis server and, optionally, a job scheduler as described in "Setting Up the Tessent Diagnosis Server" and "Distributed Diagnosis Processing," respectively.

Procedure

1. Specify the location of the local LDB. Use the add_layout command's optional -copy /tmp/*local_disk_directory* switch/string pair as in the following example:

   add_layout monitorA -dft my_layout_aware_file.ldb -copy /tmp/user

   Tessent Diagnosis Server instructs an analyzer to copy the LDB to the /tmp/*local_disk_directory* location and create a unique subdirectory for the LDB. The tool creates this subdirectory using the following method:

   ```
   TDS-host_name-processID-monitor
   ```

   where:

   - TDS — Fixed value specifying the Tessent Diagnosis Server.
   - *host_name* — Host name running the Tessent Diagnosis Server.
   - *processID* — Process ID for the Tessent Diagnosis Server process.
   - *monitor* — The name of the monitor.

   For example:

   /tmp/user/TDS-my_host-12345-my_monitor/my_layout_database.ldb

2. Run the diagnosis:

   run_diagnosis

Results

Tessent Diagnosis server results are written as both an ASCII report and CSV (comma separated values) data—see "Diagnosis Reporting."

Examples

The following dofile example illustrates using the method using a custom (generic) scheduler.

```
set generic_scheduler "bsub -q normal -o /dev/null %command"
set generic_delete "bkill %Job"
add_analyzer generic:2
add_monitor flogs flogs/ -results ./results/flogs2.ya
add_design flogs src/design.flat.gz
add_pattern flogs pat/pat.gz
add_layout flogs -dft "/filer01/layout/YA_demo.db" –copy /tmp/
start_diagnosis
delete_analyzer 1     // assumes the analyzer1 lands on lsf1
exit
```

In the example, the Tessent Diagnosis Server performs the following operations:

- Schedules two analyzers through the LSF job scheduler. Assume that analyzer1 (2) lands on the machine lsf1 and lsf2.
- Assigns a monitor (flogs) for the failure logs, and the design and patterns.
- Creates local directories on lsf1 and lsf2 with the following name (assume the host is my_host and the Tessent Diagnosis Server process ID is 12345):

  /tmp/TDS-my_host-12345-flogs/YA_demo.db

  The analyzers use these local copies of the LDB for layout-aware diagnosis.
- When the tool deletes the analyzer (1) on lsf1, the tool also removes the local copy of the LDB. The tool also does the same for lsf2 when the tool exits after finishing the diagnosis of all the failures.

Dynamic Partitioning-Based Diagnosis

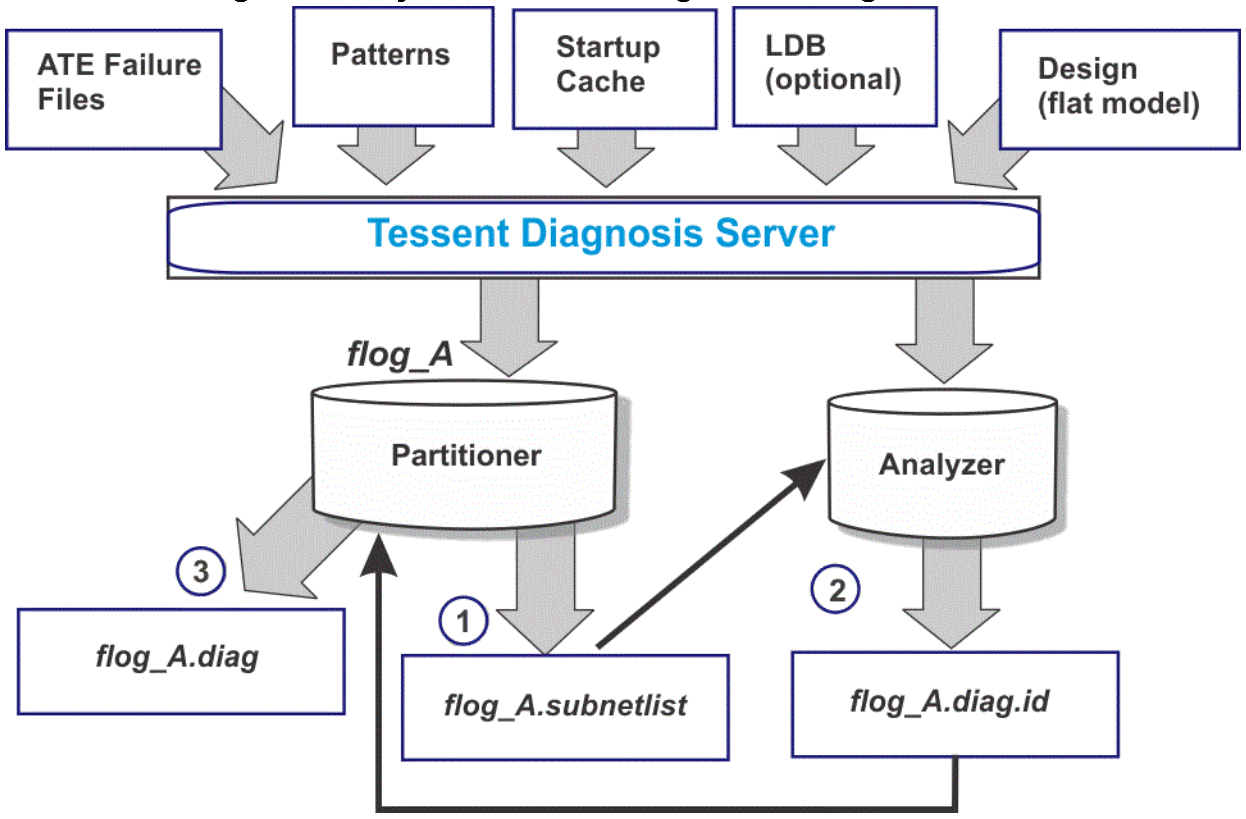
By default, the Tessent Diagnosis server spreads the diagnosis workload equally between all available analyzers. Each analyzer contains the entire netlist in memory. To lessen the amount of memory required to run a diagnosis, and to lessen runtimes, you can use dynamic partitioning-based diagnosis to partition the netlist into smaller subnetlists for each failure file and perform diagnosis on the subnetlists.

Overview

When you enable dynamic partitioning-based diagnosis, Tessent Diagnosis server initializes a dynamic partitioner. The partitioner generates partitions for failure files, which are then distributed to analyzers for processing.

Figure 6-2 shows the Tessent Diagnosis server flow for dynamic partitioning-based diagnosis.

Figure 6-2. Dynamic Partitioning-Based Diagnosis Flow



Given a failure file, flog_A, dynamic partitioning-based diagnosis performs as follows:

1. The partitioner analyzes flog_A and generates a flattened model of the subcircuit, flog_A.subnetlist.
2. The analyzer receives flog_A.subnetlist and performs diagnosis on this subcircuit for the failure file. The analyzer generates an ID-based diagnosis report, flog_A.diag.id.
3. The partitioner receives flog_A.diag.id, translates the ID-based names to string-based names, and generates the final diagnosis report, flog_A.diag.

In the dynamic partitioning-based diagnosis flow, the partitioner contains the entire netlist in memory, and hence, requires a suitable host machine with enough physical memory. Meanwhile, each analyzer contains only a subnetlist in memory, which means that each analyzer can run on a host machine with much smaller physical memory. Alternatively, for a machine with larger physical memory, multiple analyzers can run utilizing more of the CPU cores.

Prerequisites

- At least one analyzer with enough memory to host the partitioner.
- A startup cache.
- If you are performing layout-aware diagnosis, a layout database (LDB) that contains bridges and net topology information.
- You have tested the input files to ensure that they are correct.

Limitations

- It only supports one monitor per server run.
- The partitioners need extra startup time before they can create design partitions and assign jobs to analyzers.
- At-speed, clock, scan enable and compound diagnosis are not supported.

Aborted Failure Handling

Dynamic partitioning-based diagnosis aborts in the following situations:

- The sub-netlist for a failure file includes more than 20% of the gates of the original netlist. Diagnosis aborts because an analyzer with limited memory may not be able to handle such a large sub-netlist.

  Failures that abort because of oversized sub-netlists could mean that dynamic partitioning-based diagnosis is not suitable for this data set. In this case, you should use the normal Tessent Diagnosis server flow for diagnosis.

- Memory usage while diagnosing a failure on an analyzer exceeds the predefined memory limit. Diagnosis aborts to avoid crashing the analyzer.

  The tool moves aborted failure files to a directory named "oversized.ya" under the failure file directory. You can copy these aborted failures to a new directory and use the normal Tessent Diagnosis server to process them.

Preparing for Dynamic Partitioning-Based Diagnosis

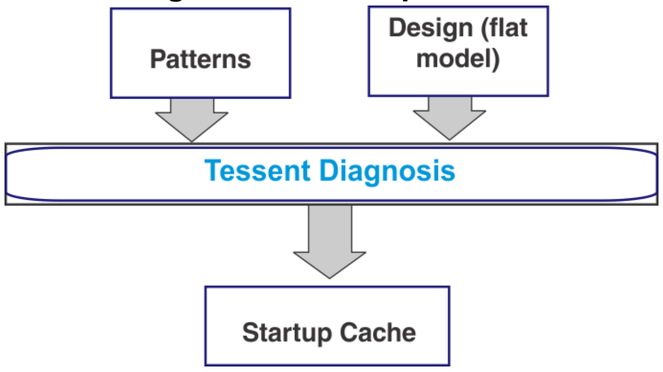
To run dynamic partitioning-based diagnosis, you must have a startup cache. If you plan to perform layout-aware diagnosis, you also must have an LDB that contains bridges and net topology information. You should also test the input files using Tessent Diagnosis before running dynamic partitioning-based diagnosis to ensure that the input files are correct.

Creating the Startup Cache

The startup cache enables you to run pattern verification only once, and it eliminates the need for each analyzer to perform additional pattern verification. Additionally, having a startup cache enables the partitioner to skip pattern verification and clock signature computation, thus improving performance.

Use Tessent Diagnosis (not Tessent Diagnosis server) to create the startup cache. Figure 6-3 shows the Tessent Diagnosis flow for creating the startup cache.

Figure 6-3. Startup Cache



Prerequisites

- A design (flat model)
- Pattern file(s)

Procedure

1. Inform the tool that the startup cache to be created is for dynamic partitioning-based diagnosis.

```
set_diagnosis_options -dynamic_partition master
```

You must specify this command before you load the test patterns. If there are existing patterns, use the delete_pattern -external command before issuing the set_diagnosis_options -dynamic_partition master command.

2. Read in your pattern file. For example:

   read_patterns ./pat/pat.stil.gz

3. Verify the pattern and create the startup cache. For example:

   verify_pattern -create_startup_cache ./design.startup_cache

Results

Tessent Diagnosis saves the pattern verification information and clock signature information into the startup cache database. In addition, it converts pattern files into binary patterns that are used by analyzers. The tool saves the newly converted binary pattern files in the same directory as the startup cache. If the original patterns are already binary patterns, they are still saved to the new binary pattern file. The binary pattern files cannot be moved elsewhere because their location information is stored in the startup cache and used during dynamic partitioning.

The newly converted binary file ends with the suffix ".bin.gz". The source binarwy files do not have to end with a ".bin" or ".bin.gz" suffix. However, if the source binary file is XYZ.bin, the new pattern file is XYZ.bin.gz. If the source binary file is already XYZ.bin.gz, the new file remains XYZ.bin.gz.

Examples

The following example creates a startup cache:

```
set_context patterns -scan_diagnosis
read_flat_model ../src/design.flat.gz
set_diagnosis_options -dynamic_partition master
read_patterns ./pat/pat.stil.gz
verify_pattern -create_startup_cache ./design.startup_cache
```

Bridges and Net Topology Information

If you plan to perform layout-aware diagnosis, you need to generate an LDB that contains pre-extracted bridge and net topology information for all the nets in the design. This information is required when you perform layout-aware diagnosis along with dynamic partitioning-based diagnosis. It is not required for logic-only diagnosis or chain diagnosis.

By default the create_layout command generates an LDB that includes pre-extracted bridge and net topology information.

Note

To ensure proper net topology extraction for net VIAs, during LEF/DEF generation specify "do not flatten" for net VIAs.

After creating the LDB, you must validate it with the open_layout command. This ensures that the Tessent Diagnosis point tool knows to use the LDB during dynamic partitioning-based diagnosis.

Test the Input Files

To ensure that Tessent Diagnosis server does not hang during processing or return unexpected results, perform one diagnosis run with Tessent Diagnosis before performing dynamic partitioning-based diagnosis.

Load the required input data as shown in Figure 6-2, run diagnosis, and ensure that the input data is correct.

Running Dynamic Partitioning-Based Diagnosis

Similar to the default Tessent Diagnosis server flow, dynamic partitioning-based diagnosis uses monitored directories and analyzers. To enable dynamic partitioning-based diagnosis, specify one or more analyzers to use as partitioners by using the add_partitioner command.

Prerequisites

- You have completed the tasks as described in section "Preparing for Dynamic Partitioning-Based Diagnosis."
- A design (flat model), a pattern file or files, and ATE failure files.

Procedure

1. From a shell, run Tessent Diagnosis server using the following syntax:

   ```
   Tessent_Tree_Path/bin/tessent -diagserver
   ```

2. Define the monitored directory and required input files with the following series of commands. For example:

   add_monitor flogs ../flogs/ -results ../results

   add_design flogs ../src/design.flat.gz

   add_pattern flogs ../pat/pat.stil.gz

3. Specify the startup cache and LDB.

   add_startup_cache flogs ./design.startup_cache

   #specify the following command if you are using layout-aware diagnosis

   add_layout flogs -dft layout.ldb

4. Analyze the resource requirements. For example:

   analyze_resource_requirements

The analyze_resouce_requirements command specifies to the tool to analyze, calculate and report the memory requirement for each partitioner and analyzer for each configuration.

5. Specify the configuration and any other options. For example:

   set_diagnosis_resource_configuration -balanced -scheduler sge

   By default, the tool uses the balanced configuration and assumes the LSF scheduler.

6. Start diagnosis.

   start_diagnosis

   When you run the start_diagnosis command, the tool begins requesting the partitioner resource from the scheduler. The partitioner and analyzer resources are requested incrementally over the run. These requests are automatically made to the scheduler up to the maximum required diagnosis configuration.

Results

Tessent Diagnosis server generates one report per failure file. See "The Diagnosis Results Directory" for more information.

You can use the report_monitor command to show you how many failure files have been partitioned. For example:

```
//  command: report_monitor

---< monitors  >-------------------------------------------
(1)  flogs
   directory: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./flogs
   results: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./flogs.ya
   : yieldinsight.adb Inactive
   design: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/../src/design.flat.gz
   pattern set 0: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/../pat/pat.gz  Verified
   layout DB: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./YA_demo5.ladb/layout.ladb
   3 total
   3 done        [ 100% ]
   0 errors      (   0% )
   3 partitioned ( 100% ) 0 sec/partition
   3 diagnosed   ( 100% ) 3 sec/diagnosis
```

Examples

The following example runs the automated dynamic-partitioning flow. You do not need to determine the number of partitioners and analyzers. The tool automatically determines the number of DP-partitioners based on the number of failure files in the given dataset. If the targeted dataset has 40 failure files, then 4 DP-partitioners and upto 20 DP-analyzers are used. For another dataset with 8 failure files, single DP-partitioner and upto 10 DP-analyzers are used by using the same dofile. By default, the LSF grid is used to schedule all jobs. After “analyze_resource_requirements” successfully determines the job scheduling, the “start_diagnosis” command starts the DP-server flow to process this dataset. The flow stops when all failure files are processed and exits. You can insert extra Tcl commands into the Tcl while loop to print out more information about the progress for monitoring.

```
// Example2: Run new DP-server flow with partitioner load of 10 using LSF
grid
add_monitor monA flogs/ -results flogs.ya
add_design monA ../src/design.flat.gz
add_pattern monA ../src/pat.gz
add_startup_cache monA ../src/scdb/design.scdb
add_layout monA ../src/ldb

set_diagnosis_resource_configuration
start_diagnosis

while { [check -queued] > 0 && ![abort] } {
  // Use Tcl commands to print out more progress messages
}
report_monitor
report_analyzer
exit
```

Setting Up Manual Dynamic Partitioning-Based Diagnosis

You can manually set up the Tessent Diagnosis server flow, dynamic partitioning-based diagnosis that uses monitored directories, and analyzers. To enable dynamic partitioning-based diagnosis, specify one or more analyzers to use as partitioners by using the add_partitioner command.

Prerequisites

- You have completed the tasks as described in section “Preparing for Dynamic Partitioning-Based Diagnosis.”
- A design (flat model), pattern file(s), and ATE failure files.

Procedure

1. From a shell, run Tessent Diagnosis server using the following syntax:

```
        Tessent_Tree_Path/bin/tessent -diagserver
```

2. Define the monitored directory and required input files with the following series of commands. For example:

   ```
   add_monitor flogs ../flogs/ -results ../results
   add_design flogs ../src/design.flat.gz
   add_pattern flogs ../pat/pat.stil.gz
   ```

3. Specify the startup cache and (optionally) an LDB that contains bridges and net topology information.

   ```
   add_startup_cache flogs ./design.startup_cache
   #specify the following command if you are using layout-aware diagnosis
   add_layout flogs -dft layout.ldb
   ```

4. Add one or more partitioners. For example:

   ```
   add_partitioner big:2
   ```

   The add_partitioner command enables dynamic partitioning. Each partitioner can process a certain number of analyzers. Specifying more than one partitioner can decrease processing time by minimizing the number of analyzers that are sitting idle at any given time. You can add partitioners at any time. Additionally, you can delete partitioners at any time.

   If you manually choose the host machine for a partitioner, then it must have enough physical memory to accommodate the entire netlist, which is roughly the same as required for an average Tessent Diagnosis point tool run. If you choose to add the machine through a job scheduler such as SGE or LSF, then Tessent Diagnosis server automatically requests an appropriate machine.

   In addition, you can specify the -dp_work_dir switch to specify a working directory. A working directory can prevent file storage issues that can occur when the partitioned files accumulate in analyzer queues waiting for diagnosis. See "add_analyzer" for more information.

   Note
   Define the partitioners before you define the analyzers.

5. Add one or more analyzers. For example:

   ```
   add_analyzer lsf:10 -priority 3
   ```

   See "add_analyzer" for information about the -hibernation switch that you can use to specify the number of minutes before idle analyzers enter hibernation mode and relinquish their licenses.

6. Start diagnosis.

   ```
   start_diagnosis
   ```

After you run the start_diagnosis command, the partitioners perform a startup process before creating subnetlists to assign to analyzers. This startup process may take a long time depending on the design size. Therefore, it is recommended that you wait until after a partitioner has partitioned the first failure file before adding the analyzers. You can do this by adding the following tcl code to the dofile:

while { [check -partitioned] == 0 && ![abort] } { }

add analyzer sge:5

Results

Tessent Diagnosis server generates one report per failure file. See "The Diagnosis Results Directory" for more information.

You can use the report_monitor command to show you how many failure files have been partitioned. For example:

```
//  command: report_monitor

---< monitors  >--------------------------------------------
(1)  flogs
   directory: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./flogs
   results: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./flogs.ya
   : yieldinsight.adb Inactive
   design: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/../src/design.flat.gz
   pattern set 0: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/../pat/pat.gz  Verified
   layout DB: /wv/ststnttmp/TNT_TMPDIR_NETWORK/
gsub_3974.11.61354f25736ede5a80af2e9f39db70de/
YA_demo5_DPMS3N_template.yielda/results.64/./YA_demo5.ladb/layout.ladb
   3 total
   3 done         [ 100% ]
   0 errors       (   0% )
   3 partitioned  ( 100% ) 0 sec/partition
   3 diagnosed    ( 100% ) 3 sec/diagnosis
```

Examples

Example 1: Manual Dynamic Partitioning-Based Diagnosis Flow

The following example illustrates the manual dynamic partitioning-based diagnosis flow. In this example, you are performing layout-aware diagnosis so you are specifying a LDB that contains bridge and net topology information. In addition, you are specifying a working directory.

```
# define the monitored directory and input files
add_monitor flogs ../flogs/ -results ../results
add_design flogs ../src/design.flat.gz
add_pattern flogs ../pat/pat.stil.gz

# specify the startup cache and LDB
add_startup_cache flogs ./design.startup_cache
add_layout flogs -dft layout.ldb

# enable dynamic partitioning and add 2 DP primaries
add_partitioner lsf:2 -dp_work_dir ../flogs/work
report_monitor
report_analyzer

start_diagnosis
while { [ check -partitioned ] == 0 && ![abort] } { }

# add 6 DP secondaries after the first failure file is partitioned
# SGE asks machines with 6GB+ memory
add analyzer sge:6
```

Tessent Diagnosis automatically derives the number of gates in the design from the flat design model, and the LSF job scheduler uses this number to deliver the appropriate partitioners. The start_diagnosis command then causes the partitioners to perform the setup and create partitions for the failure files. As soon as the first partitions become ready on the partitioners, the server starts allocating the analyzers to them, and the analyzers begin diagnosis.

Example 2

The following example runs the dynamic partitioning-based flow in the manual mode with a user-defined number of dynamic-partitioning partitioners and dynamic-partitioning analyzers.

```
add_monitor monA flogs/ -results flogs.ya
add_design monA ../src/design.flat.gz
add_pattern monA ../src/pat.gz
add_startup_cache monA ../src/scdb/design.scdb
add_layout monA ../src/ldb

add_partitioner sge:1
add_analyzer sge:2

start_diagnosis
while { [check -queued] > 0 && ![abort] } { }
report_monitor
report_analyzer
exit
```

Server Session Status

From the Tessent Diagnosis server command prompt, enter the watch command to display the server session status.

The status of all the analyzers and monitored directories are combined and dynamically displayed to the screen in the following format.

```
// command: watch
---< monitors >------------------------------
(1) Alpha
directory: /wv/dft06918/pmc/testerAA
results: /wv/dft06918/pmc/TESTERA_RESULTS
: adb.sql INACTIVE
design: /wv/dft06918/pmc/pmc.flat.gz
pattern set 0: /wv/dft06918/pmc/tester_pat.wgl
48 total
7 done [ 15% ]
0 errors ( 0% )
7 analyzed (15%)
7 diagnosed ( 15% )
----< analyzers >------------------------------
(1) wvbld03 #1 Alpha ... idle (35 secs) diagnosing
(2) wvbld05 #1 Alpha ... idle (33 secs) diagnosing
(3) wvbld07 #1 Beta ... idle (31 secs) diagnosing
(4) wvbld09 #1 Beta ... idle (29 secs) diagnosing
----< status >------------------------------
48 queued failure files for 1 monitor
4 analyzers
```

For a description of the report, see "Guidelines for Working with Analyzers" and "Guidelines for Working with Monitored Directories." This report is not written to the session log.

Press **Enter** to terminate the dynamic reporting and display the command entry prompt.

The Diagnosis Results Directory

Tessent Diagnosis server results are written as both an ASCII report and CSV (comma separated values) data.

See "Diagnosis Reporting" for more information.

By default, Tessent Diagnosis server places the diagnosis results in *monitored_directory.ya*, where *monitored_directory* is the name of the associated monitored directory and *monitored_directory.ya* is created in the same directory where *monitored_directory* resides. You can override the default directory and specify a different directory for the diagnosis results by using the -results *results_directory* switch to the add_monitor command

For example, the following command places the diagnosis results from Tessent Diagnosis server into directory /A/B/C/my_results:

```
add_monitor foo failure_files -results A/B/C/my_results
```

Duplicated Failure File Names

In the event Tessent Diagnosis server processes different failure files with the identical file name, the tool appends each diagnosed failure file with a numerical suffix beginning with 2, and then increments this suffix for multiple failure files with the identical file name.

For example, Tessent Diagnosis server processed two different failure files with the identical name (failures). In the *results* directory, Tessent Diagnosis server creates and names the following diagnosis report files:

```
-rw-rw-r-- 1 user group  744 Jan 21 14:48 failures.csv
-rw-rw-r-- 1 user group  900 Jan 21 14:46 failures.csv.2
-rw-rw-r-- 1 user group 1259 Jan 21 14:48 failures.diag
-rw-rw-r-- 1 user group 1340 Jan 21 14:46 failures.diag.2
```

Note
The latest ATE failure file Tessent Diagnosis server processes keep the unaltered file name. (For example, failures.diag.) The older the file, higher is the appended number.

Log Files

During a Tessent Diagnosis server run, the tool produces a log file named "log" in the *results* directory.

If the tool encounters errors during the run, then multiple logfiles are appended with a number (beginning with 2) representing the order they were created. For example log, log.2, log.3 and so on.

If the tool encounters no errors during the run, then Tessent Diagnosis server creates an empty log file.

Server History

The history database (HDB) provides an SQL record of events that occurred during server operation.

You can instruct the Tessent Diagnosis server to create an HDB by specifying the -hdb switch. You can query the HDB to generate reports.

When invoking Tessent Diagnosis in server mode, create a new HDB as follows:

tessent -diagserver options -hdb *history_db_name.hdb*

Caution

The tool overwrites an HDB with no warning if you repeat the command using the same name.

The HDB supports session IDs. You can append an existing HDB by adding a "+" in front of an existing HDB name to indicate append mode. Append mode opens the HDB using a new session ID. Use the following command to append to the HDB:

tessent -diagserver options -hdb +*history_db_name.hdb*

See "tessent -diagserver" in the *Tessent Shell Reference Manual* for more information.

Server History Reports

The HDB stores data about various steps performed during the diagnosis and dynamic partitioning processes, which enables detailed time and memory reporting for different parts of the diagnosis process for each failure log. From this data, you can track diagnosis performance and make throughput comparisons between dynamic partitioning and baseline diagnosis runs.

See "History Database Schema" on page 372 for information about how the data within the HDB is organized.

To query the current HDB and generate reports, specify the report_history command. By default, the tool produces an event-based report that lists the various steps involved in diagnosis and dynamic partitioning (as applicable). For example:

```
// command: report_history
History Event Report
------------------------------------------------------------------------
flog flog_num event time memory PartitionSize
------------------------------------------------------------------------
flog1.cyc 1 PartitionCreation 23.5 1.53GB 60.31
flog1.cyc 1 PartitionMark 19.3 60.31
flog1.cyc 1 PartitionWrite 4.2 60.31
flog1.cyc 1 DesignLoad 1.19 0.73GB 60.31
flog1.cyc 1 OptionsLoad 0 0.73GB 60.31
flog1.cyc 1 PatternLoad 3.2 0.74GB 60.31
flog1.cyc 1 PatternVerification 0.03 0.74GB 60.31
flog1.cyc 1 OpenLayout 0.15 0.74GB 60.31
flog1.cyc 1 Diagnosis 51.42 0.91GB 60.31
flog1.cyc 1 ReportTranslation 0.03 1.55GB 60.31
flog2.cyc 2 PartitionCreation 2.5 1.55GB 46.65
flog2.cyc 2 PartitionMark 0.8 46.65
flog2.cyc 2 PartitionWrite 1.7 46.65
flog2.cyc 2 DesignLoad 0.87 0.70GB 46.65
flog2.cyc 2 OptionsLoad 0 0.70GB 46.65
flog2.cyc 2 PatternLoad 3.2 0.70GB 46.65
flog2.cyc 2 PatternVerification 0.05 0.70GB 46.65
flog2.cyc 2 OpenLayout 0.26 0.71GB 46.65
flog2.cyc 2 Diagnosis 23.87 0.85GB 46.65
flog2.cyc 2 ReportTranslation 0.05 1.55GB 46.65
---------------------------------------------------------------------
```

Each event that a failure log passes through during the diagnosis process is a separate row within the table.

To produce a report that consolidates the diagnosis events for each failure log and lists either the time or memory consumption for each event, specify report_history -pivot time or report_history -pivot memory, respectively. For example:

```
// command: report_history -pivot time
History Event Time Report
------------------------------------------------------------------------------------------
flog       PartitionCreation       PartitionMark       PartitionWrite       DesignLoad
  OptionsLoad    PatternLoad    PatternVerification    Diagnosis         ReportTranslation
-------------------------------------------------------------------------------------------
phyb2_adom_1     0.1     0     0.1     0.04     0     3.03     0.03     0.34     0.01
phyb2_adom_0     0       0     0       0.24     0     3.03     0.02     0.26     0.01
phy_open_1       0       0     0       0.26     0     3.04     0.02     0.41     0
phyb2_and_1      0       0     0       0.26     0     3.03     0.02     1.43     0.01
phy_open_0       0       0     0       0.26     0.01  3.03     0.02     1.05     0
phyb2_and_0      0       0     0       0.26     0.01  3.03     0.02     1.41     0.01
------------------------------------------------------------------------------------------
```

The report_history -pivot memory report also includes a PartitionSize column.

The following table describes the events. With the exception of PartitionCreation and ReportTranslation, the tool performs all steps regardless of whether dynamic partitioning is enabled. In the dynamic partitioning flow, the tool performs the events for each failure log. When you are not using dynamic partitioning, the tool only performs the Diagnosis step for each failure log; the other steps are performed once for each analyzer.

Table 6-1. report_history Events

Event	Description
PartitionCreation	When using dynamic partitioning, Tessent Diagnosis creates a partition from the design based on the failure log. This step occurs for each failure log and is omitted when you are not using dynamic partitioning.
DesignLoad	The analyzer loads the partition (portion of the design) on which diagnosis is performed.
OptionsLoad	The analyzer loads the diagnosis options that were specified with the set_diagnosis_options command. Additionally, it loads UDFM files for cell-aware diagnosis, as applicable.
PatternLoad	The analyzer loads the binary patterns that the tool generates when it creates the startup cache. See "Diagnosis Startup Cache" on page 32.
PatternVerification	The analyzer loads the pattern verification results stored in the startup cache and performs verification on the patterns.
OpenLayout	The analyzer opens the LDB.
Diagnosis	The analyzer performs diagnosis on the partition corresponding to the failure log.
ReportTranslation	The partitioner reads the diagnosis results generated by the analyzer and translates the results into a completed diagnosis report. This step is omitted when you are not using dynamic partitioning.
PartitionSize	This column lists the size of the partition created for the failure log as a percentage of the design.

If any other Tessent Diagnosis server run attempts to use an HDB owned by another process or with incorrect open status, the tool issues the following warning message with the session history:

```
// Warning:  history database is already opened by another session.
Session History:
session_id  begin_time                end_time                     host      process
1           Wed May 27 15:59:52 2009  Wed May 27 16:00:34 2009     myhost    14827
1           Wed May 27 16:00:40 2009  Wed May 27 16:03:18 2009     myhost    14973
1           Wed May 27 16:02:55 2009  Wed May 27 16:03:14 2009     myhost    15675
1           Wed May 27 20:35:49 2009  <<< STILL ACTIVE >>>         myhost    18216
```

In addition to the report_history command, use the following the commands to manage the HDB.

Command	Description
cleanse_history	Removes or lists the specified rows out of the HDB tables.
query_history	Creates a Tcl string result from the history database query.

Usage Example: Analyze Diagnosis Performance and Throughput

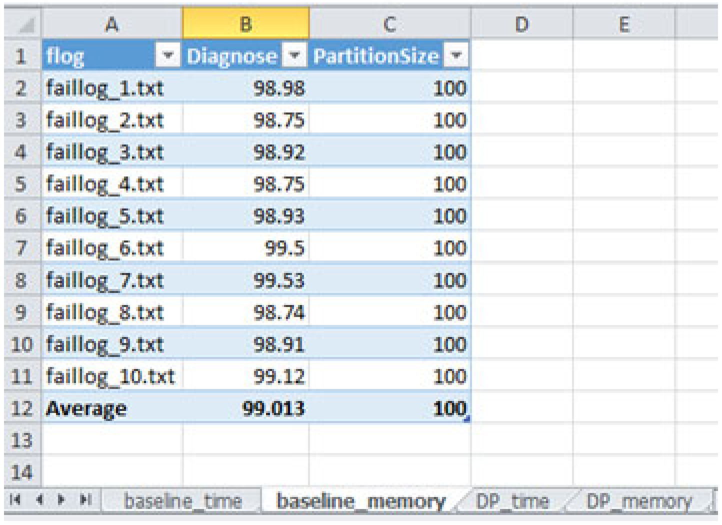
You can use the memory and time CSV-formatted history reports to analyze diagnosis performance. Generate the CSV-formatted reports with the report_history -csv command.

For example, you can compare the results of a baseline diagnosis flow against dynamic partitioning to calculate the performance improvement of using dynamic partitioning.

Suppose you have generated four history reports that you have saved to CSV files and then imported into a Microsoft Excel file (on their own sheets): time consumption for baseline diagnosis, time consumption for dynamic partitioning diagnosis, memory consumption for baseline diagnosis, and memory consumption for dynamic partitioning diagnosis.

Suppose you have also generated the average for each column of each report. The following figure shows a baseline diagnosis memory consumption table:

Figure 6-4. Memory Consumption CSV Report Imported into Excel



	A	B	C	D	E
1	flog	Diagnose	PartitionSize		
2	faillog_1.txt	98.98	100		
3	faillog_2.txt	98.75	100		
4	faillog_3.txt	98.92	100		
5	faillog_4.txt	98.75	100		
6	faillog_5.txt	98.93	100		
7	faillog_6.txt	99.5	100		
8	faillog_7.txt	99.53	100		
9	faillog_8.txt	98.74	100		
10	faillog_9.txt	98.91	100		
11	faillog_10.txt	99.12	100		
12	**Average**	**99.013**	**100**		
13					
14					

Calculating Total Diagnosis Time for Dynamic Partitioning

As shown in "Server History Reports" on page 366, the dynamic partitioning history reports provide visibility into the steps performed for each fail log. Compute the total time used by the partitioner and analyzers during the diagnosis process. To do this, add three more columns to the dynamic partitioning time consumption spreadsheet:

- PartitionerTotal: For each row, sum of the PartitionCreation step and the ReportTranslation step. In the following figure, for the first fail log, this equates to:

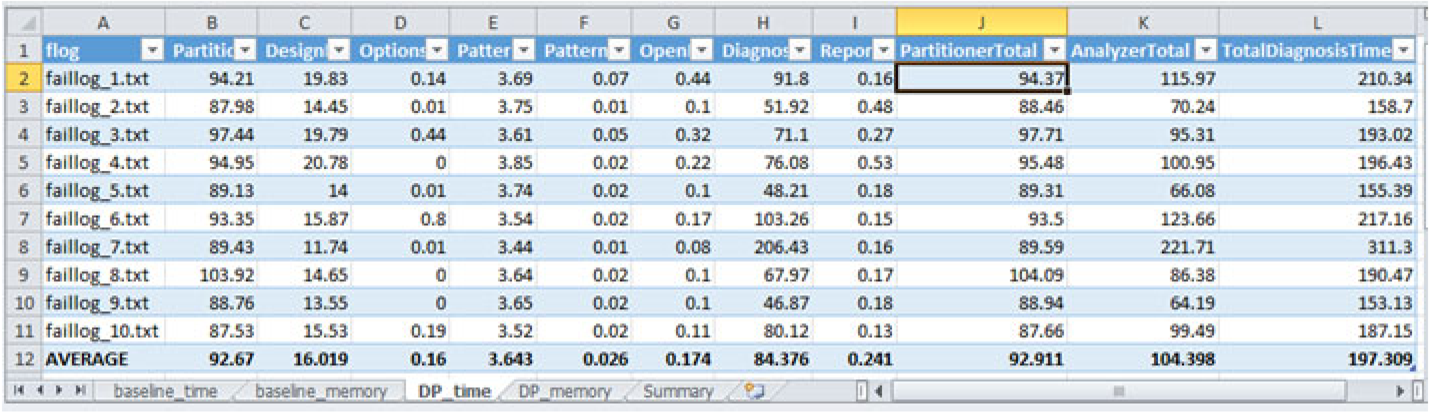
  ```
  =SUM(B2,I2)
  ```

- AnalyzerTotal: For each row, the sum of the remaining diagnosis steps—DesignLoad, OptionsLoad, PatternLoad, PatternVerification, OpenLayout, and Diagnosis. In the following figure, for the first fail log, this equates to:

  ```
  =SUM(Table4[@[ DesignLoad]:[ Diagnosis]])
  ```

- TotalDiagnosisTime: For each row, the sum of PartitionerTotal and AnalyzerTotal.

Figure 6-5. Dynamic Partitioning Time Consumption CSV Report Imported into Excel



	A	B	C	D	E	F	G	H	I	J	K	L
1	flog	Partitic	Design	Options	Patter	Pattern	Open	Diagnos	Repor	PartitionerTotal	AnalyzerTotal	TotalDiagnosisTime
2	faillog_1.txt	94.21	19.83	0.14	3.69	0.07	0.44	91.8	0.16	94.37	115.97	210.34
3	faillog_2.txt	87.98	14.45	0.01	3.75	0.01	0.1	51.92	0.48	88.46	70.24	158.7
4	faillog_3.txt	97.44	19.79	0.44	3.61	0.05	0.32	71.1	0.27	97.71	95.31	193.02
5	faillog_4.txt	94.95	20.78	0	3.85	0.02	0.22	76.08	0.53	95.48	100.95	196.43
6	faillog_5.txt	89.13	14	0.01	3.74	0.02	0.1	48.21	0.18	89.31	66.08	155.39
7	faillog_6.txt	93.35	15.87	0.8	3.54	0.02	0.17	103.26	0.15	93.5	123.66	217.16
8	faillog_7.txt	89.43	11.74	0.01	3.44	0.01	0.08	206.43	0.16	89.59	221.71	311.3
9	faillog_8.txt	103.92	14.65	0	3.64	0.02	0.1	67.97	0.17	104.09	86.38	190.47
10	faillog_9.txt	88.76	13.55	0	3.65	0.02	0.1	46.87	0.18	88.94	64.19	153.13
11	faillog_10.txt	87.53	15.53	0.19	3.52	0.02	0.11	80.12	0.13	87.66	99.49	187.15
12	**AVERAGE**	**92.67**	**16.019**	**0.16**	**3.643**	**0.026**	**0.174**	**84.376**	**0.241**	**92.911**	**104.398**	**197.309**

Comparing Diagnosis Throughput of Dynamic Partitioning and Baseline Diagnosis

Using the averages and the totals you added to the default reports generated by report_history, you can calculate the diagnosis throughput improvements when using dynamic partitioning relative to using baseline diagnosis. Throughput is generally defined as the product of memory usage and diagnosis run time.

Use the following series of calculations:

```
Baseline Throughput =

(Avg. Memory Use * Avg. Diagnosis Time)* N
```

where *N* is the number of fail logs.

```
DP Throughput =

(Avg. Memory Use Partitioner * Avg. Partitioner Total Time) +
((Avg. Memory Use Analyzer * Avg. Analyzer Total Time)* N)
```

where *N* is the number of fail logs run in parallel. That is, the ratio of analyzers to one partitioner. The average memory usage for the partitioner is the larger of the averages for the PartitionCreation task and the ReportTranslation task. The average memory usage for the analyzer is the average for the Diagnosis step because that is the most memory-intensive step for the analyzer.

Given the results for baseline throughput and dynamic partitioning throughput, you can calculate the throughput improvement of using dynamic partitioning as follows:

```
Improvement =

(Baseline Throughput * No. of fail logs supported by each partitioner)
______________________________________________________________________

                              DP Throughput
```

History Database Schema

The History Database (HDB) is an SQL database containing information in tables. Each table has a series of columns identified by names.

Table 6-2 lists the HDB tables.

Table 6-2. History Database Tables

Table Name	Description
HDB_ANALYZER	Contains analyzer data.
HDB_ERROR	Contains error data.
HDB_EVENT	Contains BeginSession, AcquireLicense, ReleaseLicense, LoadDesign, SendPatterns, VerifyPatterns, Parse, Diagnose, and ShutdownServer event data.
HDB_FILE	Contains file (flat design, patterns) information.
HDB_HEADER	Contains global HDB data.
HDB_MONITOR	Contains monitor data.
HDB_MONITORSET	Represents the state of the internal monitor of the server.
HDB_OPTION	Records the options the server uses for a diagnosis run.
HDB_QUEUE	Contains the state of the internal file queue of the Tessent Diagnosis server.
HDB_SESSION	Contains Tessent Diagnosis server startup data.

HDB_ANALYZER

Each row in the HDB_ANALYZER table represents a single processor added by an add_analyzer command.

Table 6-3 lists the HDB_ANALYZER table.

Table 6-3. HDB_ANALYZER Table

Type	Attribute	Description
analyzer_id	INTEGER	The primary key of the HDB_ANALYZER table.
analyzer_type	TEXT	Field showing whether you are dealing with “analyzer” or “partitioner”.
hostname	TEXT	The name of the computer the analyzer is running on. Example: my_host1:4, localhost:8
processor	INTEGER	Integer that specifies the number of analyzers added.
arch	TEXT	The host’s architecture. Example: sun4u
style	TEXT	One the of following job scheduler types: MANUAL LSF SGE
started	INTEGER	Date time stamp specifying the start of the job.

HDB_ERROR

The HDB_ERROR table records error information.

Table 6-4 lists the HDB_ERROR table..

Table 6-4. HDB_ERROR Table

Type	Attribute	Description
error_id	INTEGER	The primary key of the HDB_ERROR table.
event_id	INTEGER	The table row number from the HDB_EVENT table.
message	TEXT	ASCII text of the error message.

HDB_EVENT

The HDB_EVENT table records BeginSession, AcquireLicense, ReleaseLicense, LoadDesign, SendPatterns, VerifyPatterns, Parse, Diagnose, and ShutdownServer events.

Table 6-5 lists the HDB_EVENT table.

Table 6-5. HDB_EVENT Table

Type	Attribute	Description
event_id	INTEGER	The primary key of the HDB_EVENT table.
session_id	INTEGER	The primary key of the HDB_SESSION table for the event.
monitorset_id	INTEGER	The primary key of the HDB_MONITORSET table for the event.
analyzer_id	INTEGER	The primary key of the HDB_ANALYZER table for the event.
queue_id	INTEGER	The primary key of the HDB_QUEUE table for the event.
error_id	INTEGER	The primary key of the HDB_ERROR table for the event.
begintime	INTEGER	Date timestamp specifying the beginning of the event.
endtime	INTEGER	Date timestamp specifying the ending of the event.
event	TEXT	ASCII text of the event message.

HDB_FILE

The HDB_FILE table contains information for files on which the server operated.

Table 6-6 lists the HDB_FILE table.

Table 6-6. HDB_FILE Table

Type	Attribute	Description
file_id	INTEGER	The primary key of the HDB_FILE table.
path	TEXT	The directory path to the files.
basename	TEXT	The name of the file being operated on. For example, *my_flat_netlist.gz* or *my_tester_patterns.wgl*
type	TEXT	Description of the file such as design or pattern.
MD5	TEXT	The MD5 signature.
file_grp_id	INTEGER	The file group ID number.

HDB_HEADER

The HDB_HEADER table contains global information about the HDB. For timestamps, queries into the HDB can be done with either the human readable or the UTC values.

Table 6-7 lists the HDB_HEADER table.

Table 6-7. HDB_HEADER Table

Type	Attribute	Description
header_id	INTEGER	The primary key of the HDB_HEADER table.
version	INTEGER	The version of the HDB.
creation	INTEGER	The creation timestamp of the HDB.
name	TEXT	The name of the HDB.
md5	TEXT	The MD5 signature of the HDB.
password	TEXT	The password, if any, of the HDB.
isOpened	INTEGER (1 during active session)	Whether the HDB is open or not.

HDB_MONITOR

Each row in the HDB_MONITOR table represents an add_monitor command the server has processed in addition to the monitor's associated design, pattern sets, and masks.

Table 6-8 lists the HDB_MONITOR table.

Table 6-8. HDB_MONITOR Table

Type	Attribute	Description
monitor_id	INTEGER	The primary key of the HDB_MONITOR table.
synonym	TEXT	The *monitor_id* from the add_monitor command.
directory	TEXT	The *monitored_directory* from the add_monitor command.
results	TEXT	The *results_directory* from the add_monitor command.

HDB_MONITORSET

The HDB_MONITORSET table represents the state of the internal monitor of the server.

Table 6-9 lists the HDB_MONITORSET table.

Table 6-9. HDB_MONITORSET Table

Type	Attribute	Description
monitorset_id	INTEGER	The primary key of the HDB_MONITORSET table.
monitor_id	INTEGER	The primary key of the HDB_MONITOR table for the monitor.
option_grp_id	INTEGER	The primary key of the HDB_OPTION table for the for the monitor.
design_file_id	INTEGER	The design file ID number.
file_grp_id	INTEGER	The file group ID number.

HDB_OPTION

The HDB_OPTION table records the options the server uses for a diagnosis run.

Table 6-10 lists the HDB_OPTION table.

Table 6-10. HDB_OPTION Table

Type	Attribute	Description
option_id	INTEGER	The primary key of the HDB_OPTION table.
name	TEXT	The name of the argument specified with the set_diagnosis_options command.
value	TEXT	The value for the argument specified with the set_diagnosis_options command.
option_grd_id	INTEGER	The option grd ID number

HDB_QUEUE

The HDB_QUEUE represents the state of the internal file queue of the server.

The following table lists the HDB_QUEUE table.

Table 6-11. HDB_QUEUE Table

Type	Attribute	Description
queue_id	INTEGER	The primary key of the HDB_QUEUE table.
touched	INTEGER	Last processing time.
status	TEXT	Current status of the queue.
diagtime	INTEGER	The diagnosis time.

Table 6-11. HDB_QUEUE Table (cont.)

Type	Attribute	Description
monitor_id	INTEGER	The primary key of the HDB_MONITOR table for the event.
subdir	TEXT	The sub directory name.
basename	TEXT	Name of the failure file.
lot	TEXT	Lot ID from the failure file.
wafer	TEXT	Wafer ID from the failure file.
xcoord	INTEGER	The x coordinate.
ycoord	INTEGER	The y coordinate.
memory	TEXT	Memory usage information.

Explanation of HDB_QUEUE Status Entries

When Tessent Diagnosis server processes files, the tool uniquely inserts this into the table with one of the following states:

- QUEUED — The file is awaiting processing.
- DIAGNOSING — The file is currently processing.
- DIAGNOSED — The file has been processed.
- ABANDONED — The file is pending as a result of a Tessent Diagnosis server exit.
- ABORTED — The file contains syntax errors.

HDB_SESSION

Each row in the HDB_SESSION table represents a startup of the Tessent Diagnosis server.

Table 6-12 lists the HDB_SESSION table.

Table 6-12. HDB_SESSION Table

Type	Attribute	Description
session_id	INTEGER	The primary key of the HDB_SESSION table.
begin_time	INTEGER	Server start timestamp.
update_time	INTEGER	Server update timestamp.
end_time	INTEGER	Server end session timestamp.
YAversion	TEXT	Tessent Diagnosis Server software version number.

Table 6-12. HDB_SESSION Table (cont.)

Type	Attribute	Description
host	TEXT	The name of the computer the server session is running on. Example: my_host1:4, localhost:8
pid	INTEGER	Process ID for the server session.
port	INTEGER	Port for the server session.

Distributed Diagnosis Processing

You can use Load Sharing Function (LSF), Sun Grid Engine (SGE), or custom job schedulers to facilitate the distribution of the diagnosis processing on multiple host machines. Job schedulers are used to select host machines for remote processes automatically. You can also specify host machines for remote processes manually using ssh or rsc.

Specific LSF and SGE options are variable and depend on your site configuration. Consult your System Administrator for more information.

Note
This documentation refers to SGE, which is a product of Altair.

Setting up LSF or SGE Job Schedulers

You can set up the Tessent Diagnosis server to use either the LSF or SGE job scheduler before adding any analyzers to your automatic diagnosis configuration.

Prerequisites

- A configured LSF or SGE job scheduler.
- A Tessent Diagnosis license is required for each analyzer you add.

Procedure

1. Invoke Tessent Diagnosis server. For example:

   From a Linux/UNIX shell, enter:

   Tessent_Tree_Path/bin/tessent -diagserver

   where:

   - *Tessent_Tree_Path* is the path to where the Tessent Diagnosis application tree is installed.
   - -diagserver — A required switch that invokes Tessent Diagnosis in server mode.

2. Set the job_options variable to specify command options for Tessent Diagnosis server to use when submitting a job to the scheduler. For example:

   set job_options *command_options*

where:

- *command_options* — A string value that specifies the submission commands for either the LSF or SGE job scheduler. Setting the correct value requires knowledge of the scheduler configuration at your site and the job control submission syntax.

3. Use the add_analyzer command to define the host machines and the number of analyzers the job scheduler can use with Tessent Diagnosis server. For example:

   ```
   add_analyzer lsf:4 -monitor designB
   ```

   where:

   - lsf:4 — Specifies the LSF job scheduler and adds four analyzers.
   - -monitor designB — Binds all four analyzers with the designB monitored directory.

Results

During Tessent Diagnosis processing, the tool uses the specified job scheduler to job to facilitate the distribution of the diagnosis processing on multiple host machines,

Guidelines for Troubleshooting Scheduling Delays

When running Tessent Diagnosis Server with multiple analyzers on a heavily-loaded grid (LSF or SGE), you can encounter a delay in license acquisition and job scheduling. The situation where only a limited number of licenses are available poses a similar problem.

Tessent Diagnosis Server waits until it has acquired all the required licenses and all required grid resources before commencing diagnosis. If you use the add_analyzer command to specify more than one analyzer as in the following example:

```
add_analyzer generic:10
```

Then, the Tessent Diagnosis Server must acquire all 10 licenses and CPUs before commencing diagnosis, which can also lead to a delay in acquisition and job scheduling. By default, the tool has 10 minutes to complete the acquisition operation before timing out. You can circumvent the delay in license and resource acquisition by instructing the Tessent Diagnosis Server through Tcl to incrementally acquire the license and add the analyzer resource.

Figure 6-6 shows a Tessent Diagnosis Server Tcl script that incrementally checks out a license and adds the analyzer. When the tool successfully adds the analyzer, the diagnosis starts running immediately.

Figure 6-6. Tcl Script for Adding Analyzers Incrementally

```
set scheduler_timelimit 2
set max_num_analyzer 10
set loop_i 0
set loop_start_diag -1
while { [ check -analyzers ] < $max_num_analyzer && $loop_i < 200 &&
        ![check -abort] } {
  if { [ catch {  add_analyzer generic:1 } ] != 0} {
    puts "Warning: failed to add analyzer at loop [ $loop_i ] "
  }
  if { [ check -analyzers ] == 1 && $loop_start_diag == -1 } {
    start_diagnosis
    set loop_start_diag $loop_i
    puts "Note: start diagnosis after the first analyzer was added at loop
      [ $loop_i ] "
  }
  incr loop_i
}
while { [check -queued] > 0 && ![check -abort] && [check -analyzers] > 0
        { }
```

Note

In your Tcl code, you must ensure that the start_diagnosis command proceeds any looping constructs for the analyzer, specifically any add_analyzer commands.

Using this looping method, the Tessent Diagnosis Server begins diagnosis as soon as the first analyzer is acquired. The tool incrementally adds additional analyzers with each iteration through the loop as the analyzer becomes available.

For example, assume you are using Tessent Diagnosis Server on a the heavily-loaded grid that is only capable of acquiring licences and resources for 10 analyzers. Using the Tcl code in Example 6-6, the Tessent Diagnosis Server secures the first analyzer's license and CPU, and begins diagnosis immediately with this analyzer.

Subsequently, the tool incrementally adds and begins diagnosis with the other nine analyzers as licenses and CPUs become available. The Tessent Diagnosis Server keeps trying to acquire the remaining nine analyzers (for a total of ten analyzers) in the Tcl while loop a total of 200 times. When a license and resource becomes available, the tool acquires both, and immediately begins diagnosis with that resource.

Because the tool starts the diagnosis immediately when the first analyzer is available, this Tcl looping method also works well when you have a limited number of licenses. The tool continuously adds more analyzers until the limit on available licenses is reached.

Setting Up a Custom Job Scheduler

Use the following procedure to set up Tessent Diagnosis server to use a generic job scheduler before adding any analyzers to your configuration.

Procedure

1. Run Tessent Diagnosis server. For example:

   From a Linux/UNIX shell, enter:

   Tessent_Tree_Path/bin/tessent -diagserver

   where:

   - *Tessent_Tree_Path* is the path to where the Tessent Diagnosis application tree is installed.
   - -diagserver — A required switch that invokes Tessent Diagnosis in server mode.

2. Enter information required by the Tessent Diagnosis Server to use a custom job scheduler (other than SGE and LSF).

   a. Specify how to submit a job for the custom job scheduler.

      Use the following variable and value pair:

      set generic_scheduler "$<root_path>/submit reqmem=%memory %command"

      where:

      - generic_scheduler

        This variable defines the command Tessent Diagnosis Server uses when submitting a job to the grid.

        The value string can contain the following parts:

        - $<root_path>/submit

          A path you specify to the command script that you want Tessent Diagnosis Server to use when requesting a remote process from the job scheduler. The command script contains the submit command specific to the custom job scheduler

        - %memory

          A string value that specifies to find a machine with the specified amount of memory. You can specify memory with the following variations:

          - %memoryMB xxxx.xxxMB
          - %memoryM xxxx.xxxM
          - %memoryGB x.xxxGB
          - %memoryG x.xxxG
          - %memory x.xxx

        - %command

A string value that specifies the location to substitute the Siemens EDA command script that launches a remote process.

b. Specify how to delete a job previously submitted to the scheduler

Use the following variable and value pair:

```
set generic_delete "$<root_path>/delete %job"
```

where:

- generic_delete

  This variable defines the command to delete previously submitted scheduler jobs through Tessent Diagnosis Server.

- $<root_path>/delete

  A user-specified command that Tessent Diagnosis Server uses to delete a job previously submitted on the custom job scheduler when the scheduler is interrupted by Control-C.

- %job

  Tessent Diagnosis server keeps track of the job ID of each successfully submitted analyzer. The %job is a placeholder that the tool automatically fills in with the correct job ID of the respective analyzer.

3. Use the add_partitioner command to define that you are using a generic job scheduler and specify the number of analyzers to use with Tessent Diagnosis server. The tool automatically adjusts the memory requirements for the partitioner and the other analyzers by supplying the memory requirements where %memory is specified. For example:

```
add_partitioner generic:1
add_analyzer generic:4
```

Note
A Tessent Diagnosis license is required for each analyzer.

Manual Job Scheduling

In the absence of a job scheduler, you can use the rsh or ssh command to specify which network machines to host diagnosis processes.

- rsh — The network must allow connection via rsh, and your *.rhosts* file must allow rsh access from the master host without specifying a password. The *.rhosts* file on host machines must have read permission set for user. Write and execute permission can optionally be set for user, but must not be set for other and group.

rsh access is not required for Tessent Diagnosis server to create additional processes on the master host.

- ssh — The network must allow connection via ssh. To enable use of ssh, enter a Set command within the tool to set the remote_shell variable to ssh. For example:

  ```
  set remote_shell ssh
  ```

  Do this prior to issuing any other Tessent Diagnosis server commands.

  Master and remote machines must be correctly specified in the global DNS name server for reliable network operation, and you need to know either the network name or IP address of each remote machine you plan to use.

Consult the System Administrator at your site for additional information.

Running Tessent Diagnosis Server with a Local Host

You can configure the Tessent Diagnosis Server to use a local host as the analyzer. You might use this option if you have network or grid problems.

Prerequisites

- Ensure that the machine you want to use as the local host has enough virtual memory, specifically RAM and swap disk. For example, if you have a machine with 32 gigabytes (GB) of RAM and 80 GB of swap space, and the design needs 45 GB of virtual memory, then you should add at most two analyzer CPUs on the local host.

Procedure

Declare the local host as follows:

add_analyzer localhost:N

where *N* is the number of host CPUs on the machine.

Examples

The following dofile example illustrates using a machine with two CPUs:

```
add_monitor monitor_name Log_fail -result Result_directory

add_design monitor_name design.v.flat

add_pattern monitor_name pat.stil

add_layout monitor_name layout.db

add_analyzer localhost:2        // add two cpus from localhost as analyzers

start_diagnosis
```

Running Tessent Diagnosis Server in Batch Mode

You can set up and run Tessent Diagnosis server in batch mode using a dofile created with Tcl scripting features and the Tessent Diagnosis server commands.

Prerequisites

- You have configured Tessent Diagnosis Server as described in "Setting Up the Tessent Diagnosis Server."
- As needed, you have configured a job scheduler as described in "Distributed Diagnosis Processing."

Procedure

From a Linux/UNIX shell, enter:

Tessent_Tree_Path/bin/tessent -diagserver -dofile my_tcl_dofile.do

where:

- *Tessent_Tree_Path* is the path to where the Tessent Diagnosis application tree is installed.
- -diagserver — A required switch that invokes Tessent Diagnosis in server mode.
- *my_dofile*.do —is the pathname of the dofile to run.

Results

Tessent Diagnosis server invokes and runs the commands listed in the dofile. This is a sample dofile:

```
add_monitor design1_fs fail_log_server -results ../diag_rlt_design1
add_design design1_fs netlists/design_fs.v.flat
add_pattern design1_fs pat.wgl
add_monitor design2_tk fail_log_server -results ../diag_rlt_design2
add_design design2_tk netlists/design_tk.v.flat
add_pattern design2_tk pat.wgl
report_monitor
add_analyzer machine1 -monitor design1_fs
add_analyzer machine2 -monitor design1_fs
add_analyzer machine3 -monitor design2_tk
add_analyzer machine4 -monitor design2_tk
watch
report_analyzer -detail
start_diagnosis
```

The Tessent Diagnosis Server Daemon

You can invoke the Tessent Diagnosis server with a daemon option and a dofile containing Tessent Diagnosis server commands, allowing the tool to continue running even if you log out. In daemon mode, the Tessent Diagnosis server operation is identical to the standard batch mode.

Note

When using the server daemon, omit the exit command from your Tessent Diagnosis server dofile. You specify daemon operations, including exit, through the Tessent Diagnosis server invocation.

You control the Tessent Diagnosis server daemon using the following switches in conjunction with the Tessent Diagnosis server command line invocation:

tessent -diagserver options [-daemon [id]]| [-dlist] | [-dstatus id] | [-dexit id] | [-dterminate id]

- -daemon [*id*]

  An optional switch that invokes the Tessent Diagnosis server in daemon mode. The tool automatically assigns a process ID. Using the optional *id*, you can specify a process ID for the daemon.

- -dlist

  An optional switch that lists running daemons, including the id, host, and logfile for each daemon.

- -dstatus *id*

  An optional switch and integer pair that queries the status of a daemon you identify with *id*and displays the status of the monitors and analyzers.

- -dexit *id*

  An optional switch and integer pair that instructs the daemon to exit after the tool finishes running any diagnosis jobs.

- -dterminate *id*

  An optional switch and integer pair that instructs the daemon to immediately exit without finishing running any diagnosis jobs.

Server Session Customizations

Tessent Diagnosis provides many methods for customizing the server session, including setting time limits for analyzers, using server variables, configuring automatic load balancing, and using time-based licensing.

Analyzer Time Limits

By default, analyzers run for a unlimited amount of time. You can specify a different time limit with the analyzer_timelimit variable. This variable must be set prior to adding monitors.

For example:

```
set analyzer_timelimit 100
```

where 100 is an integer that specifies the analyzer time limit in seconds.

If you set analyzer_timelimit to 1, and you declare this variable before the add_monitor command, then the tool applies this global timelimit to all analyzers. The analyzer aborts if this time limit is reached.

Tessent Diagnosis Server Variables

Several variables available within Tessent Diagnosis server enable you to customize the Tessent Diagnosis server.

From the Tessent Diagnosis server command prompt, enter:

set_variable *variable_name value*

where:

- *variable_name* — The name of the variable.
- *value* — The new value to set it to.

Use the report_variable command to display the current settings for all the variables displays.

Table 6-13 lists the Tessent Diagnosis server variables.

Table 6-13. Tessent Diagnosis Server Variables

Variable Name	Data Type	Default Value	Description
analyzer_timelimit	integer	100000	Specifies the maximum time in seconds that analyzers spend on one diagnosis. For more information, see the "Analyzer Time Limits." Using the set_diagnosis_options -Time_limit switch and argument overrides this variable.

analyzer_restart	Boolean	false	Turns on and off restarting idle analyzers. Tessent Diagnosis server enables a total of three restarts per analyzer.
			
			Note: If you specify "**set / set_variable analyzer_restart on**", the tool reports the following message:
			
			`> //  command: set analyzer_restart true`
			`> ... analyzer restart engaged.`
			`> //  Warning: analyzer_restart is deprecated and will be removed in a future release.`

auto_load_balancing	Boolean	true	Turns on and off the automatic load balancing of analyzers. For more information, see the "Automatic Load Balancing" section in this document.
clock_restriction	string	default	Turns on and off ATPG clock restriction mode. Values are on or off. The clock restriction setting in the flat model of the design is the default value for this variable.
contention_check	string	default	Turns on and off contention checking during pattern verification. Values are on or off. The contention check setting in the flat model of the design is the default value for this variable. For more information, see the "set_contention_check" section in the *Tessent Shell Reference Manual*.
diagnostic_CSV	Boolean	false	Turns on and off the CSV (comma separated values) formatting of the diagnostic report.

Table 6-13. Tessent Diagnosis Server Variables (cont.)

Variable Name	Data Type	Default Value	Description
diagnostic_reports	Boolean	true	Turns on and off the generation of the diagnostic report.
dp_work_dir	string	false	Specifies a working directory for dynamic partitioning-based diagnosis. Refer to the add_analyzer -dp_work_dir option for more information.
faillog_sort_criteria	string	smallest_file	Specifies the server's re-queuing behavior for processing failure files. Choose from the following: • smallest_file — A literal that specifies re-queuing the failure files with the smallest size first. This is the default. • largest_file — A literal that specifies re-queuing the failure files with the largest size first. • oldest_file — A literal that specifies re-queuing the failure files with the oldest timestamp first. • newest_file — A literal that specifies re-queuing the failure files with newest timestamp first.
generic_delete	string		Specifies the command used to delete a job submitted via the generic_scheduler variable. For more information, see the "Setting Up a Custom Job Scheduler" section in this chapter.
generic_scheduler	string		Specifies the command script to use to request a remote process from a custom job scheduler. Formore information, see the "Setting Up a Custom Job Scheduler" section in this chapter.
gzip_path	string		Specifies the path to a gzip design. When set, the gzip_path is transmitted to the analyzers upon their launch. You must have a valid network path to the gzip design for each host that the add_analyzer command selects, otherwise the server produces an error.
job_memreq	integer	3	Specifies the minimum memory in GB required on machines LSF or SGE scheduled jobs.

Table 6-13. Tessent Diagnosis Server Variables (cont.)

Variable Name	Data Type	Default Value	Description
job_options	string		Specifies command options to append to the job submission command for the LSF or SGE job scheduler. Setting a correct value for thestring requires a knowledge of the scheduler configuration at your site and the job control submission syntax. For more information, see the "Setting up LSF or SGE Job Schedulers" section in this chapter.
license_awaken_ timelimit	integer (seconds)	10	To awaken a hibernating analyzer, the server attempts to reacquire a license for the analyzer. This variables specifies the time limit for which the server waits for the license to be acquired before trying again.
memory_monitoring	Boolean	false	Checks the analyzer's memory for each diagnosis. If set to true, then the tool adds process size and free memory information to the HDB_QUEUE table of the HDB.
monitor_filer	Boolean	true	Specifies the processed failure file management behavior.
monitor_filter	string		Specifies a string that Tessent Diagnosis server uses to filter out the failure files to process. Failure file names that contain the specified string are processed while all others are ignored. Valid POSIX Extended Regular Expression are supported. For example: set monitor_filter .*wafer13.* matches any filename that contains the string "wafer13".
remote_shell	string	rsh	Specifies which shell command is used to manually set up hosts for job scheduling. Options include: none, rsh, and ssh.
scheduler_timelimit	integer	10	Specifies the maximum time in minutes the job scheduler spends to schedule a remote machine.

Table 6-13. Tessent Diagnosis Server Variables (cont.)

Variable Name	Data Type	Default Value	Description
split_capture	string	default	Determines whether ATPG split capture is used. The split capture setting in the flat model of the design is the default value for this variable.
verify_design_layout	Boolean	true	Determines whether the tool automatically calculates and validates the MD5 signature information all at once as part of the server setup. Specifying false can improve up-front setup/validation performance, with the trade off that MD5 verification occurs later within each analyzer. Any errors are reported back to the server.

Automatic Load Balancing

Automatic load balancing is a process performed by Tessent Diagnosis server to facilitate the most efficient use of analyzers.

By default, automatic load balancing is enabled and works as follows:

- When new analyzers are added to a Tessent Diagnosis server configuration, they are assigned to monitored directories based on need.
- When all the failure files in a monitored directory are diagnosed, the assigned analyzers become idle. The idle analyzers are then reassigned to other directories based on need.
- When analyzers that are initially bound to a specific monitored directory via the add_analyzer command become idle, they are reassigned to other directories based on need. The initial binding is dissolved once the analyzer is reassigned to another monitored directory.

Set the auto_load_balancing variable to false to turn off the automatic load balancing feature. For example:

```
set auto_load_balancing false
```

To turn the auto balancing feature back on, set the auto_load_balancing variable to true.

Time-Based Licensing

Using time-based license management, you can configure the Tessent Diagnosis server to make analyzer licenses available at different times of the day.

Table 6-14 lists the commands you use for activating and configuring time-based license management in the Tessent Diagnosis server. You enter these commands at the Tessent Diagnosis server prompt or in a dofile.

Table 6-14. Time-Based Licensing Commands

Command	Description
delete_schedule	Removes previously-scheduled time-based analyzer licensing events.
report_schedule	Lists currently scheduled time-based license management events.
schedule_licenses	Specifies scheduling setup for time-based analyzer license management.

Example 1

The following example schedules five licenses at 5:30 p.m.:

```
schedule_licenses 5 -at 5:30pm
```

Example 2

The following example schedules five licenses at 12:00 a.m. every day:

```
schedule_licenses 5 -at 12:00am -daily
```

Reporting Server Status with Email

Using the Tessent Diagnosis server's email facility, you can configure the Tessent Diagnosis server to automatically send email to subscribed recipients at different times of the day.

Procedure

1. Subscribe recipients for the server email using the email command. The following example subscribes recipient1 and recipient2:

   ```
   email -subscribe recipient1@siemens.com recipient2@siemens.com
   ```

2. Use the following commands to activate and configure the email facility in the Tessent Diagnosis server. You enter these commands at the Tessent Diagnosis server prompt or in a dofile.

Table 6-15. Email Facility Commands

Command	Description
delete_schedule	Removes previously-scheduled email events.
email	Subscribes and unsubscribes email recipients.

Table 6-15. Email Facility Commands (cont.)

Command	Description
report_schedule	Lists currently-scheduled email events.
schedule_email	Specifies scheduling setup for the server's email facility.

Results

Subscribers automatically receive email notifications.

Examples

The following example shows Tcl message syntax and sends email to the subscribed recipients every hour:

```
schedule_email { query status } -every 1:00
```

The following example sends email to the subscribed recipients at 5:00 p.m.:

```
schedule_email “It’s 5 o’clock in the afternoon” -at 5:00pm
```

The following example sends email at 12:00 a.m. every day:

```
schedule_email “Midnight Query” -at 12:00am -daily
```

The following series of commands lists the currently-scheduled email and removes the first event:

```
report_schedule

//  command: report_schedule

1email
```

Command Reference

Tessent Diagnosis supports many dedicated commands that you can use to set the server up to automatically run diagnosis on failure files as they are placed into pre-specified directories.

These commands use the Tcl embedded scripting language and can be used in conjunction with all Tcl scripting features—see "Running Tessent Diagnosis Server in Batch Mode."

add_analyzer

Scope: Server mode

Sets up one or more analyzers or a job scheduler for automatic diagnosis.

Usage

add_analyzer {***host_name...*** |{**sge | lsf | generic | localhost** }} [***:processors***]
[-monitor *monitor_id* | -bind] [-priority *priority*]
[-verbose] [-hibernate *minutes*]

Arguments

- ***host_name***

  Required, repeatable string that specifies the name of a computer on which to run the analyzer. You must have rsh access to specify a host. The string localhost works as a substitute for local system host name.

- **sge | lsf | generic | localhost**

  Required literal that specifies which job scheduler to use. Depending on the job scheduler, variables must be set up prior to issuing this command. For more information, see "Distributed Diagnosis Processing". Job scheduler options include:

  sge — Sun Grid Engine (SGE) job scheduler

  Note
  This documentation refers to SGE, which is a product of Altair.

  lsf — Load Sharing Function (LSF) job scheduler

  generic — Customized job scheduler interface as described in "Setting Up a Custom Job Scheduler" on page 381

  localhost — A local host job scheduler

  Analyzer restart capability is not supported for analyzers added via custom scheduler interfaces. See "Tessent Diagnosis Server Variables" analyzer_restart variable for more information.

- ***:processors***

  Optional integer that specifies the number of analyzers to add. When specified, this argument must be appended (no spaces) to the hostname or the job scheduler argument. For example, the LSF job scheduler and two analyzers would be: lsf:2

- -monitor *monitor_id*

  Optional switch and string that specifies a monitor to bind the analyzer with. The *monitor_id* is the unique name for the monitored directory, specified with the add_monitor command.

The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

This option is not applicable to dynamic partitioning-based diagnosis, which only uses one monitor.

- -bind

  Optional switch that specifies to bind the analyzer to the monitor. For example, the analyzer and monitor may not be bound if you added the analyzers before the monitor or after specifying the start_diagnosis command.

- -priority *priority*

  Optional switch and integer pair that specifies the switching wait time (in seconds) that the analyzer remains in an idle state before it migrates to a different monitor that has work available in it. The higher the *priority* number, the longer the switching time to a new monitor. The default is 1.

  This option is not applicable to dynamic partitioning-based diagnosis, which only uses one monitor.

- -verbose

  Optional switch that reports the grid scheduler submission command with all the switches and values that have been determined by the tool.

- -hibernate *minutes*

  Optional switch and string that specifies the number of minutes before idle analyzers enter hibernation mode and relinquish their licenses. The number must be greater than one. Using different add_analyzer commands, you can specify a mixture of analyzers with no hibernation or different hibernation time limits. A hibernating analyzer attempts license acquisition under control of the license_awaken_timelimit variable as described in "Tessent Diagnosis Server Variables."

Description

The add_analyzer command starts the Tessent Diagnosis server and runs the analyzer or analyzers. The analyzers read in the design and pattern data associated with the specified monitor.

If multiple analyzers are associated with a monitor, the first analyzer to finish reading the design verifies the pattern set. If the pattern verification succeeds, all analyzers are ready to start diagnosing the failure files.

Each analyzer requires a Tessent Diagnosis license that it uses even when the analyzer is idling. To free up licenses while maintaining an active server session, specify the -hibernate switch. When an analyzer enters hibernation mode, it frees up a license while remaining ready to process failure files as they enter its monitor's queue. When a failure file becomes available, the hibernating analyzer attempts to reacquire a license. When it reacquires the license, the analyzer processes the failure file.

See also "Distributed Diagnosis Processing," "Running Tessent Diagnosis Server with a Local Host," and "Running Tessent Diagnosis Server in Batch Mode."

Examples

The following example sets up three analyzers to run on the localhost before adding partitioners.

add_analyzer localhost:3

```
//  command: add_analyzer localhost:3
//  Error: No partitioners were added. Please add partitioners first using
"add_partitioner" command.
//  command: add_partitioner localhost:1
...
//  command: add_analyzer localhost:2
...
//  command: report_partitioner
---< partitioners >-----------------------------------------------------
[1]  achilles02.wv.mentorg.com  #1 flogs  Partitioner ... idle (1 secs)
//  command: report_analyzer
---< analyzers >--------------------------------------------------------
(1)  achilles02.wv.mentorg.com  #1 flogs  Analyzer    ... idle (1 secs)
(2)  achilles02.wv.mentorg.com  #1 flogs  Analyzer    ... idle (0 secs)
```

The following example sets up four analyzers to run on host1 and binds them to the monitor1 monitored directory.

add_analyzer host1:4 -monitor monitor1

The following example sets up eight analyzers to run on the localhost and binds them to the monitor1 monitored directory.

add_analyzer localhost:8 -monitor monitor1

The following example sets up four analyzers to use the LSF job scheduler and binds them to the design1 monitored directory:

add_analyzer lsf:4 -monitor design1

The following example specifies the submit script command for submitting jobs to the custom job scheduler:

set generic_scheduler "$SGE_ROOT/bin/lx24-x86/submit reqmem=%memory %command"

You must use the "%command" string inside the generic_scheduler string to specify the location to substitute the Siemens EDA command script that launches a remote process. The %memory string provides the memory requirements.

For a detailed example refer to "Setting Up a Custom Job Scheduler" on page 381.

The following example enables you to incrementally add the hibernation mode to an analyzer that is already running. It sets analyzer number 5 to hibernate after two minutes of idling.

```
add_analyzer 5 -hibernate 2
```

add_design

Scope: Server mode

Specifies the flat design netlist used to diagnose failure files in a specified monitored directory.

Usage

add_design ***monitor_id flat_design*** [-password]

Arguments

- ***monitor_id***

  Required string that specifies the monitored directory. The *monitor_id* is the unique name for the monitored directory, specified with the add_monitor command.

  The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

- ***flat_design***

  Required string that specifies the pathname for the design netlist.

- -password

  Optional switch that enables you to enter a password and access password protected designs. The entered password must match the password saved when the flat netlist was saved.

Examples

The following example sets up the monitored directory monitor1 to use the design netlist flat_design1 for diagnosing failure files.

```
add_design monitor1 flat_design1
```

Related Topics

delete_design

add_layout

Scope: Server mode

Opens an existing Tessent Diagnosis tool-compatible LDB.

Usage

Layout-Aware Diagnosis Flow

add_layout ***monitor*** **-DFT** [-Copy *local_disk_directory*] [-chip_design_name *top_design_name*] ***layout_database_name***

Arguments

- ***monitor***

  A required string specifying the name of the monitor.

Layout-Aware Diagnosis Flow

- **-DFT**

  A required switch that specifies the layout-aware diagnosis flow.

- -Copy *local_disk_directory*

  An optional switch and string pair that specifies the local directory location (*local_disk_directory*) to store a cached copy of the LDB. Using this method results in improved performance by avoiding disk access delay over a slow network.

  You must prefix the *local_disk_directory* path with */tmp/*, for example:

  ```
  add_layout monitorA -DFT -copy /tmp/user/ my_layout_database.ldb
  ```

  The LDB must be accessible by all analyzers associated with the given monitor, and the local disk directory location for each host must be accessible by all analyzers landed on that machine. When set, the Tessent Diagnosis Server instructs an analyzer to copy the LDB to the local disk directory of the machine and performs layout-aware diagnosis on the local copy of the LDB.

  If available, all the analyzers associated with a specific monitor on the same host use the same local copy of the LDB, specifically the LDB is copied to the local directory just once per host. When done, Tessent Diagnosis Server instructs the last analyzer on a host to delete the local copy of the LDB before the analyzer is deleted.

  If the copy operation for the LDB fails, Tessent Diagnosis Server issues a warning and uses the source LDB on the shared file server for diagnosis.

  For implementation strategies and examples, see "Running Layout-Aware Diagnosis on a Local Layout Database."

- -chip_design_name *top_design_name*

  An optional switch that specifies a top-level design whose core instance information is included in the specified chip-mapped core LDB. This option only pertains to hierarchical layout-aware diagnosis and is required when you have a core that is instantiated in more than one design.

- ***layout_database_name***

  A required string that specifies the name of an existing LDB directory created with the create_layout command. For hierarchical layout-aware diagnosis, the LDB must be a chip-mapped core LDB.

Description

Layout-Aware Diagnosis Flow

Opens an existing Tessent Diagnosis tool-compatible LDB.

You must create this layout using the Tessent Diagnosis scan diagnosis tool—see the create_layout command—see "Layout-Aware Diagnosis Flow."

In hierarchical layout-aware diagnosis, you must use the -chip_design_name option when you have a core that is instantiated in more than one design. See "Diagnosis for Hierarchical Designs" for more information.

Layout-Aware Diagnosis Flow with DFM

Opens an existing Tessent Diagnosis tool-compatible LDB and if this database contains imported DFM information, the layout-aware diagnosis results are annotated with DFM hit results.

For more information, see "Diagnosis for Design for Manufacturability Analysis."

Related Topics

delete_layout

add_monitor

Scope: Server mode

Prerequisites: Specified directories must exist.

Sets up a monitored directory.

Usage

add_monitor ***monitor monitored_directory*** [-Results *results_directory*]

Arguments

- ***monitor***

  Required string that specifies a unique name for the monitored directory. *monitor* is used as shorthand to reference the monitored directory and can be used by subsequent commands.

  Note

  You cannot prefix *monitor* with a number (for example, 123monitor).

- ***monitored_directory***

  Required string that specifies the pathname to an existing directory for Tessent Diagnosis server to monitor. You can specify a relative or absolute pathname.

  Note

  Ensure that you place only fail logs into the monitored directory. After processing, the tool moves all files into their respective disposition directories. The monitor operates recursively so it also processes files in subdirectories.

- -Results *results_directory*

  Optional switch and string that specifies the pathname to a directory where you want the diagnosis results placed. You can specify a relative or absolute pathname. If the specified directory does not exist, it is created.

  By default, Tessent Diagnosis server places the diagnosis results in a directory named *monitored_directory.ya* at the same pathname as the associated monitored directory.

Description

The add_monitor command specifies which directories to monitor for incoming failure files. Once failure files are detected, Tessent Diagnosis server automatically runs diagnosis on them. Failure files that exist in the specified directory are processed immediately. If sub-directories exist under monitored_directory, Tessent Diagnosis Server runs diagnosis on the failure files in sub-directories using the same diagnosis collateral as the parent monitored_directory. The results directory structure mirrors the directory structure of the monitored_directory.

Examples

The following example defines /user/diagnosis/monitor1 as the monitored directory, names the monitored directory, *monitor1*, and specifies a directory, *results1,* for the diagnosis results.

```
add_monitor monitor1 /user/disgnosis/monitor1 -results results1
```

Related Topics

delete_monitor

report_monitor

add_pattern

Scope: Server mode

Specifies the pattern files used for the diagnosis of failure files in a specified monitored directory.

Usage

add_pattern {***monitor_id*** {***pattern*** ...}} [-NOPadding] [-mask [*maskfile*]]

Arguments

- ***monitor_id***

  Required string that specifies the monitored directory with which to bind the test patterns.

  The *monitor_id* is a unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

- ***pattern***

  Required, repeatable string that specifies the pathname for test pattern files to bind with the specified monitor. Multiple files are appended together and associated with the specified monitor and optional maskfile. If you are using multiple patterns with multiple test suites, then see "Examples" under "NO TITLE."

- -NOPadding

  An optional switch specifying that the source test pattern set contains ASCII patterns that are not padded for the scan load and unload data.

- -mask *maskfile*

  Optional switch and string pair that specifies a maskfile for the specified test pattern file.

Examples

The following example associates the *pattern1* test pattern file and the pattern1maskfile to the *monitor1* directory for the automatic diagnosis.

```
add_pattern monitor1 /user/disgnosis/pattern1 -mask /user/disgnosis/pattern1maskfile
```

add_partitioner

Scope: Server mode

Sets up one or more partitioners or a job scheduler for automatic diagnosis.

Usage

add_partitioner {***host_name...*** |{**sge | lsf | generic | localhost**}} [***:processors***]
    [-monitor *monitor_id* | -bind] [-priority *priority*]
    [-dp_work_dir *directory_path*] [-verbose] [-hibernate *minutes*]

Arguments

- ***host_name***

  Required, repeatable string that specifies the name of a computer on which to run the partitioner. You must have rsh access to specify a host. The string localhost works as a substitute for local system host name.

- **sge | lsf | generic | localhost**

  Required literal that specifies which job scheduler to use. Depending on the job scheduler, variables must be set up prior to issuing this command. For more information, see "Distributed Diagnosis Processing". Job scheduler options include:

  sge — Sun Grid Engine (SGE) job scheduler

  Note
  This documentation refers to SGE, which is a product of Altair.

  lsf — Load Sharing Function (LSF) job scheduler

  generic — Customized job scheduler interface as described in "Setting Up a Custom Job Scheduler" on page 381

  localhost — A local host job scheduler

- ***:processors***

  Optional integer that specifies the number of partitioners to add. When specified, this argument must be appended (no spaces) to the hostname or the job scheduler argument. For example, the LSF job scheduler and two partitioners would be: lsf:2

- -monitor *monitor_id*

  Optional switch and string that specifies a monitor to bind the partitioner with. The *monitor_id* is the unique name for the monitored directory, specified with the add_monitor command.

  The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

This option is not applicable to dynamic partitioning-based diagnosis, which only uses one monitor.

- -bind

  Optional switch that specifies to bind the partitioner to the monitor. For example, the partitioner and monitor may not be bound if you added the partitioner before the monitor or after specifying the start_diagnosis command.

- -priority *priority*

  Optional switch and integer pair that specifies the switching wait time (in seconds) that the partitioner remains in an idle state before it migrates to a different monitor that has work available in it. The higher the *priority* number, the longer the switching time to a new monitor. The default is 1.

  This option is not applicable to dynamic partitioning-based diagnosis, which only uses one monitor.

- -dp_work_dir *directory_path*

  Optional switch and string pair that assigns an alternative location for partitioned files. This option only pertains to dynamic partitioning-based diagnosis.

  The dynamic partitioners produce partitioned flat models in files located in the monitor's failure file directory. Because partitioning proceeds at a faster rate than diagnosis, the partitioned files accumulate until they are used. This can cause file storage issues. When you specify this option, the partitioners store the files in the specified work directory, and the partitioners read the files stored inside the work directory. Tessent Diagnosis server deletes the files from the work directory as they are processed.

  You can also specify the working directory with the dp_work_dir variable.

  Note

  For dynamic-partitioning diagnosis flow, define the partitioners prior to defining the analyzers.

- -verbose

  Optional switch that specifies to report detailed information for each partitioner currently running.

- -hibernate *minutes*

  Optional switch and string that specifies the number of minutes before idle partitioners enter hibernation mode and relinquish their licenses. The number must be greater than one. Using different add_partitioner commands, you can specify a mixture of partitioners with no hibernation or different hibernation time limits. A hibernating partitioners attempts license acquisition under control of the license_awaken_timelimit variable as described in "Tessent Diagnosis Server Variables."

Description

The add_partitioners command starts the Tessent Diagnosis server and invokes the partitioner or partitioners. Then the partitioners read in the design and pattern data associated with the specified monitor.

If multiple partitioners are associated with a monitor, the first partitioner to finish reading the design verifies the pattern set. If the pattern verification succeeds, all partitioners are ready to start diagnosing the failure files.

Each partitioner requires a Tessent Diagnosis license that it uses even when the partitioner is idling. To free up licenses while maintaining an active server session, specify the -hibernate switch. When a partitioner enters hibernation mode, it frees up a license while remaining ready to process failure files as they enter its monitor's queue. When a failure file becomes available, the hibernating partitioner attempts to reacquire a license. When it reacquires the license, the partitioner processes the failure file.

See also "Distributed Diagnosis Processing," "Running Tessent Diagnosis Server with a Local Host," and "Running Tessent Diagnosis Server in Batch Mode."

Examples

The following example sets up three analyzers to run on the localhost before adding partitioners.

add_analyzer localhost:3

```
//  command: add_analyzer localhost:3
//  Error: No partitioners were added. Please add partitioners first using
"add_partitioner" command.
//  command: add_partitioner localhost:1
...
//  command: add_analyzer localhost:2
...
//  command: report_partitioner
---< partitioners >-------------------------------------------------------
[1]  achilles02.wv.mentorg.com  #1 flogs  Partitioner ... idle (1 secs)
//  command: report_analyzer
---< analyzers >---------------------------------------------------------
(1)  achilles02.wv.mentorg.com  #1 flogs  Analyzer    ... idle (1 secs)
(2)  achilles02.wv.mentorg.com  #1 flogs  Analyzer    ... idle (0 secs)
```

The following example sets up four partitioners to run on host1 and binds them to the monitor1 monitored directory.

add_partitioner host1:4 -monitor monitor1

The following example sets up eight partitioners to run on the localhost and binds them to the monitor1 monitored directory.

add_partitioner localhost:8 -monitor monitor1

The following example sets up four partitioners to use the LSF job scheduler and binds them to the design1 monitored directory:

```
add_partitioner lsf:4 -monitor design1
```

The following example specifies the submit script command for submitting jobs to the custom job scheduler:

```
set generic_scheduler "$SGE_ROOT/bin/lx24-x86/submit reqmem=%memory %command"
```

You must use the "%command" string inside the generic_scheduler string to specify the location to substitute the Siemens EDA command script that launches a remote process. The %memory string provides the memory requirements.

For a detailed example see "Setting Up a Custom Job Scheduler" on page 381.

The following example enables you to incrementally add the hibernation mode to a partitioner that is already running. It sets partitioner number 5 to hibernate after two minutes of idling.

```
add_partitioner 5 -hibernate 2
```

add_reporting_format

Scope: Server mode

Enables the writing of diagnosis reports for a specified monitor.

Usage

add_reporting_format [*monitor1 monitor2... monitorn*] [-ENCoded] [-TEXT] [-CSV] [-LAYout_marker]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have currently defined.

- -ENCoded

  An optional switch used to generate encoded diagnosis reports. The tool encodes a suspect's pin pathname, net name, and cell name in the diagnosis output file.

- -TEXT

  A switch specifying the ASCII text format. This is the default format.

- -CSV

  A switch specifying the comma separated value (CSV) format.

- -LAYout_marker

  A switch specifying the layout coordinate format. If you use this switch, you must have loaded the layout using the add_layout command.

Description

This command also replaces the global diagnostic_CSV variable.

Related Topics

delete_reporting_format

report_reporting_format

add_reporting_xmap

Scope: Server mode

Enables the writing of SPICE-mapped nets and Verilog module hierarchy in the diagnosis reports for a specified monitor.

Usage

add_reporting_xmap [*monitor1 monitor2... monitorn*] [{-v2lvs | -cell_hierarchy}]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have currently defined.

- -v2lvs | -cell_hierarchy

  A required switch specifying the data format. You must specify at least one of the following options:

  - -v2lvs — Specifies outputting SPICE-mapped nets.
  - -cell_hierarchy — Specifies outputting Verilog module hierarchy.

  You can specify any combination of these switches. For example, the following syntax creates both the SPICE-mapped nets and the Verilog module hierarchy in the diagnosis report:

  ```
  add_reporting_xmap monitor1 -v2lvs -cell_hierarchy
  ```

Related Topics

delete_reporting_xmap

report_reporting_xmap

add_startup_cache

Scope: Server mode

Loads an existing Diagnosis Startup Cache you created with the Tessent Diagnosis scan diagnosis point tool.

Usage

add_startup_cache ***monitor_id diagnosis_startup_cache_name***

Arguments

- ***monitor_id***

  A required string that specifies the monitored directory with which to bind the Diagnosis Startup Cache.

  The *monitor_id* is a unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example, "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

- ***diagnosis_startup_cache_name***

  A required string that specifies the name of an existing Diagnosis Startup Cache.

Description

See "Diagnosis Startup Cache" for more information.

analyze_resource_requirements

Scope: Server mode

Analyzes requirements for all modes and displays them.

Usage

analyze_resource_requirements [-memory *memory*] [-job_limit *job_limit*]

Arguments

- -memory *memory*

  An optional switch and integer that specifies the memory in GB available for memory mode.

- -job_limit *job_limit*

  An optional switch and integer that specifies the maximum number of jobs that can be run. Each partitioner and analyzer is considered a job. Therefore, for one partitioner and one analyzer, the job requirement is two jobs. Use this switch to specify the license limits for the dynamic partition run..

Description

The analyze_resource_requirements command analyzes the requirements for all modes and displays the number of partitioners and the maximal number of DP-analyzers.

Examples

This example shows the output of analyze_resource_requirements for a population of 10 failure fails.

```
/// Example1
//  command: analyze_resource_requirements
//  Note: ... Analyzing resource requirements
//  Note: Since memory target wasn't specified automated target of 9GB will be used.

... Monitor monA initializing 10 failure files.
//  -----------------------  -----------------------
//  Estimated memory requirement for each resource
//  -----------------------  -----------------------
//  Partitioner              3
//  Analyzer                 2
//  -----------------------  -----------------------
//  Note: Analyzer resource estimate assumes partition size is 20% of original design. Memory
requirement for analyzer is automatically adjusted for partition size before job scheduling.
//  Note: Total number of failure files to be diagnosed: 10
//  Note: Upto 3 DP-analyzers will be used based on memory limit.
--------------------  ----------  ----------  ----------------  ----------------  --------  -
//  Maximal diagnosis # of        Total partitioner # of analyzers Total analyzers Total # of
//  load per           partitioners  memory         range           memory         memory jobs
//  partitioner                                                                     range (GB)
//                   Min      Max            Min    Max  Min  Max    Min    Max  Min   Max
//  -------------------------------------------------------------------------------------------/
/  balanced          1        3              1    10    2     20      5      23    2    11
//  memory            1        3              1     3    2      6       5       9    2     4
//  runtime           1        3              1    10    2     20       5      23    2    11
//  --------------------------------------  --------  ---------  ----------------------
```

check

Scope: Server mode

Used in Tcl script to query the active diagnosis session for a condition.

Usage

check { **-Diagnosed** | **-Queued** | **-Total** | **-Errors** | **-ANalyzers** | **-Preanalyzed** | |
-DUplicates | **-ABort** | **-PArtitioned** | -partitioners | -processors }

Arguments

- **-Diagnosed**

  Required switch that queries the active server session for the number of completed diagnoses.

- **-Queued**

  Required switch that queries the active server session for the number of queued diagnoses.

- **-Total**

  Required switch that queries the active server session for the total number of diagnoses, including completed and queued.

- **-Errors**

  Required switch that queries the active server session for the number of diagnosis errors returned.

- **-ANalyzers**

  Required switch that returns the number of active dynamic analyzers (DP mode) or total analyzers.

- **-Preanalyzed**

  Required switch that queries the active server session for the number of preanalyzed failure files.

- **-DUplicates**

  Required switch that queries the active server session for the number of duplicated failure files containing identical tracking signatures (for example, lot_id or wafer_id).

- **-ABort**

  Required switch that queries the active diagnosis session for a control-C.

- **-PArtitioned**

  Required switch that queries the active server session for the number of fail logs for which partitions have been created.

- **-partitioners**

  Required switch that returns the number of active dynamic partitioners.

- **-processors**

  Required switch that returns the total number of active processors in either mode.

Examples

The following example shows how to use the check command to abort the Tessent Diagnosis server based on the number of diagnosis jobs in the job queue.

```
while { [check -queued] > 0 && ![check -abort] } { # wait }
```

cleanse_history

Scope: Server mode

Removes or lists the specified rows from the HDB tables.

Usage

cleanse_history [-AbandonedFiles] | [-Days *days*] [-Older | -Newer] [-Timestamp *timestamp*]
[-ListOnly]

Arguments

- -AbandonedFiles

  An optional literal that removes all HDB_QUEUE entries having an ABANDONED state.

- -Days *days* [-Older | -Newer] [-Timestamp *timestamp*] [-ListOnly]

  An optional switch and integer that removes HDB_QUEUE and HDB_EVENT entries by the number of *days* back from the current day. You use the -Days days entry in conjunction with the following options:

  -Older | -Newer

  The optional -Older and -Newer switches remove the entries either *days* older or newer than the current day.

  -Timestamp *timestamp*

  An optional switch and date/time construct that specifies the timestamp criteria for the removal of HDB_QUEUE and HDB_EVENT entries. You specify the timestamp using the following format:

  YYYY/MM/DD HH:MM:SS

  Alternatively, you can specify the timestamp using a UTC integer.

  -ListOnly

  An optional literal that lists the HDB entries *days* older than the current day.

Related Topics

query_history

report_history

clear_monitor

Scope: Server mode

Resets the counters/statistics for the specified monitor.

Usage

clear_monitor [*monitors* …]

Arguments

- *monitors*

  Optional replaceable string that specifies the name of a monitor to reset.

Description

This command has no effect on the queue or previously diagnosed files. It only provides a convenient way to set up a monitor for a new batch of failure files without having to delete it.

If no monitor is specified, the counters/statistics for all monitors are reset.

Examples

The following example shows how to reset the counters/statistics for my_monitor.

```
clear_monitor my_monitor
```

Related Topics

report_monitor

clear_status

Scope: Server mode

Clears the accumulated error and warning information displayed in the status area with the report_status command.

Usage

clear_status

Arguments

None.

Examples

The following example shows how to clear the status area.

```
clear_status
```

Related Topics

report_status

delete_analyzer

Scope: Server mode

Deletes one or more specified analyzers from the automatic diagnosis configuration.

Usage

delete_analyzer {***analyzerID_1 analyzerID_2 ... analyzerID_N*** [-force] | -clear | -all}

Arguments

- ***analyzerID_1 analyzerID_2 ... analyzerID_N***

  Required, repeatable string that specifies the identification number of the analyzer to delete.

- -force

  Optional. Use this option when an analyzer fails to respond to the delete_analyzer command. The -force option is a stronger attempt at removing the named analyzer.

- -clear

  Removes analyzers that have died but are still appearing in the server's report_analyzer list. Use this command when you detect a mismatch between the server's reported analyzer list and the actual analyzers that are participating.

- -all

  Deletes all analyzers.

Description

Any diagnosis in process is completed before the analyzer is deleted.

Examples

The following example deletes analyzers 1, 2, and 3.

```
delete_analyzer 1 2 3
```

Related Topics

add_analyzer

report_analyzer

delete_design

Scope: Server mode

Removes the assigned design netlist from a specified monitored directory.

Usage

delete_design ***monitor_id***

Arguments

- ***monitor_id***

  Required string that specifies the monitored directory associated with the specified design netlist.

  The *monitor_id* is the unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Examples

The following example removes the association of the flat_design1 design netlist from the *monitor1* directory in the automatic diagnosis configuration.

```
delete_design monitor1 flat_design1
```

Related Topics

report_monitor

delete_layout

Scope: Server mode

Releases the Calibre Query Server and deletes the layout from memory.

Usage

delete_layout ***monitor***

Arguments

- ***monitor***

  Required string specifying the name of the monitor.

Description

Tessent Diagnosis server no longer generates layout markers if you use this command.

Related Topics

add_layout

delete_monitor

Scope: Server mode

Discontinues the monitoring of a specified directory.

Usage

delete_monitor ***monitor_id1*** *monitor_id2... monitor_idN*

Arguments

- ***monitor_id***

  Required, repeatable string that specifies which monitored directory to remove from the automatic diagnosis configuration.

  The *monitor_id* is the unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Description

The delete_monitor command directs the Tessent Diagnosis server to quit monitoring and processing failure files for a specified directory. Any analyzers bound to a deleted monitor are reinstated when the monitors are re-added with the add_monitor command.

Examples

The following example discontinues the monitoring of the monitor1 and monitor2 directories.

```
delete_monitor monitor1 monitor2
```

Related Topics

report_monitor

delete_partitioner

Scope: Server mode

Deletes one or more specified partitioners from the automatic diagnosis configuration.

Usage

delete_partitioner {***partitionerID_1 partitionerID_2 ... partitionerID_N*** [-force] | -clear | -all}

Arguments

- ***partitionerID_1 partitionerID_2 ... partitionerID_N***

  Required, repeatable string that specifies the identification number of the partitioner to delete.

- -force

  Optional. Use this option when a partitioner fails to respond to the delete_partitioner command. The -force option is a stronger attempt at removing the named partitioner.

- -clear

  Removes partitioners that have died but are still appearing in the server's report_partitioner list. Use this command when you detect a mismatch between the server's reported partitioner list and the actual partitioners that are participating.

- -all

  Deletes all partitioners.

Description

Any diagnosis in process is completed before the partitioners is deleted.

Examples

The following example deletes partitioners 1, 2, and 3.

```
delete_partitioner 1 2 3
```

delete_pattern

Scope: Server mode

Deletes all the test pattern files associated with a specified monitored directory from the automatic diagnosis configuration.

Usage

delete_pattern ***monitor_id***

Arguments

- ***monitor_id***

  Required string that specifies the monitored directory from where to delete test patterns.

  The *monitor_id* is a unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: “1” would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Examples

The following example deletes the association between all test pattern files and the *monitor1* directory for the automatic diagnosis configuration:

```
delete_pattern monitor1
```

Related Topics

report_monitor

delete_reporting_format

Scope: Server mode

Turns off writing of a specified reporting format for the specified monitor.

Usage

delete_reporting_format [*monitor1 monitor2... monitorn*] [-ENCoded] [-TEXT] [-CSV] [-LAYout_marker]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have defined.
- -ENCoded

  An optional switch specifying encoding a suspect's pin-pathname, net-name, and cell-name in the diagnosis output file.
- -TEXT

  A switch deleting the ASCII text format report.
- -CSV

  A switch deleting the comma separated value (CSV) format report.
- -LAYout_marker

  A switch deleting the layout marker format. This switch does not release the Calibre Query Server license or delete the layout from memory—instead, use the delete_layout command.

Description

This command turns off the format you specify. If you have specified other reporting formats, this command has no effect on them. For example, if you specify TEXT and CSV, then if you turn off CSV with this command, the tool still uses the TEXT reporting format.

Related Topics

add_reporting_format

report_reporting_format

delete_layout

delete_reporting_xmap

Scope: Server mode

Turns off writing of SPICE-mapped nets and the Verilog module hierarchy in the diagnosis report.

Usage

delete_reporting_xmap [*monitor1 monitor2... monitorn*] [{-v2lvs | -cell_hierarchy}]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have defined.

- -v2lvs | -cell_hierarchy

  A required switch to specify the type of data to turn off. You must specify at least one of the following options:

  - -v2lvs — Turns off the reporting of SPICE-mapped nets.
  - -cell_hierarchy — Turns off the reporting of the Verilog module hierarchy.

  You can specify any combination of these switches. For example, the following syntax turns off both the SPICE-mapped nets and the Verilog module hierarchy output:

  ```
  delete_reporting_xmap -v2lvs -cell_hierarchy
  ```

Description

This command turns off the data you specify.

Related Topics

add_reporting_xmap

report_reporting_xmap

delete_schedule

Scope: Server mode

Removes previously scheduled time-based analyzer licensing events and previously-scheduled email events.

Usage

delete_schedule ***job_number***

Arguments

- ***job_number***

  A required integer that specifies the licensing or email event you want to remove. You obtain the number from the report_schedule command.

Related Topics

report_schedule

schedule_licenses

email

schedule_email

dofile

Scope: Server mode

Runs the specified dofile.

Usage

dofile ***dofile*** [-history] [-CheckOnly]

Arguments

- ***dofile***

  Required string that specifies the name of a dofile to run.

- -History

  Optional switch that adds commands specified in the dofile to the history buffer. Enables you to replay the dofile commands.

- -CheckOnly

  Optional switch that checks the syntax of the dofile contents.

Description

This is Tcl syntax.

Examples

The following example runs the my_dofile dofile.

```
dofile my_dofile
```

email

Scope: Server mode

Subscribes and unsubscribes email recipients.

Usage

email ***message*** [
-To *recipient1 recipient2...recipientN*] | [-Subscribe *recipient1 recipient2...recipientN*]
| [-Unsubscribe *recipient1 recipient2...recipientN*] | [-Clear]

Arguments

- ***message***

  A required string where you specify a message in the body of the email. If a message contains more then one word, then bound *message* with " " (quotation marks): for example, "A message with more than one word."

- -To *recipient1 recipient2...recipientN*

  An optional switch and string that specifies the email recipients for *message*.

- -Subscribe *recipient1 recipient2...recipientN*

  An optional switch and string that specifies and subscribes the email recipients you list.

- -Unsubscribe *recipient1 recipient2...recipientN*

  An optional switch and string that specifies and unsubscribes the email recipients you list.

- -Clear

  An optional switch that nulls the email subscription list.

Related Topics

delete_schedule

schedule_email

report_schedule

exit

Scope: Server mode

Shuts down automatic diagnosis and exits Tessent Diagnosis server.

Usage

exit [-force]

Arguments

- -force

  An optional switch that terminates the tool session immediately without finishing active processes.

Description

All diagnosis in process are completed before automatic diagnosis is shut down. Use the -Discard option, to immediately exit the program without finishing any currently running processes.

Examples

The following example quits the tool immediately without finishing active processes.

```
exit -force
```

help

Scope: Server mode

Lists commands and associated command syntax.

Usage

help [*command* ...] [-options] [-all]

Arguments

- *command*

  Optional, repeatable string that specifies the name of a command on which to display help. By default, all the commands available in the current state of the system display.

- -options

  Optional switch that displays all options for the specified command.

- -all

  Displays all the commands regardless of the system state and notes the commands currently unavailable.

Examples

The following example displays command help for the add_analyzer command.

```
help add_analyzer
```

history

Scope: Server mode

Displays a list of previously-run commands.

Usage

history [*list_count*] [-Nonumbers] [-Reverse] [-Save *filename*]

Arguments

- *list_count*

  An optional integer that specifies for the tool to display only the specified number (*list_count*) of the recently run commands. If no *list_count* is specified, the tool displays all previously run commands.

- -Nonumbers

  An optional string that specifies for the tool to display the history list without the leading numbers. This is useful for creating dofiles. The default displays the leading numbers.

- -Reverse

  An optional switch that specifies for the tool to display the history list starting with the most recent command rather than the oldest.

- -Save *filename*

  An optional switch that saves the command history to a named file.

Description

The history command is similar to the Korn shell (ksh) history command in Unix. By default, this command displays a list of all previously-run commands, including all arguments associated with each command, starting with the oldest.

Note

The HISTFILE and HISTSIZE ksh environment variables do not control the command history of the tool.

You can perform command line editing if you set the VISUAL or EDITOR ksh environment variable to either emacs, gmacs, or vi editing. Please see the ksh(1) man page for specifics on the various editing modes.

A leading number precedes each command line in the history list that indicates the order in which the commands were entered.

query_history

Scope: Server mode

Creates a Tcl string result from the HDB query.

Usage

query_history [-Sessions] [-Monitors] [-Analyzers] [-Designs] [-QUeue] [-ERrors] [-SQL *sql_query*]

Arguments

- -Sessions

  An optional switch that returns a Tcl string result from the HDB_SESSION table.
- -Monitors

  An optional switch that returns a Tcl string result from the HDB_MONITOR table.
- -Analyzers

  An optional switch that returns a Tcl string result from the HDB_ANALYZER table.
- -Designs

  An optional switch that returns a Tcl string result from the HDB_FILE table.
- -QUeue

  An optional switch that returns a Tcl string result from the HDB_QUEUE table
- -ERrors

  An optional switch that returns a Tcl string result from the HDB_ERROR table.
- -SQL *sql_query*

  An optional switch and legal SQL query. You must put the entire SQL expression in " " (quotation marks). If you embed arguments containing spaces, then put the argument in ' ' (single quotation marks).

Examples

The following example uses the query_history command to exit Tessent Diagnosis server execution when the average diagnosis time exceeds the $limit

```
set max_num_analyzer 6
set loop_i 0
set loop_start_diag -1
set limit 50

while { [ check -analyzers ] < $max_num_analyzer && $loop_i < 200 &&
      ![check -abort] } {

  if { [ catch { add_analyzer generic:1 } ] != 0} {
  puts "Warning: failed to add analyzer at loop [ $loop_i ] "
  }

  if { [ check -analyzers ] == 1 && $loop_start_diag == -1 } {
    start_diagnosis
    set loop_start_diag $loop_i
   puts "Note: start diagnosis after the first analyzer was added at loop
   [ $loop_i ] "
  }

  incr loop_i

}

while { [check -queued] < 1 && ![check -abort] } { }
while { [check -queued] > 0 && ![check -abort] && [check -analyzers] > 0
&& [lindex [split [query_history -sql "select avg(q.diagtime) from
hdb_queue as q, hdb_monitor as m where status='ANALYZED' and
q.monitor_id=m.monitor_id and m.synonym='lym';"] "\n"] 1] < $limit } { }
exit
```

Related Topics

cleanse_history

report_history

report_analyzer

Scope: Server mode

Lists the operational state for the current analyzers.

Usage

report_analyzer *host_name*... [-verbose]

Arguments

- *host_name*

  Optional, repeatable string that specifies the name of a computer running the analyzer on which to report. By default, all the current analyzers are listed.

- -verbose

  Reports detailed information for each analyzer currently running.

Examples

The following example reports on the analyzers currently running.

report_analyzer

A report similar to the following displays:

```
//  command: report_analyzer ---< analyzers >-----------------------------
---------------------------
(1)  blackbird    #1 flogs  Analyzer   ... idle (1 secs)
(2)  blackbird    #1 flogs  Analyzer   ... idle (1 secs)
(3)  blackbird    #1 flogs  Analyzer   ... idle (1 secs)
(4)  blackbird    #1 flogs  Analyzer   ... idle (0 secs)
```

Related Topics

add_analyzer

delete_analyzer

report_history

Scope: Server mode

Queries the current HDB and generates reports.

Usage

report_history [-Pivot {time | memory}] [-Csv]
    [-Sessions] [-Monitors] [-Analyzers | -Partitioners]
    [-Designs] [-Queue] [-Events] [-ERrors] [-SQL *sql_query*]

Arguments

- -pivot {time | memory}

  An optional switch that consolidates the diagnosis events for each failure log and lists either the time or memory consumption for each event.

- -csv

  An optional switch that generates the report history in CSV format that you can pipe to a separate file to use for analysis purposes. For more information, refer to "Usage Example: Analyze Diagnosis Performance and Throughput" on page 369.

- -Sessions

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_SESSION table.

- -Monitors

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_MONITOR table.

- -Analyzers

  An optional switch that queries the current HDB and returns a list of all analyzers in the HDB_ANALYZER table.

- -Partitioners

  An optional switch that queries the current HDB and returns a list of all partitioners in the HDB_ANALYZER table.

- -Designs

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_FILE table.

- -Queue

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_QUEUE table.

- -Events

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_EVENT table.
- -ERrors

  An optional switch that queries the current HDB and returns a list of all entries in the HDB_ERROR table.
- -SQL *sql_query*

  An optional switch and legal SQL query. You must put the entire SQL expression in " " (quotation marks). If you embed arguments containing spaces, then put the argument in ' ' (single quotation marks).

Description

See "Server History" on page 366 for details.

Examples

The following example generates a memory consumption report formatted in CSV and saved to the file base_diag_memory.csv.

```
report_history -pivot memory -csv > base_diag_memory.csv
```

Related Topics

cleanse_history

query_history

report_licenses

Scope: Server mode

Reports licenses currently being used by automatic diagnosis.

Usage

report_licenses [-user]

Arguments

- -user

  Optional switch and string pair that displays all licenses currently being used by the current user. By default, all license usage displays.

Examples

The following example displays all licenses currently being used by the automatic diagnosis session.

report_licenses

The following report displays:

```
Licenses:
4 yieldascandiag
2 yieldaedtdiag
```

report_log

Scope: Server mode

Displays the contents of the named monitor’s log file.

Usage

report_log [*monitor_id*]

Arguments

- *monitor_id*

  Optional, repeatable string that specifies a monitor. If no monitors are specified, the log files of all monitors displays.

Description

Use this command to review the contents of the monitor's log file where the instances of the errors have been written.

report_monitor

Scope: Server mode

Lists the status for all currently monitored directories.

Usage

report_monitor *monitor_id* ...

Arguments

- *monitor_id*

  Optional, repeatable string that specifies a monitored directory. If no directories are specified, the status of all monitored directories displays.

  The *monitor_id* is a unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Description

The report_monitor command reports the diagnostic statistics for the monitored directories, including the associated design and test patterns. In the event that a monitor times out due to a CPU or wall time limit, the report_monitor command reports time out statistics.

Examples

The following example reports the status of all directories being monitored.

report_monitor

A report similar to the following displays for each monitored directory:

```
--- monitors ------------------------------
(1) Alpha
directory: /wv/dft06918/pmc/testerAA
results: /wv/dft06918/pmc/TESTERA_RESULTS
: adb.sql INACTIVE
design: /wv/dft06918/pmc/pmc.flat.gz
pattern set 0: /wv/dft06918/pmc/tester_pat.wgl
10 total
10 done      [ 100% ]
0 errors     (   0% )
1 timeout      ( 10% )
10 diagnosed ( 100% ) 75sec/diagnosis
```

Related Topics

delete_monitor

report_network

Scope: Server mode

Reports the network connectivity between the server and the specified host.

Usage

report_network ***hostname***

Arguments

- ***hostname***

  Required string that specifies the name of the host on which to report.

Examples

The following example reports the server connectivity for host1.

```
report_network host1
```

report_options

Scope: Server mode

Reports the optional diagnosis settings for the specified monitor.

Usage

report_options ***monitor***

Arguments

- monitor

  Required string that specifies the name of the monitor on which to report.

Examples

The following example reports the set_diagnosis_options settings for monitor1.

```
report_options monitor1
```

report_partitioner

Scope: Server mode

Lists the operational state for the current partitioners.

Usage

report_partitioner *host_name*... [-verbose]

Arguments

- *host_name*

  Optional, repeatable string that specifies the name of a computer running the partitioner on which to report. By default, all the current partitioners are listed.

- -verbose

  Reports detailed information for each partitioner currently running.

Examples

The following example reports on the partitioners currently running.

report_partitioner

A report similar to the following displays:

```
//  command: report_partitioner ---< partitioners >-----------------------
----------------------------
(1)  blackbird    #1 flogs  Partitioner   ... idle (9 secs)
(2)  blackbird    #1 flogs  Partitioner   ... idle (8 secs)
(3)  blackbird    #1 flogs  Partitioner   ... idle (8 secs)
```

report_reporting_format

Scope: Server mode

Reposts reports for the currently selected reporting formats for a specified monitor.

Usage

report_reporting_format [*monitor1 monitor2* ... *monitorn*]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have defined.

Related Topics

add_reporting_format

delete_reporting_format

report_reporting_xmap

Scope: Server mode

Reposts SPICE-mapped net data or Verilog module hierarchy data for the currently selected XMAP output formats for a specified monitor.

Usage

report_reporting_xmap [*monitor1 monitor2 ... monitorn*]

Arguments

- *monitor1 monitor2... monitorn*

  An optional string specifying the name of the monitor or multiple monitors. If you specify no monitor, then the tool applies this command to all monitors you have defined.

Description

Appends a report of SPICE-mapped pin paths and net names, and the Verilog module hierarchy, in a XMAP table to the diagnosis report.

Related Topics

add_reporting_xmap

delete_reporting_xmap

report_schedule

Scope: Server mode

Lists currently scheduled time-based license management events and currently-scheduled email events.

Usage

report_schedule

Arguments

None.

Related Topics

delete_schedule

schedule_email

email

schedule_licenses

report_status

Scope: Server mode

Reports on the total number of queued failure files, monitors, and analyzers for the active diagnosis session.

Usage

report_status

Arguments

None.

Description

Aborted analyzers are reported also. The reported information is a subset of the information reported with the watch command.

Examples

The following example reports on the total number of queued failure files, monitors, and analyzers for the active diagnosis session.

```
report_status
```

Related Topics

clear_status

watch

report_variable

Scope: Server mode

Reports the auto-processing mode variable settings.

Usage

report_variable [*variable*]

Arguments

- *variable*

  Optional string that specifies the name of the variable to display. For a list of all the variables, see "Tessent Diagnosis Server Variables."

Description

Use the Tcl built-in set command to set the variables.

Examples

The following example reports the current settings of the job_options variable.

```
report_variable job_options
```

resume_diagnosis

Scope: Server mode

Prerequisites: Diagnosis is suspended for a monitored directory with the suspend_diagnosis command.

Resumes the diagnosis of failure files in the specified monitored directory.

Usage

resume_diagnosis ***monitor_id*** ...

Arguments

- ***monitor_id***

  Required, repeatable string that specifies a suspended monitored directory.

  The *monitor _id* is the unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: “1” would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Description

The resume_diagnosis command resumes diagnosis after a suspend_diagnosis command is run. The diagnosis continues where it left off when suspended.

Examples

The following example resumes the diagnosis of the failure files in the *monitor1* and *monitor2* directories.

```
resume_diagnosis monitor1 monitor2
```

Related Topics

suspend_diagnosis

schedule_email

Scope: Server mode

Specifies scheduling setup for the server’s email facility.

Usage

schedule_email ***message*** { **-every** ***every_time*** | **-at** ***time*** [-daily] }

Arguments

- ***message***

  A required string where you specify a message in the body of the email. If a message contains more then one word, then bound the message with “ ” (quotation marks): for example, “A message with more than one word.”

  ```
  schedule_email { query status } -every 1:00
  ```

- **-every *every_time***

  A required literal and numeric time construct that specifies sending automatic email based on a fixed time regardless of day. You specify *every_time* using the following syntax:

  ```
  hh:mm
  ```

- **-at *time*** [-daily]

  A required literal and numeric-alphabetic time construct that specifies sending automatic email at a specific time of day. You specify time using the following syntax:

  ```
  hh:mmam
  ```

  or

  ```
  hh:mmpm
  ```

Description

The schedule_email command only supports a string message enclosed in “ ” (quotation marks), or the following commands enclosed with braces ({ }):

- { query status }
- { query_history -sql *sql_command* }

Related Topics

delete_schedule

report_schedule

email

schedule_licenses

Scope: Server mode

Specifies scheduling setup for time-based analyzer license management.

Usage

schedule_licenses ***max_licenses*** **-at** ***time*** [-daily] }

Arguments

- ***max_licenses***

  A required integer that specifies the maximum number of licenses the server uses for analyzers. If the *max_licenses* count decreases based on schedule, then the server deletes the analyzer to match the number of scheduled licenses. If the *max_licenses* count increases based on schedule, then the server schedules new analyzers using the job scheduler technique.

- **-at** ***time*** [-daily]

  A required literal and numeric-alphabetic time construct that specifies scheduling analyzer licenses at a specific time of day. You specify time using the following syntax:

  ```
  hh:mmam
  ```

  or

  ```
  hh:mmpm
  ```

Related Topics

delete_schedule

report_schedule

set_diagnosis_options

Scope: Server mode

Specifies optional settings for diagnosis.

Usage

set_diagnosis_options ***monitor_id***...
[-Mode { **Auto** | **Scan** | **Chain** }]
[-Verify_patterns { **ON** [**Noverbose** | **Verbose**] | **OFf** }]
[-Report { **Default** | ***percentage*** }]
[-AT_speed { **ON** | **OFf** }]
[-MAx_suspects { **Default** | ***limit*** }]
[-Time_limit { **OFf** **|** ***seconds*** }]
[-WALL_time_limit { **OFf** | ***seconds*** }]
[-Pattern_sampling { ***failing_pattern_limit*** | **OFf** } { ***passing_pattern_limit*** | **OFf** }]
[-FAILurefile_mismatch_verbosity { **Default** | ***number*** | **All** }]
[-BRidge_analysis { **ON** | **OFf** }]
[-CHEck_last_shift_value { **ON** | **OFf** }]
[-X2B_mismatch { **Ignore** | **Warning** [**Noverbose** | **Verbose**] | **Error** }]
[-B2X_mismatch { **Ignore** | **Warning** [**Noverbose** | **Verbose**] | **Error** }]
[-CELl_internal_analysis { **ON** | **OFf** }]
[-MAX_Faulty_chains ***failed_chain_threshold***]
[-CHAIN_DIAgnosis_result { **CEll** | **CHain** }]
[-CHAIN_LIB_internal_pathname { **OFf** | **ON** }]
[-JOB_memreq ***gigabytes***]
[-ABORT_DIAGNOSE_COMPOUND_faults { **ON** | **OFf** }]
[-ABORT_DIAGNOSE_MANY_faulty_chains { **ON** | **OFf** }]
[-ABORT_DIAGNOSE_MINIMUM_Chain_failing_probability ***number***]
[-ABORT_DIAGNOSE_MINIMUM_Scan_pattern_failing_probability ***number***]
[-IGNORE_TOOL_version { **OFf** | **ON** }]
[-INCLude_fail_signatures_size { ***maximum_rows_per_table*** | **MAX** }]
[-INCLUDE_BRIDGE_to_power { **ON** | **OFf** }]
[-GRoss_delay { **ON** | **OFf** }]
[-COMPOUND_Hold_time_fault_diagnosis { **ON** | **OFf** }]
[-INCLUDE_DFM_rules { **ON** | **OFf** }]
[-INCLUDE_RCD_constants { **ON** | **OFf** }]
[-EXPected_value { **ON** | **OFf** }]
[-CYcle_offset *N*]
[-MISSing_rcd_action { **Ignore** | **Warn** | **Error** }]
[-CELL_PORT_BRIDGE_analysis { Auto | ON | OFf }]
[-LANDMARK_polygon_limit ***integer***]
[-INCLUDE_CORE_INSTANCe_name { **ON** | **OFf** }]
[-CELL_FAULTS *udfm_pathname* [-RCAD ON]]
[-MBFF_TAG_SCI_TEMPlate ***template_string***]
[-MBFF_TAG_SCO_TEMPlate ***template_string***]

[-CHAINDIAG_PATTERN_SAMpling { ***integer*** | **OFf** } { ***integer*** | **OFf** }]
[-CORE_INSTANCE_ERROR { **ON** | **OFf** }]
[-INCLUDE_SRC_SINK_cells_in_marker { **ON** | **OFf** }]
[-REPORT_PIN_LOCations { **ON** | **OFf** }]
[-ADD_SPICE_LAYERS_TO_DIAGNOSIS_REPORT { **ON** | **OFf** }]

Arguments

- ***monitor_id***

  Required, repeatable string that specifies the monitored directory to modify diagnosis settings for. The diagnosis of all failure files found in the specified directory use the specified settings.

  The *monitor_id* is a unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: “1” would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

- -Mode { **Auto** | **Scan** | **Chain** }

  A switch and literal pair that determines what type of diagnosis is performed. The following options are available:

  Auto — A literal that turns on Tessent Diagnosis server to determine what type of diagnosis to run based on the contents of the failure file. If the failure file contains a chain test keyword and chain test failures, the tool performs a chain diagnosis. If the failure file contains scan test failures, the tool preforms a scan diagnosis.

  If the failures include a padding cycle of a scan chain, the tool runs a chain diagnosis.

  If chain failures are not preceded by a chain test keyword, use the -faulty_chain switch with the diagnose_failures command to run chain diagnosis.

  If the failure file contains both chain failures and scan failures, you must use the *scan test* and *chain test* keywords to identify the data. This is the default.

  Scan — A literal that turns on diagnosis only on failure files that contain scan fail data.

  If a padding cycle of a scan chain fails, the scan chain is defective, and the entire chain is masked out before running logic diagnosis. The diagnosis resolution may suffer due to masked chain(s). You should run chain diagnosis in this situation.

  Chain — A literal that turns on diagnosis only on failure files that contain a chain test section.

- -Verify_patterns { **ON** [**Noverbose** | **Verbose**] | **OFf** }

  A switch and literal pair that turns on or turns off verification of the patterns you specify when you enter a start_diagnosis command for the first time.

Note

The -verify_patterns switch is deprecated. See verify_patterns in the *Tessent Shell Reference Manual* for details and examples of creating, updating, and loading a startup cache.

Siemens strongly recommends that you do not turn off pattern verification.

ON — A literal that turns on verification of the patterns. If verification is turned on, the first diagnose_failures command you enter simulates the patterns and compares the capture values with the expected values in the patterns. The tool stops the verification process upon the first occurrence of an error and enter an error message. You can then use the existing command, report_failures -Pdet, to determine the complete set of failing patterns.

The tool performs this verification just once per tool session for a particular set of patterns, no matter how many subsequent diagnose_failures commands you enter.

Noverbose — When pattern verification is turned on and a failure file mismatch occurs, then the tool does not print out the cell pathname associated with the mismatched bit. When you use this switch, the ‘read failure’ does not produce the cell paths either.

Verbose — When pattern verification is turned on and a failure file mismatch occurs, then the tool prints out the cell pathname associated with the mismatched bit. This is the default.

OFf — A literal that turns off verification of the patterns.

Note

Do not turn off pattern verification to achieve run time reduction. Test pattern verification is required for accurate diagnosis results. You can reduce execution time by using a startup cache.

- -Report { **Default** | ***percentage*** }

  A switch and value pair that changes the diagnosis report generated by the tool. You have the following two report choices:

  Default — A literal that specifies to report the following for each symptom:

  - All suspects with score greater than or equal to 80.
  - Up to top three suspects even if the suspects’ scores are less than 80.
  - If the suspect beyond the top three has the same score with the 3rd suspect, the tool reports the suspect only if one of the following occurs:

    i. If the suspect has the same score as suspect #3, and the suspect can explain the same or higher number of failing patterns.

    a. If the suspect has the same score and fail_match, the suspect is reported only if it has the same passing pattern mismatch.

percentage — An integer in the range 1 to 100 that specifies a percentage of the suspects the tool should report for each symptom. If you specify 75, for example, and there are four suspects for a particular symptom, the tool reports the top 75% (top three) of them.

- -AT_speed { **ON** | **OFf** }

  A switch and literal pair that specifies performing at-speed failure diagnosis. See "At-Speed Failure Diagnosis" for complete information, including the conditions Tessent Diagnosis server uses when diagnosing at-speed failures.

- -MAx_suspects { **Default** | ***limit*** }

  A switch and value pair limiting the number of suspects per symptom in the diagnostics report. Choose one of the following options:

  Default — A literal that specifies reporting 100 suspects per symptom.

  limit — An integer that specifies to report this number of suspects per symptom.

- -Time_limit { **OFf** | ***seconds*** }

  A switch and value pair that specifies a time limit, in seconds, for diagnosing a single failure file. This time includes the failure file verification time, but not the pattern verification time. By default, the test patterns and each failure file is verified for consistency when a diagnosis is run.

  Specifying this switch and a time limit overrides the analyzer_timelimit server variable.

- -WALL_time_limit { **OFf** | ***seconds*** }

  A switch and value pair that specifies a wall time limit, in seconds, for diagnosing a single failure file. When you specify a wall time limit (***seconds***) and a diagnosis job exceeds the limit, then the tool aborts the diagnosis run with an error message.

  Note

  The wall time limit is independent of the CPU time limit (using the -Time_limit switch). If you set to both the CPU and wall time limit, then the tool aborts the diagnosis when either limit is reached.

  If the wall time limit is reached during diagnosis, the tool aborts the diagnosis job and no diagnosis report file is generated.

- -Pattern_sampling { ***failing_pattern_limit*** | **OFf** } { ***passing_pattern_limit*** | **OFf** }

  Optional switch with two values that specifies pattern limits for logic diagnosis. The first value is for the failing pattern limit, and the second value is for the passing pattern limit. The default is off for both values.

  Use this switch to limit the patterns used for logic diagnosis and reduce run time. By default, all failing and passing patterns are used for diagnosis. To ensure adequate diagnosis resolution, the pattern sample must be 32 or greater.

Logic diagnosis patterns with several capture cycles require more simulation time and have less resolution. To avoid losing resolution, Tessent Diagnosis server selects patterns with the least amount of capture cycles for the diagnosis. Pattern sampling options include:

failing_pattern_limit — Must be an integer of 32 or higher.

passing_pattern_limit — Must be an integer of 32 or higher. Specified number of passing patterns is selected from the first passing patterns in the set.

- -FAILurefile_mismatch_verbosity { **Default** | ***number*** | **All** }

  A switch and value pair that controls the number of pattern mismatches displayed during the verification of the failure file. You have the following report choices:

  Default — A literal that specifies to display 20 failing patterns. This is the default.

  number — An integer that specifies to report this number of failing patterns.

  All — A literal that specifies to report all failing patterns.

- -BRidge_analysis { **ON** | **OFf** }

  A switch and literal pair that determines if diagnosis includes an analysis to identify bridge suspects. A bridge suspect is identified when faulty behavior is observed for the implicated nets and the appropriate aggressor-victim conditions are met. Currently 2-way and 3-way bridges are considered and this switch is applicable to both.

  Options include:

  ON — Bridge analysis is turned on. Default setting.

  OFf — Bridge analysis is turned off.

  There are two cases where it may be desirable to turn off bridge analysis.

  Case 1 — Because the analysis is currently based only on logical responses, it is possible that a bridge suspect may not be physically possible. In cases where bridge suspects are verified as not physically possible, you may want to rerun the diagnosis with bridge analysis turned off.

  Case 2 — In cases where very few failing patterns are captured for a failing device, and bridge analysis may return a large number of suspects, you can turn off the bridge analysis to reduce the number of suspects.

- -CHEck_last_shift_value { **ON** | **OFf** }

  An optional switch and literal pair that controls whether last shift values are considered for transition launching in resistive open diagnosis as well as cell internal diagnosis.

  ON — Turns on last shift values consideration for transition launching in resistive open diagnosis. This is the default.

  OFf — Turns off considering last shift values for transition launching in resistive open diagnosis. Use this option if you believe that the shift is very slow speed such that the transition launched from last shift is irrelevant.

- -X2B_mismatch { **Ignore** | **Warning** [**Noverbose** | **Verbose**] | **Error** }

  A switch and literal pair that determines how X2B mismatches are handled during diagnosis. X2B mismatches occur during pattern verification when the simulator returns binary values instead of the expected X values.

  The tool most likely introduces X2B mismatches because it removes cell constraints when performing pattern verification and diagnosis. This turns on the tool to perform diagnosis using one flat model for potentially multiple modes, each of which have different constraints, or to use flat models that were written at different times. You can ignore these X2B mismatches.

  Ignore — This is the default. Does not display X2B mismatch information. Pattern verification passes.

  Warning Noverbose — Warns at the end of pattern verification of any X2B mismatches. Pattern verification can still pass.

  Warning Verbose — Prints during pattern verification any mismatched X2B bits. Pattern verification can still pass.

  Error — Prints the X2B mismatches for the first mismatched pattern and pattern verification fails that terminates the diagnosis or verify_patterns command.

- -B2X_mismatch { **Ignore** | **Warning** [**Noverbose** | **Verbose**] | **Error** }

  A switch and literal pair that determines whether B2X mismatches are handled during diagnosis. B2X mismatches occur during pattern verification when the simulator returns X values instead of the expected binary values. These mismatches could be introduced by different problems, for example, software enhancements to the simulator or using a mask file when reading patterns.

  Ignore — Does not display B2X mismatch information. Pattern verification passes.

  Warning Noverbose — Warns at the end of pattern verification of any B2X mismatches. Pattern verification can still pass.

  Warning Verbose — Prints during pattern verification any mismatched B2X bits. Pattern verification can still pass.

  Error — Prints the B2X mismatches for the first mismatched pattern and pattern verification fails that terminates the diagnosis or verify_patterns command.

- -CELl_internal_analysis { **ON** | **OFf** }

  A switch and literal pair that determines if Tessent Diagnosis server performs analysis to diagnose defective library cells and distinguish them from defects on nets interconnecting library cells.

- -MAX_Faulty_chains ***failed_chain_threshold***

  An optional switch and integer pair that specifies the maximum number of failing scan chains within a single datalog that are diagnosed using chain fault model. When a failure log has multiple failed_chains, there can either be multiple defects in scan chains, or a single defect affecting the scan control signals. The default failed chain threshold is 2.

- o If there are one or two failed_chains, then the tool assumes the defect is on each individual chain and uses the scan chain fault model to run diagnosis. This is the default behavior of the tool.
- o If failed_chains is greater than two, then the tool assumes the defect is on a global control signal. If "set_diagnosis_options -ABORT_DIAGNOSE_MANY_faulty_chains" is OFf, it uses the clock/ scan_enable fault model to run diagnosis. Otherwise, the tool aborts diagnosis for this case.

Changing failed chain threshold from the default value (2) causes the following effects:

- o If failed_chains is greater than *failed_chain_threshold*, then the tool assumes the defect is on a global control signal. If "set_diagnosis_options -ABORT_DIAGNOSE_MANY_faulty_chains" is OFf, it uses the clock/ scan_enable fault model to run diagnosis. Otherwise, the tools aborts diagnosis for this case.
- o If there are one, two, ... *failed_chain_threshold* faulty chains, then the tool assumes the defect is on each individual chain and uses the scan chain fault model to run diagnosis on each of the defects.

For example, a failure file contains six failed_chains. You can diagnose each of these six failed-chains as individual defects by setting this switch as follows:

```
set_diagnosis_options -max_faulty_chains 6
```

During chain diagnosis, if the tool reaches the max faulty chains limit, by default the diagnosis report includes information about the failing chains but not the failing scan cells. Suppose a failure file contains two failing scan chains and the -max_faulty_chains option is set to 1. The resulting diagnosis report displays information for the two failing chains. For example:

```
#total_symptoms=2 #total_suspects=0 total_CPU_time=0.00sec

diagnosis_mode=chain

#symptoms=2 #suspects=0  CPU_time=0.00sec  fail_log=flog1
symptom=1   #suspects=0   faulty_chain=chain1   fault_type=STUCK
symptom=2   #suspects=0   faulty_chain=chain3   fault_type=STUCK
```

- -CHAIN_DIAgnosis_result { **<u>CEll</u>** | **CHain** }

  Optional switch and literal pair that determines if a preliminary high-level chain diagnosis is performed versus a standard diagnosis. A preliminary high-level chain diagnosis only reports one failure per chain; the dominant failure.

  Options include:

  <u>CEll</u> — Runs the standard diagnosis of chain plus scan patterns and chain plus scan failures to determine cell defects. This is the default.

CHain — Runs a preliminary high-level diagnosis of the chain failures in the failure file and reports the name, length, and fault type of the chains that failed.

- -CHAIN_LIB_internal_pathname { **<u>OFf</u>** | **ON** }

  An optional switch and literal pair that reports the internal instance pathname within the library for each scan latch in a suspect scan cell.

  When set to ON, the tool reports this information in the chain diagnosis report's "lib_internal_pathname" column. If there is no instance name, the tool reports "". If you specify the add_reporting_format command with the -csv switch, then the tool also includes this information in the CSV output. See "Chain Diagnosis Section."

- -JOB_memreq ***gigabytes***

  An optional switch and integer pair that specifies the amount of memory to allocate for monitor(s). This option supersedes the job_memreq variable's value for the specified monitor(s).

- -ABORT_DIAGNOSE_COMPOUND_faults { **<u>ON</u>** | **OFf** }

  An optional switch and literal pair that specifies whether or not the tool diagnosis the logic part of compound defects during chain diagnosis. The default to "ON", meaning the tool does not diagnose the logic portion of compound defects.

  See "Abort Conditions for Chain Diagnosis."

- -ABORT_DIAGNOSE_MANY_faulty_chains { **<u>ON</u>** | **OFf** }

  An optional switch and literal pair that instructs the tool to skip diagnosing when #faulty_chains is larger than the failed chain threshold specified with the set_diagnosis_options -MAX_Faulty_chains switch. The default is "ON".

  When you set this option to OFF and the number of faulty chains is greater than the number specified by -MAX_Faulty_chains, the tool attempts to run scan enable and clock tree diagnosis that requires failure data for scan patterns. If no failure for scan patterns is available, the tool errors out with a message that indicates that the number of faulty chains is greater than the number specified by -MAX_Faulty_chains.

  See "Abort Conditions for Chain Diagnosis."

- -ABORT_DIAGNOSE_MINIMUM_Chain_failing_probability ***number***

  An optional switch and integer pair that specifies the minimum chain fail probability for the tool to abort chain diagnosis. The default is 1. The minimum is 0, which means no abort. The maximum value you can specify is 100.

  See "Abort Conditions for Chain Diagnosis."

- -ABORT_DIAGNOSE_MINIMUM_Scan_pattern_failing_probability ***number***

  An optional switch and floating point pair that specifies the minimum scan pattern fail probability for the tool to abort chain diagnosis. The default is 0.1. The minimum is 0, which means no abort. The maximum value you can specify is 100.

  See "Abort Conditions for Chain Diagnosis."

- -IGNORE_TOOL_version { **OFf** | **<u>ON</u>** }

  An optional switch and literal pair that instructs the tool to ignore the tool version when loading a startup cache. If the tool versions are different—that is, the startup cache was created by an older version—the tool issues a warning. By default, this option functions as follows with the following verify_patterns options:

  - With -create_startup_cache: Has no impact. The tool creates the same startup cache whether -ignore_tool_version is on or off.
  - With -update_startup_cache: Causes a conflict error.
  - With -load_startup_cache: Turns off the tool version check and uses the startup cache regardless of the tool version used generate or update the cache.

  When set to OFF, the tool issues an error and does not load the startup cache if the tool version used to create the startup cache is older than the current tool version.

- -INCLude_fail_signatures_size { ***maximum_rows_per_table*** | **MAX** }

  An optional switch and integer pair that specifies the maximum number of rows per table for reporting failure signature information in the diagnosis report. The default is 256. If you do not want the failure signatures included in the diagnosis report, then specify 0 (zero) with this switch. Conversely, if you require all signature information, then use the keyword MAX with this switch. See "Failure Signature Information in the Diagnosis Report."

- -INCLUDE_BRIDGE_to_power { **<u>ON</u>** | **OFf** }

  An optional switch and literal pair that specifies performing analysis and reporting of potential bridging defects between the STUCK suspect net, and a power or ground line during layout-aware diagnosis. The default is "ON".

  See "Power and Ground Bridge Reporting."

- -GRoss_delay { **ON** | **<u>OFf</u>** }

  An optional switch and literal pair that specifies performing analysis and reporting of gross delay defects. The default is "OFf".

  See "Gross Delay Defect Diagnosis."

- -COMPOUND_Hold_time_fault_diagnosis { **ON** | **<u>OFf</u>** }

  An optional switch and literal pair that specifies performing analysis and reporting of slow clock compound hold-time defects. The default is "OFf". This option only applies to hold times on shift paths, not capture paths.

  See "Slow Clock Compound Hold-Time Diagnosis."

- -INCLUDE_DFM_rules { **<u>ON</u>** | **OFf** }

  An optional switch and literal pair that specifies to perform DFM hit reporting. The default is "ON".

  See "Diagnosis for Design for Manufacturability Analysis."

- -INCLUDE_RCD_constants { **ON** | **OFf** }

  An optional switch and literal pair that specifies to populate the LDB with RCD constants. The default is "ON".

  See "Diagnosis for Root Cause Deconvolution Analysis."

- -EXPected_value { **ON** | **OFf** }

  An optional switch and literal pair that specifies whether to verify expected pattern values against actual pattern values and return the results in the failure log. When set to OFF, the tool suppresses this check and ignores any missing expected values.

  Caution

  The -expected_value switch turns off failure log verification, which means that you do not know whether there is a discrepancy between the failure log and the patterns. Use this switch with extreme caution.

- -CYcle_offset *N*

  An optional switch and integer pair that specifies to adjust the cycles in a cycle-based failure file by the value *N*. Tessent Diagnosis uses the adjusted cycles for failure verification and diagnosis. See "Cycle Offset Adjustment for Failure Files."

- -MISSing_rcd_action { **Ignore** | **Warn** | **Error** }

  An optional switch and literal pair that checks whether the LDB is populated with RCD constants. The tool checks the LDB at the beginning of the diagnosis run and performs one of the following actions when the RCD constants for the current flat file and pattern set are not present:

  Ignore — Takes no action and diagnosis proceeds.

  Warn — Issues a warning and diagnosis proceeds. This is the default.

  Error — Issues an error and diagnosis halts.

  Use this switch to ensure that the LDB contains the RCD constants required for RCD analysis with Tessent YieldInsight. The warning and error outputs remind you to run create_feature_statistics, as needed, to generate the RCD constants.

- -CELL_PORT_BRIDGE_analysis { Auto | ON | OFf }

  An optional switch and literal pair that specifies whether to perform cell port bridge diagnosis. By default, the tool does not perform cell port bridge diagnosis for layout-aware diagnosis but does perform cell bridge diagnosis for cell-aware diagnosis. For more information, refer to "Cell Bridge Port Diagnosis Reporting."

- -LANDMARK_polygon_limit ***integer***

  An optional switch and integer that controls the number of landmark polygons that are written. The tool uses a proximity-to-defect algorithm to limit the number of landmark polygons for global signals such as power, ground, and scan enable. Specify an integer greater than 99. The default is 10000.

- -INCLUDE_CORE_INSTANCe_name { **ON** | **OFf** }

  An optional switch and literal pair that specifies whether to generate the layout-aware hierarchical diagnosis report with chip-level pin and net names rather than the default core-level names. For more information, refer to "Running Layout-Aware Diagnosis Using a Core-Level LDB."

- -CELL_FAULTS *udfm_pathname*

  An optional switch and string pair that specifies a UDFM (.udfm) file generated by Tessent CellModelGen that is required for performing cell-aware diagnosis. For details, refer to "Cell-Aware Diagnosis."

  When specified, the tool performs a series of validation checks to ensure that within Tessent CellModelGen, the UDFM file was generated using the proper switches and values, and issues warnings if this is not the case. In addition, for flat models, the tool verifies the flat model against the UDFM and issues a warning if the flat model contains cells not defined in the UDFM file.

- -RCAD ON

  An optional switch that turns on RCAD constant creation and performing cell-aware diagnosis that includes the RCAD constants. Specify this switch with the -cell_faults switch. For details, refer to "Performing Cell-Aware Diagnosis With RCD" on page 247.

- -MBFF_TAG_SCI_TEMPlate ***template_string***

  An option and string pair that specifies to set the default internal pin tag naming convention for SCI pins for elements inside multi-bit flip-flops in the format "*template_string N*", where *N* is the index of a master element within the multi-bit flip-flop, starting with 1. For usage details, refer to "Multi-Bit Flip-Flop Handling" on page 112.

- -MBFF_TAG_SCO_TEMPlate ***template_string***

  An option and string pair that specifies to set the default internal pin tag naming convention for SCOs for elements inside multi-bit flip-flops in the format "*template_string N*", where *N* is the index of a master element within a multi-bit flip-flop, starting with 1. For usage details, refer to "Multi-Bit Flip-Flop Handling" on page 112.

- -CHAINDIAG_PATTERN_SAMpling { ***integer*** | **OFf** } { ***integer*** | **OFf** }

  An optional switch and integer pair that specifies to limit diagnosis analysis to the specified number of failing and passing patterns, where the first integer is the number of failing patterns and the second integer is the number of passing patterns. The minimum value for *integer* is 32. This option reduces chain diagnosis run time with a potential minor impact on the quality of the diagnosis results. You can turn off sampling for either failing or passing patterns.

- -CORE_INSTANCE_ERROR { **ON** | **OFf** }

  An optional switch and literal pair that specifies whether core name and core instance path mismatches between the tracking information in the failure file and an LDB are handled as warnings or errors. This option is used for hierarchical diagnosis.

The default is on, in which mismatches are handled as errors. In addition, when opening an LDB that contains core instances, the layout fails to open with an error if the -chip_design_name option is unspecified.

- -INCLUDE_SRC_SINK_cells_in_marker { **<u>ON</u>** | **OFf** }

  An optional switch and literal pair that specifies whether to include source/sink cell Open suspect information in layout marker files. For more information, refer to "Source/Sink Polygon Layout Markers for Open Diagnosis Suspects" on page 253.

- -REPORT_PIN_LOCations { **ON** | **<u>OFf</u>** }

  An optional switch and literal pair that specifies whether to include a PIN_LOCATION table in the XMAP table of a layout-aware diagnosis report. The default is off, which does not print the table.

- -ADD_SPICE_LAYERS_TO_DIAGNOSIS_REPORT { **ON** | **<u>OFf</u>** }

  An optional switch and literal pair that specifies whether to include a spice_layer column in the CELL_DEFECT_LOCATION table of a layout-aware diagnosis report. The default is off, which does not print the column. Turn on this switch and enable cell-aware diagnosis to add the spice_layer column to the report.

Description

The set_diagnosis_options command specifies global settings that determine the behavior of subsequent automatic diagnosis sessions associated with a specified monitored directory. These global settings remain in effect until another set_diagnosis_options command is run.

Examples

The following example runs only scan diagnosis on failure files contained in monitored directory, monitor1.

```
set_diagnosis_options monitor1 -mode scan
```

The following example sets the passing pattern limit to 64.

```
set_diagnosis_options -pattern_sampling off 64
```

Related Topics

Tessent Diagnosis Server Variables

set_diagnosis_resource_configuration

Applies diagnosis resource requirements for the specified mode and options.

Usage

set_diagnosis_resource_configuration [-balanced | -memory [*memory_target*] | -runtime]
[-scheduler *sge* | *lsf*] [-job_limit *job_limit*]

Arguments

- -balanced | -memory [*memory_target*] | -runtime

  An optional switch with optional integer that specifies the mode. The following modes are available:

  -balanced: The tool tries to balance resources used and run time. This is the default.

  -memory [*memory_target*]: The tool sets the best partitioners and analyzers targets within the provided memory budget *memory_target*. If *memory_target* is not provided, the tool applies a budget of three times the predicted partitioner usage.

  -runtime: The tool's primary concern is the wall time for the DP run.

- -scheduler *sge* | *lsf*

  An optional switch and literal that specifies the SGE or LSF grid scheduler. The default is LSF.

- -job_limit *job_limit*

  An optional switch and integer that specifies the limit for the available number of jobs.

Examples

Example 1

This example specifies that a single partitioner is used. Up to 10 dynamic-partitioning analyzers will be added for this run. Therefore, there are 10 possible scenarios with 1 to 10 analyzers. The tool reports the number of jobs and total memory usage for each scenario. The tool dynamically determines whether a new analyzer is needed to maximize the throughput and allocates a new analyzer when it is needed. The actual number of analyzers used can be up to 10 times the number of partitioners.

```
//  command: set_diagnosis_resource_configuration
//  Note: ... Analyzing resource requirements
//  -----------------------  -----------------------
//  Estimated memory requirement for each resource
//  -----------------------  -----------------------
//  Partitioner              3
//  Analyzer                 2
//  -----------------------  -----------------------
//  Note: Analyzer resource estimate assumes partition size is 20% of
original design. Memory requirement for analyzer is automatically adjusted
for partition size before job scheduling.//  Note: Total number of failure
files to be diagnosed: 10

#Partitioner  #Analyzer  #Job  Memory(GB)
------------  ---------  ----  ----------
           1          1     2           5
           1          2     3           7
           1          3     4           9
           1          4     5          11
           1          5     6          13
           1          6     7          15
           1          7     8          17
           1          8     9          19
           1          9    10          21
           1         10    11          23
------------  ---------  ----  ----------
//  Note: Run DP-server flow using 1 partitioner and upto 10 analyzers ...
//        lsf will be used for job scheduling.
```

set_monitor_options

Scope: Server mode

Specifies the number of shifts for patterns for each monitor before diagnosis begins.

Usage

set_monitor_options ***monitor*** **-number_shifts** { ***default*** | ***integer*** }
[-use_full_fail_log_path_in_server_diagnosis_report { **ON** | **OFf** }]

Arguments

- ***monitor***

  Required string that specifies a monitor, which is a named reference to a monitored directory of failure files. See the add_monitor command for more information.

- **-number_shifts** { ***default*** | ***integer*** }

  Required switch and string or integer pair that specifies the number of shifts.

  By default, the tool uses the number of shifts stored in the flat model. If you specify a valid number of shifts with ***integer***, then the tool uses that number of shifts.

- -use_full_fail_log_path_in_server_diagnosis_report { **ON** | **OFf** }

  Optional switch and literal pair that specifies whether to print the full path of the fail log in the diagnosis report. The default is off, which prints only the fail log filename.

  ```
  #symptoms=1 suspects=3 … fail_log=wafer1.flog
  ```

  Use "on" to print the fail log filename and its full path.

  ```
  #symptoms=1 suspects=3 … fail_log=/full/path/to/wafer1.flog
  ```

Description

Note
You can only use this command to modify the number of shifts for an EDT design. There is currently no support for this for non-EDT designs.

By default, the tool uses the number of shifts stored in the flat model. Alternatively, if you specify a valid number of shifts, then the tool applies this number to each analyzer associated with the monitor.

Examples

The following example specifies 100 shifts for the flogs monitor.

```
add_monitor flogs ../flogs/ -results ../results/flogs.ya
set_monitor_options flogs -number_shifts 100 // specify the number of shifts
add_design flogs ../src/design.flat.gz
```

```
add_pattern flogs ../pat/pat.gz
add_analyzer localhost:1
start_diagnosis
```

start_diagnosis

Scope: Server mode

Prerequisites: Monitored directory and associated test pattern file and design netlist must be set up.

Initiates the diagnosis of failure files in the specified monitored directory.

Usage

start_diagnosis [*monitor_id*...] [-count *count_number*]

Arguments

- *monitor_id*

  Optional, repeatable string that specifies a monitored directory. If you do not use this string, the tool automatically assigns analyzers specified with the add_analyzer command to all monitors in a round-robin manner.

  The *monitor _id* is the unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

- -count *count_number*

  Optional switch and value pair that specifies the number of diagnoses to perform. Once the specified number is reached, automatic diagnosis is suspended.

Description

Create CSV diagnosis output format by setting the diagnostic_CSV variable true (the default is false), or use the add_reporting_format command to change diagnosis reporting formats.

Examples

The following example initiates the diagnosis of failure files in the *monitor1* directory.

```
start_diagnosis monitor1
```

Related Topics

add_reporting_format

diagnostic_CSV

add_pattern

suspend_diagnosis

Scope: Server mode

Suspends the diagnosis of the failure files in the specified directory.

Usage

suspend_diagnosis ***monitor_id*** ...

Arguments

- ***monitor_id***

  Required, repeatable string that specifies a monitored directory.

  The *monitor _id* is the unique name for the monitored directory, specified with the add_monitor command. The *monitor_id* can also be the number representing the order that a monitored directory was defined. For example: "1" would identify the first monitored directory defined. Use the report_monitor command to determine the order number for a monitored directory.

Examples

The following example suspends the diagnosis of failure files in the *monitor1* and *monitor2* directories.

```
suspend_diagnosis monitor1 monitor2
```

Related Topics

resume_diagnosis

version

Scope: Server mode

Displays the version of the Tessent Diagnosis server software that you are currently running.

Usage

version

Arguments

None.

Examples

The following example displays the Tessent Diagnosis server version.

```
version
```

watch

Scope: Server mode

Initiates dynamic reporting to monitor screen.

Usage

watch

Arguments

None.

Description

The watch command displays the progress of the automatic diagnosis on screen. When dynamic reporting is enabled, you cannot enter any commands.

Press Enter to resume normal command input.

Examples

The following example displays the progress of the automatic diagnosis.

```
watch
```

Chapter 7 Reversible Scan Chain Diagnosis

Reversible scan is a 2-dimensional scan architecture in which the scan chain performs both forward and backward scan shift to diagnose scan faults. In simple terms, each scan cell can receive shift values from its neighbor cell either on its left or on its right.

The scan chain is stitched together in both directions, enabling shifting scan data from left to right (denoted as "L2R") or right to left (denoted as "R2L"), by a control signal DIR that changes scan shift direction. A simple scan chain example with 3 scan cells is illustrated in Figure 7-1 and Figure 7-2.

The scan path in red color in Figure 7-1 indicates L2R shift from Cell 2 to Cell 0, whereas the scan path in green color in Figure 7-2 indicates R2L shift from Cell 0 to Cell 2.

Figure 7-1. Reversible Scan Chain L2R Shift

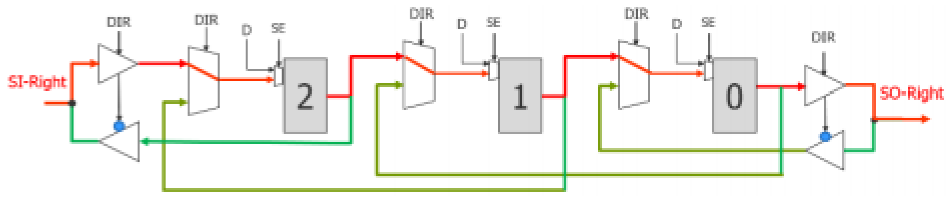
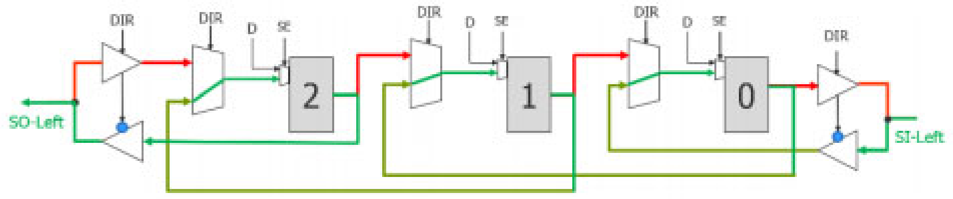


Figure 7-2. Reversible Scan Chain R2L Shift



Benefits of Reversible Scan

Using one dimensional reversible scan architecture, you can design specific scan chain patterns to achieve perfect diagnostic resolution. Test chip evaluations show 4X increase in diagnostic resolution and 6X decrease in physical area of a single chain suspect.

As diagnosis only uses chain patterns, it also improves test time and diagnosis times.

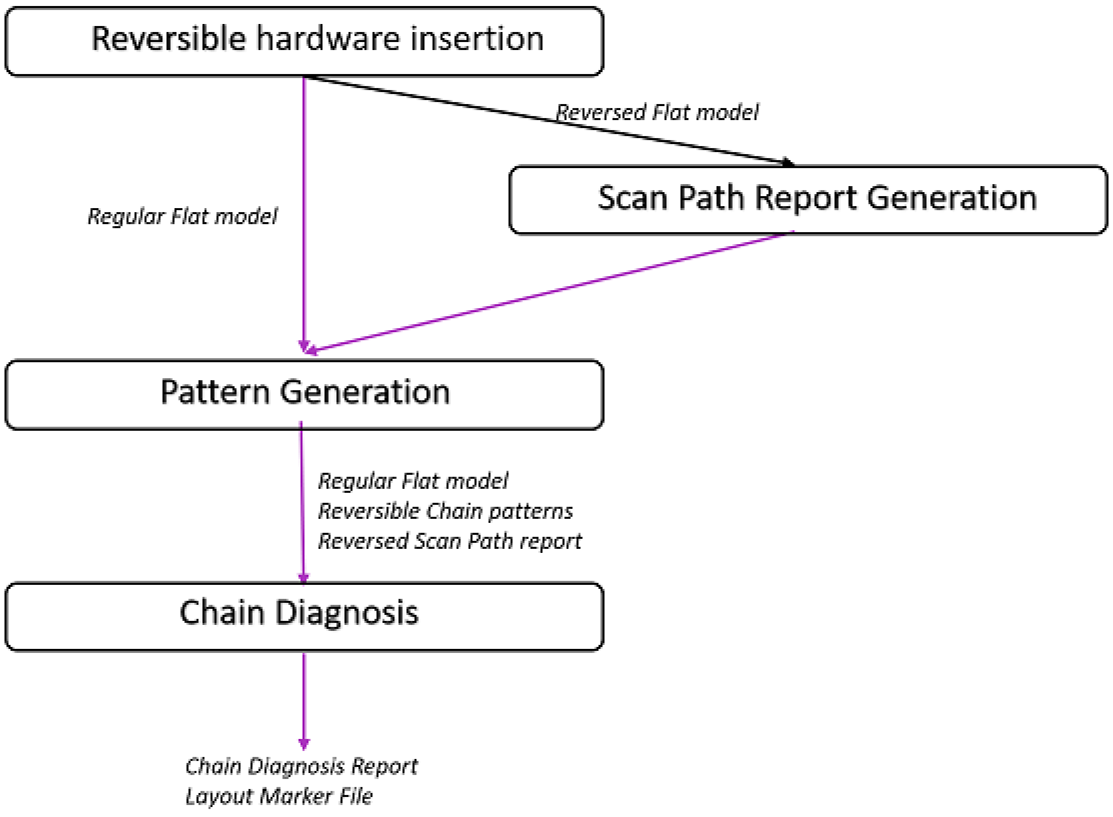
Design Considerations

The overall hardware utilization cost for reversible scan is about 5%. A big part of the overhead is the costly re-stitching of connections outside of MBFFs (multi-bit flip-flops). You can mitigate this by either making MBFFs non-reversible or only partially reversible.

Reversible Scan Chain Diagnosis Flow

The flow chart below shows the steps of the reversible scan chain diagnosis flow.

Figure 7-3. Reversible Scan Chain Diagnosis Flow Chart



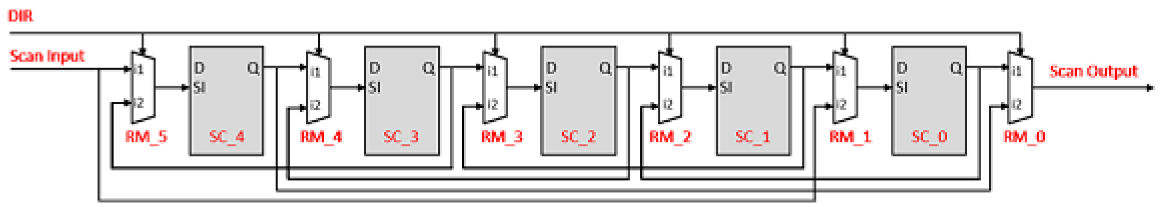
Reversible Scan Chain Insertion

Tessent Shell does not support insertion of reversible scan chains. You can do the insertion during the default scan insertion procedure or by restitching existing scan chains using the custom flow in the “dft insertion mode”.

For more information, refer to set_context in the *Tessent Shell Reference Manual* and Scan Insertion Flows in the *Tessent Scan and ATPG User’s Manual*..

Reversible scan chains are stitched together in both directions to enable shifting of scan data from left to right (denoted as “L2R”) or right to left (denoted as “R2L”). A simple scan chain example with 5 scan cells (SC_0-SC_4) is illustrated in Figure 7-4. Compared to the conventional scan chain architecture (link to scan insertion flow), the reverse scan insertion procedure adds two-input multiplexers to control shift direction for subsequent scan cells. Inputs i1 and i2 of each inserted multiplexer are connected to the outputs of previous scan cell and shifted by two positions towards the scan output, respectively. The output of the inserted multiplexer is connected to scan input of subsequent scan cell.

Figure 7-4. Scan Chain Example



The insertion procedure has the following exceptions to handle implementations with single direction scan input or scan output ports:

- Input i1 of the last reverse multiplexer (RM_5) is connected to the scan chain input.
- Output of the first reverse multiplexer (RM_0) is connected to scan chain output.
- Input i2 of the first reverse multiplexer (RM_0) is connected to output of last scan cell (SC_4).
- Input i2 of the next to last reverse multiplexer (RM_1) is connected to scan chain input.

Limitations

Scan chains with multiple clock domains or lockup latches are not supported in the insertion procedure. Avoid using them in the reversible scan diagnosis flow.

Reversible Scan Suspect Types

Reversed scan chains enable you to unload shifted values in the opposite direction from which they were loaded. The tool can observe good values being unloaded up until the faulty values begin. In most cases, the tool can pinpoint the faulty cell. When high intermittency occurs, the tool narrows the suspect range to a subset of cells.

Tessent Diagnosis analyzes the flush patterns and produces more topologically detailed suspect types. When a fault affects only one shift path, two new suspect type categories appear in the diagnosis report to indicate these types of failures:

- **REV_OUT** — Indicates that a failure occurred only in the standard shift direction.
- **REV_IN** — Indicates that a failure occurred only in the reversed shift direction.

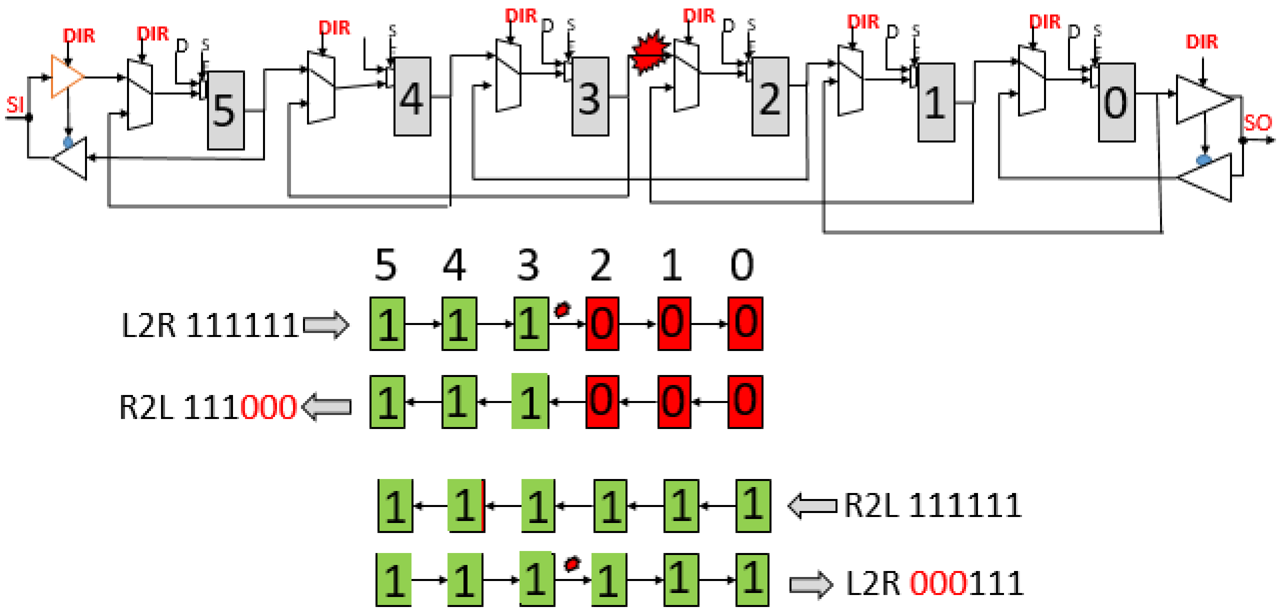
These categories can apply to the fault types STUCK_AT, SLOW, FAST, SLOW_TO_RISE/FALL, and FAST_TO_RISE/FALL.

Failures can also occur in both directions. In the diagnosis report, they appear as IN+CELL+OUT suspects.

Standard Shift Defects

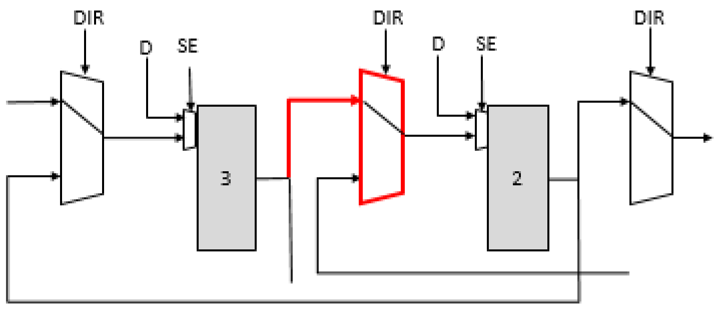
For standard shift defects, the standard shift pattern fails, and the tool passes the reversed shift pattern. The following figure shows a standard shift defect. In the standard shift direction, the tool observes faulty values after three good cycles. This correlates with the defective cell position in the chain at cell 3.

Figure 7-5. Standard Shift Defect



In this case, the tool limits the suspect topology to a single-direction MUX and its A0 input net. It omits the cell_number 3 polygons, which is beneficial for physical failure analysis (PFA) efforts.

Figure 7-6. Standard Shift Defect Suspect Topology



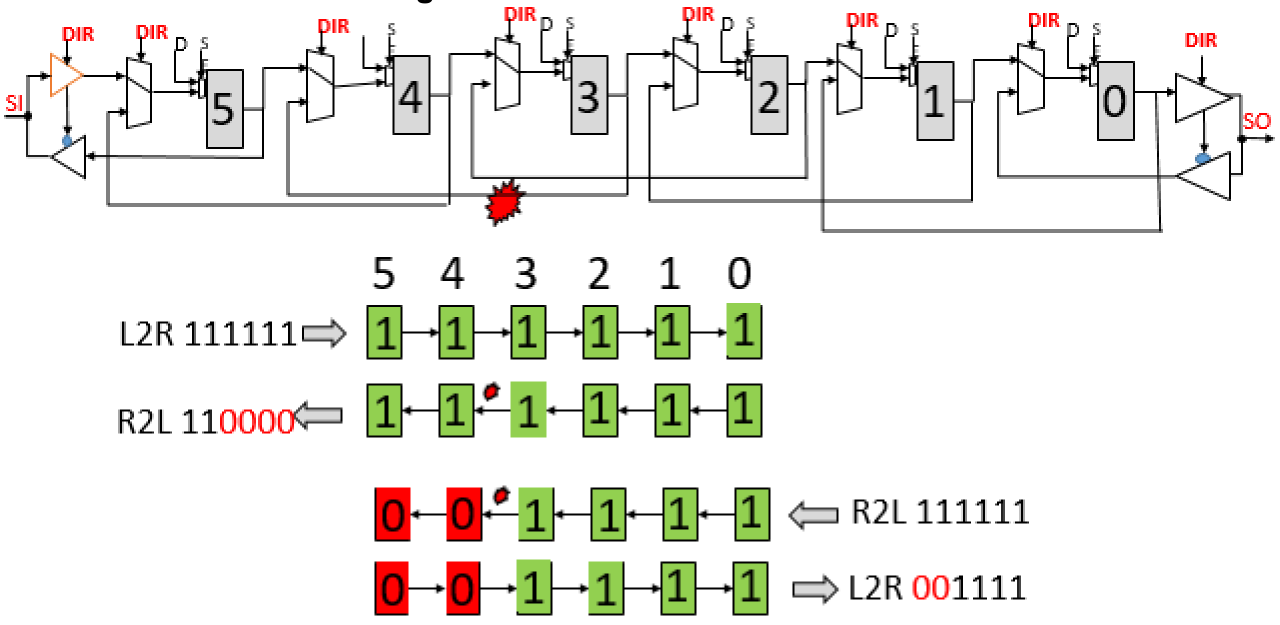
The diagnosis report displays the REV_OUT fault type category for this type of failure. For example:

```
diagnosis_mode=chain
#symptoms=1 #suspects=1 CPU_time=2.87sec fail_log=fal

symptoms=1 #suspects=1 faulty_chain=100 fault_type=STUCK
suspect score type          value pin_        cell_   net_      cell_  chain_  memory_  shift_
                                  pathname    name    pathname number name     type     clock
-----------------------------------------------------------------------------------------
1       100   STUCK(REV_OUT) 1 /reg/Q     SDFF_X1  /bigreg17[21] 3  chain4    MASTER  /clk1
-----------------------------------------------------------------------------------------
```

Reversed Shift Defects

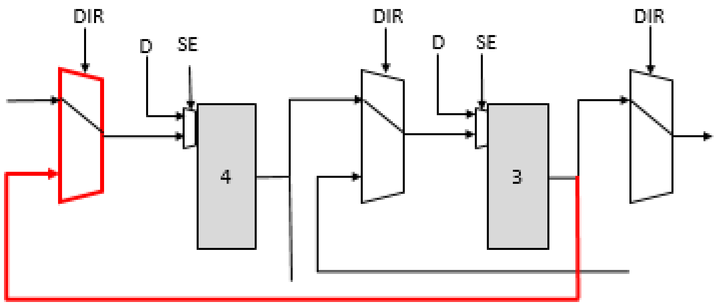
For reversed shift defects, the reversed shift pattern fails, and the tool passes the standard shift pattern. The following figure shows a reversed shift defect. In the reversed shift direction, the tool observes faulty values after four good cycles. This correlates with the defective cell position in the chain at cell 4.

Figure 7-7. Reversed Shift Defect



The suspect topology is limited to the input MUX and its A1 input net.

Figure 7-8. Reversed Shift Defect Suspect Topology



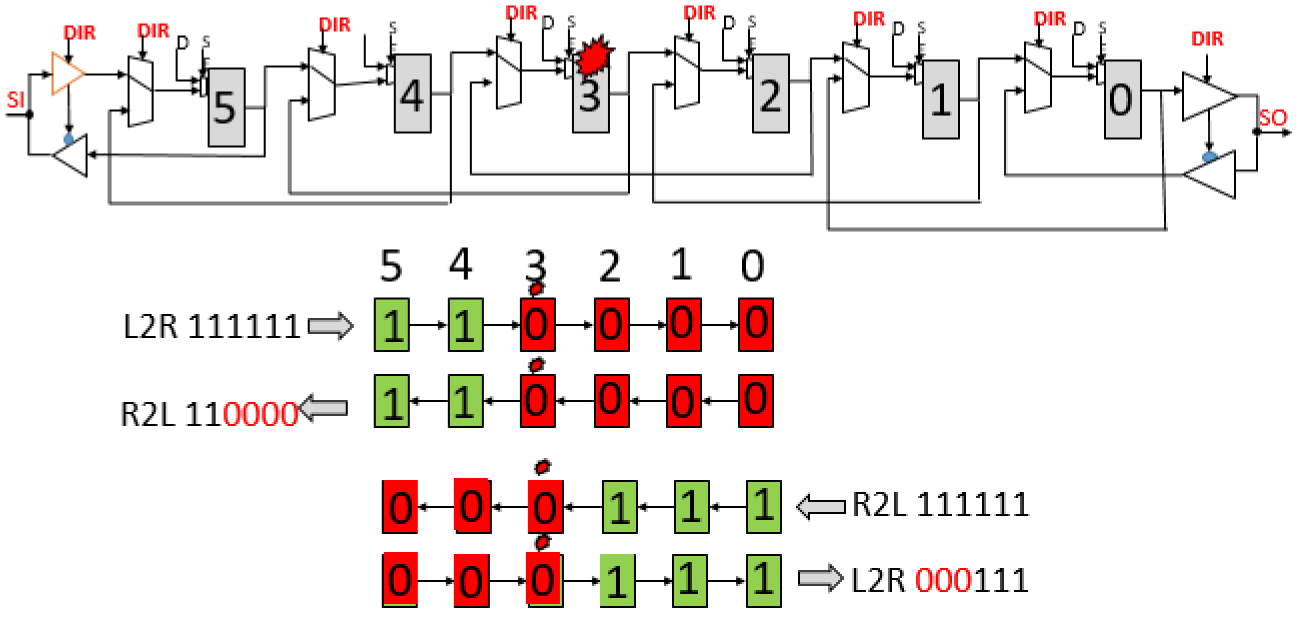
The diagnosis report displays the REV_IN fault type category for this type of failure. For example:

```
diagnosis_mode=chain
#symptoms=1 #suspects=1 CPU_time=2.87sec fail_log=fal

symptoms=1 #suspects=1 faulty_chain=100 fault_type=STUCK
suspect score type         value pin_       cell_    net_      cell_  chain_   memory_  shift_
                                 pathname   name     pathname number name      type     clock
------------------------------------------------------------------------------------------
1       100    STUCK(REV_IN) 1 /reg/SI     SDFF_X1 /r_mux_146_Z 4  chain4    MASTER  /clk1
------------------------------------------------------------------------------------------
```

Defects in Both Shift Directions

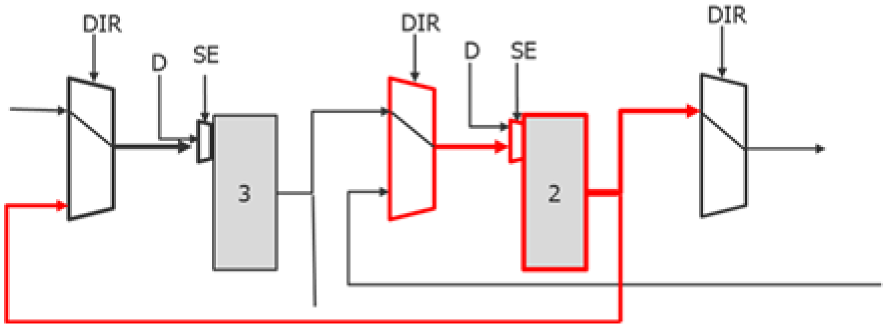
When a fault affects both shift directions, the failures from both ends converge onto a single cell and both patterns fail. In the following example, the standard and reversed patterns both fail at cell 3.

Figure 7-9. Defect in Both Shift Directions



When both chain diagnosis patterns for a reversed scan chain fail, the diagnosis suspect topology is limited to the scan cell itself. Additionally, the tool includes the cell coordinates of the input side scan selection MUX as well as the entire network of output scan port.

Figure 7-10. Both Shift Directions Suspect Topology



The following diagnosis report shows how the results appear:

```
diagnosis_mode=chain
#symptoms=1 #suspects=1 CPU_time=2.87sec fail_log=fal

symptoms=1 #suspects=1 faulty_chain=100 fault_type=STUCK
suspect score type          value pin_       cell_   net_      cell_  chain_  memory_  shift_
                                  pathname   name    pathname number name     type     clock
-----------------------------------------------------------------------------------------
1       100   STUCK(IN+CELL+OUT) 1 /reg/Q    SDFF_X1 /bigreg17[21] 2  chain4   MASTER   /clk1
-----------------------------------------------------------------------------------------
```

Note
Multiple suspects that originate from a single MBFF instance that does not have reversed shift capability are reported as a single symptom.

Defects on Chain Input or Output

When a PAD defect affects the chain SI or SO signal, the tool observes faulty values in both shift directions that do not converge to any cell. The chain's first and last cells are both suspects because the defect can be on the scan-in or scan-out.

The diagnosis report displays one symptom using the IN+CELL and the CELL+OUT scenarios. See "Symptom Descriptions" on page 107 for more information about scenarios.

The following example is for a chain with 49 cells, cell 48 being closest to the scan-in:

```
diagnosis_mode=chain
#symptoms=1 #suspects=2 CPU_time=0.23sec fail_log=chain.flog

symptom=1 #suspects=2 faulty_chain=chain4 fault_type=STUCK
suspect score type          value pin_      cell_ net_     cell_  chain_ memory_ shift_
                                  pathname name  pathname number name   type    clock
------------------------------------------------------------------------------------
1       100   STUCK(CELL+OUT) 0 /reg/Q    sff   /reg_Q   0      chain4 MASTER  /clk
2       100   STUCK(IN+CELL)  0 /reg/SI   sff   /reg_SI  48     chain4 MASTER  /clk
------------------------------------------------------------------------------------
```

Diagnosing Reversed Scan Chain Patterns

For diagnosis, there is no limit to the number of concurrently failing reversed scan chains.

Restrictions and Limitations

- You cannot read in and then write out reversed patterns with the write_patterns command. The tool assumes the standard cell loading direction, so the written file is non-reversed.
- The tool does not support designs with IJTAG interfaces.
- The tool does not support dynamic partitioning.
- The tool does not support failure generation via the "write_failures" command. You must perform serial pattern simulation.
- The tool provides a warning when a truncated failure log indicates that some chains were untested. It is similar to the following:

  ```
  //  Warning: Failure truncation caused some chains on channel
  <channel_name> to be left untested. Last tested pattern is <#>.
  ```

  For more information, see "Failure Truncation Handling" on page 65.

Prerequisites

- You have generated chain patterns for reversible scan chains from a design that includes shift direction global signals. Refer to create_reversible_patterns for information on generating reversible test patterns.
- You have run the "report_scan_path -output_file" command against the reversed flat model. The resulting *scan path report (.spr)* file provides visibility into the reversed scan chain configuration.

Procedure

To enable reversed scan chain diagnosis, prior to specifying the diagnose_failures command, run the set_diagnosis_options command with the following two new options:

- set_diagnosis_options -shift_direction_port *name*

  Use the -shift_direction_port option to specify the name of the shift direction port. The tool verifies the pattern values on the port so that it distinguishes single-direction chain patterns (to be used for direction and fault type detection) from reversed chain patterns (to be used for down-to-cell diagnosis).

  Specifying this option triggers reversed chain diagnosis.

- set_diagnosis_options -reverse_scan_configuration *spr_file*

  Use the -reverse_scan_configuration option to specify the scan path report file that contains the reversed configuration. The tool only enables you to load one flat model, so this report provides the necessary additional information about the reversed scan path.

  The tool reports an error if you specify the shift direction port name but not the scan path report file.

  The tool loads the file once per diagnosis session. If the tool cannot validate the file, it reports an error and tries again on the next diagnose_failures command. Errors could occur in the following cases:

  - The tool generated the scan path report with unsupported metadata.
  - The length of the scan chains differs from the standard configuration.
  - The file is invalid because it was written out with non-default options.

Generating Reversed Scan Path Description

You must provide the scan configuration of the reversed shift direction for the pattern generation process and diagnosis algorithms. The flat model only contains L2R shift connections, so you need to provide an additional side file to the tool, which is the Scan Path Report, generated using the flat model containing R2L shift connections.

Procedure

Run the following commands to generate the Scan Path report:

```
read_verilog <design_with_reversible_hw>.v
read_cell_library < >
dofile <edt_dofile>
add_pin_constraint <shift_direction_port> C1
set_system_mode analysis
report_scan_path -output_file design.rev.spr
```

Results

The "report_scan_path" command generates the compressed scan description file with the ".gz" suffix. The tool automatically stores design information in the scan description file in the form of metadata to provide consistency checking during loading.

You must provide the "design.rev.spr.gz" file during the ATPG step and diagnosis steps.

Frequently Asked Questions

This section contains frequently asked questions and possible solutions about reversible chain diagnosis.

1. **I have a design with dedicated channels for OCC chains. The OCC chains are directly connected from or to the SI or SO port, while other chains are connected from or to the decompressor or compressor. Can I apply RVS to this structure**?

   If you insert reversible logic outside of Tessent for OCC chains, contact Siemens EDA for support. However, it is recommended that you get the flat model back as soon as possible for testing. At the very least, provide a skeleton design with such re-stitched OCCs and a few compressed reversible chains. The Tessent reversible scan insertion flow does not currently support modifying OCC chains because of tool hardware manipulation restrictions for these elements. In general, this only comprises a small number of registers compared to hundreds of thousands in regular chains. If this is important for you, please contact Siemens EDA.

2. **Does RVS support inserted pipeline registers between SI port and decompressor, and between compressor and SO port?**

   Yes, you can apply RVS to this structure.

3. **We have masked only the OCC chains, but we also have non-reversible elements such as embedded scan chain memories. Should we also mask the chains, including embedded scan chain memories?**

Do this only if a single memory spans the entire chain length. If there are multiple memory segments, RVS will work, but the resolution will only apply to the segment. Individual bits are not distinguishable.

4. **Should we mask all scan chains that contain non-reversible elements?**

   No. Refer to #3 above.

Appendix A dlogutil Utility

The dlogutil utility generates failure files from standard test data format (STDF) files.

The Tessent Diagnosis dlogutil utility extracts scan failures from STDF-V4 2007(.1)-formatted files and generates failure files that you can use for diagnosis with Tessent Diagnosis. The following sections describe the features of the dlogutil utility.

- Command line syntax and switches.
- Utility commands and variables.
- Features for SSN on-chip compare.

dlogutil Invocation

Invokes the dlogutil utility, a Tessent Diagnosis utility.

Usage

DLOGUTIL
  [-DOFile *dofile_name*]
  [-LOGfile *logfile_name* [-REPlace]]

Arguments

- -DOFile *dofile_name*

  An optional switch and string pair that specifies the name of a dofile to run upon invocation.

- -LOGfile *logfile_name*

  An optional switch and string pair that specifies the name of the file to which you want dlogutil utility to write all session information. The default is to display session information to the standard output. The version banner that indicates the tool, version, date, and platform on which the log file was produced is included at the beginning of the file.

- -REPlace

  An optional switch that overwrites the -Logfile *logfile_name* if a log file of the same name already exists.

Description

You use the dlogutil utility to extract scan failures from STDF-V4 2007(.1)- formatted files and create failure files for use with Tessent Diagnosis. Subsequently, you use the dlogutil output for diagnosis with Tessent Diagnosis.

Examples

```
DLOGUTIL -dofile my_test.std
```

dlogutil Utility Commands

Use the dlogutil utility commands to extract scan failures from STDF-V4 2007(.1)-formatted files and output failure files that you can use for diagnosis with Tessent Diagnosis.

extract_stdf_failures

Extracts failures from an STDF file to generate failure files.

Usage

extract_stdf_failures ***stdf_filename faillog_base*** [-REPlace] [-ignorePRR]

Arguments

- ***stdf_filename***

  A required string that specifies the name of the STDF file.

  Tip
  You can use gzipped files without decompressing them. Ensure the file name ends with the ".gz" extension for the tool to automatically use gzip processing.

- ***faillog_base***

  A required string that specifies the base name to use for each failure file. For every failing part, the utility names the failure file faillog_base_*i*, where *i* is the unique ID for this part.

- -REPlace

  An optional switch that specifies to replace an existing failure file.

- -ignorePRR

  An optional switch to ignore the part results record (PRR) and generate failure files for all parts.

Description

When you enter this command, the *dlogutil* utility parses the given STDF file using the default schema and loads the STDF file into memory. During parsing, the tool ignores invalid records.

The stdf file can include either a single part or multiple parts.

The *dlogutil* utility automatically extracts valid scan failures for each part from the STR records based on the STDF V4-2007(.1) standard. The tool writes extracted fails for each part to a failure file that it names based on your *faillog_base* string.

Examples

Example 1

The following example extracts fails from the *my_test.std* file in the *stdf* directory and generates a failure file in the *results* directory for every failing part using the *my_part* base name.

```
extract_stdf_failures stdf/my_test.std results/my_part
```

Example 2

The following example generates failure files for every test in the *compressed.stdf.gz* file. It uses the *failure_file* base name when generating a failure file, and it replaces an existing failure file if it has the same name.

```
extract_stdf_failures compressed.stdf.gz failure_file -replace -ignorePRR
```

load_fail_map

Loads the Logic BIST chain mapping file from the LVDB.

Usage

load_fail_map ***<LBIST_controller_ID>.lbist_chain_info_file***

Arguments

- ***<LBIST_controller_ID>.lbist_chain_info_file***

  A required string that specifies the *.lbist_chain_infofile* to load.

Description

The load_fail_map command is used when converting a top-level failure file to a core-level failure file that Tessent Diagnosis can read. Normally, the logic BIST fail mapping file is stored at the following location:

```
./LV_WORKDIR/<LBIST controller id>.lbist_chain_info
```

See "Converting the Top-Level Failure File to a Core-Level Failure File" for complete information.

Note

You must load the chain mapping file before you use the map_fail_log command.

Examples

load_fail_map BP4.WBP0.lbist_chain_info

load_stdf_file

Loads the STDF file using the default schema based on the STDF V4-2007(.1) standard.

Usage

load_stdf_file ***stdf_filename***

Arguments

- ***stdf_filename***

  A required string that specifies the name of the STDF file.

  Tip

  You can use gzipped files without decompressing them. Ensure the file name ends with the ".gz" extension for the tool to automatically use gzip processing.

Description

The tool parses the given STDF file using the specified schema and loads the file into memory.

Examples

Example 1

The following example loads the *my_test.std* file found in the *stdf* directory.

```
load_stdf_file stdf/my_test.std
```

Example 2

The following example loads the *compressed.stdf.gz* file from the current working directory.

```
load_stdf_file compressed.stdf.gz
```

map_fail_log

Converts a top-level failure file to a core-level failure file for use by Tessent Diagnosis diagnosis.

Usage

map_fail_log ***top_level_failure_file*** [-out ***core_level_failure_file***]
[-replace]

Arguments

- ***top_level_failure_file***

  A required string that specifies the name of the top-level failure file to convert to a core-level failure file.

- -out ***core_level_failure_file***

  An optional switch and string pair that specifies the name of the converted core-level failure file converted by the tool.

- -replace

  An optional switch that overwrites the output core-level failure file if it already exists.

Description

See "Converting the Top-Level Failure File to a Core-Level Failure File" for complete information.

Examples

The following example loads the Tessent SiliconInsight top-level failure file using the map_fail_log command:

```
dlogutil> map_fail_log top_level_failure_file.flog -out core.flog -replace
```

report_stdf_conditions

Reports the testing condition under which the loaded STDF file was generated.

Usage

report_stdf_conditions

Arguments

None.

Description

Reports the testing condition under which the loaded STDF file was generated, including the temperature, shift or capture frequency, voltage, and so on. By default, the testing condition for the first part is reported.

report_stdf_parts

Summarizes the STDF parts tested in table format.

Usage

report_stdf_parts

Arguments

None.

Description

Summarizes the following STDF parts tested in table format:

- Part number
- Range of records for this part
- Tester head/site number
- Number of tests
- Testing time
- Hard/soft bin
- Part ID
- Testing results

report_stdf_pattern_sequences

Reports the pattern sets that are applied for testing.

Usage

report_stdf_pattern_sequences

Arguments

None.

Description

Reports the pattern sets that are applied for testing, including the pattern set name, pattern count, and cycle offset.

write_atdf_file

Writes a pre-loaded STDF file into an ASCII ATDF file.

Usage

write_atdf_file ***atdf_filename*** [-replace]

Arguments

- ***atdf_filename***

  A required string that specifies the name of the ATDF file.

- -replace

  An optional argument that enables the tool to overwrite the existing file.

Description

The tool traverses all records of the pre-loaded STDF file and writes the content of each record into the ATDF file you specify with ***atdf_filename***.

A short summary is provided first. For each field of a given record, the field name and the field content are separated by a colon.

Examples

```
write_atdf_file my_test.atd -replace
```

dlogutil Utility Variables

In addition to commands, dlogutil supports many variables.

To set a variable, from the dlogutil utility command prompt, enter:

set_variable ***variable_name value***

where:

- *variable_name* — The name of the variable.
- *value* — The new value to set it to.

To display a variable, from the dlogutil utility command prompt, enter:

report_variable

A report of the current settings for all the variables displays.

The variables are:

stdf_cap_data_mapping

Changes the dlogutil utility default handling for captured values that are not known to Tessent Diagnosis by supporting user-defined mappings for the unknown values.

Usage

set_variable stdf_cap_data_mapping ***value1 new_value1 value2 new_value2 ...***

Data Type

String.

Default Value

None.

Arguments

- ***value1 new_value1 value2 new_value2 ...***

  A required Tcl list that specifies the mapping in pairs such that the first value is the unknown captured value and the second value is a supported mapped value for the unknown value.

Description

By default, Tessent Diagnosis recognizes H, L, and Z values for the expected values and actual values for failing bits. If it encounters any other captured values, it issues an error and does not generate failure files for failing bits with the unknown values.

To generate valid failure files for parts with unknown captured values, you can map unknown captured values to the following strings, where the "Mapped Value" column lists the options for ***new_value1***, ***new_value2***, and so on, in the usage syntax:

Table A-1. Supported Mapping Values for Unknown Captured Values

Mapped Value	Description
H	Maps to H.
L	Maps to L.
Z	Maps to Z.
PASS	Treats this failing bit as passing and excludes this bit when generating the failure file.
FAIL	Treats this failing bit as failing. The tool attempts to derived its value from the corresponding expected value. If the expected value is H or L, the captured value is mapped to L or H, respectively. Otherwise, the tool issues an error.

Define each unknown captured value once. Multiple mappings for each unknown captured value cause errors.

Examples

Suppose your STDF file, my_test.stdf, includes failures with expected values that are unknown to Tessent Diagnosis: G (glitch) and M (mid-band voltage).

```
extract_stdf_failures ./my_test.stdf my_flog_try1

...

set_variable stdf_cap_data_mapping “G PASS M Z”

extract_stdf_failures ./my_test.stdf my_flog_try2

...

set_variable stdf_cap_data_mapping "G FAIL M PASS"

extract_stdf_failures ./my_test.stdf my_flog_try3
```

The first extraction run only generates failure files for failing bits whose expected values are valid (H, L, or Z). The tool reports the unknown values (G or M) but does not generate a failure file for the failing parts with any unknown values.

In the second run, dlogutil maps the unknown captured value M to Z, and drops all failing bits with captured values of G. Now failure files can be extracted for all failing parts.

In the third run, all failing bits with captured values of G are automatically turned into failing bits assuming that their corresponding expected values are H or L. And all failing bits with captured values of M are dropped. Similarly failure files are extracted for all failing parts.

stdf_fail_trunc_handling

Changes the dlogutil utility default handling for truncated failures from STDF-V4 2007(.1) failure files based on the failure file tracking keyword.

Usage

set_variable
    stdf_fail_trunc_handling {[all] | [last_cycle_logged] | [unknown]}

Data Type

String.

Default Value

None.

Arguments

None.

stdf_selected_parts

Specifies all failing parts that should be considered under subsequent failure extraction.

Usage

set_variable stdf_selected_parts ***selected_part_id_list***

Data Type

String.

Default Value

None.

Arguments

- ***selected_part_id_list***

  A required Tcl list that specifies the part IDs for all failing parts that the tool targets for subsequent failure extraction.

Description

You specify ***selected_part_id_list*** using a Tcl list.

In the Tcl list, you must specify each part using the part's unique part ID that is numbered starting from 0. If you have an STDF loaded, then use the report_stdf_parts to obtain the parts.

If you submit an empty list or do not specify one, then the tool extracts failures for all failing parts.

Examples

```
set_variable stdf_selected_parts { 0 1 3 5 }
```

stdf_selected_psr_ids

Specifies all pattern sequences that the tool should consider for subsequent failure extraction.

Usage

set_variable stdf_selected_psr_ids ***selected_test_psr_index_list***

Data Type

String.

Default Value

None.

Arguments

- ***selected_test_psr_index_list***

  A required Tcl list that specifies the PSR index for all pattern sequences that are targeted for subsequent failure extraction.

Description

You specify ***selected_test_psr_index_list*** using a Tcl list.

In the Tcl list, you must specify each pattern sequence using the pattern sequences's unique PSR_INDX defined in its PSR record. If you have an STDF file loaded, then use the report_stdf_pattern_sequences to obtain the PSR_INDX.

If you submit an empty list or do not specify one, then the tool extracts failures for all pattern sequences. Only the failures associated with the targeted pattern sequences are extracted and written into the failure files.

stdf_test_name_source

Appends the pattern file name to the test suite name for STDF-V4 2007(.1) failures files produced by Verigy testers.

Usage

set_variable stdf_test_name_source **PSR_NAM**

Data Type

String.

Default Value

None.

Arguments

- **PSR_NAM**

  A required string that specifies where the test name is extracted.

dlogutil Features for SSN On-Chip Compare

The dlogutil utility has several features that support SSN on-chip compare.

Using a Test Pattern in Multiple Tests

Using a test pattern in multiple tests for the same part is a valid use case. SSN's on-chip compare feature requires this use case.

You can specify all pattern sequences that the tool considers for failure extraction with the stdf_selected_psr_ids variable. The dlogutil utility reads the pattern sequence records (PSR) and extracts all related fails. It does not report an error if it finds a PSR more than once.

The dlogutil utility can determine the individual runs of the same pattern based on the test number field of the scan test record (STR). The test number field name is TEST_NUM. The utility generates separate failure files for each run and appends "*.test_#*" to the base name (where # is the test number).

The STR has a limit to the number of fails it can hold. If the tester has more fails to record, it writes an additional STR immediately after the full STR. This STR and any subsequent STRs from the same test have the continuation flag bit set to indicate that the fails it holds are from the same test as its predecessor.

Additionally, the dlogutil utility passes test condition fields to each failure file's tracking_info section.

Core Instance Information

If your ATE can use datalog test records (DTR), you can log data to STDF for dlogutil to extract into the failure files. You can log a list of failing cores to STDF for the top-level failure file for SSN on-chip compare. Use the core's SSH instance name from the ICL design.

Note

See the Primary Phase and Retest Phase sections of "Testing and Failure Logging Process" on page 560 for more information.

For example, if three cores are contributing failures to the SSN bus and their SSH instance names are corea_instance1, corea_instance2, and corea_instance3, use the ATE to log three DTRs containing the following (one DTR for each line) immediately after the test's STRs:

```
TESSENT ssn_on_chip_compare_enabled_failing_instances begin
corea_instance1 corea_instance2 corea_instance3
TESSENT ssn_on_chip_compare_enabled_failing_instances end
```

The dlogutil utility reads the DTRs only if they immediately follow the test's STRs. For this example, the dlogutil utility writes the SSN core instance information into the chip-level failure file that it generates for the STRs:

```
ssn_on_chip_compare_enabled_failing_instances_begin
   corea_instance1
   corea_instance2
   corea_instance3
ssn_on_chip_compare_enabled_failing_instances_end
```

Retesting to Collect Failures for Specific Cores

If your ATE can use the COND_LST field of STRs, you can log data to STDF for dlogutil to extract into the failure files. The first test phase for SSN on-chip compare is the primary phase. The retest phase collects failures for specific cores. You can log the test phase to STDF for the chip-level failure files.

Note

See the Primary Phase and Retest Phase sections of "Testing and Failure Logging Process" on page 560 for more information.

For example, if a core contributes failures to the SSN bus, use the ATE to log the following line into the COND_LST field of the test's STRs for the primary phase:

```
ssn_on_chip_compare_test_phase=primary
```

Use this line for the retest phase:

```
ssn_on_chip_compare_test_phase=retest
```

The dlogutil utility reads this information from the STRs and writes the test phases into the chip-level failure file before it writes the on-chip compare failing instance names. The utility generates the following for the primary phase:

```
ssn_on_chip_compare_test_phase primary
ssn_on_chip_compare_enabled_failing_instances_begin
   corea_instance1
   corea_instance2
   corea_instance3
ssn_on_chip_compare_enabled_failing_instances_end
```

The utility generates the following for the retest phase:

```
ssn_on_chip_compare_test_phase retest
ssn_on_chip_compare_enabled_failing_instances_begin
   corea_instance1
   corea_instance2
   corea_instance3
ssn_on_chip_compare_enabled_failing_instances_end
```

Appendix B
Layout-Aware Diagnosis Layout Verification Rules

When you perform layout-aware diagnosis, Tessent Diagnosis uses layout verification rules to perform up-front validation of the design and layer information from the LEF/DEF input.

Refer to "Layout Verification Reporting" for more information about layout verification reporting.

Refer to "Layout Verification Examples" for a series of examples that walk you through layout verification debugging.

The Layout Verification Rules

Tessent Diagnosis uses various rule categories for validation of the design and layer information before creating the LDB for subsequent use during diagnosis. By default, the handling for all of the layout rules is a Warning.

The layout verification rule categories are:

Chip Boundary Rules

Chip boundary rules pertain to the top-level module chip boundary issues.

Table B-1 lists the chip boundary rules.

Table B-1. Chip Boundary Rules

Rule Name	Description
DesignTopModuleMatch	The names of the top level modules must match between design and layout. This rule is only run if you do not specify any difference in hierarchy between design and layout. See the create_layout command for information on specifying hierarchical differences.
DesignTopModulePinMatch	All IO pins at the boundary of the chip (specifically, the top level module) in the design must exist in the top level module in the layout. This rule is only run if you do not specify any difference in hierarchy between design and layout. See the create_layout command for information on specifying hierarchical differences.

Instance Rules

Instance rules pertains to library cell and other cell instantiations.

Table B-2 lists the instance rules.

Table B-2. Instance Rules

Rule Name	Description
DesignCellMatch	For all the library cells that are instantiated at least once in the design, a corresponding LEF macro with the same name must be found in layout.
DesignCellPinMatch	For every library cell instantiated at least once in the design that passes DesignCellMatch rule, the pins must match between the cell definition in design and the corresponding (specifically, matching name) LEF macro in layout.
DesignInstanceMatch	All instances in the design (library cell instances as well as higher level instances) must exist in the layout. This check is not repeated for the instances belonging to any higher level design instance that does not exist in the layout.
DesignInstanceTypeMatch	For every leaf instance (specifically, instances of library cells) in the design that passes the DesignInstanceMatch rule, the library cell name in the design must match the LEF macro name for the corresponding instances in the layout. This check is not run for the library cell instances belonging to any higher level design instance that does not pass the DesignInstanceMatch rule (that is, missing from the layout).
DesignModuleCell	A higher-level design module (specifically, a non-library cell module) must not have the same name as a LEF macro in the layout. For example, if the ATPG library is defined at a low level compared to the LEF library. The ATPG library might be composed of simple gates such as AND, OR, NOT, and on the layout side, the LEF macros might be defined at a higher level, for example, ANDOR, half adder, and so on. If this rule is violated, then the tool most likely produces DesignInstanceMatch violations because the lower level instances in the design are not found in the layout.

Layer Definition Rules

Layer definition rules pertain to layer specifications for nets, pins, and so on.

Table B-3 lists the layer definition rules.

Table B-3. Layer Definition Rules

Rule Name	Description
LayerExistenceNet	A specified net must use a layer that is specified in the LEF files.

Table B-3. Layer Definition Rules (cont.)

Rule Name	Description
LayerExistencePin	A specified pin must use a layer that is specified in the LEF files.
LayerExistenceMacro	A specified macro must use a layer that is specified in the LEF files.
LayerExistenceNondefaultrule	A specified nondefaultrule must use a layer that is specified in the LEF files.
LayerExistenceVia	A specified via must use a layer that is specified in the LEF files.
LayerExistenceViarule	A specified viarule must use a layer that is specified in the LEF files.
LayerExistenceViarulelayer	A specified via rule layer must use a layer that is specified in the LEF files.
LayerDuplicate	A specified layer is already defined in a previously-specified LEF file. The tool ignores the duplicate layer.

Macro Definition Rules

Macro definition rules pertain to missing and duplicated macros.

Table B-4 lists the macro definition rules.

Table B-4. Macro Definition Rules

Rule Name	Description
MacroExistence	A specified component must use a macro that is defined in the specified LEF files.
MacroDuplicate	A specified macro is already defined in a previously-specified LEF file or files. The tool ignores the duplicate macro definition.
MacroDesign	A specified DEF design is already defined in a previously specified LEF file. The tool ignores any duplicate definition in any LEF files.

Net Rules

Net rules pertain to checking nets and pins.

Table B-5 lists the net rules.

Table B-5. Net Rules

Rule Name	Description
DesignNetMatch	All nets in the design, within the common area, must exist in the layout.
DesignNetPinMatch	For each pin connected to a net (that passes the DesignNetMatch rule) in the design there must be a pin of the same name connected to the corresponding net in the layout.
LayoutNetPinMatch	For each pin connected to a net (that passes the DesignNetMatch rule) in the layout there must be a pin of the same name connected to the corresponding net in the design.
NetExistenceNondefaultrule	A specified NONDEFAULTRULE at a net must be defined in the LEF or DEF files. If this rule is violated, then Tessent Diagnosis uses the standard width/spacing values from the layer definition in the LEF files. This can result in inaccurate polygon coordinates in the Tessent Diagnosis diagnosis report, marker file, and other tool-produced data.
NetExistenceStyle	A specified style at a net must be defined in the DEF files.
Net_use	In DEF file, net use between POWER and GROUND must be consistent. Inconsistency in NET-USE statements associated with DEF-design=*design* results in a violation of this rule.
NetExistenceVia	A specified via at a net must be defined in the LEF/DEF files.
Net_pin_use	In DEF file, net and pin use between POWER and GROUND must be consistent. Inconsistency in NET-USE and PIN-USE statements associated with DEF-design=*design* results in a violation of this rule.
Net_port_use	Net and port use between POWER and GROUND must be consistent between the DEF file and the LEF MACRO port. Inconsistency between NET-USE statements associated with DEF-design=*design* and PORT-USE statements in the LEF file results in a violation of this rule.
PinExistenceMacro	A specified net connected to a macro must use a pin that is defined in the LEF files.

Taper Rules

Taper rules pertains to duplicated and missing taper rules.

Table B-6 lists the taper rules.

Table B-6. Taper Rules

Rule Name	Description
TaperDuplicate	A specified taper rule is already defined in a previously-defined LEF file or files. The tool ignores the duplicate taperrule.
TaperExistence	A specified taper rule in a macro at a pin must be defined in the LEF file or files.

Via Definition Rules

Via definition rules pertain to duplicated and missing vias.

Table B-7 lists the via definition rules.

Table B-7. Via Definition Rules

Rule Name	Description
ViaDuplicate	A specified via is already defined in a previously-specified LEF file or files. The tool ignores the duplicate via definition.
ViaExistenceMacro	A specified via in a LEF macro must be defined in the LEF/DEF files.

LEF/DEF Parser Warning Rules

LEF/DEF parser warning rules check for missing statements in the LEF and DEF files in addition to other parser warnings.

Table B-8 lists the LEF/DEF parser warning rules.

Table B-8. LEF/DEF Parser Warning Rules

Rule Name	Description
LefParseMissingVersion	This rule is violated if the VERSION statement is missing from a LEF file. Without this statement version 5.6 is assumed. If the actual LEF file version is other than 5.6, the handling becomes unreliable. You can fix this by adding an appropriate VERSION statement in the corresponding LEF file.

Table B-8. LEF/DEF Parser Warning Rules (cont.)

Rule Name	Description
LefParseMissingCaseSensitive	This rule is violated if the NAMESCASESENSITIVE statement is missing from a LEF file. In this case the names in the LEF files are assumed to be case sensitive. You can fix this violation by adding an appropriate NAMESCASESENSITIVE statement in the corresponding LEF file.
LefParseMissingBusBitChar	This rule is violated if the BUSBITCHARS statement is missing from a LEF file. In this case the default characters '[ ]' are assumed to be the bus bit characters. You can fix this violation by adding an appropriate BUSBITCHARS statement in the corresponding LEF file.

LefParseMissingMacroPIN
Polygons
	This rule is violated if there is some PIN or PORT definition in a MACRO without any polygons defined for the PIN or PORT. The LEF file excerpt below gives an example of this situation:
	`...`
	`MACRO buf04`
	`  ...`
	`  PIN VCC`
	`        DIRECTION OUTPUT ;`
	`        USE POWER ;`
	`        PORT`
	`           LAYER route_2 ;`
	`        END`
	`    END VCC`
	`...`
	`END`
	The PIN/PORT polygons are used for open defect diagnosis. Missing information about these polygons can result in incomplete open diagnosis for a net connected to the corresponding PIN/PORT. You can fix this by adding the missing polygon information about the PIN/PORT in the corresponding LEF file.

LefParseEndLibrary	This rule is violated if there are some LEF constructs (for example, MACRO definitions) after an 'END LIBRARY' statement in a LEF file. Any such constructs are ignored by the parser and are not used during database creation. You can fix this by ensuring that there is no data in the LEF file after an END LIBRARY statement.

Table B-8. LEF/DEF Parser Warning Rules (cont.)

Rule Name	Description
DefParseMissingVersion	This rule is violated if the VERSION statement is missing from a DEF file. Without this statement version 5.7 is assumed. If the actual DEF file version is other than 5.7, the handling becomes unreliable. You can fix this by adding an appropriate VERSION statement in the corresponding DEF file.
DefParseMissingCaseSensitive	This rule is violated if the NAMESCASESENSITIVE statement is missing from a DEF file. In this case the names in the DEF files are assumed to be case sensitive. You can fix this violation by adding an appropriate NAMESCASESENSITIVE statement in the corresponding DEF file.
DefParseMissingBusBitChar	This rule is violated if the BUSBITCHARS statement is missing from a DEF file. In this case the default characters '[]' are assumed to be the bus bit characters. You can fix this violation by adding an appropriate BUSBITCHARS statement in the corresponding DEF file.
DefParseMissingDividerChar	This rule is violated if the DIVIDERCHAR statementis missing from a DEF file. In this case the default character '/ 'is assumed as the hierarchy divider character. You can fix this violation by adding an appropriate DIVIDERCHAR statement in thecorresponding DEF file.
DefParseMissingDesign	This rule is violated if the DESIGN statement is missing from a DEF file. In this case the top module name for the DEF file is unknown. For hierarchical DEF files this is fatal error because the hierarchy cannot be properly processed without this top module name. For non-hierarchical DEF files the top module name cannot be matched between design and layout if this rule is violated.You can fix this violation by adding an appropriate DESIGN statement in the corresponding DEF file.
DefParseMissingSpecialNet	This rule is violated if there is some SPECIALNET type definition in the DEF file; however no nets are defined for this SPECIAL NET type. This may indicate missing data for layout-aware diagnosis. You can fix this by adding appropriate net definitions for the SPECIAL NET type in the corresponding DEF file.

Table B-8. LEF/DEF Parser Warning Rules (cont.)

Rule Name	Description
DefParseEndDesign	This rule is violated if there are some DEF constructs (for example, NET definitions) after an 'END DESIGN' statement in a DEF file. Any such constructs are ignored by the parser and are used during database creation. You can fix this by ensuring that there is no data in the DEF file after an END DESIGN statement.

Instance, Net, and Pin Path Names in Layout Rule Violation Reports

In the layout rule violation mismatch report, Tessent Diagnosis reports the design instance/net/pin path names for various layout rule violations.

Table B-9 lists the layout rule violations that report instand/net/pin path names.

Table B-9. Rules Reporting Instance, Net, and Pin Path Name Violations

DesignInstanceMatch	DesignInstanceTypeMatch
DesignNetMatch	DesignNetPinMatch
LayoutNetPinMatch	

By default, Tessent Diagnosis reports these instance/net/pin path names in Siemens EDA DFT format.

Name Mismatch Reporting

At times the corresponding instance/net/pin path names in the layout format differ from the names in the DFT format. To remedy this mismatch, Tessent Diagnosis adds the DFT and layout names to the rule violation reporting.

For the rules in Table B-9, Tessent Diagnosis adds the DFT and layout names to the text of the violations as follows:

DesignInstanceMatch Violation

DesignInstanceMatch-<ID>: Design instance ***instance path name in DFT format*** (Expected Layout Name: ***instance path name in layout format***) does not exist in layout.

DesignInstanceTypeMatch Violation

DesignInstanceTypeMatch-<ID>: Design instance ***instance path name in DFT format*** (Expected Layout Name: ***instance path name in layout format***) type ***design cell name*** does not match with type ***layout cell name*** used in layout.

DesignNetMatch Violation

DesignNetMatch-<ID>: Design net ***net path name in DFT format*** (Expected Layout Name: ***net path name in layout format***) does not exist in layout.

DesignNetPinMatch Violation

If both the net and pin names differ between the DFT and layout formats:

DesignNetPinMatch-<ID>: Design pin ***pin path name in DFT format*** (Expected Layout Name: ***pin path name in layout format***) at net ***net path name in DFT format*** (Expected Layout Name: ***net path name in layout format***) does not exist in layout.

DesignNetPinMatch Violation

If only the net name differs between the DFT and layout format.

DesignNetPinMatch-<ID>: Design pin ***pin path name in DFT format*** at net ***net path name in DFT format*** (Expected Layout Name: ***net path name in layout format***) does not exist in layout.

DesignNetPinMatch Violation

If only the pin name differs between DFT and layout format.

DesignNetPinMatch-<ID>: Design pin ***pin path name in DFT format*** (Expected Layout Name: ***pin path name in layout format***) at net ***net path name in DFT format*** does not exist in layout.

LayoutNetPinMatch Violation

LayoutNetPinMatch-<ID>: Design pin ***pin path name in layout format*** at net ***net path name in DFT format*** (Expected Layout Name: ***net path name in layout format***) does not exist in design.

Instance, Net, and Pin Layout Path Names Suppression in Violation Reporting

You can suppress the reporting of instance/net/pin layout path names in the violations with the report_layout_rules command.

Specify the report_layout_rules command with the -no_layout_names switch as follows:

report_layout_rules layout_rule_id-occurrance# -no_layout_names

or

report_layout_rules -all_fails -no_layout_names

Instance, Net, and Pin Path Name Violation Examples

The following examples demonstrate instance, net, and pin path name violations.

Instance Name Violation Example

In this case the instance name in DFT format differs from that in layout format because the last part of the name is flat. Because the layout is being created from the LEF/DEF the instance path name in layout format is expected to be as follows:

```
/core0/cntl_1/mac\/adder\/AND0
```

Note that in LEF/DEF, then naming convention individual hierarchy delimiters are escaped.

Report the corresponding DesignInstanceMatch rule violation using the following command:

FAULT> report_layout_rules DesignInstanceMatch-1

```
// DesignInstanceMatch-1: Design instance /core0/cntl_1/mac/adder/AND0
(Expected Layout Name: /core0/cntl_1/mac\/adder\/AND0) not found in
layout.
```

You can suppress reporting of the layout name for this violation using the report_layout_rules command as follows:

FAULT> report_layout_rules DesignInstanceMatch-1 –no_layout_names

```
// DesignInstanceMatch-1: Design instance /core0/cntl_1/mac/adder/AND0 not
found in layout.
```

Net Name Violation Example

In this example, the following net is not found in the layout:

```
'/core0/cache_10/bank2/\dbit[32] '
```

In this case, the net name in DFT format differs from the layout format.

Report the corresponding DesignNetMatch rule violation using the following command:

FAULT> report_layout_rules DesignNetMatch-1

```
DesignNetMatch-<ID>: Design net /core0/cache_10/bank2/dbit[32](Expected
Layout Name: /core0/cache_10/bank2/dbit\[32\]) does not exist in layout.
```

You can suppress reporting of the layout name for this violation using the report_layout_rules command as follows:

FAULT> report_layout_rules DesignNetMatch-1 –no_layout_names

```
DesignNetMatch-<ID>: Design net /core0/cache_10/bank2/dbit[32] does not
exist in layout.
```

Pin Name Violation Example

In this example, the following design pin is not found in the layout:

Pin Name: '/core0/pll_10/freq_1/OR1/C'

On net: '/core0/pll_10/freq_1/\ctl_net[10] '

In this case, the pin name in layout format is same as in DFT format, however the net name differs.

Report the corresponding DesignNetPinMatch rule violation using the report_layout_rules command as follows:

```
FAULT> report_layout_rules DesignNetPinMatch-1
```

```
DesignNetPinMatch-1: Design pin /core0/pll_10/freq_1/OR1/C at net /core0/
pll_10/freq_1/ctl_net[10] (Expected Layout Name: /core0/pll_10/freq_1/
ctl_net\[10\]) does not exist in layout.
```

Layout Verification Examples

Some of the more common issues you could encounter during layout verification and LDB creation include common area, low percentage match, extra END LIBRARY in the LEF file, and missing DEF files.

Common Area Example

The common area is established by the Tessent Diagnosis tool after instance matching and represents a defined area of the layout that the tool uses for net-based matching.

Example B-1 illustrates the mismatch report, which reports a design-to-layout match of 95.55 percent. For clarity, the specific rule violations are in bold font. This example highlights the various sources of layout and design mismatches.

Figure B-1. Example Mismatch Report for Layout and Design

```
//  Mismatch Report
//  13394   (  4.45%)  design cell instances undefined
//                                                   (DesignInstanceMatch rule)
// 287622   ( 95.55%)  design cell instances matched with layout (common
//                                                                    area)
// ------------------------
// 301016              total number of design cell instances
//
//
//   8767   (  2.55%)  nets outside of common area (Hierarchy difference,
//                                                    DesignInstanceMatch rule)
//   5893   (  1.70%)  nets at the boundary of common area (Hierarchy
//                                                               difference,
//                                                    DesignInstanceMatch rule)
//     20   (  0.01%)  net not found at layout (DesignNetMatch rule)
//     29   (  0.01%)  nets with mismatched pins (LayoutNetPinMatch rule)
// 328939   ( 95.72%)  nets matched with layout
// ------------------------
// 343648              total number of design nets
//
//  Note: Design and layout match to 95.55%
```

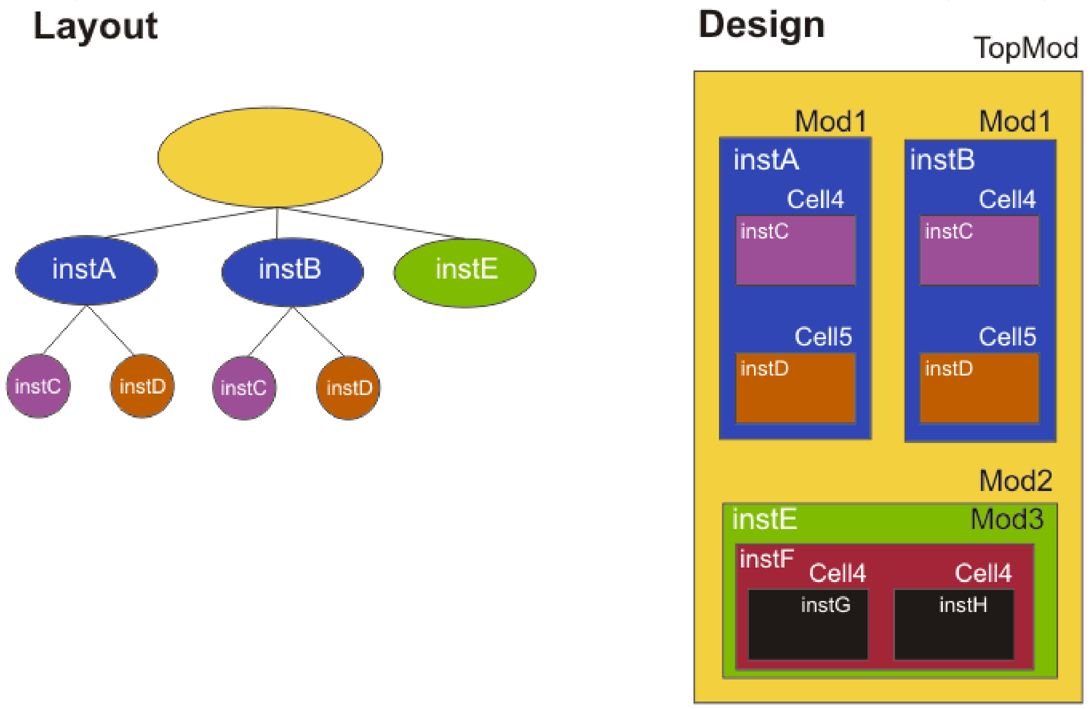
As described in "Mismatch Report" section, the report contains two sections: the first section reports instance information, the second section reports net information.

Example B-1 identifies the following information regarding instances:

- 13,394 undefined cells instances in the design that have no corresponding match in the layout, hence the DesignInstanceMatch rule violation.
- 287,622 matched design cell instances with the layout (common area).

Figure B-2 illustrates the common area for this example.

Figure B-2. Common Area for the Layout and Corresponding Design



In Figure B-2, the Tessent Diagnosis tool establishes the common area of the layout to match with the design. The design contains additional instances (instF, instG, and instH) that have no corresponding component in the layout; this produces a DesignInstanceMatch rule violation. Any elements below instF are outside the common area.

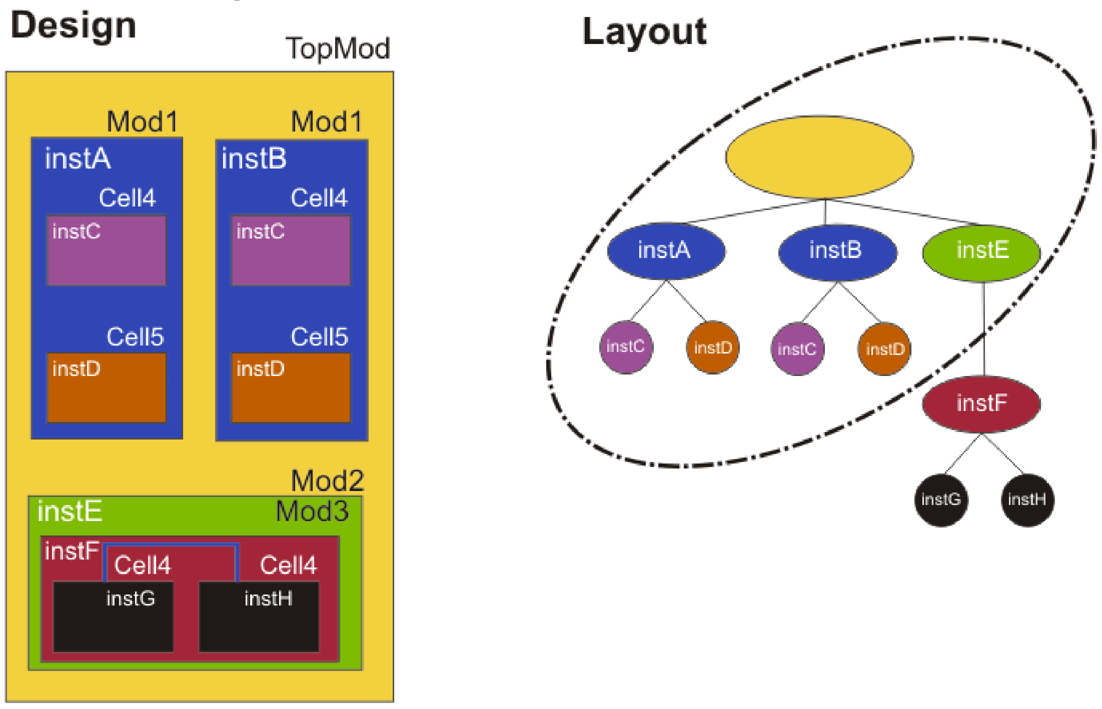
In Example B-1, the net information section reports the following information. For clarity, the specific rule violations are in bold font

```
//   8767  (  2.55%)  nets outside of common area (Hierarchy difference,
//                                                 DesignInstanceMatch rule)
//   5893  (  1.70%)  nets at the boundary of common area (Hierarchy
//                                                           difference,
//                                                 DesignInstanceMatch rule)
//     20  (  0.01%)  net not found at layout (DesignNetMatch rule)
//     29  (  0.01%)  nets with mismatched pins (LayoutNetPinMatch rule)
// 328939  ( 95.72%)  nets matched with layout
// ------------------------
// 343648             total number of design nets
```

In this example, the two items warranting attention are the nets outside the common area, and the nets on the boundary of the common area.

Figure B-3 illustrates nets outside the common area.

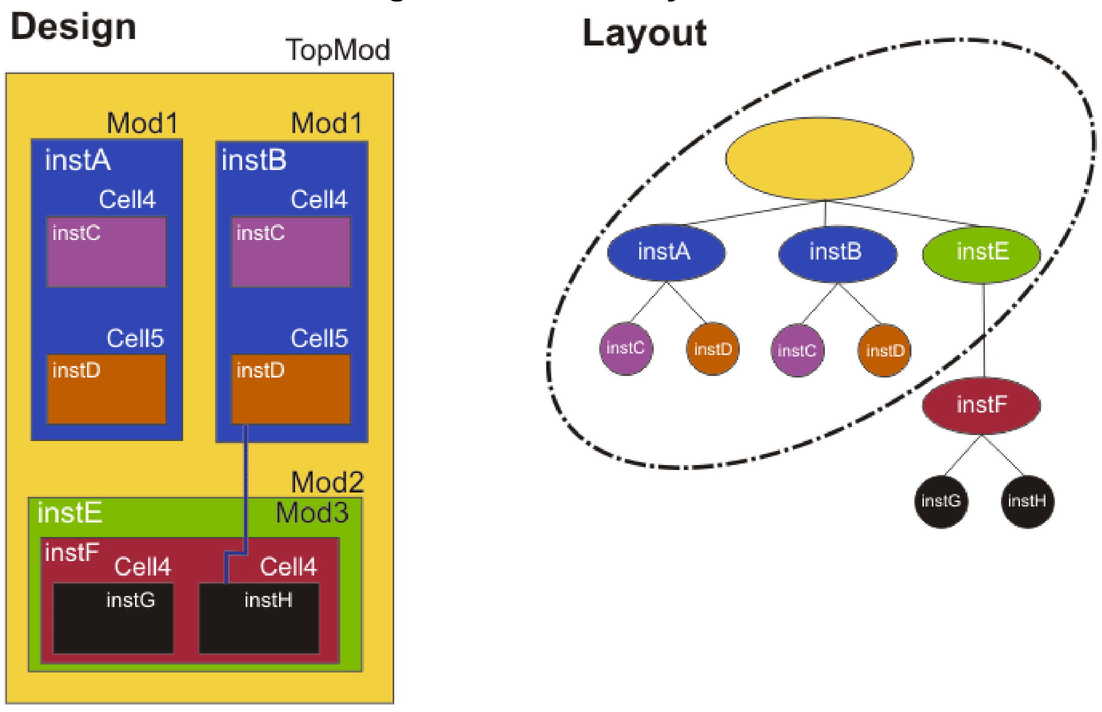
Figure B-3. Net Outside Common Area Example



In the figure, instF, instG, and instH cause a DesignInstanceMatch rule violation because these instances are present in the design, but not within the common area of the layout. In Figure B-3, the net shown in blue is an example of a net outside the common area.

Figure B-4 provides an example of boundary nets, which are nets at the boundary of the common area.

Figure B-4. Boundary Nets



In the figure, the net shown in blue is an example of a boundary net.

Low Percentage Match Example

The tool warns you when you have a low mismatch percentage, which you can then debug by first debugging low instance match numbers and then debugging any remaining net problems.

The following example shows the mismatch report for a low percentage match. The relevant portions of the example report are in bold font.

Figure B-5. Mismatch Report Example for a Low Percentage Match

```
Mismatch Report
// 4882  (100.00%) design cell instances undefined
//                                              (DesignInstanceMatch rule)
//    0  (  0.00%)  design cell instances matched with layout (common area)
// ------------------------
//  4882           total number of design cell instances
//  5413  (100.00%) nets outside of common area (DesignInstanceMatch rule)
//  0     (0.00%)   nets matched with layout
// ------------------------
//  5413           total number of design nets
//
//  Warning: Design and layout only match to 0.00%
//           Mismatch may result in incomplete layout aware diagnosis.
```

You debug this type of design-to-layout mismatch by using the following steps:

1. Start with the mismatch report
    a. First debug low instance match number
    b. Then debug any remaining net problems
2. Look at other rule violations for clues

In this example, you begin the debugging process by using the report_layout_rules command to view the specific DesignInstanceMatch violations:

FAULT> report_layout_rules DesignInstanceMatch

```
DesignInstanceMatch: #fails=4882 #checks=4882 handling=warning (design
instance path name does not match with layout).
...
//  Warning: DesignInstanceMatch-3225: Design instance uPORT/ix1490 does
//  not exist in layout.
...
```

According to the warning message, uPORT/ixl490 exists in the design, but has no corresponding component in the DEF file.

A review of the DEF file shows the file contains an extra layer of hierarchy (cpu_i) that is not present in the layout.

```
DESIGN cpu_edt_top ;
     COMPONENTS 7600 ;
      …….
     - cpu_i/uPORT/ix1490     ao21
       + PLACED ( 1498500 4554000 ) N ;
     ……..
    END COMPONENTS
END DESIGN
```

You solve this mismatch problem by using the -extra_layout_hierarchy switch to the create_layout command as in the following example:

FAULT> create_layout my_layout -extra_layout_hierarchy cpu_i -lef my_lef.lef -def my_def.def

Extra END LIBRARY in the LEF File Example

An extra END LIBRARY in the LEF file causes MacroExistence and DesignInstanceMatch rule violations.

The following example shows an example mismatch report caused by an extra END LIBRARY in the LEF file.

```
//  Layout Rule Violation Summary
//  Warning: Rule DesignInstanceMatch violated 48 times out of 5344
//  checks.
//  Warning: Rule MacroExistence violated 5335 times out of 7331 checks.
//  Warning: Rule LefParseEndLibrary violated 1 times out of 1 checks.
//
//  Mismatch Report
//    5344  (100.00%)  design cell instances undefined (MacroExistence
//                                                              rule,
//                                                DesignInstanceMatch rule)
//       0  (  0.00%)  design cell instances matched with layout (common
//                                                              area)
// ------------------------
//    5344             total number of design cell instances
//
//    5931  (100.00%)  nets outside of common area (MacroExistence rule,
//                                                 DesignInstanceMatch rule)
//       0  (  0.00%)  nets matched with layout
// ------------------------
//    5931             total number of design nets
//
//  Warning: Design and layout only match to 0.00%
//           Mismatch may result in incomplete layout aware diagnosis.
```

In the example, all design instances are undefined. More importantly, however, is the presence of the MacroExistence rule warning in the report; this is a clue that the design instance mismatches were due to a problem with the LEF files.

A review of the LEF file shows an extra END LIBRARY as in the following:

```
VERSION 5.6 ;
     MACRO oai33
     ...
     END oai33

END LIBRARY

     MACRO inv01
     ...
     END oai33

  ...

END LIBRARY
```

Missing DEF File Example

When you have a missing DEF file during LDB generation, the mismatch report includes the percentage of nets outside of the common area and the percentage of nets at the boundary of the common area.

The following example shows an example mismatch report caused by a missing DEF file. The relevant portions of the example report are in bold font.

```
//  Mismatch Report
//   11789   (  7.51%) design cell instances undefined (MacroExistence
//                     rule, DesignInstanceMatch rule)
//   145163 ( 92.49%) design cell instances matched with layout (common
//                                                                   area)
// ------------------------
//   156952            total number of design cell instances
//
//   10600   (  6.64%) nets outside of common area (MacroExistence rule,
//                                         DesignInstanceMatch rule)
//    4973   (  3.11%) nets at the boundary of common area (MacroExistence
//                                   rule, DesignInstanceMatch rule)
//       1   (  0.00%) nets not found at layout (DesignNetMatch rule)
//      71   (  0.04%) nets with mismatched pins (LayoutNetPinMatch rule)
//  144110   ( 90.21%) nets matched with layout
// ------------------------
//  159755             total number of design nets
//
//  Warning: Design and layout only match to 90.21%
//           Mismatch may result in incomplete layout aware diagnosis.
```

To debug this problem, use the report_layout_rules command to review the warnings for both the DesignInstanceMatch and MacroExistence rule violations as follows:

FAULT> report_layout_rules DesignInstanceMatch

```
report_layout_rules DesignInstanceMatch
DesignInstanceMatch: #fails=11722 #checks=156952 handling=warning (design
instance path name does not match with layout).
............
//  Design instance design_jtag/level1/foo_pm does not exist in
//  layout.
//  Design instance design_jtag/level1/foo_pm does not exist in
//  layout.
//  Design instance design_jtag/level1/foo_pm does not exist in
// layout.
```

FAULT> report_layout_rules MacroExistence

```
MacroExistence: #fails=387102 #checks=839382 handling=warning (macro not
exist).
............
//  The specified COMPONENT my_design_jtag\/level1\/foo_pm uses a
//  MACRO foo which is not defined in the specified LEF files.
```

It is now clear that the DEF file for the foo component is missing, and, consequently, the layout information is also missing.

Guidelines for Including or Excluding Design Modules From Mismatch Reporting

Under certain circumstances, it may be known that your design contains modules for which there is no corresponding layout information, specifically the DEF files.

Tessent Diagnosis reports the following layout rule violations for modules with no corresponding layout information:

- DesignInstanceMatch and DesignNetMatch for library cell instances and nets inside all instances of such modules.
- DesignNetPinMatch violations for nets on the boundary of all instances of such modules.

For instances in which the layout information is known to be missing, it is desirable to filter out the associated layout rule violations. You can filter the reporting of the mismatch violations in these cases by defining an excluded area using the following switches of the report_layout_rules command:

- -EXClude *design_module_name* — If you specify this switch with a design module, then any instance in the design that can trace its inclusion in a higher-level instance of an excluded module is considered to be in the excluded area. In other words, the excluded area comprises all design elements contained below instances of excluded modules.
- -INClude *design_module_name* — If you specify this switch, then any instance in the design that cannot trace its inclusion in a higher-level instance of an included module is considered to be in the excluded area. In other words, the excluded area comprises all design elements that are not below instances of included modules.

When using the -EXClude or -INClude switches with the report_layout_rules command, Tessent Diagnosis suppresses the reporting of the DesignInstanceMatch, DesignNetMatch, and DesignNetPinMatch violations in the excluded area as defined above. Consequently, you can analyze the other mismatch verification rule violations besides the violations associated with modules containing missing layout information.

In addition, Tessent Diagnosis reports the percentage match between the design and layout outside of the instance of the modules that are missing layout information.

Layout Rule Violation Report Generation

When you define an excluded area, you can regenerate the layout rule violation report using the –MISmatch_report switch to the report_layout_rules command.

When Tessent Diagnosis completes the report regeneration with the excluded area, the report provides enhanced metrics about the excluded area as in the following example.

Note

The total number of design cell instances does not count the excluded instances, and the total numer of design nets does not count the excluded nets.

```
//  Mismatch Report
//        115656  (  6.85%) design cell instances excluded from reporting
// For the remaining instances:
//       1186152  ( 75.43%) design cell instances undefined
//                          (DesignInstanceMatch rule)
//        386275  ( 24.57%) design cell instances matched with layout
//                          (common area)
// ------------------------
//       1572427            total number of design cell instances
//        113816  (  6.77%) nets in area excluded from reporting
//        1179  (  0.07%) nets at boundary of area excluded from reporting
// For the remaining nets:
//       1074672  ( 68.57%) nets outside of common area
//                          (DesignInstanceMatch rule)
//         10964  (  0.70%) nets at the boundary of common area
//                          (DesignInstanceMatch rule)
//          4825  (  0.31%) nets with mismatched pins (DesignNetPinMatch
                            rule, LayoutNetPinMatch rule)
//        476845  ( 30.42%) nets matched with layout
// ------------------------
//       1567306            total number of design nets
```

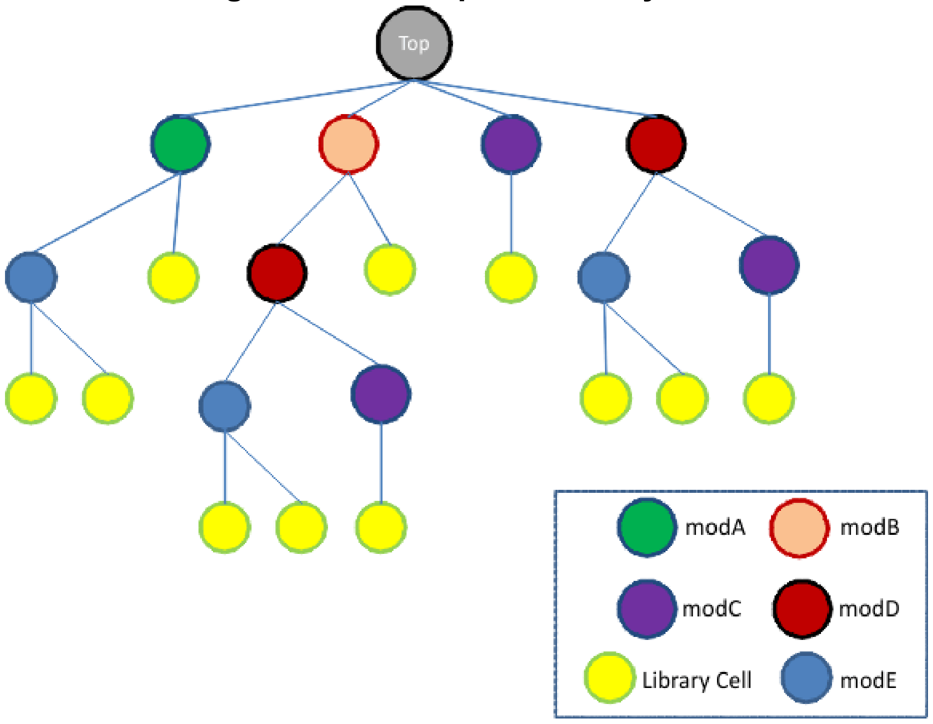
Excluded Area Examples

Tessent Diagnosis can determine the excluded area from the list of excluded/included modules you provide.

Example 1

The following figure shows a design’s hierarchy tree. Each node in the tree is an instance, and the nodes are color coded by the corresponding modules.

Figure B-6. Example Hierarchy Tree

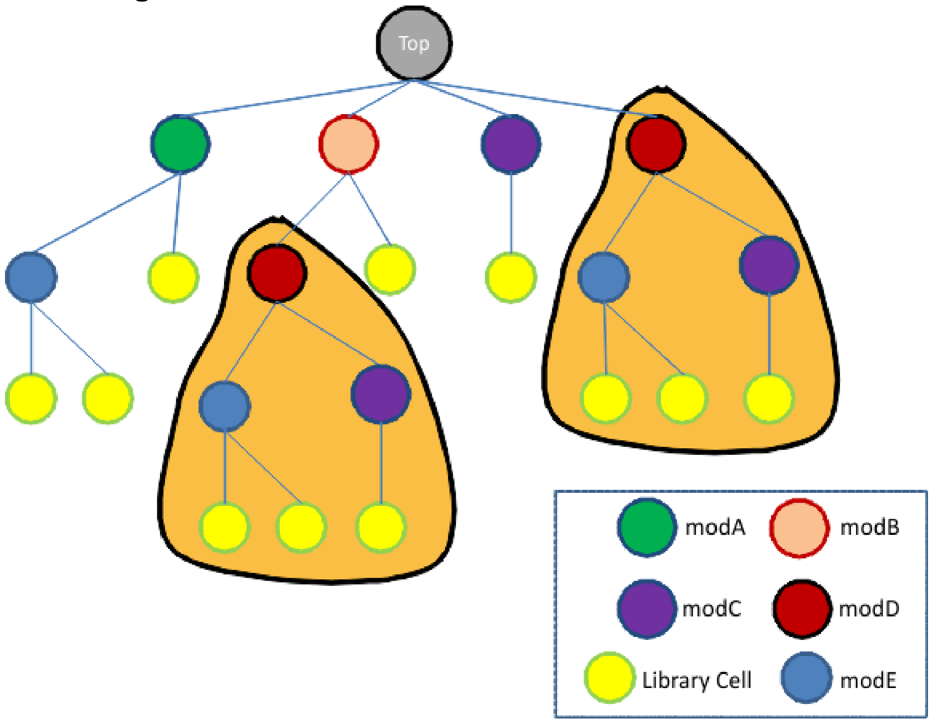


Continuing with Figure B-6, you exclude modD using report_layout_rules command with the following syntax:

report_layout_rules … -exclude modD

Figure B-7 shows the results of the exclude operation, specifically the shaded region.

Figure B-7. Excluded Area With ModD Excluded



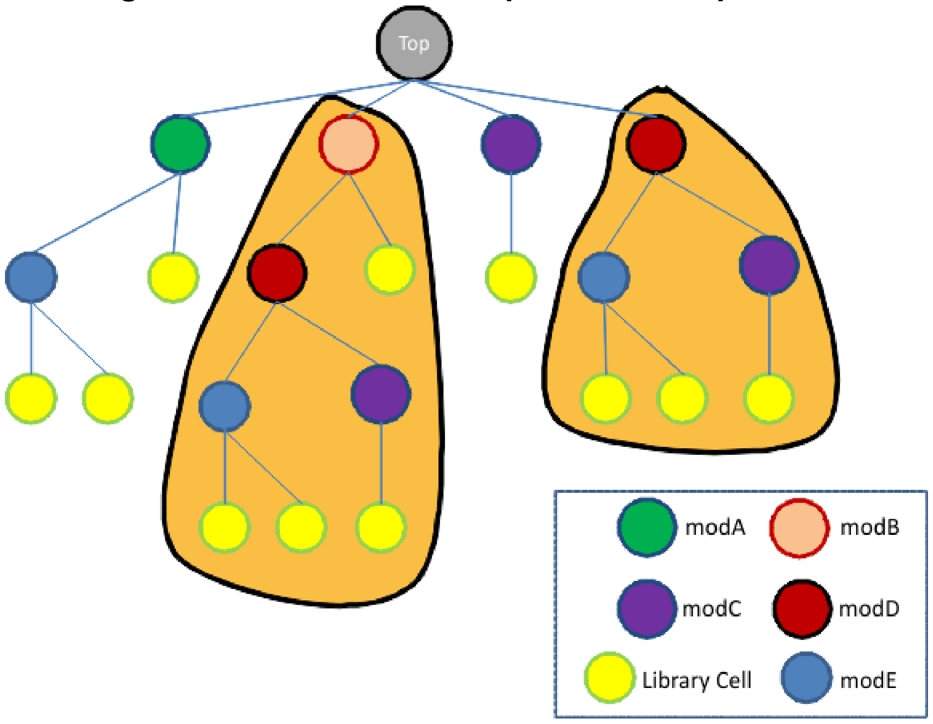
Example 2

Using the design hierarchy as shown below, this example uses the report_layout_rules command to exclude modB and modD.

report_layout_rules … -exclude modD modB

Figure B-8 shows the results of the exclude operation, including the excluded area.

Figure B-8. Results of Multiple Exclude Operation



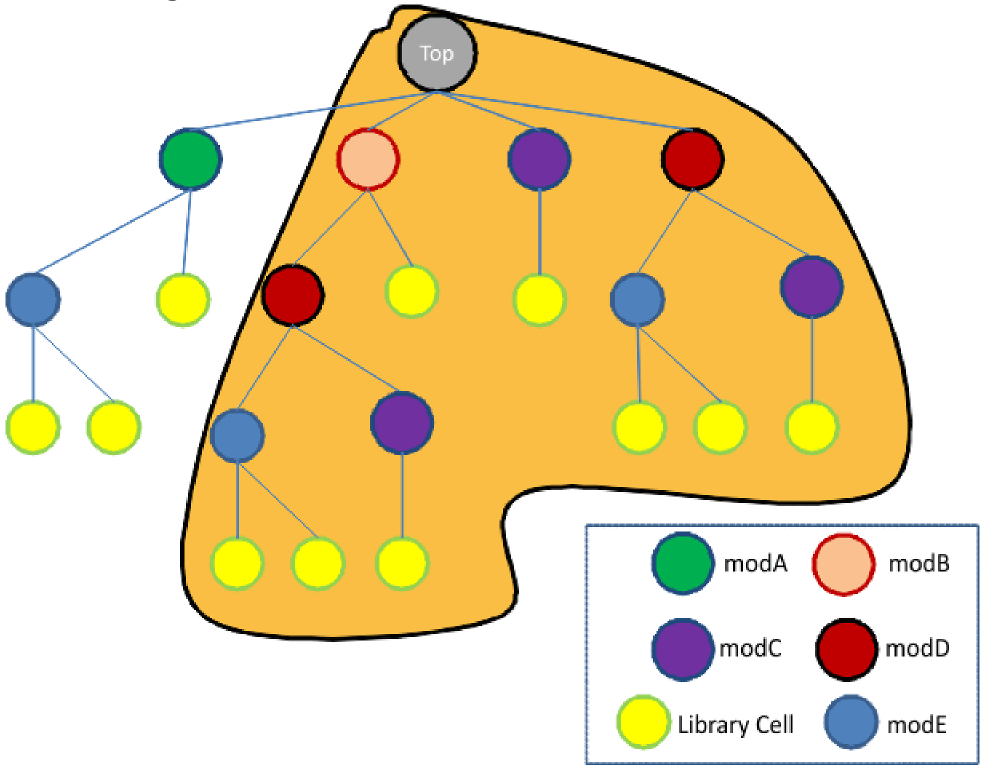
Example 3

Using the design hierarchy in the figure below, this example specifies modA as an included module using the following command:

report_layout_rules … -include modA

Figure B-9 shows the resulting excluded area.

Figure B-9. Excluded Area With modA Included



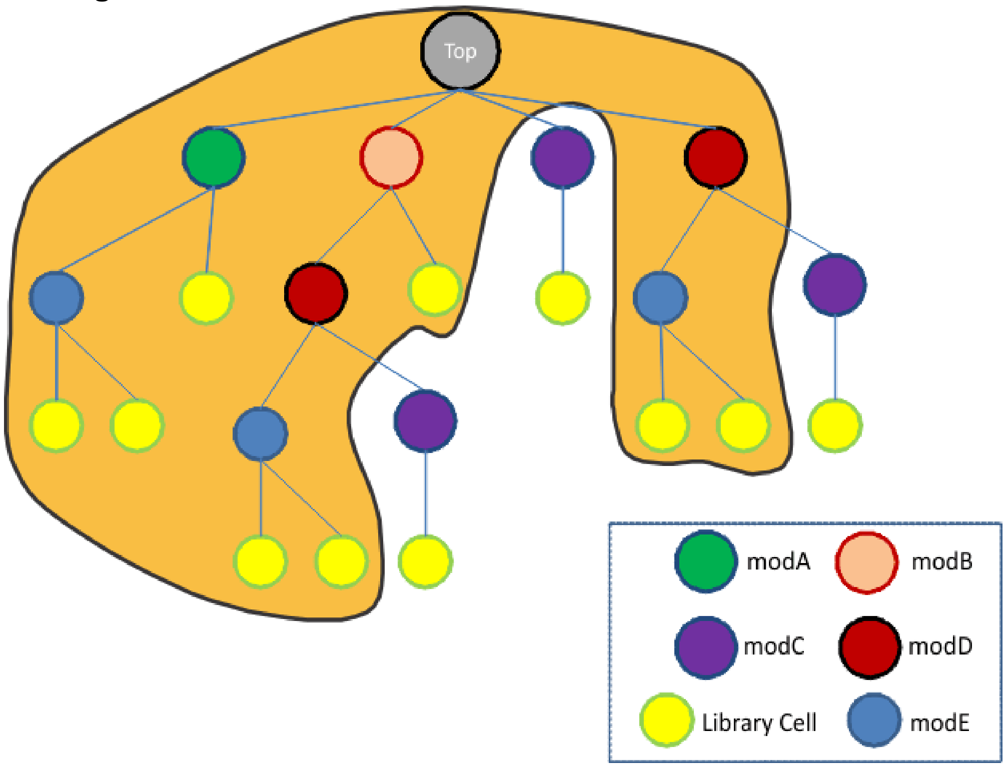
Example 4

Using the design hierarchy in the figure below, this example specifies modC as an included module using the following command:

```
report_layout_rules … -include modC
```

Figure B-10 shows the resulting excluded area.

Figure B-10. Excluded Area When modC an Included Area



report_layout_rules Usage Examples

The report_layout_rules command can help you debug mismatches.

The following examples are based on a design for which DEF information is missing for the following two modules:

- DRAMA — A memory module.
- DRAM_CTL — A memory controller.

Example 1

The following example demonstrates regenerating the layout rule violation mismatch report without specifying any excluded or included design modules.

```
FAULT> report_layout_rules -mismatch_report
```

```
//  Note: Recovering mismatch information...
//
//  Layout Rule Violation Summary
//  Warning: Rule DesignTopModulePinMatch violated 308 times out of 308
//            checks.
//  Warning: Rule DesignModuleCell violated 2 times out of 405 checks.
//  Warning: Rule DesignCellMatch violated 28 times out of 608 checks.
//  Warning: Rule DesignInstanceMatch violated 1301808 times out of
//           1688083 checks.
//  Warning: Rule DesignNetPinMatch violated 7200 times out of 1489200
//           checks.
//  Warning: Rule LayoutNetPinMatch violated 107 times out of 1544875
          checks.
//  Warning: Rule ViaDuplicate violated 406 times out of 420 checks.
//  Note: The command 'report_layout_rules' can be used for detailed
//           information on rule violations
//
//  Mismatch Report
//       1301808  ( 77.12%)design cell instances undefined
//           (DesignInstanceMatch rule)
//        386275  ( 22.88%)design cell instances matched with layout
//           (common area)
// ------------------------
//       1688083         total number of design cell instances
//
//       1125560  ( 66.91%)nets outside of common area
//             (DesignInstanceMatch rule)
//         75071  (  4.46%)nets at the boundary of common area
//            (DesignInstanceMatch rule)
//          4825  (  0.29%)nets with mismatched pins (DesignNetPinMatch
//          rule, LayoutNetPinMatch rule)
//        476845  ( 28.34%)nets matched with layout
// ------------------------
//       1682301         total number of design nets
//
//  Warning: Design and layout only match to 22.88%
//           Mismatch may result in incomplete layout aware diagnosis.
```

From the layout rule violation mismatch report, a total of 1,301,808 design library cell instances are missing from the layout. Tessent Diagnosis identifies these cell instances as missing because no corresponding design instances of the cells are in the instances of DRAMA and DRAM_CTL modules.

Example 2

Continuing from Example 1, the following example uses the report_layout_rules command to exclude the DRAMA module from the layout rule violation mismatch report and reports the violations of the DesignInstanceMatch from this operation.

FAULT> report_layout_rules DesignInstanceMatch -exclude DRAMA

```
DesignInstanceMatch: #fails=115656 #checks=1688083 handling=warning
(design instance path name does not match with layout).
//  Warning: DesignInstanceMatch-148270: Design instance core0/
dram_cnt_block/AND1 does not exist in layout.
//  Warning: DesignInstanceMatch-148271: Design instance core0/
dram_cnt_block/AND2 does not exist in layout.
//  Warning: DesignInstanceMatch-148272: Design instance core0/
dram_cnt_block/OR1 does not exist in layout.
//  Warning: DesignInstanceMatch-148273: Design instance core0/
dram_cnt_block/BUF2 does not exist in layout.
```

The number of violations is now 115,656, which means that the remaining (1,301,808 – 115,656 = 1,186,152) violations are in instances of the DRAMA module. The violation IDs (in bold font) do not change, but remain the same as if no excluded area was specified.

Example 3

Continuing from Example 2, the following example uses the report_layout_rules command to regenerate the layout rule violation mismatch report with module DRAMA excluded.

FAULT> report_layout_rules -mismatch_report -exclude DRAMA

```
// Note: Recovering mismatch information...

//

//  Layout Rule Violation Summary

//  Warning: Rule DesignTopModulePinMatch violated 308 times out of 308
checks.

//  Warning: Rule DesignModuleCell violated 2 times out of 405 checks.

//  Warning: Rule DesignCellMatch violated 28 times out of 608 checks.

           // Warning: Rule DesignInstanceMatch violated 115656 times out
of 1688083 checks.

//  Warning: Rule DesignNetPinMatch violated 7200 times out of 1489200
checks.

//  Warning: Rule LayoutNetPinMatch violated 107 times out of 1544875
checks.
```

```
//  Warning: Rule ViaDuplicate violated 406 times out of 420 checks.
//  Note: The command 'report_layout_rules' can be used for detailed
information on rule violations
//
//  Mismatch Report
            // 1186152  ( 70.27%)design cell instances excluded from
reporting
//  For the remaining instances:
            // 115656  ( 23.04%)design cell instances undefined
(DesignInstanceMatch rule)
//  386275  ( 76.96%)design cell instances matched with layout (common
area)
// ------------------------
//  501931 total number of design cell instances (not counting the
excluded instances)
//
            // 1074672  ( 63.88%)nets in area excluded from reporting
            // 11798  (  0.70%)nets at boundary of area excluded from
reporting
//  For the remaining nets:
```

```
24 // 50888  (  8.54%) nets outside of common area (DesignInstanceMatch rule)

25 // 63273  ( 10.62%) nets at the boundary of common area
    (DesignInstanceMatch rule)

26 // 4825  (  0.81%) nets with mismatched pins (DesignNetPinMatch rule,
   LayoutNetPinMatch rule)

27 // 476845  ( 80.03%) nets matched with layout

28 // ------------------------

29 // 595831         total
   number of design nets (not counting the excluded nets)

30 //

31
             // Warning: Design and layout only match to 76.96%

32 //Mismatch may result in incomplete layout aware diagnosis.
```

This example contains a number of lines in bold font, the meaning of which is as follows:

- Line 7 shows that the number of DesignInstanceMatch rule violations drops to 115,656 from 1,301,808 because the remaining violations lie in some instance of the DRAMA module and are excluded from the report.
- Line 14 identifies 1,186,152 design library cell instances that are in the excluded area.
- Line 16 contains the number of undefined instances, which has dropped to 115,656. The number specifies that 23.04% of the design cell instances are outside of the excluded area.
- Lines 21 and 22 provides the number of nets inside (1,074,672) and on the boundary (11,798) of the excluded area, respectively. In this example, 80.03% of nets outside the excluded area match with the layout.
- Line 31 shows the overall match percentage (76.96%) outside the excluded area.

Example 4

The following example demonstrates using the report_layout_rules command to re-generate the layout rule violation mismatch report with both the DRAMA and DRAM_CTL modules excluded.

FAULT> report layout rules -mismatch_report -exclude DRAMA DRAM_CTL

```
// Note: Recovering mismatch information...
//
//  Layout Rule Violation Summary
//  Warning: Rule DesignTopModulePinMatch violated 308 times out of 308
checks.
//  Warning: Rule DesignModuleCell violated 2 times out of 405 checks.
//  Warning: Rule DesignCellMatch violated 28 times out of 608 checks.
//  Warning: Rule DesignNetPinMatch violated 7200 times out of 1489200
checks.
//  Warning: Rule LayoutNetPinMatch violated 107 times out of 1544875
checks.
//  Warning: Rule ViaDuplicate violated 406 times out of 420 checks.
//  Note: The command 'report_layout_rules'
can be used for detailed information on rule violations
//
//  Mismatch Report
// 1301808  ( 77.12%)design cell instances excluded from reporting
//  For the remaining instances:
```

```
15 //  386275  (100.00%)design cell instances matched with layout (common
   area)

16 // ------------------------

17 // 386275  total number of design cell instances (not counting the
   excluded instances)

18 //

19 // 1188488  ( 70.65%) nets in area excluded from reporting

20 // 12143   (  0.72%) nets at boundary of area excluded from reporting

21 //  For the remaining nets:

22 // 4825  (  1.00%) nets with mismatched pins (DesignNetPinMatch rule,
   LayoutNetPinMatch rule)

23 // 476845  ( 99.00%) nets matched with layout

24 // ------------------------

25 // 481670 total number of design nets (not counting the excluded nets)

26 //

27 //  Note: Design and layout match to 99.00%
```

The layout rule violation mismatch report shows on Line 15 that all the instances outside the excluded area match with the layout. This is consistent because the DEF files for only DRAMA and DRAM_CTL are missing. By excluding these known violations, you can focus on the remaining DesignNetPinMatch and LayoutNetPinMatch rule violations on Line 22.

Example 5

The following example demonstrates using the report_layout_rules command to report the layout rule violations in a given module using the command's –INClude switch.

FAULT> report_layout_rules -mismatch_report -include DRAMA

```
// Note: Recovering mismatch information...
//
//  Layout Rule Violation Summary
//  Warning: Rule DesignTopModulePinMatch violated 308 times out of 308 checks.
//  Warning: Rule DesignModuleCell violated 2 times out of 405 checks.
//  Warning: Rule DesignCellMatch violated 28 times out of 608 checks.
//  Warning: Rule DesignInstanceMatch violated 1186152 times out of 1688083 checks.
//  Warning: Rule LayoutNetPinMatch violated 107 times out of 1544875 checks.
//  Warning: Rule ViaDuplicat violated 406 times out of 420 checks.
//  Note: The command 'report_layout_rules'
can be used for detailed information on rule violations
//
//  Mismatch Report
//  501931  ( 29.73%) design cell instances excluded from reporting
//  For the remaining instances:
// 1186152  (100.00%)design cell instances undefined (DesignInstanceMatch rule)
// 0  (  0.00%) design cell instances matched with layout (common area)
// ------------------------
// 1186152 total number of design cell instances (not counting the excluded instances)
```

```
//
// 607629  ( 36.12%)nets in area excluded from reporting
//  For the remaining nets:
//  1074672  (100.00%)nets outside of common area (DesignInstanceMatch rule)
//  0  (  0.00%)nets matched with layout
// ------------------------
//  1074672 total number of design nets (not counting the excluded nets)
//
//  Warning: Design and layout only match to 0.00%
//  Mismatch may result in incomplete layout aware diagnosis.
```

The only the violations that are inside instances of the DRAMA module are included in the percentage match calculation.

Appendix C
Diagnosis Report Signature Formats

The Tessent Diagnosis diagnosis report's XMAP data has two formats depending on the signature type: failure signature and MD5 signature.

Failure Signature Format

The failure signature information in the diagnosis report begins with the CHANNEL_BEGIN section.

See "Failure Signature Information in the Diagnosis Report" for complete information.

The following example describes the XMAP format for the failure signature information (in **bold**) in the Tessent Diagnosis diagnosis report.

```
XMAP_TABLE_BEGIN
:      version
:      [ALL_FAILURE_INFO_BEGIN
:           [EDT = ON | OFF]
:// EDT = ON | OFF is optional. "EDT = ON" is the default indicates the
:// diagnosis report is for an EDT design, and "EDT = ON" does not show
:// in the diagnosis report. The only time the "EDT = ..." statement
:// is present is when "EDT = OFF" is specified, which indicates the
:// diagnosis report is for a non-EDT design.
:           TOTAL_FAILURE_BITS = <F>
:           MD5 signature data
:      ALL_FAILURE_INFO_END]
:      [CHANNEL_BEGIN
:          TOTAL_CHANNELS = <C>
:          channel         pin_name FBR
:          PO              <PO_PIN1> <PO_PIN1_FBR> <channel_1>
:                          <channel_pin_1> <channel_1_FBR>
:          ...
:          <channel_X>     <channel_pin_X> <channel_X_FBR>
:      CHANNEL_END]
:      [CHANNEL_OFFSET_BEGIN
:          TOTAL_CHANNEL_OFFSETS = <CO>
:          channel     pin_name offset       FBR
:          PO          <PO_PIN1> -1           <PO_PIN1_FBR>
:          <channel_1> <pin_1> <offset_0>   <channel_1(offset_0)_FBR>
:          ...
:          <channel_X> <pin_X> <offset_Y>   <channel_X(offset_Y)_FBR>
:      CHANNEL_OFFSET_END]
:      [OFFSET_BEGIN
:           TOTAL_OFFSETS = <O>
:           offset           FBR
:           <offset_1>       <offset_1_FBR>
:           ...
:           <offset_k>       <offet_k_FBR>
:      OFFSET_END]
:      [OFFSET_PATTERN_BEGIN
:           TOTAL_OFFSET_PATTERNS = <OP>
:           offset      pattern_id  FBR
:           <offset_1>  <pat0>        <offset_1_pat0_FBR>
:           ...
:           <offset_k>  <patW>       <offset_k_patW_FBR>
:       OFFSET_PATTERN_END]
:       [PATTERN_BEGIN
:          TOTAL_PATTERN_IDS = <P>
:          pattern_id       FBR
:          <pat0>           <pat0_FBR>
:          ...
:         <patY>            <patY_FBR>
:        PATTERN_END]
:      [CELL_XMAP_BEGIN
:      format
:             cell type data
:      CELL_XMAP_END]
:      [SPICE_XMAP_BEGIN
:         format
:        SPICE name mapping data
:      SPICE_XMAP_END]
:XMAP_TABLE_END
```

:

MD5 Signature Format

The MD5 signature format is similar to the failure signature format.

See "MD5 Signature Information in the Diagnosis Report" for complete information.

The following example describes the XMAP format for the MD5 signature information in the Tessent Diagnosis diagnosis report.

```
XMAP_TABLE_BEGIN
     version
     [ALL_FAILURE_INFO_BEGIN
          [EDT = ON | OFF]
// EDT = ON | OFF is optional. "EDT = ON" is the default indicates the
// diagnosis report is for an EDT design, and "EDT = ON" does not show
// in the diagnosis report. The only time the "EDT = ..." statement
// is present is when "EDT = OFF" is specified, which indicates the
// diagnosis report is for a non-EDT design.
          TOTAL_FAILURE_BITS = <F>
          design:<abs_path_flat_model> = <netlist md5 string>
          [pattern:<abs_path_pattern_1> = <pattern file 1 md5 string>
          pattern:<abs_path_pattern_2> = <pattern file 2 md5 string>]
……
    ALL_FAILURE_INFO_END]
     [CELL_XMAP_BEGIN
     format
            cell type data
     CELL_XMAP_END]
     [SPICE_XMAP_BEGIN
       format
       SPICE name mapping data
     SPICE_XMAP_END]
     [CHANNEL_BEGIN
         ...
     CHANNEL_END]
     [CHANNEL_OFFSET_BEGIN
         ...
     CHANNEL_OFFSET_END]
     [OFFSET_BEGIN
         ...
     OFFSET_END]
     [OFFSET_PATTERN_BEGIN
         ...
     OFFSET_PATTERN_END]
     [PATTERN_BEGIN
         ...
     PATTERN_END]
XMAP_TABLE_END
```

Appendix D
STDF-V4 2007 ATDF Record Examples

STDF-V4 2007(.1) ATDF records can appear for Teradyne and Verigy ATEs.

Teradyne Record ATDF Examples

Teradyne ATDF records include a Name Map Record.

Version Update Record (VUR)

```
Record[ 0 ]: VUR, Type/Subtype=0/30, Length=8
  UPD_NAME : V4-2007
```

Name Map Record (NMR)

```
Record[ 1 ]: NMR, Type/Subtype=1/91, Length=273
  CONT_FLG : 1
  TOTM_CNT : 48
  LOCM_CNT : 23
  PMR_INDX : 1 2 3 4 5 6 7 8 9 10
             11 12 13 14 15 16 17 18 19 20
             21 22 23
  ATPG_NAM : scan_in1 scan_in10 scan_in11 scan_in12 scan_in13 scan_in14
             scan_in15 scan_in16 scan_in17 scan_in18 scan_in19 scan_in2
             scan_in20 scan_in21 scan_in22 scan_in23 scan_in24 scan_in3
             scan_in4 scan_in5 scan_in6 scan_in7 scan_in8
```

Pattern Sequence Record (PSR)

```
Record[ 4 ]: PSR, Type/Subtype=1/90, Length=326
  CONT_FLG : 0
  PSR_INDX : 1
  PSR_NAM : my_ATPG_patterns.PAT
  OPT_FLG : 0
  TOTP_CNT : 1
  LOCP_CNT : 1
  PAT_BGN : 0
  PAT_END : 8274
  PAT_FILE : my_stil_patterns.stil
  PAT_LBL : my_ATPG_patterns.PAT
  FILE_UID : My Failures
  ATPG_DSC : Tessent FastScan v9.5...
  SRC_ID : PatternExec
```

Scan Test Record (STR)

```
Record[ 5 ]: STR, Type/Subtype=15/30, Length=7004
  CONT_FLG : 0
  TEST_NUM : 4100
  HEAD_NUM : 0
  SITE_NUM : 0
  PSR_REF : 1
  TEST_FLG : 192
  LOG_TYP : SWB
  TEST_TXT : My Testing
  ALARM_ID : 'NULL'
  PROG_TXT : An example of testing
  RSLT_TXT : 'NULL'
  Z_VAL : 3
  FMU_FLG : 8
  MASK_MAP :
  FAL_MAP :
  CYCL_CNT : 8274
  TOTF_CNT : 566
  TOTL_CNT : 566
  CYC_BASE : 0
  BIT_BASE : 0
  COND_CNT : 2
  LIM_CNT : 0
  CYC_SIZE : 8
  PMR_SIZE : 2
  CHN_SIZE : 0
  PAT_SIZE : 0
  BIT_SIZE : 0
  U1_SIZE : 0
  U2_SIZE : 0
  U3_SIZE : 0
  UTX_SIZE : 0
  CAP_BGN : 32767
  LIM_INDX :
  LIM_SPEC :
  COND_LST :  DC_COND=DC Context = GrossLevels.Nominal
AC_COND=Spec(_per_100) = 0.0000002
  CYC_CNT : 566
  CYC_OFST :  176 177 178 179 180 180 181 182 184 185
              ...
              8038 8040 8042 8046 8070 8072
  PMR_CNT : 566
  PMR_INDX :  27 33 27 33 27 33 33 33 33 27
              ...
              29 29 29 29 25 25
  CHN_CNT : 0
  CHN_NUM :
  EXP_CNT : 566
  EXP_DATA :  76 76 76 72 72 72 72 72 72 72
              ...
              72 72 72 72 72 76
  CAP_CNT : 566
  CAP_DATA :  72 72 72 76 76 76 76 76 76 76
              ...
              76 76 76 76 76 72
  NEW_CNT : 0
```

```
    NEW_DATA :
    PAT_CNT : 0
    PAT_NUM :
    BPOS_CNT : 0
    BIT_POS :
    USR1_CNT : 0
    USR1 :
    USR2_CNT : 0
```

Verigy Record ATDF Examples

Verigy ATDF records are relatively concise.

Version Update Record (VUR)

```
  Record[ 2 ]: VUR, Type/Subtype=0/30, Length=8
    UPD_NAME : V4-2007
```

Pattern Sequence Record (PSR)

```
  Record[ 1045 ]: PSR, Type/Subtype=1/90, Length=64963
    CONT_FLG : 1
    PSR_INDX : 0
    PSR_NAM : testing
    OPT_FLG : 14
    TOTP_CNT : 300
    LOCP_CNT : 200
    PAT_BGN :  0  0  0  0  0  0  0  0  0  0
               0  0  0  0  0  0  0  0  0  0
               ...
               0  0  0  0  0  0  0  0  0
    PAT_END :  0  0  0  0  0  0  0  0  0  0
               ...
               0  0  0  0  0  0  0  0  0
    PAT_FILE :  NULL NULL NULL NULL NULL NULL NULL NULL NULL NULL
                ...
                NULL NULL NULL NULL NULL NULL NULL NULL NULL
    PAT_LBL :  pattern 0 pattern 1 pattern 2 pattern 3 pattern 4 pattern 5
               ...
               pattern 197 pattern 198 pattern 199
```

Appendix E
Layout-Aware Diagnosis Marker File Semantics

Layout-Aware Diagnosis with Tessent Diagnosis enables you to write out the layout location of each suspect. Tessent Diagnosis provides this layout information in addition to the usual layout-aware diagnosis report and places it into a marker file.

About the Marker File

Tessent Diagnosis writes the marker file in a format that can be read into and displayed in any layout viewer that supports the Calibre RVE format. If a layout viewer does not support the RVE format, it may be necessary to convert from the RVE format into another format. This section describes the semantics of the marker file such that you can use of the information for your purposes.

From a high level description, a marker file contains two main pieces of information:

1. X, Y and layer polygons for the suspected defect locations, that enclose the potential defect location. These locations are called defect bounding boxes.
2. X, Y, and layer polygons that describe the layout of the nets and cells associated with the defect suspects. This set of polygons are called the landmark polygons.

The marker file follows the Calibre® DRC file syntax. Therefore, Calibre DRC-RVE can read it and display its content through Calibre DRV.

The full syntax of the file is documented in the *Calibre Verification User's Manual* section "ASCII nmDRC Results Database Format." This section explains the semantics of the marker file for the LEF/DEF-based flow.

Marker File Semantics

Each marker file starts with the name of the top module or design. Calibre DRV checks this name when you want to highlight some of the Tessent Diagnosis results. It prompts you for input and confirmation if the top-module names do not match. After the top-module name, separated by a white space, the data base precision is given.

Below is an example based on portions of a marker file. Note that the line numbers were added for the purpose of this discussion, line numbers are not part of the marker file syntax.

```
1     cpu_edt_top 2000
2     /cpu_i/uPORT/nx2332 route_2 3
3     32 32 0 Tue Feb 24 06:55:57 GMT 2009
4     p 1 4
5     ALL 3
6     SUSPECT-1-1 3
7     SUSPECT-1-1.1 3
8     SYMPTOM-1 3
9     2027000 4765000
10    2033000 4765000
11    2033000 4893000
12    2027000 4893000
13    p 2 4
14    ALL 3
15    SUSPECT-1-1 3
16    SUSPECT-1-1.1 3
17    SYMPTOM-1 3
18    2061000 4765000
19    2067000 4765000
20    2067000 5312000
21    2061000 5312000
```

Line 1 is the name of the top module or design. Lines 2 and 3 are the preamble for a sequence of polygon information in the following lines. Line 2 consists of three parts, separated by white spaces. The first part is a string representing the “DRC-rule” name. Tessent Diagnosis places either the name of the net (see line 2) or an identifier of the defect bounding box of a particular suspect (see lines 42 and 53) there. The second string on the line is the name of the layer the following polygons are on, followed by a layer ID that is just a unique, numerical identifier (>0) for the layer name string. The layer ID is used later on in the property statements, as shown in lines 5 through 8.

Line 3 opens with the number of polygons to follow, that all belong to the entity declared on the line above, here the net “/cpu_i/uPORT/nx2332” on layer “route_2”. In this example, the net has 32 polygons on this layer, thus the syntax is “32 32 0”. Each number separated by a white space. After this, the date and time is printed in the shown format.

The set of lines describing one polygon starts with line 4 and ends with line 12. The next polygon description, if there is one, follows directly thereafter as shown in lines 13 to 21. Each sequence starts with an enumerator, here “p 1”, refers to polygon 1 (of 32 in this example), followed by the number of coordinate pairs (four in this case). All string-number-pairs between line 4 and the beginning of the coordinate pairs (line 9) are “DRC properties”. The number of such lines is arbitrary and can be zero. Tessent Diagnosis fills in the property statements to tie the polygon to a particular symptom and suspect. The key words Tessent Diagnosis uses follow “ALL layerID”, “SUSPECT-id1-id2 layerID”, and “SYMPTOM-id1 layerID”. id1 always references the symptom enumerator from the original diagnosis report, and id2 references back to the suspect number from the original diagnosis report. In the example, the polygon described by lines 9 through 12 is part of symptom 1 (SYMPTOM-1), suspect 1 of symptom 1 (SUSPECT-1-1) and suspect 1.1 of symptom 1 (SUSPECT-1-1.1), each of them is on the layer with the ID “3”.

Tessent Diagnosis compacts the marker file in the sense that each polygon is listed exactly once. Tessent Diagnosis adds as many property statements as necessary. Therefore, it is common to see that a polygon is part of several symptoms and suspects.

Lines 9 through 12 define the four corner coordinates of the polygon. The coordinates are given in the common X Y semantic and in a counter-clockwise direction, starting with the lower left corner.

Directly following the last coordinate line, the next polygon definition, ‘2 of 32’, starts on line 13. The last polygon of the sequence, polygon ‘32 of 32’, is given in lines 22 through 30 to show the transition to the next set of data (line 31). Note that polygons 3 through 31 have not been presented here. The next set of data starts in line 31 with a line equivalent to line 2. In this example, there are 15 polygons following (as declared on line 32), all the polygons belong to the net “/cpu_i/uPORT/nx2332”, on layer “v_lay2”, with the layer ID “4” (line 31).

```
22    p 32 4
23    ALL 3
24    SUSPECT-1-1 3
25    SUSPECT-1-1.1 3
26    SYMPTOM-1 3
27    2618000 5427000
28    2626000 5427000
29    2626000 5435000
30    2618000 5435000
31    /cpu_i/uPORT/nx2332 v_lay2 4
32    15 15 0 Tue Feb 24 06:55:57 GMT 2009
33    p 1 4
34    ALL 4
35    SUSPECT-1-1 4
36    SUSPECT-1-1.1 4
37    SYMPTOM-1 4
38    2620000 5307000
39    2624000 5307000
40    2624000 5311000
41    2620000 5311000
```

Lines 42 and 53 give examples, that a set of data may not start with a net name, but with a defect bounding box identifier. Tessent Diagnosis computes these polygons out of the layout data, they don’t exist in the layout. Tessent Diagnosis adds these polygons to the marker file.

```
42    SUSPECT-1-1.1 route_2 11
43    1 1 0 Tue Feb 24 06:55:57 GMT 2009
44    p 1 4
45    ALL 11
46    SYMPTOM-1 11
47    SUSPECT-1-1 11
48    SUSPECT-1-1.1 11
49    2618000 5305000
50    2626000 5305000
51    2626000 5313000
52    2618000 5313000
53    SUSPECT-1-1.1 v_lay2 12
54    1 1 0 Tue Feb 24 06:55:57 GMT 2009
55    p 1 4
56    ALL 12
57    SYMPTOM-1 12
57    SUSPECT-1-1 12
58    SUSPECT-1-1.1 12
59    2620000 5307000
60    2624000 5307000
61    2624000 5311000
62    2620000 5311000
```

Lines 42 through 52 define the defect bounding box polygon for symptom 1, suspect 1.1, on layer “route_2”, which has the ID “11”. Lines 54 through 63 define the defect bounding box for the same symptom and suspect, but on layer “v_lay2”, with ID “12”.

You might have noticed that there are two IDs for the layer names. For example, layer “route_2” has the IDs 3 and 11. This is for the purpose of distinguishing the actual layers of nets, from the virtual layers of the defects. Thus, all defect bounding boxes of “route_2” have the ID 11, and all net polygons on “route_2” have ID 3. The net layer IDs usually start with 1 — but this cannot be assumed — and enumerate all routing and via layers starting with metal 1 (layer ID ‘m’), in sequence, to the top-most metal layer with id ‘m+n’. The first virtual defect layer has the ID ‘m+n+1’ and relates to the first layer ID ‘m’. The virtual layer IDs enumerate all virtual layers parallel to the actual layers from ‘m+n+1’ through ‘m+n+n’.

The marker file ends with the last polygon. There is no end-of-file syntax element.

Appendix F
Generating DEF from Other Tools

You can use Synopsys IC Compiler and AtopTech APRISA tools to generate DEF files for layout-aware diagnosis.

Generating DEF from IC Compiler

When you export a design by saving design data in a hierarchical Verilog netlist file, you can save the physical constraints in a DEF file. To do this, you need to include and exclude particular DEF sections. Including a section exports and saves it into the DEF file. Excluding a section leaves it out of the DEF file.

Procedure

1. Specify the DEF file name and select the DEF version.

   The choices are 5.6, 5.5, 5.4, and 5.3. The default version is 5.5. The preferred version is 5.6.

2. Ensure that the default value of 1000 is selected for the unit conversion factor.

   The tool writes this value in the DEF UNITS DISTANCE MICRONS statement.

3. If you have rotated vias and non-rotated rules (NDRs), ensure that you export incremental DEF for rotated vias and NDRs so that they are included in the DEF file.

4. Ensure that the following options are deselected so that their corresponding DEF sections are excluded from the DEF file:

   a. Rows/Tracks/GCells

   b. Regions/Groups

   c. Blockages

   d. Scan chain

   e. Fixed cells only/Placed cells only

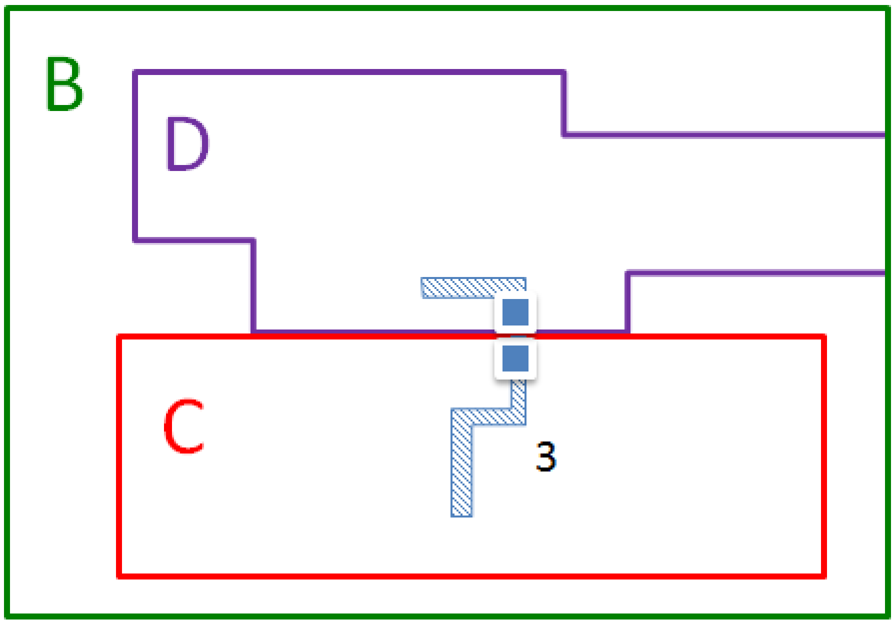
   f. Notch/Gap

   g. P/G metal fill

However, you must include non-routed nets that are required to connect two sub-DEF files. The followinf figure shows that in B.def, net 3 connects C.def to D.def without having been routed through B.def, as follows:

```
-net3 (C pin1) (D pin2)
```

Figure F-1. Non-Routed Nets Included in DEF File



5. Ensure that the following options are selected so that their corresponding DEF sections are included in the DEF file:

    a. Pins

    b. Floating metal fill

    c. Output all design nondefault rules

        This section is supported only for DEF version 5.6.

    d. Vias

        However, if you have an option to include all vias, deselect it.

    e. Components

    f. Nonstandard cells

    g. Nets

        However, ensure that you exclude any output diode pins.

    h. Special nets

        However, exclude notch gaps or power and ground metal fill.

6. Set the following file processing options.

    a. Enable the legalization capability.

    b. Compress the output, which provides a .gzip format.

    c. Enable the verbose option to display all information and warning messages in the session transcript.

Generating DEF from ATopTech APRISA

In ATopTech APRISA, generate DEF files with the write_def command.

Procedure

The write_def command writes out a DEF file down to, but not including, the standard cells. The LEF file provided by the library vendor includes the complete standard cell information. Specify the write_def command as follows:

```
write_def DEF_filename
```

Note

Do not use the write_def -all switch. The -all switch includes standard cell definitions in the DEF in addition to those already in the LEF.

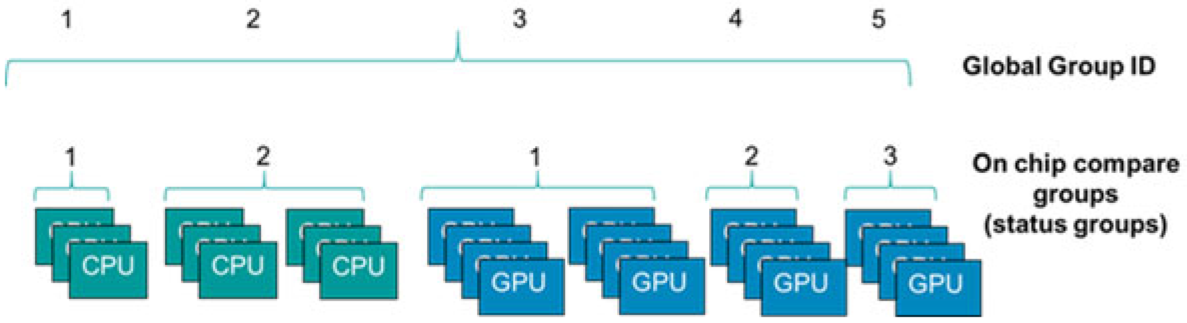
Appendix G
Logging Failures for SSN On-Chip Compare

You can use the on-chip compare capability of the SSH to test multiple identical cores simultaneously. Identical core instances can be grouped into one or more on-chip compare status groups.

See the section "On-Chip Compare Mode Setup" of the topic On-Chip Compare With SSN in the *Tessent Shell User's Manual* for more details on On-Chip Compare status groups. The tool uses this global group ID information to provide the correct core instance names that are part of a status group in the failure file prior to failure mapping.

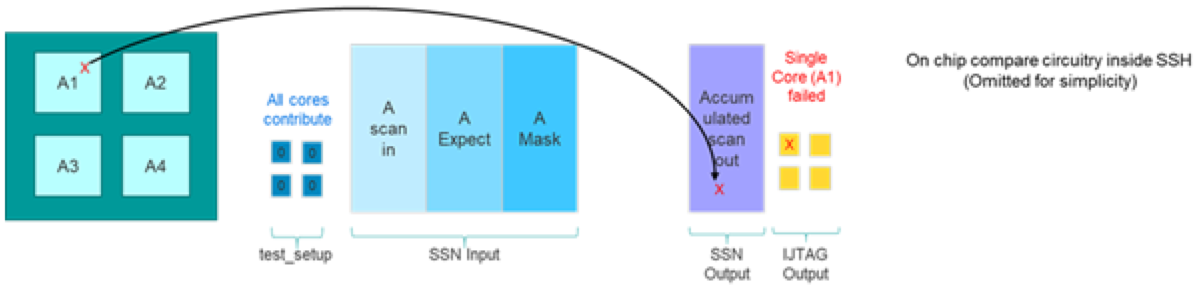
Figure G-1 shows two distinct cores, CPU and GPU, each with multiple instances in the design. The global group ID 1 comprises three CPU core instances, and the global group ID 2 comprises six CPU core instances. Similarly, the global group ID 3 comprises eight GPU core instances, and global group ID 4 and global group ID 5 each comprise 4 GPU core instances.

Figure G-1. Global and Status Groups



You apply scan test patterns to all core instances in a status group. The SSH asserts a register bit, called a sticky status bit, when it detects a miscompare during the scan test. Each sticky status bit indicates the pass or fail status of a single core instance within a status group. You can unload the sticky status bits via the IJTAG output (TDO) pin.

Figure G-2. Single Core Failure Example



As shown in Figure G-2, you test all core instances in a status group together; each instance, by default, "contributes" failing cycles that occur in the status group to the SSN output. If the single-core instance A1 fails during the test, the SSN output contains failing cycles for core instance A1, and the tool asserts the sticky status bit associated with core A1 at the end of the test. If multiple core instances fail within a status group, by default, the SSN output comprises the failing cycles of all the contributing core instances.

Scan diagnosis requires failing cycle information for each failing core instance. The tester logs these failing cycles in the failure log. To perform scan diagnosis on the design, you run the tester failure log through a failure mapping process to create a failure file that can be used during diagnosis. During the failure mapping process, the tool translates the tester failure log to the failing pattern and cycle for the specific failing core instance.

Scan diagnosis for on-chip compare may require special handling of the test application and failure mapping, as described in the following sections.

Testing and Failure Logging Process

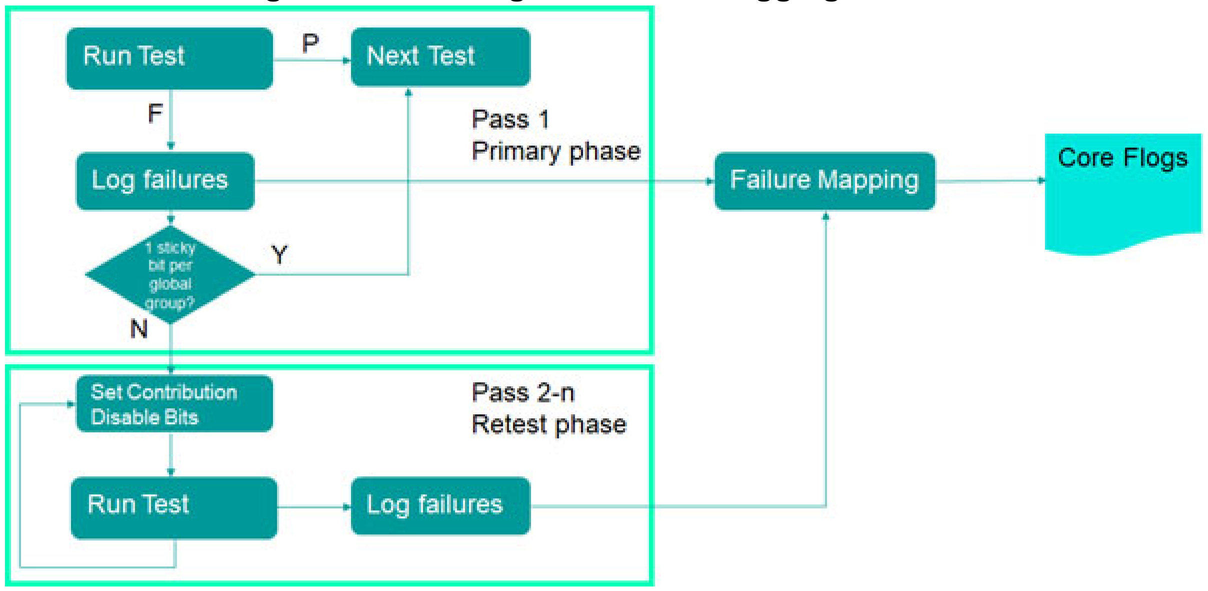
This section provides an overview of the testing and failure logging process for multiple instances contributing to a status group.

When testing designs that use on-chip compare, you first apply the test patterns to the design. This is the primary phase. If there are no failing cycles or patterns, the tests have passed.

If you observe any failing cycles or patterns, you use the asserted bits in the global status group to determine the next steps for test application and failure logging prior to running diagnosis.

The following figure shows an overview of the testing and failure logging process for each global status group.

Figure G-3. Testing and Failure Logging Process



The following scenarios describe actions you must take for successful scan diagnosis.

Scenario 1: No Sticky Status Bits are Asserted in any Global Status Group

If the testing did not assert any sticky status bits for any of the global status groups during the primary test phase, all core instances in each of the respective status groups passed. Failures observed on the SSN output correspond to the cores in the design that are not part of the on-chip compare groups. You can pass the failure files through failure mapping using the retargeted TCD file to produce a core-level failure log for diagnosis.

Scenario 2: A Single Sticky Status Bit is Asserted

In the primary test phase, if the tool asserts a single sticky status bit in a global status group, it indicates there is only one failing core instance. Multiple global status groups can each have a single sticky status bit in this scenario.

The failures logged for the on-chip compare groups have all the information needed for scan diagnosis after failure mapping.

Before running failure mapping, the failure file must indicate the phase of the test and the failing core instances for each of the global status groups. This information is contained in the failure file used for failure mapping using the keywords ssn_on_chip_compare_test_phase primary, ssn_on_chip_compare_enabled_failing_instances_begin, and ssn_on_chip_compare_enabled_failing_instances_end.

```
format cycle
ssn_on_chip_compare_test_phase primary
ssn_on_chip_compare_enabled_failing_instances_begin
   corea_ssh3
   corea_ssh5
   corea_ssh7
   coreb_ssh2
   coreb_ssh3
ssn_on_chip_compare_enabled_failing_instances_end
…
  failures_begin
```

You must also include the list of all failing core instances for each status group. You can produce the failing core instance names by analyzing the sticky status bit and the annotations in the STIL file. as described in Scenario 3.

In the primary phase, if multiple failure core instances contribute to a single status group, failure mapping does not map the multiple failing instances in a status group because there is insufficient information for each of the failing instances. The tool reports the following warning:

```
//Warning: Multiple SSH ICL instance names have been specified for global
compare group <#> of SSH ICL module '<ssh_module_name>' in the
'ssn_on_chip_compare_enabled_failing_instances' section.
Please note that the failures of those instances will be ignored, i.e.,
those failures will not be mapped to the core level.
To map the failures of this compare group, please perform the retest and
specify one failing SSH ICL instance name of this global compare group in
the 'ssn_on_chip_compare_enabled_failing_instances' section of that
failure file.
```

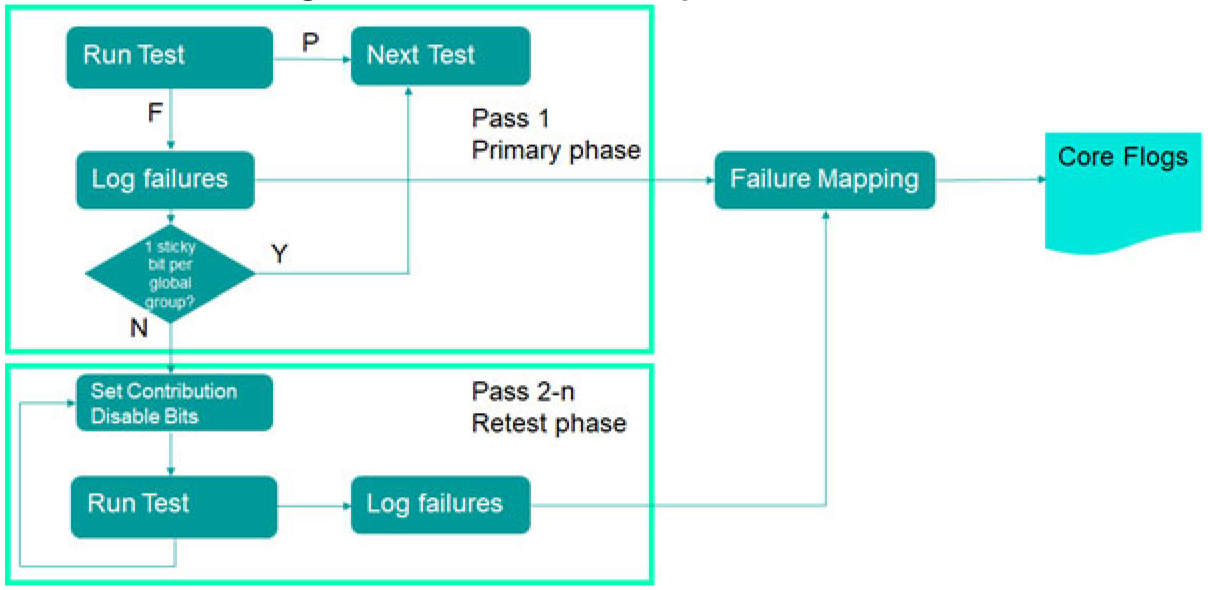
Scenario 3: Multiple Failing Instances per Status Group

When there are multiple failing instances within a status group, the tool combines the failures from all instances into the same packets on the SSN bus, and there is no way to distinguish which bits are associated with each core instance. This necessitates entering a failure collection loop where only one core instance contributes to the status group. Do the following:

1. Run the test and read the sticky_status bits from the SSN_end section to determine which instances failed.
2. Patch the contribution disable bits in the SSN_Setup section to disable all cores except for one of the failing cores in each status group.
3. Re-run the test and log the failures.
4. Repeat steps 2 and 3 until all failing core instances have been logged.

The testing process consists of two phases: primary and retest. The primary phase is the initial run of the SSN test. In this phase, all core instances contribute to their respective status groups. The tool collects the failure data for all cores in the test. If there is a need to re-run the test to collect specific core failures, then this phase is referred to as the retest phase. For each test, there is only one primary phase, but you can have as many retest phases as necessary to collect the failure data.

Figure G-4. OCC With Multiple Core Failures



Pattern Annotations

There are many annotations in the SSN pattern files that define the SSN structure and location of certain operations. For the process of failure collection with on-chip compare, there are three sections that you must use. While the patterns can be written in either STIL or WGL, only the STIL syntax is described here.

Active SSH Section

The first section is the Active SSH section. This is located at the beginning of the pattern file where the payload vectors are written. The tool uses this section to determine the core instances in each status group. The two key pieces of information are the ICL instance name and the global status group. You must create a list of cores in each global status group. The tool uses this information during the retest phase to ensure that it selects no more than one core instance per status group. This section is shown here:

```
        Ann {*   Begin_Active_Ssh_Section *}
        Ann {*     instance                                              =
                   {GPS_2/gps_baseband_rtl1_tessent_ssn_scan_host_1_inst} *}
        Ann {*     icl_instance                                          =
                   {GPS_2.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst} *}
        Ann {*     bus_width                                             = 2 *}
        Ann {*     packet_size                                           = 16 *}
        Ann {*     bits_per_packet                                       = 16 *}
        Ann {*     capture_packets                                       = 7 *}
        Ann {*     packets_per_capture_pulse                             = 2 *}
    …
        Ann {*     initial_bit0_position                                 = 0 *}
        Ann {*     initial_bit0_position_of_packet                       = 0 *}
        Ann {*     cycles_until_first_packet                             = 7 *}
        Ann {*     delay_cycles_in_packet                                = 0 *}
        Ann {*     offset                                                = 8 *}
        Ann {*     extra_shift_packets                                   = 0 *}
        Ann {*     delay_packets                                         = 0 *}
        Ann {*     clock_multiplier                                      = 1 *}
        Ann {*     on_chip_compare                                       = on *}
        Ann {*     on_chip_compare_capture_group_count                   = 1 *}
        Ann {*     on_chip_compare_capture_group                         = 1 *}
        Ann {*     on_chip_compare_capture_global_group_count            = 1 *}
        Ann {*     on_chip_compare_capture_global_group                  = 1 *}
        Ann {*     min_shift_clock_low_width                             = 4 *}
        Ann {*     min_capture_clock_low_width                           = 12 *}
        Ann {*     total_shift_count                                     = 146 *}
        Ann {*     from_scan_out_bits                                    = 4 *}
        Ann {*   End_Active_Ssh_Section *}
```

Note

Use the "on_chip_compare_capture_global_group" and not the "on_chip_compare_capture_group". The global group is unique across all cores, but the capture group is local to the core type.

IJTAG Bit Location Annotations

You can identify the sticky_status and contribution disable bits by annotations in the pattern file. These are instructions on the IJTAG network and have the "TESSENT_PRAGMA" keyword. They are either a read operation on the TDO pin or a write operation on the TDI pin. Since the IJTAG network operates at a much slower speed than the SSN bus, the IJTAG portions of the pattern can use cycle-scaling to keep the same period on the ATE. When you do this, it spreads each IJTAG cycle over a number of cycles as defined by the tck_ratio. This example illustrates a pattern where the IJTAG cycles are spread into 8 SSN cycles. In the IJTAG portions of the pattern, you have the following annotation:

```
Ann {* TESSENT_PRAGMA procedure ssn_setup -tck_ratio 8 *}
```

To determine which ATE tester cycle they are located on, you must identify in which ATE cycle the annotation occurs and then read the "relative cycles" value. Then, multiply this value by the TCK ratio and add it to the cycle where the annotation occurs to determine the cycle where that variable begins.

Ann {* TESSENT_PRAGMA variable

```
Ann {* TESSENT_PRAGMA variable
GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst.disable_on_chip_comp
are_contribution -type write -var_bits {0} -pin TDI -relative_cycles {64}
*}
```

1. Identify cycle with annotation
2. Begin edit at annotation cycle + **relative_cycles** X tck_ratio
3. Make edits for **tck_ratio** cycles

Note

If there is no tck_ratio annotation in the pattern, then it is not scaled and the tck_ratio is 1.

Sticky Status Bits

The tool shifts the sticky_status bits out of the IJTAG network on TDO during the SSN_end section of the test. There is one bit for each core instance that uses on-chip compare to communicate which instance is failing. The following example illustrates the sticky_status bit annotation in the ssn_end section of the test:

```
Ann {* TESSENT_PRAGMA variable
GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst.sticky_status -type
read -var_bits {0} -pin TDO -relative_cycles {12} *}
```

You must multiply the relative cycles by the tck_ratio and then add this to the cycle where the annotation occurred to determine the beginning of that variable. In this case, the sticky status bit for the core "GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst" begins 96 cycles after this annotation. If this bit fails during the test, this core instance has failed.

Contribution Disable Bits

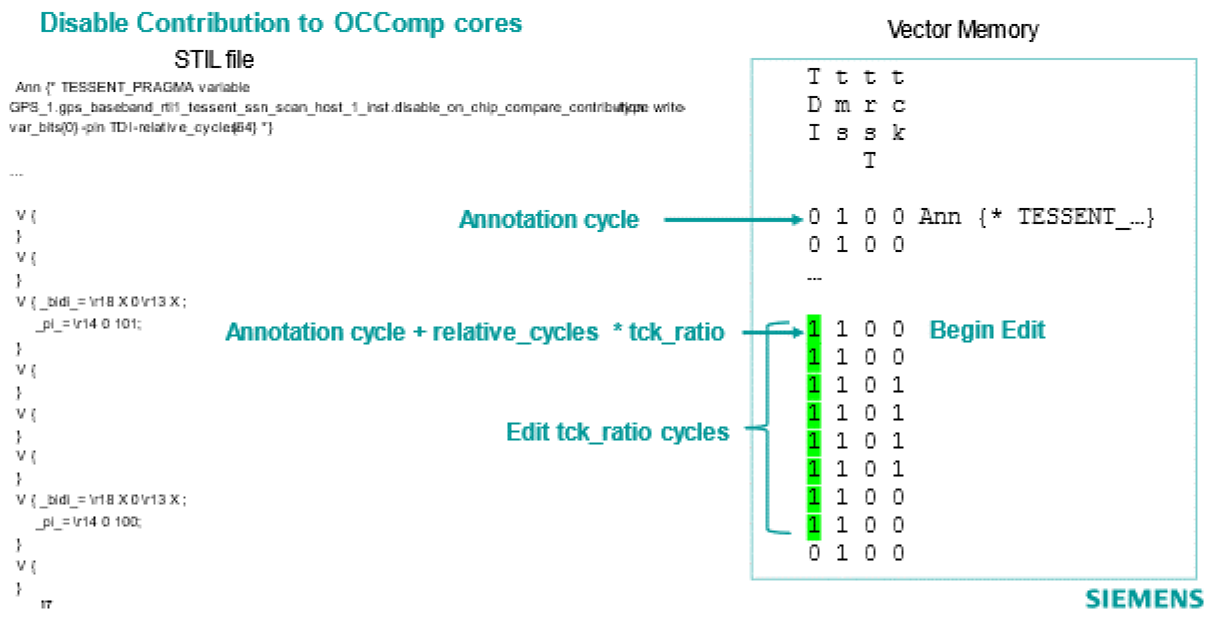
During the retest phase, the ATE test program must disable the output of some cores from contributing to the SSN bus. It does this by setting the contribution disable bits for that particular core. This example illustrates the contribution disable bit for the instance "GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst".

```
Ann {* TESSENT_PRAGMA variable
GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst.disable_on_chip_comp
are_contribution -type write -var_bits {0} -pin TDI -relative_cycles {64}
*}
```

Again, to determine where this variable begins shifting into TDI, you multiply the relative cycles by the tck_ratio and add this to the cycle where the annotation occurs. In this case, it begins 512 cycles after the annotation.

This is where the ATE test program must “patch” the vector data to change the value being shifted, but the tck_ratio determines the number of cycles to be patched.

Figure G-5. Contribution Disable Bits


Disable Contribution to OCComp cores
STIL file
Vector Memory
Annotation cycle
Annotation cycle + relative_cycles * tck_ratio
Edit tck_ratio cycles
Begin Edit
SIEMENS

```
Ann {* TESSENT_PRAGMA variable
GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst.disable_on_chip_compare_contribution write-
var_bits(0) -pin TDI -relative_cycles(64) *}

...

V {
}
V {
}
V { _bidi_ = \r18 X 0 \r13 X ;
    _pi_ = \r14 0 101;
}
V {
}
V {
}
V {
}
V { _bidi_ = \r18 X 0 \r13 X ;
    _pi_ = \r14 0 100;
}
V {
}
```

```
T t t t
D m r c
I s s k
    T

0 1 0 0 Ann {* TESSENT_...}
0 1 0 0
...
1 1 0 0
1 1 0 0
1 1 0 1
1 1 0 1
1 1 0 1
1 1 0 1
1 1 0 0
1 1 0 0
0 1 0 0
```

Primary Phase

During the primary phase, the ATE test program enables all of the core instances for each compare group so that it writes any failures to the SSN bus, which are observed on the ATE. The ATE test program logs these failures and you must ensure that these failures conform to the Tessent failure file format.

Note

If your ATE can log these failures to STDF, and provided that the ATE can use the supported records and fields to log the core instances and the primary phase, you can use the Tessent dlogutil utility to extract the data into Tessent failure files.

See “dlogutil Features for SSN On-Chip Compare” on page 503 for details.

The ATE test program must read the sticky status bits to determine which on-chip compare core instances have failed. You must ensure that these instances are written into the Tessent failure file as follows:

```
ssn_on_chip_compare_enabled_failing_instances begin
   failing_instance_name
   failing_instance_name
   failing_instance_name
ssn_on_chip_compare_enabled_failing_instances end
```

In this phase, there should be one failing instance written to the failure file for each failing sticky_status bit. You must also ensure that the ATE test program writes the test phase to the Tessent failure file as follows:

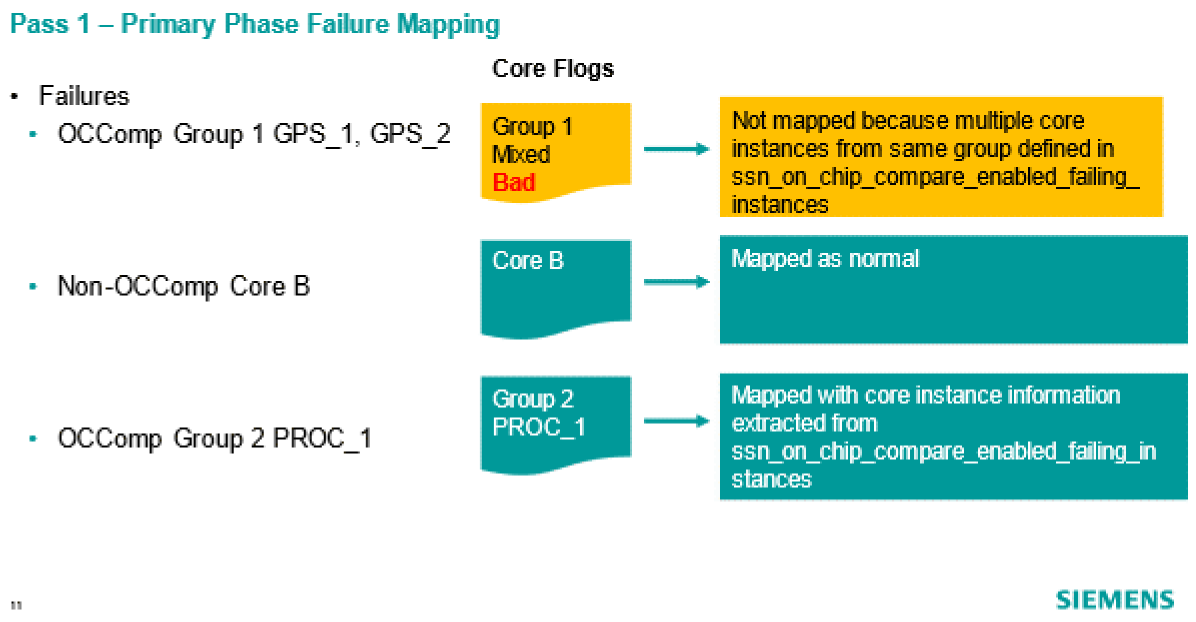
```
ssn_on_chip_compare_test_phase primary
```

Tessent failure mapping uses these keywords in the failure mapping step to ensure that it maps the failures to the correct core instances for diagnosis. In the case where multiple failing instances contribute to the same status group, Tessent failure mapping does not map these failures and reports a warning. Consider the following example, where one SSN test contains one non-on-chip compare group and two on-chip compare groups. Group 1 is an on-chip compare group where two core instances are failing. Core B is a non-on-chip compare core and Group 2 is an on-chip compare group with only one failing core instance. The following example shows the keywords are written to the fail log:

```
ssn_on_chip_compare_test_phase primary
ssn_on_chip_compare_enabled_failing_instances begin
   GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst
   GPS_2.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst
   PROC_2.processor_core_rtl1_tessent_ssn_scan_host_1_inst
ssn_on_chip_compare_enabled_failing_instances end
```

This then goes to the failure mapping step where the tool generates the core-level fail logs. The test phase keyword tells failure mapping to map the failures for the non-on-chip compare cores and not the on-chip compare cores with multiple failing cores in the status group. In this example, the tool does not map the Group 1 failures. It maps the failures from the non-on-chip compare Core B, and the failures from the processor core (Group 2) because only one instance failed from that group.

Figure G-6. Primary Phase Failure Mapping


Pass 1 – Primary Phase Failure Mapping
Core Flogs
Failures
OCComp Group 1 GPS_1, GPS_2
Group 1 Mixed Bad
Not mapped because multiple core instances from same group defined in ssn_on_chip_compare_enabled_failing_instances
Non-OCComp Core B
Core B
Mapped as normal
OCComp Group 2 PROC_1
Group 2 PROC_1
Mapped with core instance information extracted from ssn_on_chip_compare_enabled_failing_instances
SIEMENS

If a status group has more than one failing instance, then you must go through the retest phase to capture the failure data for diagnosis. The test program determines which core instances need the retest phase.

Retest Phase

The retest phase is solely for the collection of failure data from on-chip compare groups with multiple failing instances. If only one instance failed from the group, that group does not need to be retested and the ATE test program must disable all instances from those groups. This ensures that you do not diagnose the same failing instance multiple times when it is not necessary.

For each group with multiple failing instances, disable all cores except one of the failing instances by setting the contribution disable bit in the SSN_setup portion of the test. Then re-run the test and write the failures to the Tessent failure file.

Note

If your ATE can log these failures to STDF, and provided that the ATE can use the supported records and fields to log the retest phase and core instance, you can use the Tessent dlogutil utility to extract the data into Tessent failure files.

See "dlogutil Features for SSN On-Chip Compare" on page 503 for details.

You must ensure that the Tessent failure file contains the test phase, as shown below:

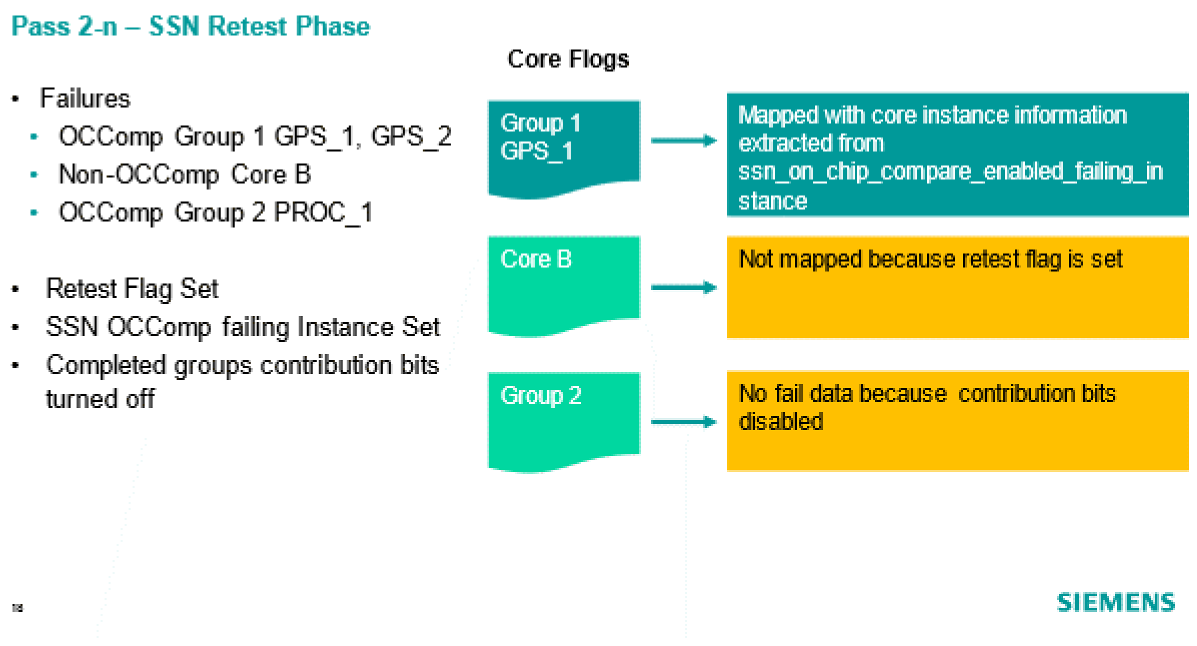
```
ssn_on_chip_compare_test_phase retest
```

In the retest phase, the ATE test program runs the same test, which means that it tests all cores within a status group. It sets the sticky_status bits for all failing cores which, therefore, does not indicate which instance it logs. The ATE test program must write the core instance that remains enabled to the Tessent failure file as follows:

```
ssn_on_chip_compare_enabled_failing_instances begin
   GPS_1.gps_baseband_rtl1_tessent_ssn_scan_host_1_inst

ssn_on_chip_compare_enabled_failing_instances end
```

During the retest phase, the ATE test program can only log one instance per status group. If there are multiple instances from a status group written to the failure file, Tessent failure mapping reports an error. Because the on-chip compare cores are already mapped during the primary phase, Tessent failure mapping does not map them during the retest phase to avoid duplication.

Figure G-7. Pass 2-n Retest Phase


Pass 2-n – SSN Retest Phase
Failures
OCComp Group 1 GPS_1, GPS_2
Non-OCComp Core B
OCComp Group 2 PROC_1
Retest Flag Set
SSN OCComp failing Instance Set
Completed groups contribution bits turned off
Core Flogs
Group 1 GPS_1
Mapped with core instance information extracted from ssn_on_chip_compare_enabled_failing_instance
Core B
Not mapped because retest flag is set
Group 2
No fail data because contribution bits disabled
SIEMENS

The ATE test program must reset the contribution disable bits before testing the next device.

Failure Mapping

Use the "set_failure_mapping_options -on_chip_compare_verbose_messages" command and switch to control the listing of ICL instance names in on-chip compare error and warning messages. The default setting is off. Turn this setting on to include the ICL instance names, which can be long depending on the IJTAG network in your design.

The tool performs the following checks during failure mapping. Example error or warning messages that may result are shown.

1. Each ICL instance name corresponds to an SSH.

   a. The tool can report the following error message:

   ```
   // Error: The specified SSH ICL name ’<ssh_name>’ does not match
   any SSH in the current design.

      Please verify that your pattern file matches the top-level
   failure file and/or that the names specified in the
   ’ssn_on_chip_compare_enabled_failing_instances’ section are
   correct and re-read the appropriate files.
   ```

2. Specified SSH instances are operating in on-chip compare mode.

   a. The tool can report the following error message:

   ```
   // Error: The specified SSH ICL name ’<ssh_name>’ is not
   configured for on-chip compare mode in the current design.

      Please verify that your pattern file matches the top-level
   failure file and/or that the names specified in the
   ’ssn_on_chip_compare_enabled_failing_instances’ section are
   correct and re-read the appropriate files.
   ```

3. A compare group with multiple on-chip compare instances has failures in the top-level failure file. However, the group has no SSH specified in the ssn_on_chip_compare_enabled_failing_instances section.

   The tool performs this check on core instances with failures in the failure file and ignores passing instances. It performs this check for both the primary and retest phases.

   a. The tool can report the following error messages in the primary phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages off”:

   ```
   // Error: Global compare group <#> of SSH ICL module
   ’<ssh_module_name>’ has failures in the failure file, but none of
   its SSH ICL instance names were specified in the
   ’ssn_on_chip_compare_enabled_failing_instances’ section.

      Please ensure that all failing instances are represented in
   the ’ssn_on_chip_compare_enabled_failing_instances’ section of
   the failure file.
   ```

b. The tool can report the following error messages in the primary phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages on”:

```
// Error: Global compare group <#> of SSH ICL module
'<ssh_module_name>' has failures in the failure file, but none of
the following SSH ICL instance names were specified in the
'ssn_on_chip_compare_enabled_failing_instances' section:
'<ssh1_icl_name_of_ssh_module_name_in_group_#>'
'<ssh2_icl_name_of_ssh_module_name_in_group_#>'
…
'<sshN_icl_name_of_ssh_module_name_in_group_#>'

   Please ensure that all failing instances are represented in
the 'ssn_on_chip_compare_enabled_failing_instances' section of
the failure file.
```

c. The tool can report the following error messages in the retest phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages off”:

```
// Error: Global compare group <#> of SSH ICL module
'<ssh_module_name>' has failures in the failure file, but none of
its SSH ICL instance names were specified in the
'ssn_on_chip_compare_enabled_failing_instances' section.

   Please ensure that only one instance is contributing to its
status group and that instance is represented in the
'ssn_on_chip_compare_enabled_failing_instances' section of the
failure file.
```

d. The tool can report the following error messages in the retest phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages on”:

```
// Error: Global compare group <#> of SSH ICL module
'<ssh_module_name>' has failures in the failure file, but none of
the following SSH ICL instance names were specified in the
'ssn_on_chip_compare_enabled_failing_instances' section:
'<ssh1_icl_name_of_ssh_module_name_in_group_#>'
'<ssh2_icl_name_of_ssh_module_name_in_group_#>'
…
'<sshN_icl_name_of_ssh_module_name_in_group_#>'

   Please ensure that only one instance is contributing to its
status group and that instance is represented in the
'ssn_on_chip_compare_enabled_failing_instances' section of the
failure file.
```

4. A compare group has multiple SSH instances.

   The tool performs this check on core instances with failures in the failure file and ignores passing instances. It performs this check for both the primary and retest phases.

a. The tool can report the following warning message in the primary phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages off”:

```
// Warning: Multiple SSH ICL instance names have been specified
for global compare group <#> of SSH ICL module
'<ssh_module_name>' in the
'ssn_on_chip_compare_enabled_failing_instances' section.

Please note that the failures of those instances will be ignored,
i.e., those failures will not be mapped to the core level.

To map the failures of this compare group, please perform the
retest and specify one failing SSH ICL instance name of this
global compare group in the
'ssn_on_chip_compare_enabled_failing_instances' section of that
failure file.
```

b. The tool can report the following warning message in the primary phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages on”:

```
// Warning: Multiple SSH ICL instance names have been specified
for global compare group <#> of SSH ICL module
'<ssh_module_name>' in the
'ssn_on_chip_compare_enabled_failing_instances' section:
'<ssh1_icl_name_of_ssh_module_name_in_group_#>'
'<ssh2_icl_name_of_ssh_module_name_in_group_#>'
…
'<sshN_icl_name_of_ssh_module_name_in_group_#>'

Please note that the failures of those instances will be ignored,
i.e., those failures will not be mapped to the core level.

To map the failures of this compare group, please perform the
retest and specify one failing SSH ICL instance name of this
global compare group in the
‘ssn_on_chip_compare_enabled_failing_instances' section of that
failure file.
```

c. The tool can report the following error message in the retest phase when you specify “set_failure_mapping_options -on_chip_compare_verbose_messages off”:

```
// Error: Multiple SSH ICL instance names have been specified for
global compare group <#> of SSH ICL module '<ssh_module_name>' in
the 'ssn_on_chip_compare_enabled_failing_instances' section.

Please ensure that only one instance is contributing to its
status group and that instance is represented in the
'ssn_on_chip_compare_enabled_failing_instances' section of the
failure file.
```

d. The tool can report the following error message in the retest phase when you specify "set_failure_mapping_options -on_chip_compare_verbose_messages on":

```
// Error: Multiple SSH ICL instance names have been specified for
global compare group <#> of SSH ICL module '<ssh_module_name>' in
the 'ssn_on_chip_compare_enabled_failing_instances' section:
<ssh1_icl_name_of_ssh_module_name_in_group_#>
<ssh2_icl_name_of_ssh_module_name_in_group_#>
…
<sshN_icl_name_of_ssh_module_name_in_group_#>

Please ensure that only one instance is contributing to its
status group and that instance is represented in the
'ssn_on_chip_compare_enabled_failing_instances' section of the
failure file.
```

5. Use of the ssn_on_chip_compare_test_phase keyword.

a. The tool can report the following error message:

```
// Error: The 'ssn_on_chip_compare_test_phase' keyword was not
specified in the failure file.

   This keyword is required when there are multiple SSH instances
in the same global compare group.

   Please specify the 'ssn_on_chip_compare_test_phase primary |
retest' keyword/value pair in the failure file and re-read the
file.
```

Note

If you specify the ssn_on_chip_compare_test_phase keyword, the tool does not require the ssn_on_chip_compare_enabled_failing_instances section in the primary phase as long as there are no compare groups with multiples instances contributing to their status. If you do not specify the ssn_on_chip_compare_enabled_failing_instances section, the tool assumes all on-chip compare SSHs are enabled.

However, if you specify the ssn_on_chip_compare_test_phase keyword in the retest phase, the tool requires the ssn_on_chip_compare_enabled_failing_instances section.

6. Use of the ssn_on_chip_compare_enabled_failing_instances section in the retest phase.

a. The tool can report the following error message:

```
// Error: The 'ssn_on_chip_compare_enabled_failing_instances'
section was not specified in the failure file for the retest
phase.

   Please ensure that only one instance is contributing to its
status group and that instance is represented in the
'ssn_on_chip_compare_enabled_failing_instances' section of the
failure file.
```

Collecting Failure Data on Automated Test Equipment (ATE)

Proper data collection collects the scan failures from the SSN payload patterns. The sticky_status bits must also be collected from the SSN_end portion of the test. You must take care to ensure that the sticky_status bits are collected even if the failures in the payload patterns fill the failure buffer during test.

ATE With Per-Pin Failure Buffers

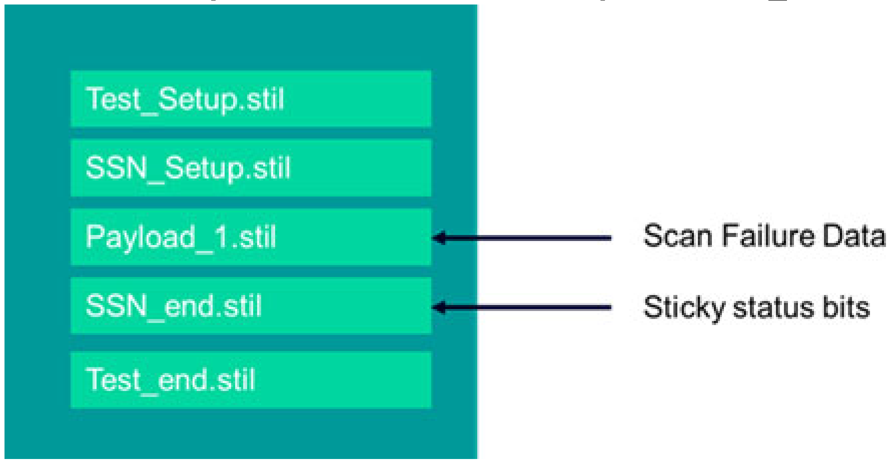
Some ATE can set a failure buffer on a per-pin basis. This enables the remaining pins to continue collecting the failure data even after the ATE has reached the buffer limit on other pins, and ensures that the sticky_status bits can be logged for all patterns. This feature is a configurable option on some ATE. We recommend that you consult with the field applications teams for your ATE to ensure it is implemented correctly.

ATE With Central Failure Buffers

If you cannot set buffer limits on a per-pin basis on your ATE, you must split the pattern and run the SSN_end portion separately from the payload to collect the failures on the sticky_status bits. During this operation, you must maintain power to the DUT between the patterns to ensure that the sticky_status bits are not lost. For details on how to write the patterns to separate the SSN_end portion from the payload, see How To Write JTAG and Payload Procedures Separately in the *Tessent Shell User's Manual*.

The resulting pattern looks like Figure G-8.

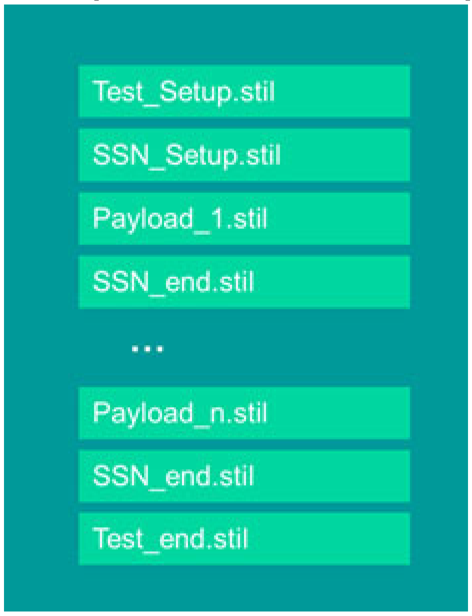
Figure G-8. Example of Pattern Set With Separate SSN_end Portion



Collecting Failure Data When the Payload is Split

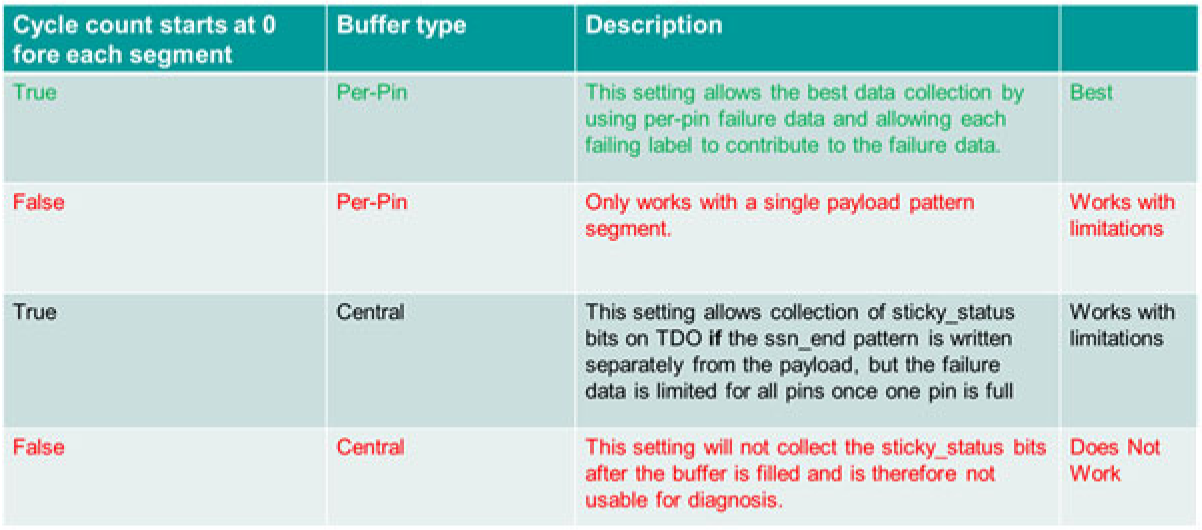
Many times, you may want to split large payload patterns into multiple smaller patterns. With SSN, this can impact the failure collection process and may require extra steps. After each payload segment, you must run the SSN_end pattern to collect the sticky_status bits and prepare the SSN for the next payload pattern. Figure G-9 shows an example of the resulting patterns.

Figure G-9. Example of Pattern Set With Split Payload



For failure mapping to work correctly when the payload is split into multiple files, make sure the ATE creates the failure log where the cycle count begins at 0 for each payload segment. For details, see the chart in Figure G-10.

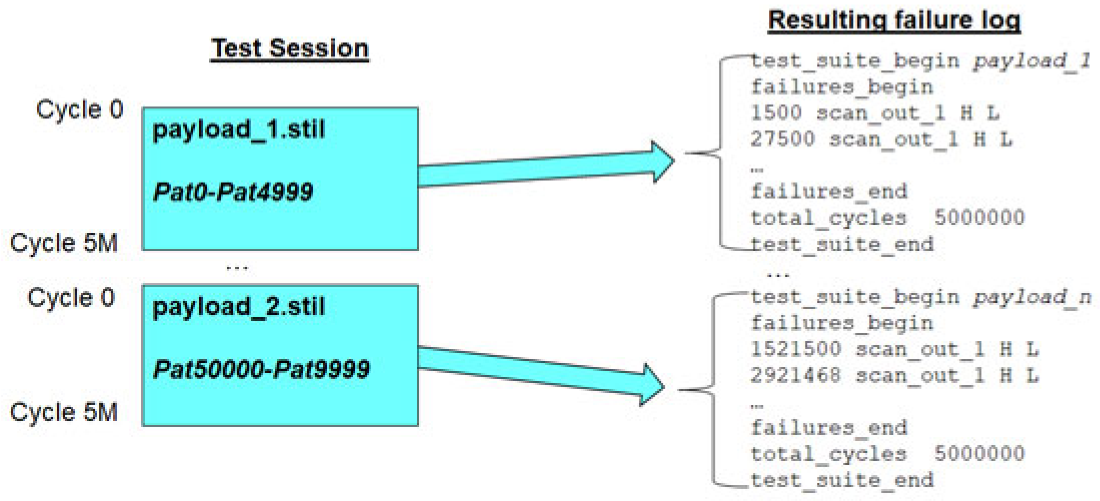
Figure G-10. Cycle Count for Multiple Payload Segments



Cycle count starts at 0 fore each segment	Buffer type	Description	
True	Per-Pin	This setting allows the best data collection by using per-pin failure data and allowing each failing label to contribute to the failure data.	Best
False	Per-Pin	Only works with a single payload pattern segment.	Works with limitations
True	Central	This setting allows collection of sticky_status bits on TDO if the ssn_end pattern is written separately from the payload, but the failure data is limited for all pins once one pin is full	Works with limitations
False	Central	This setting will not collect the sticky_status bits after the buffer is filled and is therefore not usable for diagnosis.	Does Not Work

When creating the Tessent failure file for this type of setup, the fail log must contain only the failure data from the payload patterns, because the tool can only read the payload patterns for failure mapping. You must place the failures from each payload pattern into a test suite in the Tessent failure file.

Figure G-11. Test Suites for Each Payload

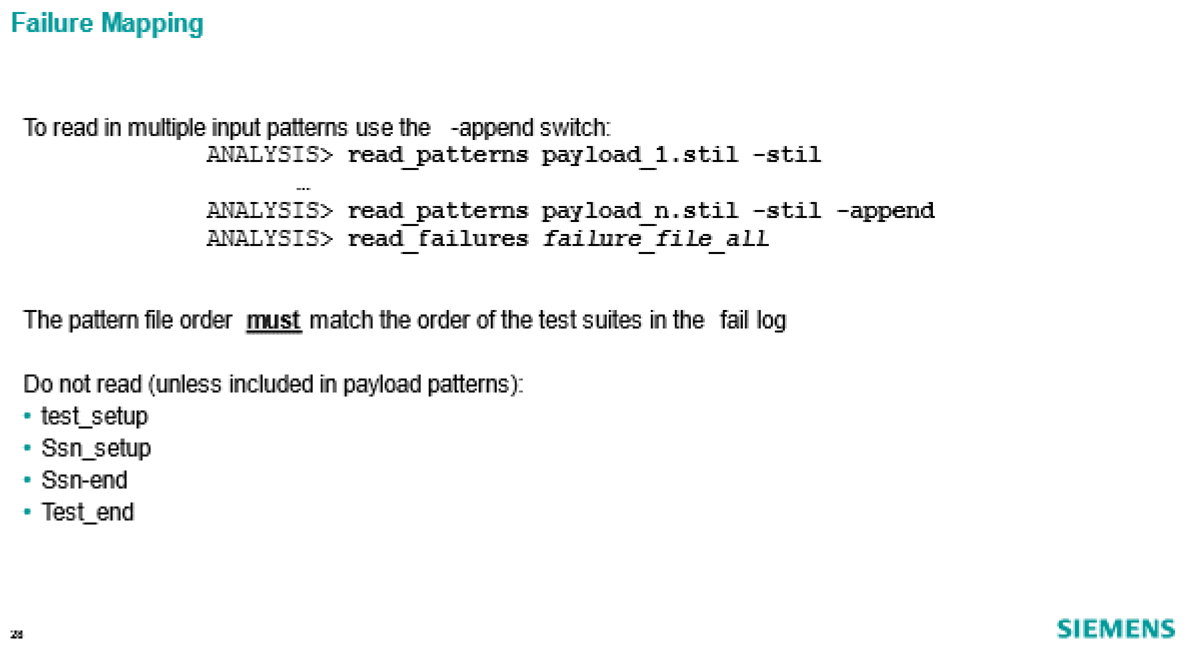


```
test_suite_begin payload_1
failures_begin
1500 scan_out_1 H L
27500 scan_out_1 H L
...
failures_end
total_cycles  5000000
test_suite_end
 ...
test_suite_begin payload_n
failures_begin
1521500 scan_out_1 H L
2921468 scan_out_1 H L
...
failures_end
total_cycles  5000000
test_suite_end
```

Note

Tessent failure file mapping can only read the payload patterns. Make sure the order in which it reads the patterns matches the order of test suites in the Tessent failure file.

Figure G-12. Failure Mapping



Failure Mapping

To read in multiple input patterns use the -append switch:

```
ANALYSIS> read_patterns payload_1.stil -stil
     ...
ANALYSIS> read_patterns payload_n.stil -stil -append
ANALYSIS> read_failures failure_file_all
```

The pattern file order **must** match the order of the test suites in the fail log

Do not read (unless included in payload patterns):

- test_setup
- Ssn_setup
- Ssn-end
- Test_end

Appendix H Getting Help

There are several ways to get help when setting up and using Tessent software tools. Depending on your need, help is available from documentation, online command help, and Siemens EDA Support.

The Tessent Documentation System

At the center of the documentation system is the InfoHub that supports both PDF and HTML content. From the InfoHub, you can access all locally installed product documentation, system administration documentation, videos, and tutorials. For users who want to use PDF, you have a PDF bookcase file that provides access to all the installed PDF files.

For information on defining default HTML browsers, setting up browser options, and setting the default PDF viewer, refer to the "Documentation Options" in the *Siemens® Software and Mentor® Documentation System* manual.

You can access the documentation in the following ways:

- **Shell Command** — On Linux platforms, enter mgcdocs at the shell prompt or invoke a Tessent tool with the -manual invocation switch.
- **File System** — Access the Tessent InfoHub or PDF bookcase directly from your file system, without invoking a Tessent tool. For example:

  HTML:

  ```
  firefox <software_release_tree>/doc/infohubs/index.html
  ```

  PDF:

  ```
  acroread <software_release_tree>/doc/pdfdocs/_tessent_pdf_qref.pdf
  ```

- **Application Online Help** — ou can get contextual online help within most Tessent tools by using the "help -manual" tool command. For example:

  > help dofile -manual

  This command opens the appropriate reference manual at the "dofile" command description.

Global Customer Support and Success

A support contract with Siemens EDA is a valuable investment in your organization's success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

`https://support.sw.siemens.com`

If your site is under a current support contract, but you do not have a Support Center login, register here:

`https://support.sw.siemens.com/register`

Index

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.